# MACHINE LEARNING

## A First Course for Engineers and Scientists

Andreas Lindholm, Niklas Wahlström,
Fredrik Lindsten, Thomas B. Schön

Second Edition

This version: September 1, 2026

# Contents

# Acknowledgements

Many people have helped us throughout the writing of this book.

For the first edition of the book, we received valuable feedback from many students and other teacher colleagues. We are, of course, very grateful for each and every comment we have received; in particular, we want to mention David Sumpter, David Widmann, Adrian Wills, Johannes Hendricks, Mattias Villani, Dmitrijs Kass, and Joel Oskarsson. For the first edition we also received useful feedback on the technical content of the book, including the practical insights in Chapter 13, from Agrin Hilmkil (then at Peltarion), Salla Franzén and Alla Tarighati (then at SEB), Lawrence Murray (then at Uber), James Hensman and Alexis Boukouvalas (then at Secondmind), Joel Kronander and Nand Dalal (then at Nines), and Peter Lindskog and Jacob Roll (at Arriver, now part of Qualcomm). We also received valuable comments from Arno Solin on Chapter 9 and 10, and Joakim Lindblad on Chapter 6. Several people helped us with the figures illustrating the examples in Chapter 1, namely Antônio Ribeiro (Figure 1.1), Fredrik K. Gustafsson (Figure 1.4), and Theodoros Damoulas (Figure 1.5). Thank you all for your help!

Ahead of writing the second edition, we also received useful feedback from numerous anonymous GitHub users based on the first edition of the book, which all has been considered. For the second edition we have also received useful feedback from Birger Moëll, Ekta Vats and Oskar Allerbo.

During the writing of this book, we enjoyed financial support from AI Competence for Sweden, the Swedish Research Council (projects: 2016-04278, 2016-06079, 2017-03807, 2020-04122, 2021-04321), the Swedish Foundation for Strategic Research (projects: ICA16-0015, RIT12-0012), the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, ELLIIT, and the Kjell och Märta Beijer Foundation.

Finally, we are thankful to Lauren Cowles at Cambridge University Press for helpful advice and guidance through the publishing process and to Chris Cartwright for careful and helpful copyediting.

# Notation

| Symbol | Meaning |
|---|---|
| | *General mathematics* |
| $b$ | a scalar |
| $\mathbf{b}$ | a vector |
| $\mathbf{B}$ | a matrix |
| $^\mathsf{T}$ | transpose |
| $\text{sign}(x)$ | the sign operator; $+1$ if $x > 0$, $-1$ if $x < 0$ |
| $\nabla$ | del operator; $\nabla f$ is the gradient of $f$ |
| $\|\mathbf{b}\|_2$ | Euclidean norm of $\mathbf{b}$ |
| $\|\mathbf{b}\|_1$ | taxicab norm of $\mathbf{b}$ |
| $p(z)$ | probability density (if $z$ is a continuous random variable) or probability mass (if $z$ is a discrete random variable) |
| $p(z\|x)$ | the probability density (or mass) for $z$ conditioned on $x$ |
| $\mathcal{N}(z; m, \sigma^2)$ | the probability density for a random variable $z$ with a normal distribution with mean $m$ and variance $\sigma^2$ |
| $\mathbb{E}$ | expected value |
| | *The supervised learning problem* |
| $\mathbf{x}$ | input |
| $y$ | output |
| $\mathbf{x}_\star$ | test input |
| $y_\star$ | test output |
| $\widehat{y}(\mathbf{x}_\star)$ | a prediction of $y_\star$ |
| $\varepsilon$ | noise |
| $n$ | number of data points in training data |
| $\mathcal{T}$ | training data $\{\mathbf{x}_i, y_i\}_{i=1}^n$ |
| $L$ | loss function |
| $J$ | cost function |
| | *Supervised methods* |
| $\boldsymbol{\theta}$ | parameters to be learned from training data |
| $g(\mathbf{x})$ | model of $p(y \mid \mathbf{x})$ (most classification methods) |
| $\lambda$ | regularisation parameter |
| $\phi$ | link function (generalised linear models) |

| | |
|---|---|
| $h$ | activation function (neural networks) |
| $\mathbf{W}$ | weight matrix (neural networks) |
| $\mathbf{b}$ | offset vector (neural networks) |
| $\gamma$ | learning rate |
| $B$ | number of members in an ensemble method |
| $\kappa$ | kernel |
| $\boldsymbol{\phi}$ | nonlinear feature transformation (kernel methods) |
| $d$ | dimension of $\boldsymbol{\phi}$; number of features (kernel methods) |

*Evaluation of supervised methods*

| | |
|---|---|
| $E$ | error function |
| $E_{\text{new}}$ | new data error |
| $E_{\text{train}}$ | training data error |
| $E_{k\text{-fold}}$ | estimate of $E_{\text{new}}$ from $k$-fold cross validation |
| $E_{\text{hold-out}}$ | estimate of $E_{\text{new}}$ from hold-out validation data |


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September* 1, 2026, `http://smlbook.org`

# 1 Introduction

Machine learning is about learning, reasoning, and acting based on data. This is done by constructing computer programs that process data, extract useful information, make predictions regarding unknown properties, which can inform what actions to take or decisions to make. What turns data analysis into machine learning is that the process is automated and that the computer program is *learnt* from data. This means that generic computer programs are used, which are adapted to application-specific circumstances by automatically adjusting the settings of the program based on observed *training data*. It can therefore be said that machine learning is a way of *programming by example*. The beauty of machine learning is that it is quite arbitrary what the data represents, and we can design general methods that are useful for a wide range of practical applications. We illustrate this versatility via a range of examples below.

The "generic computer program" referred to above corresponds to a *mathematical model* of the data. When we develop and describe different machine learning methods, we do so using the language of mathematics. The model is a compact representation of the data that captures the key properties of the phenomenon we are studying. When implementing the method in practice, this mathematical model is translated into code. However, to understand what the computer program actually does, it is essential to understand the underlying mathematics.

The model is learned based on the available training data using a *learning algorithm* which automatically adjusts the settings, or *parameters*, of the model to agree with the data. In summary, the three cornerstones of machine learning are:

| 1. The data 2. The mathematical model 3. The learning algorithm. |
|---|

In this introductory chapter, we will give a taste of the machine learning landscape by illustrating these cornerstones with examples ranging from clinical diagnostics, sports analytics and material sciences to environmental engineering, computer vision and structural biology. We also give some advice on how to proceed through the rest of the book and, at the end, provide references to good books on machine learning for the interested reader who wants to dig further into this topic.

## 1.1 Machine Learning Exemplified

Machine learning is a multifaceted subject. Before going into the technical details, we will provide an intuitive answer to the question *"What is machine learning?"* by discussing application examples where it has achieved state-of-the-art results. We

start with a high-stakes application in cardiology.

**Example 1.1 Diagnosing heart abnormalities**

Problems with the human heart or blood vessels, so-called cardiovascular diseases, are the most common cause of death. Those problems often influence the electrical activity of the heart, which can be measured using electrodes attached to the body and reported in an electrocardiogram (ECG). There are different formats for ECGs, but a typical format used at hospitals involves 10 electrodes put on a patient's body and recorded during 10 seconds at about 500 Hz. In Figure 1.1 we show examples of (parts of) the measured signals from a few of the electrodes for three different hearts. The measurements stem from a healthy heart (top), a heart suffering from atrial fibrillation (middle), and a heart suffering from right bundle branch block (bottom). Atrial fibrillation makes the heart beat irregularly making it hard for the heart to pump blood in a normal way. Right bundle branch block corresponds to a delay or blockage in the electrical pathways of the heart.

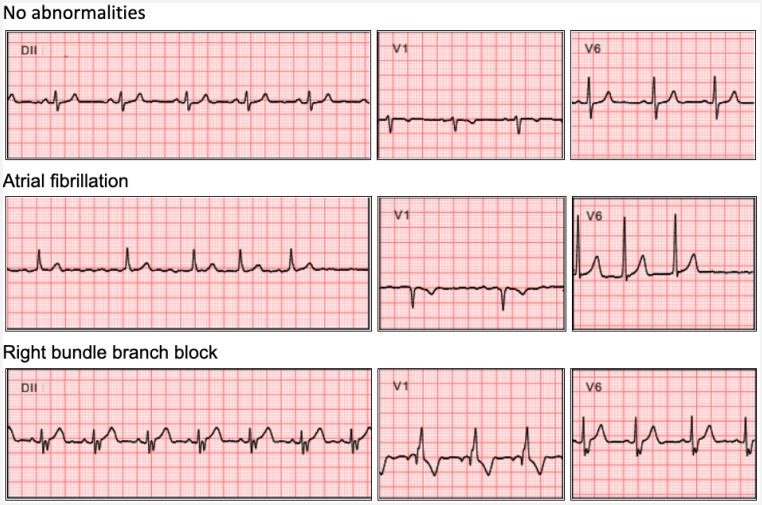



**Fig. 1.1**

By analysing the ECG, a healthcare professional can gain valuable information about the condition of the heart, diagnose the patient and plan the treatment. The interpretation of an ECG is, however, non-trivial and requires a qualified cardiologist. Certain cases can also be very time-critical, and a delayed diagnosis and treatment can increase the risk for severe consequences for the patient.

To this end, there is a need for a computer program that reads the ECG signals, analyses the data, and returns a very accurate *prediction* of the status of the patient's heart. Such models, capable of automating parts of the work of a skilled cardiologist, are finding applications globally. The needs are most acute in low- and middle-income countries where there is a shortage of skilled cardiologists, but there is also a use case for such models in wearable devices such as smart watches that are able to record ECGs.

The key challenge in building such a computer program is that it is far from obvious which computations are needed to turn the raw ECG signal into a prediction about


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September* 1, 2026, `http://smlbook.org`

the heart condition. There are such conventional computer programs available, that build on a certain set of programmatic rules defined by experienced cardiologists, but it has turned out extremely challenging for those programs to reach the top human performance.

However, when tackling this problem through the machine learning approach, the situation turns out differently. Instead of asking a cardiologist to specify a set of rules for how to classify an ECG signal as normal or abnormal, we simply ask the cardiologist (or a group of cardiologists) to *label* a large number of recorded ECG signals with labels corresponding to the underlying heart condition. This is a much easier (albeit possibly tedious) way for the cardiologists to communicate their experience and encode it in a way that is interpretable by a computer.

The task of the learning algorithm is then to automatically adapt the computer program in such a way that its predictions agree with the cardiologists' labels on the labelled training data. The hope is that, if it succeeds on the training data (where we already know the answer), then it should be possible to use the predictions made by the program on previously unseen data (where we *do not* know the answer) as well.

This is the approach taken by Ribeiro et al. (2020), who developed a machine learning model for ECG prediction. In their study, the training data consists of more than 2 300 000 ECG records that each come with a label – no abnormalities, atrial fibrillation, right bundle branch block, etc. – according to the status of the heart. Based on this data, a machine learning model was trained to automatically classify a new ECG recording.

It is far from obvious how the performance of such a model should be evaluated, since the hardest cases are genuinely also tricky for trained cardiologists. The approach taken by **w**as to ask three experienced cardiologists to examine and classify 827 ECG recordings from distinct patients, and the majority vote was taken as the source of truth. This dataset was then evaluated by the model, some other cardiologists of different levels of experience, as well as medical students. The average performance was then compared across all groups and the result was that the model achieved the same or better result than all the different human groups.

Moreover it has turned out that the machine learning approach can surpass the human performance not only in diagnostic accuracy, but also in what diagnoses it can read from the ECG. For example Gustafsson et al. (2026) developed a model that can diagnose heart attacks based on which artery in the heart that is occluded and how acute that occlusion is. Information of that kind can be used for making a better decision on patient treatment, than the conventional scheme for heart attack diagnosis.

The models used in the mentioned studies are deep neural networks, more specifically so-called convolutional neural networks (Chapter 7). Such models are typically used for images, but the researchers adapted them to work for the ECG signals for these studies.

The ECG example introduces several concepts central to this book. First, the *training data* consists of both the signal (the *input*) and a manually assigned label (the *output*). Training a model from labelled data points is referred to as *supervised learning*. The goal is for the model to *generalize beyond the training data*, providing accurate predictions on new, unseen patients. How to train models that are capable of generalising, and how to evaluate to what extent they do so, is a central theoretical question studied throughout this book (see in

particular Chapter 4).

Let us use the ECG example to illustrate the general idea how to train and use a machine learing model in Figure 1.2.

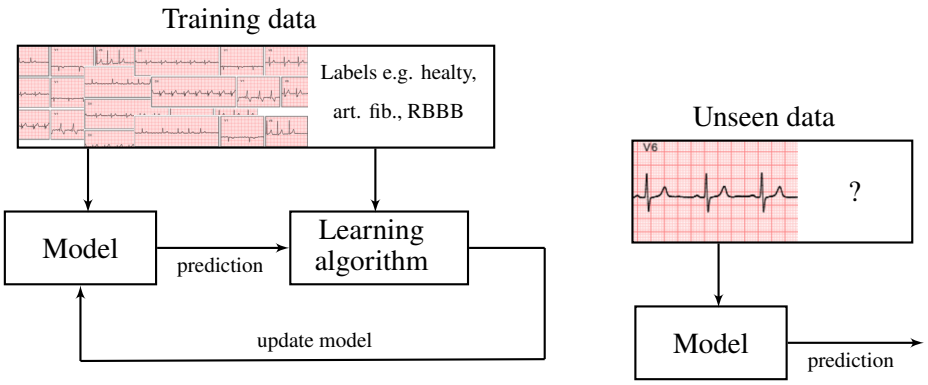



**Figure 1.2:** Illustrating the supervised machine learning process with training to the left and then the use of the trained model to the right. **Left:** Values for the unknown parameters of the model are set by the learning algorithm such that the model best describes the available training data. **Right:** The learned model is used on new, previously unseen data, where we hope to obtain a correct classification. It is thus essential that the model is able to generalise to new data that is not present in the training data.

Another key concept in the ECG example is the notion of a *classification* problem. Classification is a supervised machine learning task that amounts to predicting a certain class, or label, for each data point. In classification problems, there are only a finite number of possible output values. If there are only two possible outputs, such as 'normal' or 'abnormal', we refer to it as a *binary classification problem*, since there are only two possible classes. If there are more than two classes, such as one for each possible diagnosis, we instead face a *multi-class classification problem*.

Classification is, however, not the only flavour of supervised machine learning that we will encounter. We will also study another type of problem, referred to as a *regression problem*. Regression differs from classification in that the output – that is, the quantity we want the model to predict – is a numerical value, such as blood pressure, body temperature or a so-called formation energy as in the following example.

**Example 1.2 Formation energy of crystals**

The discovery of new materials with unique properties drives much of our technological progress. Touch screens and electric-vehicle batteries, for instance, grew out of advances in materials science. Such materials were traditionally found through experiments, but experiments are slow and expensive, which limits how many new materials can be discovered. Computational methods have therefore taken on a larger role, where the idea is to screen a large number of hypothetical materials, predict the properties of interest, and then try only the most promising candidates in real


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September* 1, 2026, `http://smlbook.org`

experiments.

Crystalline solids – crystals, for short – are a central class of inorganic material. Their atoms sit in a highly ordered microscopic structure. To understand a crystal's properties, it is therefore not enough to know the proportion of each element; we also need to know how the atoms are arranged. One basic property is the *formation energy*: the energy nature must spend to assemble the crystal from its individual elements. Nature favours minimum-energy configurations, so a crystal whose formation energy is much higher than that of other crystals built from the same elements is unlikely to form stably in practice. Since we can imagine far more materials than could ever exist, predicting the formation energy lets us discard the unstable majority and spend effort only on candidates that might actually be made.

One classical method for computing the formation energy is density functional theory (DFT), which dates back to the 1960s. Built on quantum-mechanical modelling, DFT enabled the first breakthrough in computational materials science and made high-throughput screening possible. But it is computationally expensive: even with modern supercomputers, only a small fraction of all potentially interesting materials have been analysed.

Machine learning offers a way around this limitation, and could spark a second computational revolution in the field. A model trained to predict the formation energy directly – in a fraction of the time DFT requires – lets us investigate far more candidate materials.

As a concrete example, Faber et al. (2016) used kernel ridge regression (see Chapter 9) to predict the formation energy of around 2 million so-called elpasolite crystals. The model takes a candidate crystal as input – essentially a description of the positions and elemental types of its atoms – and returns a predicted formation energy, a real continuous number. To train the model, 10 000 crystals were selected at random and their formation energies computed with DFT. The model was then trained to match the DFT energies as closely as possible on this training set, and used to predict the energy of the remaining ~99.5% of the elpasolites. Among these, 90 structures had a favourable energy, making them potentially stable in nature.

The same problem has since been tackled at much larger scale with deep neural networks (see Chapter 6). Merchant et al. (2023) trained a network that takes a candidate crystal as input, like the kernel ridge regression method above, and predicts its formation energy. The neural network model was trained on a much larger scale of data and produced in the end more than 2 million crystals predicted to be stable. Around 380 000 of these expanded the known catalogue of stable inorganic crystals by an order of magnitude, and more than 700 have since been synthesised in the laboratory. The kernel ridge regression method and the neural network solve the same problem; they only differ in the machine learning model used and in the scale at which it is applied.

Comparing the two examples discussed above, we can make a few interesting observations. The most important similarity is the ability of a model to *generalize* from the training data to new data where we do not have any ground truth yet. On the side of differences, the ECG model is asked to predict a certain class (say, normal or abnormal) whereas the materials discovery model is asked to predict a numerical value (the formation energy of a crystal). These are the two main types of

prediction problems that we will study in this book, referred to as classification and regression, respectively. While conceptually similar, we often use slight variations of the underpinning mathematical models, depending on the problem type. It is therefore important to be aware of the distinction.

Both types are supervised learning problems, though. That is, we train a predictive model to mimic the predictions made by a 'supervisor'. However, it is interesting to note that the supervision is not necessarily done by a human domain expert. Indeed, for the formation energy model, the training data was obtained by running automated (but costly) density functional theory computations. In other situations, we might obtain the output values naturally when collecting the training data. For instance, assume that you want to build a model for predicting the outcome of a soccer match based on data about the players in the two teams. This is a classification problem (the output is 'win', 'lose', or 'tie'), but the training data does not have to be manually labelled, since we get the labels directly from historical matches. Similarly, if you want to build a regression model for predicting the price of an apartment based on its size, location, condition, etc., then the output (the price) is obtained directly from historical sales.

Finally, it is worth noting that, although the examples discussed above correspond to very different application domains, the problems are quite similar from a machine learning perspective. Indeed, the general procedure outlined in Figure 1.2 is also applicable, with minor modifications, to the materials discovery problem. This generality and versatility of the machine learning methodology is one of its main strengths and beauties.

In this book, we will make use of statistics and probability theory to describe the models used for making predictions. Using probabilistic models allows us to systematically represent and cope with the *uncertainty* in the predictions. In the ECG example above, it could (perhaps) be argued that there is a 'correct answer' of the heart diagnosis. However, even in situations where there is a correct answer, machine learning models rely on various assumptions, and they are trained from data using computational learning algorithms. With probabilistic models we are able to represent the uncertainty in the model's predictions, whether it originates from the data, the modelling assumptions, or the computation. Furthermore, in many applications of machine learning, the output is uncertain in itself, and there is no such thing as a definite answer. To highlight the need for probabilistic predictions, let us consider an example from sports analytics.

**Example 1.3 Expected Goals (xG) in soccer**

Modern professional soccer is heavily instrumented. High-frequency event data and optical tracking allow teams to quantify every action on the pitch. One of the most successful applications of this data is the *Expected Goals* (xG) metric, which is now frequently presented during live TV match coverage.

Consider the problem of predicting whether or not a specific shot will result in a


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September* 1, 2026, `http://smlbook.org`

goal. To build a model, we define the input based on the geometry of the shot: the distance from the goal and the angle between the lines drawn from the player's position to the two goal posts (see Figure 1.3). This is a binary classification problem where the outcome is either a goal (1) or no goal (0).

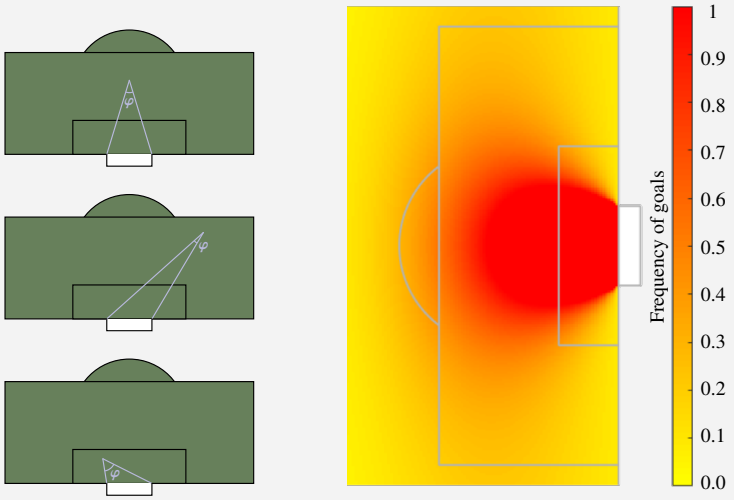



**Fig. 1.3**

Because soccer is a high-variance sport, knowing the player's position is insufficient to predict the outcome with certainty. Instead of a hard classification, it is possible to design a machine learning model to output a *probability*: the xG value (Sumpter 2016). We can achieve this using a *logistic regression* model (introduced in Chapter 3), which maps the input features to a value between 0 and 1. The heat map in the right panel of Figure 1.3 illustrates the resulting predicted probabilities across the field.

While logistic regression provides a solid baseline, state-of-the-art xG models typically employ more sophisticated ensembles. Specifically, gradient boosting is commonly used, thanks to its ability to capture non-linear interactions between features like shot height, defender proximity, and game state. We will explore gradient boosting in Chapter 8.

Again, the most important ability for a model predicting the expected goals (as well as a heart diagnosis and a formation energy) is the ability to *generalize well* and find the underlying patterns in the data – the very point of computing expected goals is to compare how a team actually performed in a specific match against what its chances were worth on average. For that comparison to be meaningful, the model needs to generalize the patterns learned from past data. Beyond the importance of generalization, the three examples so far also illustrate some of the most fundamental aspects of machine learning: classification, regression and the need for probabilistic models.

We will now turn our eye to a few highly specialised modern machine learning applications. A common theme in many modern applications is high-dimensional data, such as images or even videos, that has intricate dependencies across space and time. In the next example, we move from predicting single values for individual soccer

players to predicting a class for every pixel in an image.

**Example 1.4 Segmentation: Pixel-wise class prediction**

When it comes to machine vision, an important capability is to be able to associate each pixel in an image with a corresponding class; see Figure 1.4 for an illustration in an autonomous driving application. This is referred to as *semantic segmentation*. In autonomous driving, it is used for separating cars, road, pedestrians, etc. The output is then used as input to other algorithms, for instance path planning or collision avoidance. Segmentation is, for instance, also used in medical imaging to tell apart different organs and tumors, and has also become important in modern defense and warfare applications.

**Fig. 1.4**

To train a semantic segmentation model, the training data consist of a large number of images (inputs). For each such image, there is a corresponding output image of the same size, where each pixel has been labelled by hand to belong to a certain class. The supervised machine learning problem then amounts to using this data to find a mapping that is capable of taking a new unseen image and produce a corresponding output in the form of a predicted class for each pixel. Essentially, this is a type of classification problem, but all pixels need to be classified simultaneously while respecting the spatial dependencies across the image to result in a coherent segmentation. The bottom part of Figure 1.4 shows the prediction generated by such an algorithm, where the aim is to classify each pixel as either car (blue), traffic sign (yellow), pavement (purple), or tree (green).

The best performing solutions for this task today rely on cleverly crafted deep neural networks (see Chapter 6 and 7). Traditionally, these models are "specialists" trained on a fixed set of labels for a specific domain (Long et al. 2015). However, a significant recent shift is the emergence of vision *foundation models*. Much like the Large Language Models (LLMs) discussed in Chapter 7, these are massive networks trained on vast, diverse datasets to develop a generalized understanding of their input.


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September* 1, 2026, `http://smlbook.org`

A prominent example is the Segment Anything Model (SAM, Kirillov et al. 2023). Unlike traditional classifiers, it can identify the contours of objects in almost any scene (including those it has never encountered before) by using a simple spatial input such as a single point or a bounding box to initiate the process. This represents a transition from building niche models for every new problem towards using universal models that understand the fundamental structure of images.

In the next example, we raise the bar even higher, since here the model needs to be able to explain dependencies not only over space, but also over time, in a so-called spatio-temporal problem. Since the output here is an air quality index, a continuous number, this is a regression problem.

### Example 1.5 Estimating air pollution levels across London

Ambient air pollution is a critical public health challenge. For engineers, the task is to transform sparse measurements into high-resolution, real-time "pollution maps." This requires shifting from traditional, expensive government stations to *hyper-local* sensing.

In recent years, the data landscape has been revolutionized by crowd-sourced networks. These consist of thousands of low-cost sensors maintained by citizens. While this provides unprecedented spatial density, it introduces a "multi-fidelity" problem: how do we fuse highly accurate (but sparse) official data with high-density (but noisy and uncalibrated) crowd-sourced data?

The resulting machine learning task is a *regression* problem across time and space. While large-scale deep learning models like GraphCast (Lam et al. 2023) have set new benchmarks for global forecasting, they often require massive compute and can struggle to provide the granular *uncertainty quantification* needed for local policy decisions.

In this urban context, Hamelijnck et al. (2019) instead use a Gaussian process (GP) (Chapter 10). The GP is a non-parametric, probabilistic model that is particularly powerful for such sensor fusion.

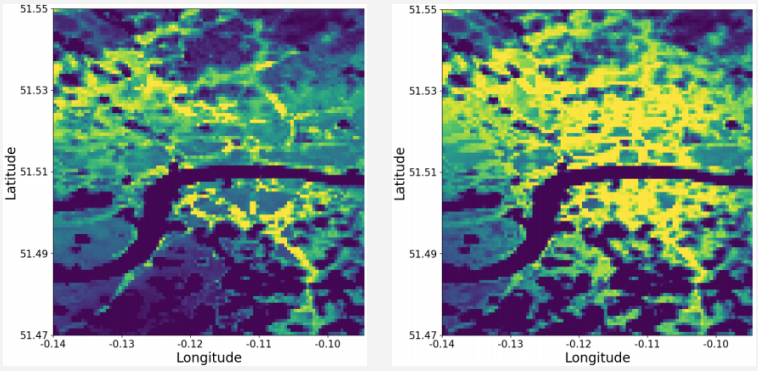



**Fig. 1.5**

Figure 1.5 illustrates the spatio-temporal estimation of $NO_2$ levels in London at two different points in time. The model does not just predict a number; it provides a

probability distribution. For an engineer, the most valuable output is often the *variance* (uncertainty). In areas with few sensors, the model signals its own lack of confidence, preventing over-reliance on a single noisy data point.

The examples so far have been *discriminative*. That term refers to the situation where a method returns a "claim" for each input, such as a heart diagnosis or an air quality index. Several machine learning methods are more capable than that: so-called *generative models* can generate new, realistic-looking data, even for highly complex problems such as the building blocks of life, which will be our final example.

**Example 1.6 Predicting the molecular machinery of life**

Proteins are the fundamental "nanomachines" of life, responsible for everything from metabolizing food to fighting viruses. A protein's function is determined almost entirely by its three-dimensional shape. For decades, determining this shape was a monumental engineering challenge; experimental methods like X-ray crystallography or cryo-electron microscopy can take years of PhD-level effort and cost hundreds of thousands of dollars for a single molecule.

The protein folding problem involves predicting the 3D structure of a protein given only its one-dimensional sequence of amino acids. In 2020, this was largely solved for individual proteins by AlphaFold 2 (Jumper et al. 2021), whose developers were awarded the 2024 Nobel Prize in Chemistry. However, life does not happen in isolation. To design a new drug or understand a disease, we must model how proteins interact with DNA, RNA, and small molecules (ligands).

This is the role of AlphaFold 3 (Abramson et al. 2024). Rather than being a specialist model for a single task, it acts as a *biological foundation model*. It uses a generative approach (Chapter 11 and 12) to model the joint distribution of the 3D coordinates of all atoms in a complex system simultaneously. Concretely, this means that the model does not simply read off a single fixed answer; instead it *generates* a structure by sampling atomic coordinates from this learned distribution. By training on the Protein Data Bank, a decades-long global effort of collected experimental data, the model has learned the underlying physics and chemistry of molecular interactions.

Instead of running computationally expensive molecular dynamics simulations that might take weeks on a supercomputer to show how a drug binds to a protein, AlphaFold 3 can generate a plausible bound structure in seconds.

The architecture powering this relies on a combination of transformers (Chapter 7) and diffusion models (Chapter 12). For engineers, this opens a new frontier in generative design, where we can potentially program new enzymes to break down plastics or design highly specific medicines with minimal side effects.

We have now seen six different examples of applications where machine learning can make a significant difference. All examples have fundamentally been *supervised* meaning that they are trained from data containing pairs of inputs and outputs, such as ECG recordings vs heart diagnoses, crystal recipes vs formation energy, location on the soccer pitch vs probability of scoring a goal, an image vs a segmentation mask, detailed but uneven measurements vs air quality index or a sequence of amino

acids vs a 3D shape. The field of machine learning is not limited to those problems, but there is also genuinely *unsupervised* learning, for problems where there is only input data without any corresponding output, which we will encounter in Chapter 11. However, many of the most powerful and successful machine learning methods are supervised. In fact, a great deal of the recent machine learning success consists of clever engineering in re-formulating problems such that supervised machine learning methods can be applied to them, even if the original problem might not have seemed like a supervised problem. One such idea is self-supervised learning (Section 2.4), which is a fundamental idea in, among others, large language models (Chapter 7).

## 1.2 About This Book

The aim of this book is to convey the spirit of machine learning, without requiring any previous experience in the field. We focus on the underlying mathematics as well as the practical aspects. This book is a textbook; it is not a reference work or a programming manual. It therefore contains only a careful (yet comprehensive) selection of machine learning methods and no programming code. There are by now many well-written and well-documented code packages available, and it is our firm belief that with a good understanding of the mathematics and the inner workings of the methods, the reader will be able to make the connection between this book and their favorite code package in their favorite programming language.

We take a statistical perspective in this book, meaning that we discuss and motivate methods in terms of their statistical properties. It therefore requires some previous knowledge in statistics and probability theory, as well as calculus and linear algebra. We hope that reading the book from start to end will give the reader a good starting point for working as a machine learning engineer and/or pursuing further studies within the subject.

The book is written such that it can be read back to back. There are, however, multiple possible paths through the book that are more selective depending on the interest of the reader. Figure 1.6 illustrates the major dependencies between the chapters. In particular, the most fundamental topics are discussed in Chapters 2, 3, and 4, and we do recommend the reader to read those chapters before proceeding to the later chapters that contain technically more advanced topics (Chapters 5–10). **??** goes beyond the supervised setting of machine learning, and Chapter 13 focuses on some of the more practical aspects of designing a successful machine learning solution and has a less technical nature than the preceding chapters. Finally, Chapter 14 (written by David Sumpter) discusses certain ethical aspects of modern machine learning.

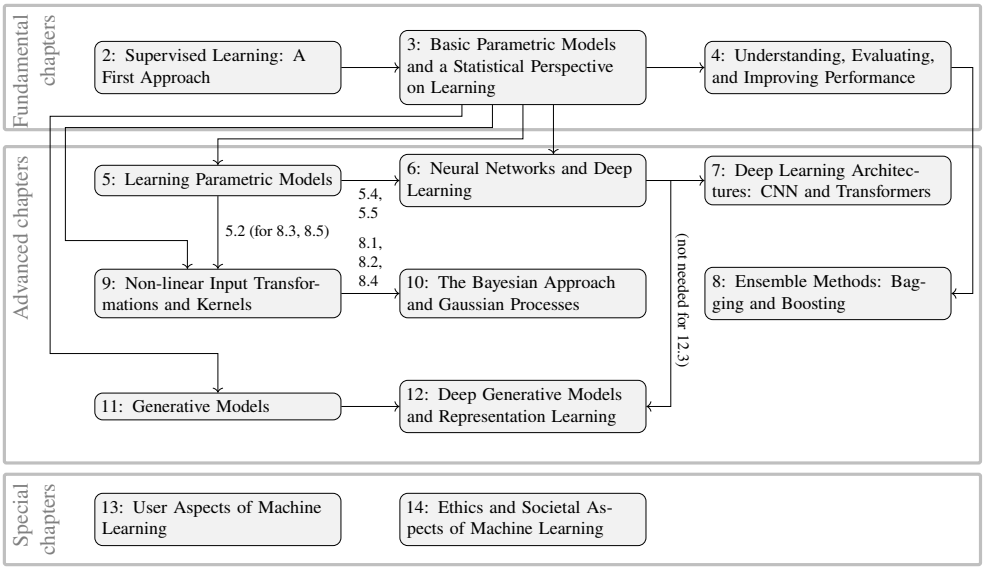



**Figure 1.6:** The structure of this book, illustrated by blocks (chapters) and arrows (recommended order in which to read the chapters). We do recommend everyone to read (or at least skim) the fundamental material in Chapters 2, 3, and 4 first. The path through the technically more advanced Chapters 5–12, can be chosen to match the particular interest of the reader. For Chapters 13 and 14, we recommend reading at least the fundamental chapters first.


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September* 1, 2026, `http://smlbook.org`

## 1.3 Further Reading

There are by now quite a few extensive textbooks available on the topic of machine learning, which introduce the area in different ways compared to how we do so in this book. We will only mention a few here. The book of Hastie et al. (2009) introduces the area of statistical machine learning in a mathematically solid and accessible manner. A few years later, the authors released a different version of their book (James et al. 2013), which is mathematically significantly lighter, conveying the main ideas in an even more accessible manner. These books do not venture long either into the world of Bayesian methods or the world of neural networks. However, there are several complementary books that do exactly that – see e.g. C. M. Bishop and H. Bishop (2024), Murphy (2022), and Murphy (2023). MacKay (2003) provides a rather early account drawing interesting and useful connections to information theory. It is still very much worth looking into. The book by Shalev-Shwartz and Ben-David (2014) provides an introduction with a clear focus on the underpinning theoretical constructions, connecting very deep questions – such as 'what is learning?' and 'how can a machine learn' – with mathematics. It is a perfect book for those of our readers who would like to deepen their understanding of the theoretical background of that area. We also mention the work of Efron and Hastie (2016), where the authors take a constructive historical approach to the development of the area, covering the revolution in data analysis that emerged with computers. Good introductions to the mathematics of machine learning are provided by Strang (2019) and Deisenroth et al. (2019).

# 2 Supervised Learning: A First Approach

In this chapter, we will introduce the supervised machine learning problem as well as two basic machine learning methods for solving it. The methods we will introduce are called $k$-nearest neighbours and decision trees. These two methods are relatively simple, and we will derive them on intuitive grounds. Still, these methods are useful in their own right and are therefore a good place to start. Understanding their inner workings, advantages, and shortcomings also lays a good foundation for the more advanced methods that are to come in later chapters.

## 2.1 Supervised Machine Learning

In supervised machine learning, we have some *training data* that contains examples of how some *input*[1] variable $\mathbf{x}$ relates to an *output*[2] variable $y$. By using some mathematical model or method, which we adapt to the training data, our goal is to *predict* the output $y$ for a new, previously unseen, set of *test data* for which only $\mathbf{x}$ is known. We usually say that we *learn* (or *train*) a model from the training data, and that process involves some computations implemented on a computer.

### Learning from Labelled Data

In most interesting supervised machine learning applications, the relationship between input $\mathbf{x}$ and output $y$ is difficult to describe explicitly. It may be too cumbersome or complicated to fully unravel from application domain knowledge, or even unknown. The problem can therefore usually not be solved by writing a traditional computer program that takes $\mathbf{x}$ as input and returns $y$ as output from a set of rules. The supervised machine learning approach is instead to learn the relationship between $\mathbf{x}$ and $y$ from data, which contains examples of observed pairs of input and output values. In other words, supervised machine learning amounts to learning from examples.

The data used for learning is called *training data*, and it has to consist of several input–output data points (samples) $(\mathbf{x}_i, y_i)$, in total $n$ of them. We will compactly

[1] The input is commonly also called feature, attribute, predictor, regressor, covariate, explanatory variable, controlled variable, and independent variable.

[2] The output is commonly also called target, response, regressand, label, explained variable, predicted variable, or dependent variable.

write the training data as $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^n$. Each data point in the training data provides a snapshot of how $y$ depends on $\mathbf{x}$, and the goal in supervised machine learning is to squeeze as much information as possible out of $\mathcal{T}$. In this book, we will only consider problems where the individual data points are assumed to be (probabilistically) *independent*. This excludes, for example, applications in time series analysis, where it is of interest to model the correlation between $\mathbf{x}_i$ and $\mathbf{x}_{i+1}$.

The fact that the training data contains not only input values $\mathbf{x}_i$ but also output values $y_i$ is the reason for the term 'supervised' machine learning. We may say that each input $\mathbf{x}_i$ is accompanied by a label $y_i$, or simply that we have *labelled data*. For some applications, it is only a matter of jointly recording $\mathbf{x}$ and $y$. In other applications, the output $y$ has to be created by labelling of the training data inputs $\mathbf{x}$ by a domain expert. For instance, to construct a training dataset for the cardiovascular disease application introduced in Chapter 1, a cardiologist needs to look at all training data inputs (ECG signals) $\mathbf{x}_i$ and label them by assigning to the variable $y_i$ to correspond to the heart condition that is seen in the signal. The entire learning process is thus 'supervised' by the domain expert.

We use a vector boldface notation $\mathbf{x}$ to denote the input, since we assume it to be a $p$-dimensional vector, $\mathbf{x} = [x_1 \;\; x_2 \;\; \cdots \;\; x_p]^\mathsf{T}$, where $^\mathsf{T}$ denotes the transpose. Each element of the input vector $\mathbf{x}$ represents some information that is considered to be relevant for the application at hand, for example the outdoor temperature or the unemployment rate. In many applications, the number of inputs $p$ is large, or put differently, the input $\mathbf{x}$ is a high-dimensional vector. For instance, in a computer vision application where the input is a greyscale image, $\mathbf{x}$ can be all pixel values in the image, so $p = h \times w$ where $h$ and $w$ denote the height and width of the input image.[3] The output $y$, on the other hand, is often of low dimension, and throughout most of this book, we will assume that it is a scalar value. The *type* of the output value, numerical or categorical, turns out to be important and is used to distinguish between two subtypes of the supervised machine learning problems: *regression* and *classification*. We will discuss this next.

### Numerical and Categorical Variables

The variables contained in our data (input as well as output) can be of two different types: *numerical* or *categorical*. A numerical variable has a natural ordering. We can say that one instance of a numerical variable is larger or smaller than another instance of the same variable. A numerical variable could, for instance, be represented by a continuous real number, but it could also be discrete, such as an integer. Categorical variables, on the other hand, are always discrete, and importantly, they lack a natural

[3] For image-based problems it is often more convenient to represent the input as a matrix of size $h \times w$ than as a vector of length $p = hw$, but the dimension is nevertheless the same. We will get back to this in Chapter 6 when discussing the convolutional neural network, a model structure tailored to image-type inputs.


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

**Table 2.1:** Examples of numerical and categorical variables.

| **Variable type** | **Example** | **Handled as** |
|---|---|---|
| Number (continuous) | 32.23 km/h, 12.50 km/h, 42.85 km/h | Numerical |
| Number (discrete) with natural ordering | 0 children, 1 child, 2 children | Numerical |
| Number (discrete) without natural ordering | 1 = Sweden, 2 = Denmark, 3 = Norway | Categorical |
| Text string | Hello, Goodbye, Welcome | Categorical |

ordering. In this book we assume that any categorical variable can take only a finite number of different values. A few examples are given in Table 2.1 above.

The distinction between numerical and categorical is sometimes somewhat arbitrary. We could, for instance, argue that having no children is qualitatively different from having children, and use the categorical variable 'children: yes/no' instead of the numerical '0, 1 or 2 children'. It is therefore a decision for the machine learning engineer whether a certain variable is to be considered as numerical or categorical.

The notion of categorical vs. numerical applies to both the output variable $y$ and to the $p$ elements $x_j$ of the input vector $\mathbf{x} = [x_1 \ x_2 \ \cdots \ x_p]^\mathsf{T}$. All $p$ input variables do not have to be of the same type. It is perfectly fine (and common in practice) to have a mix of categorical and numerical inputs.

### Classification and Regression

We distinguish between different supervised machine learning problems by the type of the output $y$.

> *Regression* means that the output is numerical, and *classification* means that the output is categorical.

The reason for this distinction is that the regression and classification problems have somewhat different properties, and different methods are used for solving them.

Note that the $p$ input variables $\mathbf{x} = [x_1 \ x_2 \ \cdots \ x_p]^\mathsf{T}$ can be either numerical or categorical for both regression and classification problems. It is only the type of the output that determines whether a problem is a regression or a classification problem. A method for solving a classification problems is called a *classifier*.

For classification, the output is categorical and can therefore only take values in a finite set. We use $M$ to denote the number of elements in the set of possible output values. It could, for instance, be {`false`, `true`} ($M = 2$) or {`Sweden`, `Norway`, `Finland`, `Denmark`} ($M = 4$). We will refer to these elements as *classes* or *labels*. The number of classes $M$ is assumed to be known in the classification problem. To prepare for a concise mathematical notation, we use integers $1, 2, \ldots, M$ to denote the output classes if $M > 2$. The ordering of the integers is arbitrary and does *not* imply any ordering of the classes. When there are only $M = 2$ classes, we

have the important special case of *binary* classification. In binary classification we use the labels $-1$ and 1 (instead of 1 and 2). Occasionally we will also use the equivalent terms *negative* and *positive* class. The only reason for using a different convention for binary classification is that it gives a more compact mathematical notation for some of the methods, and it carries no deeper meaning. Let us now have a look at a classification and a regression problem, both of which will be used throughout this book.

### Example 2.1 Classifying songs

Say that we want to build a 'song categoriser' app, where the user records a song, and the app answers by reporting whether the song has the artistic style of either the Beatles, Kiss, or Bob Dylan. At the heart of this fictitious app, there has to be a mechanism that takes an audio recording as an input and returns an artist's name.

If we first collect some recordings with songs from the three groups/artists (where we know which artist is behind each song: a labelled dataset), we could use supervised machine learning to *learn* the characteristics of their different styles and therefrom *predict* the artist of the new user-provided song. In supervised machine learning terminology, the artist name (`the Beatles`, `Kiss`, or `Bob Dylan`) is the output $y$. In this problem, $y$ is categorical, and we are hence facing a classification problem.

One of the important design choices for a machine learning engineer is a detailed specification of what the input $\mathbf{x}$ really is. It would in principle be possible to consider the raw audio information as input, but that would give a very high-dimensional $\mathbf{x}$ which (unless an audio-specific machine learning method is used) would most likely require an unrealistically large amount of training data in order to be successful (we will discuss this aspect in detail in Chapter 4). A better option could therefore be to define some summary statistics of audio recordings and use those so-called *features* as input $\mathbf{x}$ instead. As input features, we could, for example, use the length of the audio recording and the 'perceived energy' of the song. The length of a recording is easy to measure. Since it can differ quite a lot between different songs, we take the logarithm of the actual length (in seconds) to get values in the same range for all songs. Such feature transformations are commonly used in practice to make the input data more homogeneous.

The energy of a song[a] is a bit more tricky, and the exact definition may even be ambiguous. However, we leave that to the audio experts and re-use a piece of software that they have written for this purpose[b] without bothering too much about its inner workings. As long as this piece of software returns a number for any recording that is fed to it, and always returns the same number for the same recording, we can use it as an input to a machine learning method.

In Figure 2.1 we have plotted a dataset with 230 songs from the three artists. Each song is represented by a dot, where the horizontal axis is the logarithm of its length (measured in seconds) and the vertical axis the energy (on a scale 0–1). When we later return to this example and apply different supervised machine learning methods to it, this data will be the training data.

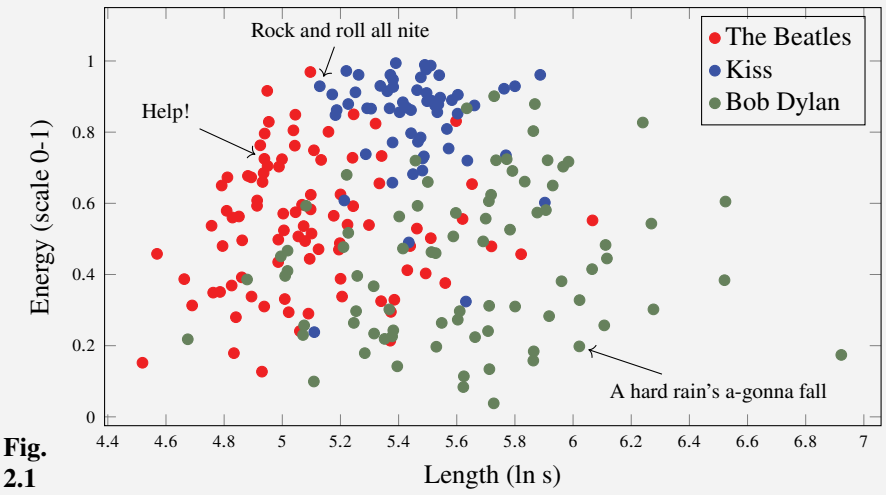



**Fig. 2.1**

[a] We use this term to refer to the perceived musical energy, not the signal energy in a strict sense.

[b] Specifically, we use `http://api.spotify.com/` here.

## Example 2.2 Car stopping distances

Ezekiel and Fox (1959) present a dataset with 62 observations of the distance needed for various cars at different initial speeds to brake to a full stop.[a] The dataset has the following two variables:

- `Speed`: The speed of the car when the brake signal is given.
- `Distance`: The distance traveled after the signal is given until the car has reached a full stop.

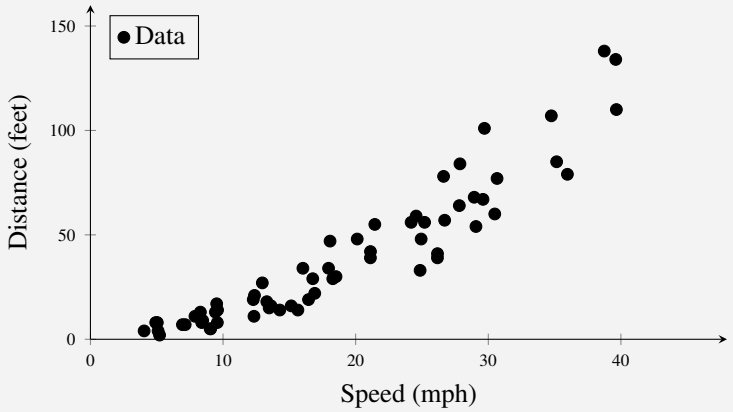



**Fig. 2.2**

To make a supervised machine learning problem out of this, we interpret `Speed` as the input variable $x$ and `Distance` as the output variable $y$, as shown in Figure 2.2. Note that we use a non-bold symbol for the input here since it is a scalar value and not

a vector of inputs in this example. Since $y$ is numerical, this is a regression problem. We then ask ourselves what the stopping distance would be if the initial speed were, for example, 33 mph or 45 mph, respectively (two speeds at which no data has been recorded). Another way to frame this question is to ask for the *prediction* $\widehat{y}(x_\star)$ for $x_\star = 33$ and $x_\star = 45$.

[a] The data is somewhat dated, so the conclusions are perhaps not applicable to modern cars.

## Generalising Beyond Training Data

There are two primary reasons why it can be of interest to mathematically model the input–output relationships from training data.

(i) To *reason about and explore* how input and output variables are connected. An often-encountered task in sciences such as medicine and sociology is to determine whether a correlation between a pair of variables exists or not ('does eating seafood increase life expectancy?'). Such questions can be addressed by learning a mathematical model and carefully reasoning about the likelihood that the learned relationships between input $\mathbf{x}$ and output $y$ are due only to random effects in the data or if there appears to be some substance to the proposed relationships.

(ii) To *predict* the output value $y_\star$ for some new, previously unseen input $\mathbf{x}_\star$. By using some mathematical method which generalises the input–output examples seen in the training data, we can make a prediction $\widehat{y}(\mathbf{x}_\star)$ for a previously unseen test input $\mathbf{x}_\star$. The hat $\widehat{\ }$ indicates that the prediction is an estimate of the output.

These two objectives are sometimes used to roughly distinguish between classical statistics, focusing more on objective (i), and machine learning, where objective (ii) is more central. However, this is not a clear-cut distinction since predictive modelling is a topic in classical statistics too, and explainable models are also studied in machine learning. The primary focus in this book, however, is on making predictions, objective (ii) above, which is the foundation of supervised machine learning. Our overall goal is to obtain as accurate predictions $\widehat{y}(\mathbf{x}_\star)$ as possible (measured in some appropriate way) for a wide range of possible test inputs $\mathbf{x}_\star$. We say that we are interested in methods that *generalise well* beyond the training data.

A method that generalises well for the music example above would be able to correctly tell the artist of a new song which was not in the training data (assuming that the artist of the new song is one of the three that was present in the training data, of course). The ability to generalise to new data is a key concept of machine learning. It is not difficult to construct models or methods that give very accurate predictions if they are only evaluated on the training data (we will see an example in the next section). However, if the model is not able to generalise, meaning that the predictions are poor when the model is applied to new test data points, then the model is of little

use in practice for making predictions. If this is the case, we say that the model is *overfitting* to the training data. We will illustrate the issue of overfitting for a specific machine learning model in the next section, and in Chapter 4 we will return to this concept using a more general and mathematical approach.

## 2.2 A Distance-Based Method: *k*-NN

It is now time to encounter our first actual machine learning method. We will start with the relatively simple $k$-nearest neighbours ($k$-NN) method, which can be used for both regression and classification. Remember that the setting is that we have access to training data $\{\mathbf{x}_i, y_i\}_{i=1}^n$, which consists of $n$ data points with input $\mathbf{x}_i$ and corresponding output $y_i$. From this we want to construct a prediction $\widehat{y}(\mathbf{x}_\star)$ for what we believe the output $y_\star$ would be for a new $\mathbf{x}_\star$, which we have not seen previously.

### The *k*-Nearest Neighbours Method

Most methods for supervised machine learning build on the intuition that *if the test data point* $\mathbf{x}_\star$ *is close to training data point* $\mathbf{x}_i$*, then the prediction* $\widehat{y}(\mathbf{x}_\star)$ *should be close to* $y_i$. This is a general idea, but one simple way to implement it in practice is the following: first, compute the Euclidean distance[4] between the test input and all training inputs, $\|\mathbf{x}_i - \mathbf{x}_\star\|_2$ for $i = 1, \ldots, n$; second, find the data point $\mathbf{x}_j$ with the *shortest distance* to $\mathbf{x}_\star$, and use its output as the prediction, $\widehat{y}(\mathbf{x}_\star) = y_j$.

This simple prediction method is referred to as the 1-nearest neighbour method. It is not very complicated, but for most machine learning applications of interest it is too simplistic. In practice we can rarely say *for certain* what the output value $y$ will be. Mathematically, we handle this by describing $y$ as a random variable. That is, we consider the data as *noisy*, meaning that it is affected by random errors referred to as noise. From this perspective, the shortcoming of 1-nearest neighbour is that the prediction relies on only one data point from the training data, which makes it quite 'erratic' and sensitive to noisy training data.

To improve the 1-nearest neighbour method, we can extend it to make use of the $k$ nearest neighbours instead. Formally, we define the set $\mathcal{N}_\star = \{i : \mathbf{x}_i$ is one of the $k$ training data points closest to $\mathbf{x}_\star\}$ and aggregate the information from the $k$ outputs $y_j$ for $j \in \mathcal{N}_\star$ to make the prediction. For regression problems, we take the average of all $y_j$ for $j \in \mathcal{N}_\star$, and for classification problems, we use a majority vote.[5] We illustrate the $k$-nearest neighbours ($k$-NN) method by Example 2.3 and summarise it in Method 2.1.

[4] The Euclidean distance between a test point $\mathbf{x}_\star$ and a training data point $\mathbf{x}_i$ is $\|\mathbf{x}_i - \mathbf{x}_\star\|_2 = \sqrt{(x_{i1} - x_{\star 1})^2 + (x_{i2} - x_{\star 2})^2}$. Other distance functions can also be used and will be discussed in Chapter 9. Categorical input variables can be handled, as we will discuss in Chapter 3.

[5] Ties can be handled in different ways, for instance by a coin-flip, or by reporting the actual vote count to the end user, who gets to decide what to do with it.

Methods that explicitly use the training data when making predictions are referred to as *nonparametric*, and the $k$-NN method is one example of this. This is in contrast with *parametric* methods, where the prediction is given by some function (a model) governed by a fixed number of parameters. For parametric methods, the training data is used to *learn* the parameters in an initial training phase, but once the model has been learned, the training data can be discarded since it is not used explicitly when making predictions. We will introduce parametric modelling in Chapter 3.

---

**Data:** Training data $\{\mathbf{x}_i, y_i\}_{i=1}^n$ and test input $\mathbf{x}_\star$
**Result:** Predicted test output $\widehat{y}(\mathbf{x}_\star)$

1 Compute the distances $\|\mathbf{x}_i - \mathbf{x}_\star\|_2$ for all training data points $i = 1, \ldots, n$
2 Let $\mathcal{N}_\star = \{i : \mathbf{x}_i \text{ is one of the } k \text{ data points closest to } \mathbf{x}_\star\}$
3 Compute the prediction $\widehat{y}(\mathbf{x}_\star)$ as

$$\widehat{y}(\mathbf{x}_\star) = \begin{cases} \text{Average}\{y_j : j \in \mathcal{N}_\star\} & \text{(Regression problems)} \\ \text{MajorityVote}\{y_j : j \in \mathcal{N}_\star\} & \text{(Classification problems)} \end{cases}$$

---

**Method 2.1:** $k$-nearest neighbour, $k$-NN

**Example 2.3 Predicting colours with $k$-NN**

We consider a synthetic binary classification problem ($M = 2$). We are given a training dataset with $n = 6$ observations of $p = 2$ input variables $x_1, x_2$ and one categorical output $y$, the colour `Red` or `Blue`,

| $i$ | $x_1$ | $x_2$ | $y$ |
|---|---|---|---|
| 1 | −1 | 3 | `Red` |
| 2 | 2 | 1 | `Blue` |
| 3 | −2 | 2 | `Red` |
| 4 | −1 | 2 | `Blue` |
| 5 | −1 | 0 | `Blue` |
| 6 | 1 | 1 | `Red` |

and we are interested in predicting the output for $\mathbf{x}_\star = [1 \;\; 2]^\mathsf{T}$. For this purpose we will explore two different $k$-NN classifiers, one using $k = 1$ and one using $k = 3$.

First, we compute the Euclidean distance $\|\mathbf{x}_i - \mathbf{x}_\star\|_2$ between each training data point $\mathbf{x}_i$ (red and blue dots) and the test data point $\mathbf{x}_\star$ (black dot), and then sort them in ascending order.

Since the closest training data point to $\mathbf{x}_\star$ is the data point $i = 6$ (`Red`), this means that for $k$-NN with $k = 1$, we get the prediction $\widehat{y}(\mathbf{x}_\star) = \texttt{Red}$. For $k = 3$, the three nearest neighbours are $i = 6$ (`Red`), $i = 2$ (`Blue`), and $i = 4$ (`Blue`). Taking a majority vote among these three training data points, `Blue` wins with 2 votes against 1, so our

prediction becomes $\widehat{y}(\mathbf{x}_\star) = \texttt{Blue}$. In Figure 2.3, $k = 1$ is represented by the inner circle and $k = 3$ by the outer circle.

| $i$ | $\|\mathbf{x}_i - \mathbf{x}_\star\|_2$ | $y_i$ |
|---|---|---|
| 6 | $\sqrt{1}$ | Red |
| 2 | $\sqrt{2}$ | Blue |
| 4 | $\sqrt{4}$ | Blue |
| 1 | $\sqrt{5}$ | Red |
| 5 | $\sqrt{8}$ | Blue |
| 3 | $\sqrt{9}$ | Red |

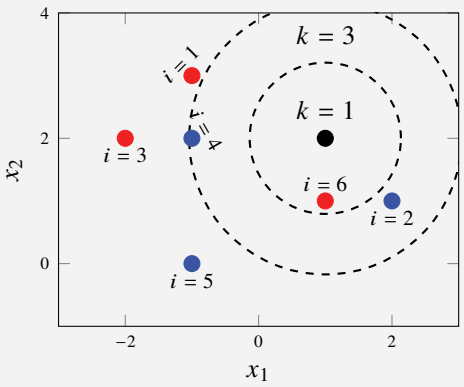



**Fig. 2.3**

## Decision Boundaries for a Classifier

In Example 2.3 we only computed a prediction for one single test data point $\mathbf{x}_\star$. That prediction might indeed be the ultimate goal of the application, but in order to visualise and better understand a classifier, we can also study its *decision boundary*, which illustrates the prediction for all possible test inputs. We introduce the decision boundary using Example 2.4. It is a general concept for classifiers, not only $k$-NN, but it is only possible to visualise easily when the dimension of $\mathbf{x}$ is $p = 2$.

**Example 2.4 Decision boundaries for the colour example**

In Example 2.3 we computed the prediction for $\mathbf{x}_\star = [1 \;\; 2]^\mathsf{T}$. If we were to shift that test point by one unit to the left at $\mathbf{x}_\star^{\text{alt}} = [0 \;\; 2]^\mathsf{T}$, the three closest training data points would still include $i = 6$ and $i = 4$, but now $i = 2$ is exchanged for $i = 1$. For $k = 3$ this would give two votes for Red and one vote for Blue, and we would therefore predict $\widehat{y} = \texttt{Red}$. In between these two test data points $\mathbf{x}_\star$ and $\mathbf{x}_\star^{\text{alt}}$, at $[0.5 \;\; 2]^\mathsf{T}$, it is equally far to $i = 1$ as to $i = 2$, and it is undecided if the 3-NN classifier should predict Red or Blue. (In practice this is most often not a problem, since the test data points rarely end up exactly at the decision boundary. If they do, this can be handled by a coin-flip.) For all classifiers, we always end up with points in the input space where the class prediction abruptly changes from one class to another. These points are said to be on the *decision boundary* of the classifier.

Continuing in a similar way, changing the location of the test input across the entire input space and recording the class prediction, we can compute the complete decision boundaries for Example 2.3. We plot the decision boundaries for $k = 1$ and $k = 3$ in Figure 2.4.

In Figure 2.4 the decision boundaries are the points in input space where the class prediction changes, that is, the borders between red and blue. This type of figure gives a concise summary of a classifier. However, it is only possible to draw such a plot in

the simple case when the problem has a 2-dimensional input $\mathbf{x}$. As we can see, the decision boundaries of $k$-NN are not linear. In the terminology we will introduce later, $k$-NN is thereby a non-linear classifier.

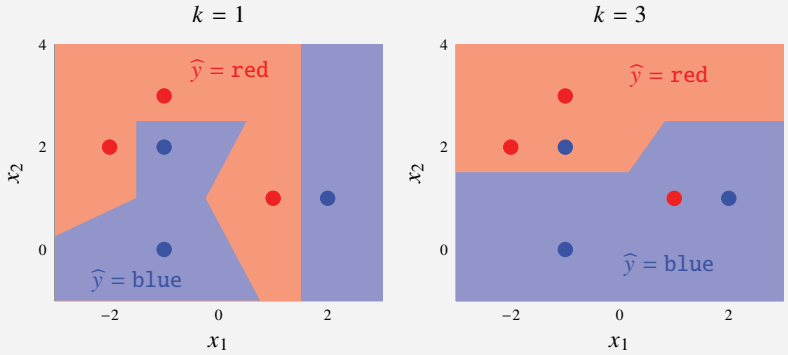



**Fig. 2.4**

## Choosing *k*

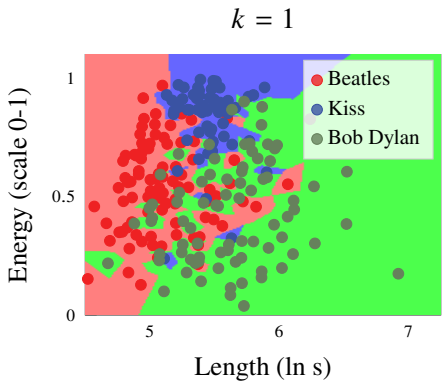
The number of neighbours $k$ that are considered when making a prediction with $k$-NN is an important choice the user has to make. Since $k$ is not learned by $k$-NN itself, but is design choice left to the user, we refer to it as a *hyperparameter*. Throughout the book, we will use the term 'hyperparameter' for similar tuning parameters for other methods.

The choice of the hyperparameter $k$ has a big impact on the predictions made by $k$-NN. To understand the impact of $k$, we study how the decision boundary changes as $k$ changes in Figure 2.5, where $k$-NN is applied to the music classification Example 2.1 and the car stopping distance Example 2.2, both with $k = 1$ and $k = 20$.

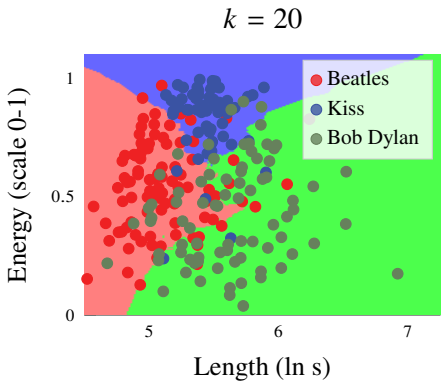
With $k = 1$, all training data points will, by construction, be correctly predicted, and the model is adapted to the exact $\mathbf{x}$ and $y$ values of the training data. In the classification problem there are, for instance, small green (Bob Dylan) regions within the red (the Beatles) area that are most likely misleading when it comes to accurately predicting the artist of a new song. In order to make good predictions, it would probably be better to instead predict red (the Beatles) for a new song in the entire middle-left region since the vast majority of training data points in that area are red. For the regression problem, $k = 1$ gives quite shaky behaviour, and also for this problem, it is intuitively clear that this does not describe an actual effect, but rather that the prediction is adapting to the noise in the data.

The drawbacks of using $k = 1$ are not specific to these two examples. In most real world problems there is a certain amount of randomness in the data, or at least insufficient information, which can be thought of as a random effect. In the music example, the $n = 230$ songs were selected from all songs ever recorded from these artists, and since we do not know how this selection was made, we may consider it random. Furthermore, and more importantly, if we want our classifier to generalise to completely new data, like new releases from the artists in our example (overlooking the obvious complication for now), then it is not reasonable to assume that the length

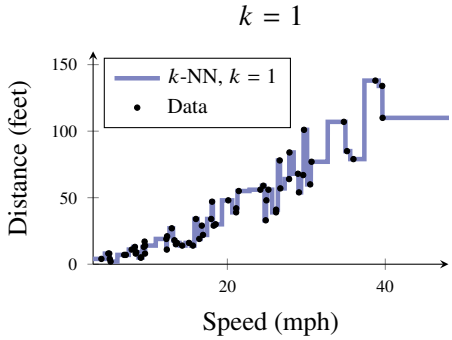

*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September* 1, 2026, `http://smlbook.org`



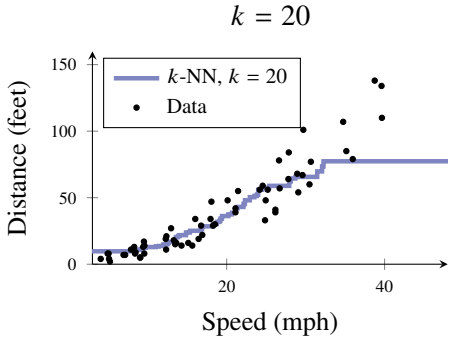

**(a)** Decision boundaries for the music classification problem using $k = 1$. This is a typical example of overfitting, meaning that the model has adapted too much to the training data so that it does not generalise well to new previously unseen data.




**(b)** The music classification problem again, now using $k = 20$. A higher value of $k$ gives a smoother behaviour which, hopefully, predicts the artist of new songs more accurately.




**(c)** The black dots are the car stopping distance data, and the blue line shows the prediction for $k$-NN with $k = 1$ for any $x$. As for the classification problem above, $k$-NN with $k = 1$ overfits to the training data.




**(d)** The car stopping distance, this time with $k = 20$. Except for the boundary effect at the right, this seems like a much more useful model which captures the interesting effects of the data and ignores the noise.

**Figure 2.5:** $k$-NN applied to the music classification Example 2.1 (a and b) and the car stopping distance Example 2.2 (c and d). For both problems $k$-NN is applied with $k = 1$ and $k = 20$.

and energy of a song will give a complete picture of the artistic styles. Hence, even with the best possible model, there is some ambiguity about which artist has recorded a song if we only look at these two input variables. This ambiguity is modelled as random noise. Also for the car stopping distance, there appear to be a certain amount of random effects, not only in $x$ but also in $y$. By using $k = 1$ and thereby adapting very closely to the training data, the predictions will depend not only on the interesting patterns in the problem but also on the (more or less) random effects that have shaped the training data. Typically we are not interested in capturing these effects, and we refer to this as *overfitting*.

With the $k$-NN classifier, we can mitigate overfitting by increasing the region of the neighbourhood used to compute the prediction, that is, increasing the hyperparameter $k$. With, for example, $k = 20$, the predictions are no longer based only on the closest neighbour but are instead a majority vote among the 20 closest neighbours. As a consequence, all training data points are no longer perfectly classified, but some of the songs end up in the wrong region in Figure 2.5b. The predictions are, however, less adapted to the peculiarities of the training data and thereby less overfitted, and Figure 2.5b and d are indeed less 'noisy' than Figure 2.5a and c. However, if we make $k$ too large, then the averaging effect will wash out all interesting patterns in the data as well. Indeed, for sufficiently large $k$ the neihgbourhood will include all training data points, and the model will reduce to predicting the mean of the data for any input.

Selecting $k$ is thus a trade-off between flexibility and rigidity. Since selecting $k$ either too big or too small will lead to a meaningless classifiers, there must exist a sweet spot for some moderate $k$ (possibly 20, but it could be less or more) where the classifier generalises the best. Unfortunately, there is no general answer to the $k$ for which this happens, and this is different for different problems. In the music classification problem, it seems reasonable that $k = 20$ will predict new test data points better than $k = 1$, but there might very well be an even better choice of $k$. For the car stopping problem, the behaviour is also more reasonable for $k = 20$ than $k = 1$, except for the boundary effect for large $x$, where $k$-NN is unable to capture the trend in the data as $x$ increases (simply because the 20 nearest neighbours are the same for all test points $x_\star$ around and above 35). A systematic way of choosing a good value for $k$ is to use cross-validation, which we will discuss in Chapter 4.

***Time to reflect 2.1*** *The prediction $\widehat{y}(\mathbf{x}_\star)$ obtained using the $k$-NN method is a piecewise constant function of the input $\mathbf{x}_\star$. For a classification problem, this is natural, since the output is categorical (see, for example, Figure 2.5 where the coloured regions correspond to areas of the input space where the prediction is constant according to the colour of that region). However, $k$-NN will also have piecewise constant predictions for regression problems. Why?*

## Input Normalisation

A final important practical aspect when using $k$-NN is the importance of normalisation of the input data. Imagine a training dataset with $p = 2$ input variables $\mathbf{x} = [x_1 \;\; x_2]^\mathsf{T}$ where all values of $x_1$ are in the range $[100, \; 1100]$ and the values for $x_2$ are in the much smaller range $[0, \; 1]$. It could, for example, be that $x_1$ and $x_2$ are measured in different units. The Euclidean distance between a test point $\mathbf{x}_\star$ and a training data point $\mathbf{x}_i$ is $\|\mathbf{x}_i - \mathbf{x}_\star\|_2 = \sqrt{(x_{i1} - x_{\star 1})^2 + (x_{i2} - x_{\star 2})^2}$. This expression will typically be dominated by the first term $(x_{i1} - x_{\star 1})^2$, whereas the second term $(x_{i2} - x_{\star 2})^2$ tends to have a much smaller effect, simply due to the different magnitude of $x_1$ and


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

$x_2$. That is, the different ranges lead to $x_1$ being considered much more important than $x_2$ by $k$-NN.

To avoid this undesired effect, we can re-scale the input variables. One option, in the mentioned example, could be to subtract 100 from $x_1$ and thereafter divide it by 1 000 and create $x_{i1}^{\text{new}} = \frac{x_{i1}-100}{1\,000}$ such that $x_1^{\text{new}}$ and $x_2$ both are in the range $[0,\ 1]$. More generally, this normalisation procedure for the input data can be written as

$$x_{ij}^{\text{new}} = \frac{x_{ij} - \min_\ell(x_{\ell j})}{\max_\ell(x_{\ell j}) - \min_\ell(x_{\ell j})}, \qquad \text{for all } j = 1, \ldots, p, \quad i = 1, \ldots, n. \tag{2.1}$$

Another common normalisation approach (sometimes called standardising) is by using the mean and standard deviation in the training data:

$$x_{ij}^{\text{new}} = \frac{x_{ij} - \bar{x}_j}{\sigma_j}, \qquad \forall j = 1, \ldots, p, \quad i = 1, \ldots, n, \tag{2.2}$$

where $\bar{x}_j$ and $\sigma_j$ are the mean and standard deviation for each input variable, respectively.

It is crucial for $k$-NN to apply some type of input normalisation (as was indeed done in Figure 2.5), but it is a good practice to apply this also when using other methods, for numerical stability if nothing else. It is, however, important to compute the scaling factors ($\min_\ell(x_{\ell j})$, $\bar{x}_j$, etc.) using training data only and to also apply that scaling to future test data points. Failing to do this, for example by performing normalisation before setting test data aside (which we will discuss more in Chapter 4), might lead to wrong conclusions on how well the method will perform in predicting future (not yet seen) data points.

## 2.3 A Rule-Based Method: Decision Trees

The $k$-NN method results in a prediction $\widehat{y}(\mathbf{x}_\star)$ that is a piecewise constant function of the input $\mathbf{x}_\star$. That is, the method partitions the input space into disjoint regions, and each region is associated with a certain (constant) prediction. For $k$-NN, these regions are given implicitly by the $k$-neighbourhood of each possible test input. An alternative approach, that we will study in this section, is to come up with a set of *rules* that defines the regions explicitly. For instance, considering the music data in Example 2.1, a simple set of high-level rules for constructing a classifier would be: inputs to the right in Figure 2.1 are classified as green (Bob Dylan), in the left as red (The Beatles), and in the upper part as blue (Kiss). We will now see how such rules can be learned systematically from the training data.

The rule-based models that we consider here are referred to as *decision trees*. The reason is that the rules used to define the model can be organised in a graph structure referred to as a binary tree. The decision tree effectively divides the input space into multiple disjoint regions, and in each region, a constant value is used for the prediction $\widehat{y}(\mathbf{x}_\star)$. We illustrate this with an example.

## Example 2.5 Predicting colours with a decision tree

We consider a classification problem with two numerical input variables $\mathbf{x} = [x_1 \ x_2]^\mathsf{T}$ and one categorical output $y$, the colour `Red` or `Blue`. For now, we do not consider any training data or how to actually learn the tree but only how an already existing decision tree can be used to predict $\widehat{y}(\mathbf{x}_\star)$.

The rules defining the model are organised in the graph in Figure 2.6, which is referred to as a binary tree. To use this tree to predict a label for the test input $\mathbf{x}_\star = [x_{\star 1} \ x_{\star 2}]^\mathsf{T}$, we start at the top, referred to as the *root node* of the tree (in the metaphor, the tree is growing upside down, with the root at the top and the leaves at the bottom). If the condition stated at the root is true, that is, if $x_{\star 2} < 3.0$, then we proceed down the left *branch*, otherwise along the right *branch*. If we reach a new *internal node* of the tree, we check the rule associated with that node and pick the left or the right branch accordingly. We continue and work our way down until we reach the end of a branch, called a *leaf node*. Each such final node corresponds to a constant prediction $\widehat{y}_m$, in this case one of the two classes `Red` or `Blue`.

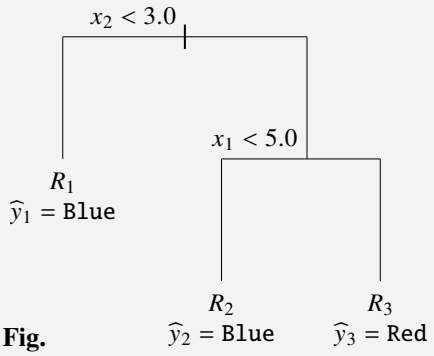



**Fig. 2.6** A classification tree. At each internal node, a rule of the form $x_j < s_k$ indicates the left branch coming from that split, and the right branch then consequently corresponds to $x_j \geq s_k$. This tree has two internal nodes (including the root) and three leaf nodes.

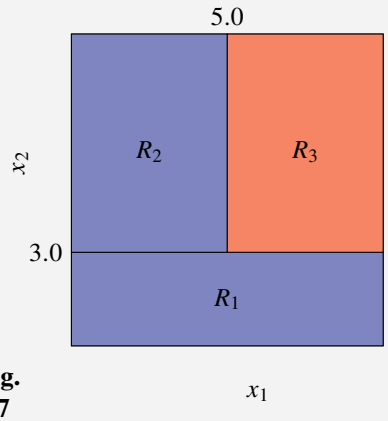



**Fig. 2.7** A region partition, where each region corresponds to a leaf node in the tree. Each border between regions corresponds to a split in the tree. Each region is coloured with the prediction corresponding to that region, and the boundary between red and blue is therefore the decision boundary.

The decision tree partitions the input space into axis-aligned 'boxes', as shown in Figure 2.7. By increasing the depth of the tree (the number of steps from the root to the leaves), the partitioning can be made finer and finer and thereby describes more complicated functions of the input variable.

Pseudo-code for predicting a test input with the tree in Figure 2.6 would look like:

```
if x_2 < 3.0 then
    return Blue
else
    if x_1 < 5.0 then
        return Blue
```


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September* 1, 2026, `http://smlbook.org`

```
        else
            return Red
        end
    end
```

As an example, if we have $\mathbf{x}_\star = [2.5 \;\; 3.5]^\mathsf{T}$, in the first split we would take the right branch since $x_{\star 2} = 3.5 \geq 3.0$, and in the second split we would take the left branch since $x_{\star 1} = 2.5 < 5.0$. The prediction for this test point would be $\widehat{y}(\mathbf{x}_\star) = \texttt{Blue}$.

To set the terminology, the endpoint of each branch $R_1$, $R_2$, and $R_3$ in Example 2.5 are called *leaf nodes*, and the internal splits, $x_2 < 3.0$ and $x_1 < 5.0$, are known as *internal nodes*. The lines that connect the nodes are referred to as *branches*. The tree is referred to as *binary* since each internal node splits into exactly two branches.

With more than two input variables, it is difficult to illustrate the partitioning of the input space into regions (Figure 2.7), but the tree representation can still be used in the very same way. Each internal node corresponds to a rule where one of the $p$ input variables $x_j$, $j = 1, \ldots, p$, is compared to a threshold $s$. If $x_j < s$, we continue along the left branch, and if $x_j \geq s$, we continue along the right branch.

The constant predictions that we associate with the leaf nodes can be either categorical (as in Example 2.5 above) or numerical. Decision trees can thus be used to address both classification and regression problems.

Example 2.5 illustrated how a decision tree can be used to make a prediction. We will now turn to the question of how a tree can be learned from training data.

### Learning a Regression Tree

We will start by discussing how to learn (or, equivalently, train) a decision tree for a regression problem. The classification problem is conceptually similar and will be explained later.

As mentioned above, the prediction $\widehat{y}(\mathbf{x}_\star)$ from a regression tree is a piecewise constant function of the input $\mathbf{x}_\star$. We can write this mathematically as,

$$\widehat{y}(\mathbf{x}_\star) = \sum_{\ell=1}^{L} \widehat{y}_\ell \mathbb{I}\{\mathbf{x}_\star \in R_\ell\}, \tag{2.3}$$

where $L$ is the total number of regions (leaf nodes) in the tree, $R_\ell$ is the $\ell$th region, and $\widehat{y}_\ell$ is the constant prediction for the $\ell$th region. Note that in the regression setting, $\widehat{y}_\ell$ is a numerical variable, and we will consider it to be a real number for simplicity. In the equation above, we have used the indicator function, $\mathbb{I}\{\mathbf{x} \in R_\ell\} = 1$ if $\mathbf{x} \in R_\ell$ and $\mathbb{I}\{\mathbf{x} \in R_\ell\} = 0$ otherwise.

Learning the tree from data corresponds to finding suitable values for the parameters defining the function (2.3), namely the regions $R_\ell$ and the constant predictions $\widehat{y}_\ell$,

$\ell = 1, \ldots, L$, as well as the total size of the tree $L$. If we start by assuming that the shape of the tree, the partition $(L, \{R_\ell\}_{\ell=1}^L)$, is known, then we can compute the constants $\{\widehat{y}_\ell\}_{\ell=1}^L$ in a natural way, simply as the average of the training data points falling in each region:

$$\boxed{\widehat{y}_\ell = \text{Average}\{y_i : \mathbf{x}_i \in R_\ell\}.}$$

It remains to find the shape of the tree, the regions $R_\ell$, which requires a bit more work. The basic idea is, of course, to select the regions so that the tree fits the training data. This means that the output predictions from the tree should match the output values in the training data. Unfortunately, even when restricting ourselves to seemingly simple regions such as the 'boxes' obtained from a decision tree, finding the tree (a collection of splitting rules) that optimally partitions the input space to fit the training data as well as possible turns out to be computationally infeasible. The problem is that there is a combinatorial explosion in the number of ways in which we can partition the input space. Searching through all possible binary trees is not possible in practice unless the tree size is so small that it is not of practical use.

To handle this situation, we use a heuristic algorithm known as *recursive binary splitting* for learning the tree. The word recursive means that we will determine the splitting rules one after the other, starting with the first split at the root and then building the tree from top to bottom. The algorithm is *greedy*, in the sense that the tree is constructed one split at a time, without having the complete tree 'in mind'. That is, when determining the splitting rule at the root node, the objective is to obtain a model that explains the training data as well as possible after a single split, without taking into consideration that additional splits may be added before arriving at the final model. When we have decided on the first split of the input space (corresponding to the root node of the tree), this split is kept fixed, and we continue in a similar way for the two resulting half-spaces (corresponding to the two branches of the tree), etc.

To see in detail how one step of this algorithm works, consider the situation when we are about to do our very first split at the root of the tree. Hence, we want to select one of the $p$ input variables $x_1, \ldots, x_p$ and a corresponding cutpoint $s$ which divide the input space into two half-spaces,

$$R_1(j, s) = \{\mathbf{x} \,|\, x_j < s\} \qquad \text{and} \qquad R_2(j, s) = \{\mathbf{x} \,|\, x_j \geq s\}. \tag{2.4}$$

Note that the regions depend on the index $j$ of the splitting variable as well as the value of the cutpoint $s$, which is why we write them as functions of $j$ and $s$. This is the case also for the predictions associated with the two regions,

$$\widehat{y}_1(j, s) = \text{Average}\{y_i : \mathbf{x}_i \in R_1(j, s)\} \quad \text{and} \quad \widehat{y}_2(j, s) = \text{Average}\{y_i : \mathbf{x}_i \in R_2(j, s)\},$$

since the averages in these expressions range over different data points depending on the regions.

For each training data point $(\mathbf{x}_i, y_i)$,we can compute a prediction error by first determining which region the data point falls in and then computing the difference


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September* 1, 2026, `http://smlbook.org`

between $y_i$ and the constant prediction associated with that region. Doing this for all training data points, the sum of squared errors can be written as

$$\sum_{i:\mathbf{x}_i \in R_1(j,s)} (y_i - \widehat{y}_1(j,s))^2 + \sum_{i:\mathbf{x}_i \in R_2(j,s)} (y_i - \widehat{y}_2(j,s))^2 . \tag{2.5}$$

The square is added to ensure that the expression above is non-negative and that both positive and negative errors are counted equally. The squared error is a common *loss function* used for measuring the closeness of a prediction to the training data, but other loss functions can also be used. We will discuss the choice of loss function in more detail in later chapters.

To find the optimal split, we select the values for $j$ and $s$ that minimise the squared error (2.5). This minimisation problem can be solved easily by looping through all possible values for $j = 1, \ldots, p$. For each $j$, we can scan through the finite number of possible splits and pick the pair $(j, s)$ for which the expression above is minimised. As pointed out above, when we have found the optimal split at the root node, this splitting rule is fixed. We then continue in the same way for the left and right branches independently. Each branch (corresponding to a half-space) is split again by minimising the squared prediction error over all training data points following that branch.

In principle, we can continue in this way until there is only a single training data point in each of the regions – that is, until $L = n$. Such a fully grown tree will result in predictions that exactly match the training data points, and the resulting model is quite similar to $k$-NN with $k = 1$. As pointed out above, this will typically result in too erratic a model that has overfitted to (possibly noisy) training data. To mitigate this issue, it is common to stop the growth of the tree at an earlier stage using some stopping criterion, for instance by deciding on $L$ beforehand, limiting the maximum depth (number of splits in any branch), or adding a constraint on the minimum number of training data points associated with each leaf node. Forcing the model to have more training data points in each leaf will result in an averaging effect, similar to increasing the value of $k$ in the $k$-NN method. Using such a stopping criterion means that the value of $L$ is not set manually but is determined adaptively based on the result of the learning procedure.

A high-level summary of the method is given in Method 2.2. Note that the learning in Method 2.2 includes a recursive call, where in each recursion we grow one branch of the tree one step further.

### Classification Trees

Trees can also be used for classification. We use the same procedure of recursive binary splitting but with two main differences. Firstly, we use a majority vote instead of an average to compute the prediction associated with each region:

$$\widehat{y}_\ell = \text{MajorityVote}\{y_i : \mathbf{x}_i \in R_\ell\}.$$

**Learn** a decision tree using recursive binary splitting

**Data:** Training data $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^n$
**Result:** Decision tree with regions $R_1, \ldots, R_L$ and corresponding predictions $\widehat{y}_1, \ldots, \widehat{y}_L$

Let $R$ denote the whole input space
Compute the regions $(R_1, \ldots, R_L) = \texttt{Split}(R, \mathcal{T})$
Compute the predictions $\widehat{y}_\ell$ for $\ell = 1, \ldots, L$ as

$$\widehat{y}_\ell = \begin{cases} \text{Average}\{y_i : \mathbf{x}_i \in R_\ell\} & \text{(Regression problems)} \\ \text{MajorityVote}\{y_i : \mathbf{x}_i \in R_\ell\} & \text{(Classification problems)} \end{cases}$$

**Function** `Split(`$R$`,`$\mathcal{T}$`)`**:**
  **if** *stopping criterion fulfilled* **then**
    **return** $R$
  **else**
    Go through all possible splits $x_j < s$ for all input variables $j = 1, \ldots, p$.
    Pick the pair $(j, s)$ that minimises (2.5)/(2.6) for regression/classification problems.
    Split region $R$ into $R_1$ and $R_2$ according to (2.4).
    Split data $\mathcal{T}$ into $\mathcal{T}_1$ and $\mathcal{T}_2$ accordingly.
    **return** `Split(`$R_1$`,`$\mathcal{T}_1$`)`, `Split(`$R_2$`,`$\mathcal{T}_2$`)`
  **end**
**end**

**Predict** from a decision tree

**Data:** Decision tree with regions $R_1, \ldots, R_L$, training data $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^n$, test data point $\mathbf{x}_\star$
**Result:** Predicted test output $\widehat{y}(\mathbf{x}_\star)$

Find the region $R_\ell$ which $\mathbf{x}_\star$ belongs to.
Return the prediction $\widehat{y}(\mathbf{x}_\star) = \widehat{y}_\ell$.

**Method 2.2:** Decision trees

Secondly, when learning the tree, we need a different splitting criterion than the squared prediction error to take into account the fact that the output is categorical. To define these criteria, note first that the split at any internal node is computed by solving an optimisation problem of the form

$$\arg\min_{j,s} \; n_1 Q_1 + n_2 Q_2, \tag{2.6}$$


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September* 1, 2026, `http://smlbook.org`

where $n_1$ and $n_2$ denote the number of training data points in the left and right nodes of the current split, respectively, and $Q_1$ and $Q_2$ are the costs (derived form the prediction errors) associated with these two nodes. The variables $j$ and $s$ denote the index of the splitting variable and the cutpoint as before. All of the terms $n_1$, $n_2$, $Q_1$, and $Q_2$ depend on these variables, but we have dropped the explicit dependence from the notation for brevity. Comparing (2.6) with (2.5), we see that we recover the regression case if $Q_\ell$ corresponds to the mean-squared error in node $\ell$.

To generalise this to the classification case, we still solve the optimisation problem (2.6) to compute the split, but choose $Q_\ell$ in a different way which respects the categorical nature of a classification problem. To this end, we first introduce

$$\widehat{\pi}_{\ell m} = \frac{1}{n_\ell} \sum_{i:\mathbf{x}_i \in R_\ell} \mathbb{I}\{y_i = m\}$$

to be the proportion of training observations in the $\ell$th region that belong to the $m$th class. We can then define the *splitting criterion*, $Q_\ell$, based on these class proportions. One simple alternative is the *misclassification rate*

$$Q_\ell = 1 - \max_m \widehat{\pi}_{\ell m}, \tag{2.7a}$$

which is simply the proportion of data points in region $R_\ell$ which do not belong to the most common class. Other common splitting criteria are the *Gini index*

$$Q_\ell = \sum_{m=1}^{M} \widehat{\pi}_{\ell m}(1 - \widehat{\pi}_{\ell m}) \tag{2.7b}$$

and the *entropy* criterion,

$$Q_\ell = -\sum_{m=1}^{M} \widehat{\pi}_{\ell m} \ln \widehat{\pi}_{\ell m}. \tag{2.7c}$$

In Example 2.6 we illustrate how to construct a classification tree using recursive binary splitting and with the entropy as the splitting criterion.

**Example 2.6 Learning a classification tree (continuation of Example 2.5)**

We consider the same setup as in Example 2.5, but now with the following dataset:

We want to learn a classification tree by using the entropy criterion in (2.7c) and growing the tree until there are no regions with more than five data points left.

**First split:** There are infinitely many possible splits we can make, but all splits which give the same partition of the data points are equivalent. Hence, in practice we only have nine different splits to consider in this dataset. The data (dots) and these possible splits (dashed lines) are visualised in Figure 2.8.

We consider all nine splits in turn. We start with the split at $x_1 = 2.5$, which splits

the input space into two regions, $R_1 = x_1 < 2.5$ and $R_2 = x_1 \geq 2.5$. In region $R_1$ we

| $x_1$ | $x_2$ | $y$ |
|---|---|---|
| 9.0 | 2.0 | Blue |
| 1.0 | 4.0 | Blue |
| 4.0 | 6.0 | Blue |
| 4.0 | 1.0 | Blue |
| 1.0 | 2.0 | Blue |
| 1.0 | 8.0 | Red |
| 6.0 | 4.0 | Red |
| 7.0 | 9.0 | Red |
| 9.0 | 8.0 | Red |
| 9.0 | 6.0 | Red |

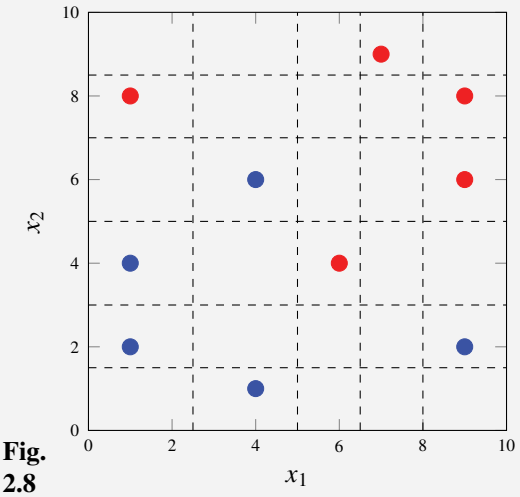



**Fig. 2.8**

have two blue data points and one red, in total $n_1 = 3$ data points. The proportion of the two classes in region $R_1$ will therefore be $\widehat{\pi}_{1\text{B}} = 2/3$ and $\widehat{\pi}_{1\text{R}} = 1/3$. The entropy is calculated as

$$Q_1 = -\widehat{\pi}_{1\text{B}} \ln(\widehat{\pi}_{1\text{B}}) - \widehat{\pi}_{1\text{R}} \ln(\widehat{\pi}_{1\text{R}}) = -\frac{2}{3} \ln\left(\frac{2}{3}\right) - \frac{1}{3} \ln\left(\frac{1}{3}\right) = 0.64.$$

In region $R_2$ we have $n_2 = 7$ data points with the proportions $\widehat{\pi}_{2\text{B}} = 3/7$ and $\widehat{\pi}_{2\text{R}} = 4/7$. The entropy for this region will be

$$Q_2 = -\widehat{\pi}_{2\text{B}} \ln(\widehat{\pi}_{2\text{B}}) - \widehat{\pi}_{2\text{R}} \ln(\widehat{\pi}_{2\text{R}}) = -\frac{3}{7} \ln\left(\frac{3}{7}\right) - \frac{4}{7} \ln\left(\frac{4}{7}\right) = 0.68,$$

and inserted in (2.6), the total weighted entropy for this split becomes

$$n_1 Q_1 + n_2 Q_2 = 3 \cdot 0.64 + 7 \cdot 0.68 = 6.69.$$

We compute the costs for all other splits in the same manner and summarise them in the table below:

| Split ($R_1$) | $n_1$ | $\widehat{\pi}_{1\text{B}}$ | $\widehat{\pi}_{1\text{R}}$ | $Q_1$ | $n_2$ | $\widehat{\pi}_{2\text{B}}$ | $\widehat{\pi}_{2\text{R}}$ | $Q_2$ | $n_1 Q_1 + n_2 Q_2$ |
|---|---|---|---|---|---|---|---|---|---|
| $x_1 < 2.5$ | 3 | 2/3 | 1/3 | 0.64 | 7 | 3/7 | 4/7 | 0.68 | 6.69 |
| $x_1 < 5.0$ | 5 | 4/5 | 1/5 | 0.50 | 5 | 1/5 | 4/5 | 0.50 | 5.00 |
| $x_1 < 6.5$ | 6 | 4/6 | 2/6 | 0.64 | 4 | 1/4 | 3/4 | 0.56 | 6.07 |
| $x_1 < 8.0$ | 7 | 4/7 | 3/7 | 0.68 | 3 | 1/3 | 2/3 | 0.64 | 6.69 |
| $x_2 < 1.5$ | 1 | 1/1 | 0/1 | 0.00 | 9 | 4/9 | 5/9 | 0.69 | 6.18 |
| $x_2 < 3.0$ | 3 | 3/3 | 0/3 | 0.00 | 7 | 2/7 | 5/7 | 0.60 | **4.18** |
| $x_2 < 5.0$ | 5 | 4/5 | 1/5 | 0.50 | 5 | 1/5 | 4/5 | 0.06 | 5.00 |
| $x_2 < 7.0$ | 7 | 5/7 | 2/7 | 0.60 | 3 | 0/3 | 3/3 | 0.00 | **4.18** |
| $x_2 < 8.5$ | 9 | 5/9 | 4/9 | 0.69 | 1 | 0/1 | 1/1 | 0.00 | 6.18 |


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September* 1, 2026, `http://smlbook.org`

From the table, we can read that the two splits at $x_2 < 3.0$ and $x_2 < 7.0$ are both equally good. We choose to continue with $x_2 < 3.0$.

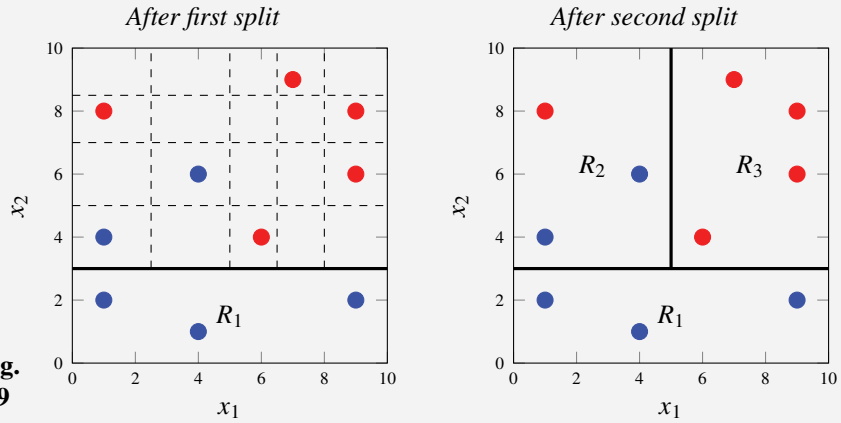



**Fig. 2.9**

**Second split:** We note that only the upper region has more than five data points. Also, there is no point splitting region $R_1$ further since it only contains data points from the same class. In the next step, we therefore split the upper region into two new regions, $R_2$ and $R_3$. All possible splits are displayed in Figure 2.9 to the left (dashed lines), and we compute their costs in the same manner as before:

| Splits ($R_1$) | $n_2$ | $\widehat{\pi}_{2B}$ | $\widehat{\pi}_{2R}$ | $Q_2$ | $n_3$ | $\widehat{\pi}_{3B}$ | $\widehat{\pi}_{3R}$ | $Q_3$ | $n_2Q_2 + n_3Q_3$ |
|---|---|---|---|---|---|---|---|---|---|
| $x_1 < 2.5$ | 2 | 1/2 | 1/2 | 0.69 | 5 | 1/5 | 4/5 | 0.50 | 3.89 |
| $x_1 < 5.0$ | 3 | 2/3 | 1/3 | 0.63 | 4 | 0/4 | 4/4 | 0.00 | **1.91** |
| $x_1 < 6.5$ | 4 | 2/4 | 2/4 | 0.69 | 3 | 0/3 | 3/3 | 0.00 | 2.77 |
| $x_1 < 8.0$ | 5 | 2/5 | 3/5 | 0.67 | 2 | 0/2 | 2/2 | 0.00 | 3.37 |
| $x_2 < 5.0$ | 2 | 1/2 | 1/2 | 0.69 | 5 | 1/5 | 4/5 | 0.50 | 3.88 |
| $x_2 < 7.0$ | 4 | 2/4 | 2/4 | 0.69 | 3 | 0/3 | 3/3 | 0.00 | 2.77 |
| $x_2 < 8.5$ | 6 | 2/6 | 4/6 | 0.64 | 1 | 0/1 | 1/1 | 0.00 | 3.82 |

The best split is the one at $x_1 < 5.0$, visualised above to the right. None of the three regions has more than five data points. Therefore, we terminate the training. The final tree and its partitions were displayed in Example 2.5. If we want to use the tree for prediction, we predict blue if $\mathbf{x}_\star \in R_1$ or $\mathbf{x}_\star \in R_2$ since the blue training data points are in the majority in each of these two regions. Similarly, we predict red if $\mathbf{x}_\star \in R_3$.

When choosing between the different splitting criteria mentioned above, the misclassification rate sounds like a reasonable choice since that is typically the criterion we want the final model to do well on.[6] However, one drawback is that it does not favour pure nodes. By pure nodes we mean nodes where most of the data points belong to a certain class. It is usually an advantage to favour pure nodes in the

[6] This is not always true, for example for imbalanced and asymmetric classification problems; see Section 4.5.

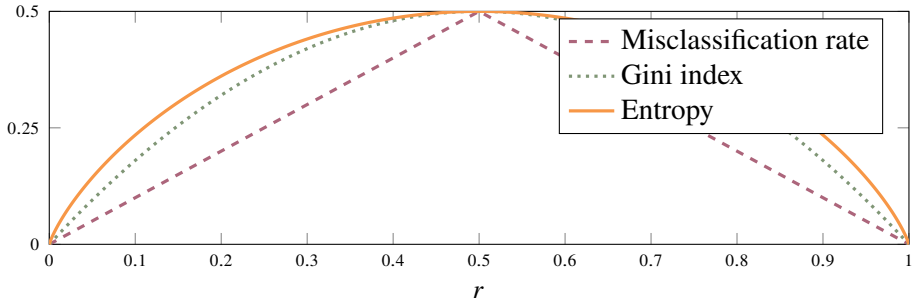



**Figure 2.10:** Three splitting criteria for classification trees as a function of the proportion of the first class $r = \pi_{\ell 1}$ in a certain region $R_\ell$ as given in (2.8). The entropy criterion has been scaled such that it passes through (0.5,0.5).

greedy procedure that we use to grow the tree, since this can lead to fewer splits in total. Both the entropy criterion and the Gini index favour node purity more than the misclassification rate does.

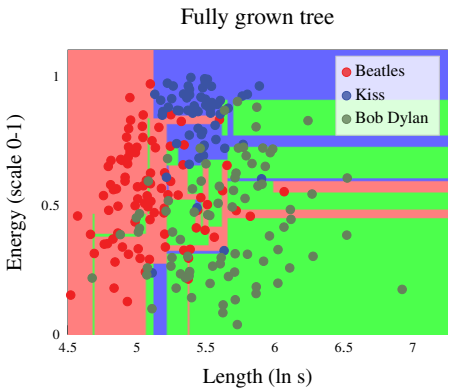
This advantage can also be illustrated in Example 2.6. Consider the first split in this example. If we were to use the misclassification rate as the splitting criterion, both the split $x_2 < 5.0$ and the split $x_2 < 3.0$ would provide a total misclassification rate of 0.2. However, the split at $x_2 < 3.0$, which the entropy criterion favoured, provides a pure node $R_1$. If we now went with the split $x_2 < 5.0$, the misclassification after the second split would still be 0.2. If we continued to grow the tree until no data points were misclassified, we would need three splits if we used the entropy criterion, whereas we would need five splits if we used the misclassification criterion and started with the split at $x_2 < 5.0$.

To generalise this discussion, consider a problem with two classes, where we denote the proportion of the first class as $\pi_{\ell 1} = r$ and hence the proportion of the second class as $\pi_{\ell 2} = 1 - r$. The three criteria (2.7) can then be expressed in terms of $r$ as

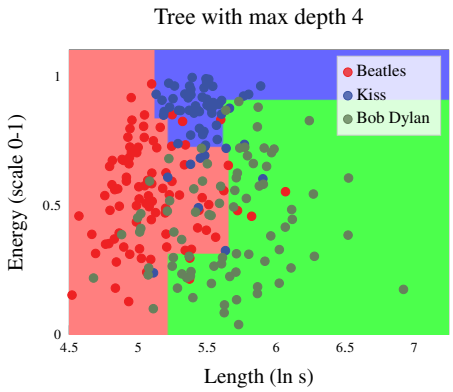
$$\begin{aligned} \text{Misclassification rate:} \quad & Q_\ell = 1 - \max(r, 1-r), \\ \text{Gini index:} \quad & Q_\ell = 2r(1-r), \\ \text{Entropy:} \quad & Q_\ell = -r \ln r - (1-r)\ln(1-r). \end{aligned} \tag{2.8}$$

These functions are shown in Figure 2.10. All three citeria are similar in the sense that they provide zero loss if all data points belong to either of the two classes and maximum loss if the data points are equally divided between the two classes. However, the Gini index and entropy have a higher loss for all other proportions. In other words, the gain of having a pure node ($r$ close to 0 or 1) is higher for the Gini index and the entropy than for the misclassification rate. As a consequence, the Gini index and the entropy both tend to favour making one of the two nodes pure (or close to pure) since that provides a smaller total loss, which can make a good combination with the greedy nature of the recursive binary splitting.

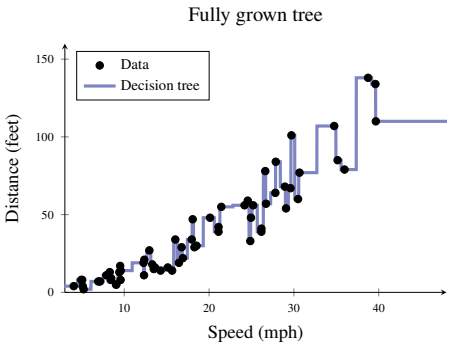

*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September* 1, 2026, `http://smlbook.org`



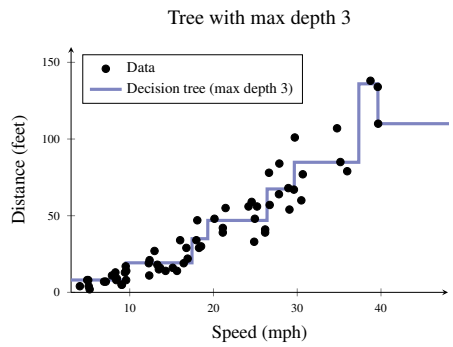

**(a)** Decision boundaries for the music classification problem for a fully grown classification tree with the Gini index. This model overfits the data.




**(b)** The same problem and data as in 2.11a for which a tree restricted to depth 4 has been learned, again using the Gini index. This models will hopefully make better predictions for new data.




**(c)** The prediction for a fully grown regression tree. As for the classification problem above, this model overfits to the training data.




**(d)** The same problem and data as in 2.11c for which a tree restricted to depth 3 has been learned.

**Figure 2.11:** Decision trees applied to the music classification Example 2.1 (a and b) and the car stopping distance Example 2.2 (c and d).

## How Deep Should a Decision Tree be?

The depth of a decision tree (the maximum distance between the root node and any leaf node) has a big impact on the final predictions. The tree depth impacts the predictions in a somewhat similar way to the hyperparameter $k$ in $k$-NN. We again use the music classification and car stopping distance problems from Examples 2.1 and 2.2 to study how the decision boundaries change depending on the depth of the trees. In Figure 2.11, the decision boundaries are illustrated for two different trees. In Figure 2.11a and c, we have not restricted the depth of the tree and have grown it until each region contains only data points with the same output value – a so-called fully grown tree. In Figure 2.11b and d, the maximum depth is restricted to 4 and 3, respectively.

Similarly to choosing $k = 1$ in $k$-NN, for a fully grown tree, all training data points will, by construction, be correctly predicted since each region only contains data points with the same output. As a result, for the music classification problem, we get thin and small regions adapted to single training data points, and for the car stopping distance problem, we get a very irregular line passing exactly through the observations. Even though these trees give excellent performance on the training data, they are not likely to be the best models for new, as yet unseen data. As we discussed previously in the context of $k$-NN, we refer to this as overfitting.

In decision trees, we can mitigate overfitting by using shallower trees. Consequently, we get fewer and larger regions with an increased averaging effect, resulting in decision boundaries that are less adapted to the noise in the training data. This is illustrated in Figure 2.11b and d for the two example problems. As for $k$ in $k$-NN, the optimal size of the tree depends on many properties of the problem, and it is a trade-off between flexibility and rigidity. Similar trade-offs have to be made for almost all methods presented in this book, and they will be discussed systematically in Chapter 4.

How can the user control the growth of the tree? Here we have different strategies. The most straightforward strategy is to adjust the stopping criterion, that is, the condition that should be fulfilled for not proceeding with further splits in a certain node. As mentioned earlier, this criterion could be that we do not attempt further splits if there are less than a certain number of training data points in the corresponding region, or, as in Figure 2.11, we can stop splitting when we reach a certain depth. Another strategy to control the depth is to use *pruning*. In pruning, we start with a fully grown tree, and then in a second post-processing step, prune it back to a smaller one. We will, however, not discuss pruning further here.

## 2.4 Self-Supervised Learning

The supervised problem might come across as a somewhat limited problem formulation, and one might question whether it is worth putting so much effort in developing methods for such a seemingly narrow scope.

As it turns out, the supervised problem formulation is way more powerful than it may seem at a first glance, and an important part of many recent advances in applied machine learning is about formulating seemingly non-supervised problems as supervised problems. One such example is *self-supervised learning*. In real-world situations we may have a long stretch of text and want a model that can continue writing it, or we may have a long time series of measurements and want a forecast of what comes next. At first glance these situations seem not to fit the supervised formulation, but with a clever re-formulation called self-supervised, they actually do. Self-supervised learning allows all supervised methods, from $k$-NN and trees to deep learning and kernel methods in later chapters, to be applied also to these kinds of problems.

Recall that the supervised problem assumes there is a training dataset of input–output pairs in $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^{n}$. If we want to predict the continuation of a single sequence


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September* 1, 2026, `http://smlbook.org`

of categorical or numerical data $y_1, y_2, \ldots, y_N$, it might seem like an issue that there is no explicit input variable $\mathbf{x}$ but only a sequence of $y$-values. Now the idea of self-supervised learning is to manufacture a supervised problem out of such unlabelled data by exploiting the structure already present in the data. If we are interested in predicting the next value in a sequence from its past, we simply *define* the input to be a window of past values and the output to be the next value. Suddenly we have a supervised problem on our hands, and we can apply any supervised method to solve it. The term *self-supervised* reflects the fact that the inputs and outputs originate from the same underlying data. No separate labelling step is needed, and the data is, in a sense, supervising itself.

**Example 2.7 Time-series forecasting as self-supervised learning**

Assume that we have observed a single time series $y_1, y_2, \ldots, y_T$ and want to use machine learning for predicting the next value. To turn this into a self-supervised problem, we define the input at time $t$ to be the vector of $p$ past values, $\mathbf{x}_t = [y_{t-p} \ \ldots \ y_{t-1}]^\mathsf{T}$, and the output to be $y_t$. With $T = 6$ observed values and a window length of $p = 2$, the sequence

$$y_1 = 1.2, \quad y_2 = 1.5, \quad y_3 = 1.7, \quad y_4 = 2.1, \quad y_5 = 2.0, \quad y_6 = 2.4$$

gives rise to four training samples:

| Input $\mathbf{x}_t$ | Output $y_t$ |
|---|---|
| $[1.2 \ \ 1.5]^\mathsf{T}$ | 1.7 |
| $[1.5 \ \ 1.7]^\mathsf{T}$ | 2.1 |
| $[1.7 \ \ 2.1]^\mathsf{T}$ | 2.0 |
| $[2.1 \ \ 2.0]^\mathsf{T}$ | 2.4 |

A decision tree could now be applied to this dataset, yielding a so-called autoregressive tree, or more generally an *autoregressive model of order* $p$. Note that the four training pairs are generated entirely from the recorded sequence; no separate labelling step is required.

This trick is by no means limited to numerical sequences as in this example, and this idea will be applied to text problems and language models in Chapter 7.

## 2.5 Further Reading

The reason why we started this book by $k$-NN is that it is perhaps the most intuitive and straightforward way to solve a classification problem. The idea is at least a thousand years old and was described already by Ḥassan Ibn al-Haytham (latinised as Alhazen) around the year 1030 in *Kitāb al-Manāẓir* (*Book* of *Optics*) (Pelillo 2014), as an explanation of how the human brain perceives objects. As with many good ideas, the nearest neighbour idea has been re-invented many times, and a more modern

description of $k$-NN can be found in Cover and Hart (1967).

Also, the basic idea of decision trees is relatively simple, but there are many possible ways to improve and extend them as well as different options for how to implement them in detail. A somewhat longer introduction to decision trees is found in Hastie et al. (2009), and a historically oriented overview can be found in Loh (2014). Of particular significance is perhaps CART (Classification and Regression Trees, Breiman et al. (1984)), as well as ID3 and C4.5 (Quinlan 1986), Quinlan (1993).


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September* 1, 2026, `http://smlbook.org`

# 3 Basic Parametric Models and a Statistical Perspective on Learning

In the previous chapter, we introduced the supervised machine learning problem, as well as two methods for solving it. In this chapter, we will consider a generic approach to learning referred to as *parametric* modelling. In particular, we will introduce *linear regression* and *logistic regression*, which are two such parametric models. The key point of a parametric model is that it contains some parameters $\boldsymbol{\theta}$, which are learned from training data. However, once the parameters are learned, the training data may be discarded, since the prediction only depends on $\boldsymbol{\theta}$.

## 3.1 Linear Regression

Regression is one of the two fundamental tasks of supervised learning (the other one is classification). We will now introduce the *linear regression* model, which might (at least historically) be the most popular method for solving regression problems. Despite its relative simplicity, it is a surprisingly useful and is an important stepping stone for more advanced methods, such as deep learning (see Chapter 6).

As discussed in the previous chapter, regression amounts to learning the relationships between some input variables $\mathbf{x} = [x_1 \ x_2 \ \dots \ x_p]^\mathsf{T}$ and a numerical output variable $y$. The inputs can be either categorical or numerical, but we will start by assuming that all $p$ inputs are numerical as well, and discuss categorical inputs later. In a more mathematical framework, regression is about learning a *model* $f$,

$$y = f(\mathbf{x}) + \varepsilon, \tag{3.1}$$

mapping the input to the output, where $\varepsilon$ is an error term that describes everything about the input–output relationship that cannot be captured by the model. With a statistical perspective, we consider $\varepsilon$ as a random variable, referred to as *noise*, that is independent of $\mathbf{x}$ and has mean value of zero. As a running example of regression, we will use the car stopping distance regression problem introduced in the previous chapter as Example 2.2.

### The Linear Regression Model

The linear regression model assumes that the output variable $y$ (a scalar) can be described as an affine[1] combination of the $p$ input variables $x_1, x_2, \ldots, x_p$ plus a noise term $\varepsilon$,

$$y = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_p x_p + \varepsilon. \tag{3.2}$$

We refer to the coefficients $\theta_0, \theta_1, \ldots \theta_p$ as the *parameters* of the model, and we sometimes refer to $\theta_0$ specifically as the intercept (or offset) term. The noise term $\varepsilon$ accounts for random errors in the data not captured by the model. The noise is assumed to have mean zero and to be independent of $\mathbf{x}$. The zero-mean assumption is nonrestrictive, since any (constant) non-zero mean can be incorporated in the offset term $\theta_0$.

To have a more compact notation, we introduce the parameter vector $\boldsymbol{\theta} = [\theta_0 \quad \theta_1 \quad \ldots \quad \theta_p]^\mathsf{T}$ and extend the vector $\mathbf{x}$ with a constant one in its first position, such that we can write the linear regression model (3.2) compactly as

$$y = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_p x_p + \varepsilon = \begin{bmatrix}\theta_0 & \theta_1 & \ldots & \theta_p\end{bmatrix} \begin{bmatrix}1 \\ x_1 \\ \vdots \\ x_p\end{bmatrix} + \varepsilon = \boldsymbol{\theta}^\mathsf{T}\mathbf{x} + \varepsilon. \tag{3.3}$$

This notation means that the symbol $\mathbf{x}$ is used both for the $p+1$ and the $p$-dimensional versions of the input vector, with or without the constant one in the leading position, respectively. This is only a matter of book-keeping for handling the intercept term $\theta_0$. Which definition is used will be clear from the context and carries no deeper meaning.

The linear regression model is a *parametric* function of the form (3.3). The parameters $\boldsymbol{\theta}$ can take arbitrary values, and the actual values that we assign to them will control the input–output relationship described by the model. *Learning* of the model therefore amounts to finding suitable values for $\boldsymbol{\theta}$ based on observed training data. Before discussing how to do this, however, let us first look at how to use the model for predictions once it has been learned.

The goal in supervised machine learning is making predictions $\widehat{y}(\mathbf{x}_\star)$ for new, previously unseen, test inputs $\mathbf{x}_\star = [1 \;\; x_{\star 1} \;\; x_{\star 2} \;\; \cdots \;\; x_{\star p}]^\mathsf{T}$. Let us assume that we have already learned some parameter values $\widehat{\boldsymbol{\theta}}$ for the linear regression model (how this is done will be described next). We use the symbol $\widehat{\ }$ to indicate that $\widehat{\boldsymbol{\theta}}$ contains learned values of the unknown parameter vector $\boldsymbol{\theta}$. Since we assume that the noise

[1] An affine function is a linear function plus a constant offset.


Machine Learning – A First Course for Engineers and Scientists
Second Edition, online draft version September 1, 2026, http://smlbook.org

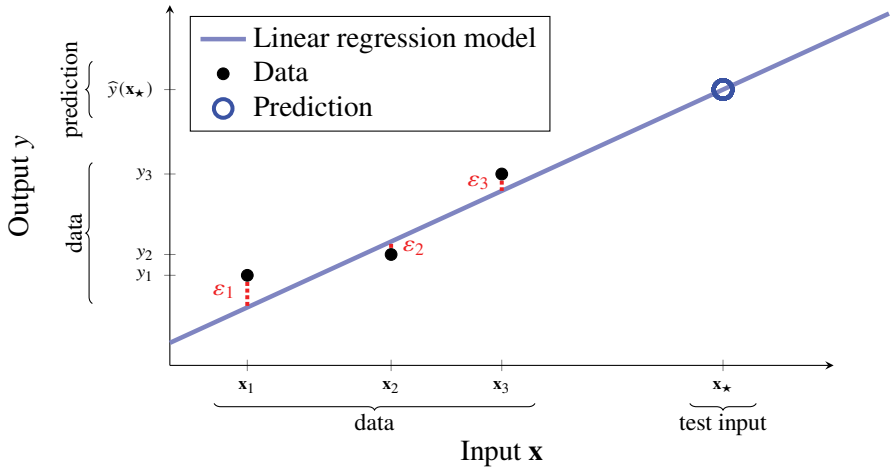



**Figure 3.1:** Linear regression with $p = 1$: The black dots represent $n = 3$ data points, from which a linear regression model (blue line) is learned. The model does not fit the data perfectly, so there is a remaining error corresponding to the noise $\varepsilon$ (red). The model can be used to *predict* (blue circle) the output $\widehat{y}(\mathbf{x}_\star)$ for a test input $\mathbf{x}_\star$.

term $\varepsilon$ is random with zero mean and independent of all observed variables, it makes sense to replace $\varepsilon$ with 0 in the prediction. That is, a prediction from the linear regression model takes the form

$$\widehat{y}(\mathbf{x}_\star) = \widehat{\theta}_0 + \widehat{\theta}_1 x_{\star 1} + \widehat{\theta}_2 x_{\star 2} + \cdots + \widehat{\theta}_p x_{\star p} = \widehat{\boldsymbol{\theta}}^\mathsf{T} \mathbf{x}_\star. \tag{3.4}$$

The noise term $\varepsilon$ is often referred to as an *irreducible error* or an *aleatoric*[2] uncertainty in the prediction. We illustrate the predictions made by a linear regression model in Figure 3.1.

### Training a Linear Regression Model from Training Data

Let us now discuss how to train a linear regression model, that is to learn $\boldsymbol{\theta}$, from training data $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^n$. We collect the training data, which consists of $n$ data point with inputs $\mathbf{x}_i$ and outputs $y_i$, in the $n \times (p+1)$ matrix $\mathbf{X}$ and $n$-dimensional vector $\mathbf{y}$,

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1^\mathsf{T} \\ \mathbf{x}_2^\mathsf{T} \\ \vdots \\ \mathbf{x}_n^\mathsf{T} \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, \quad \text{where each } \mathbf{x}_i = \begin{bmatrix} 1 \\ x_{i1} \\ x_{i2} \\ \vdots \\ x_{ip} \end{bmatrix}. \tag{3.5}$$

[2] From the Latin word *aleator*, meaning dice-player.

**Example 3.1 Car stopping distances**

We continue Example 2.2 and train a linear regression model for the car stopping distance data. We start by forming the matrices $\mathbf{X}$ and $\mathbf{y}$. Since we only have one input and one output, both $x_i$ and $y_i$ are scalar. We get

$$\mathbf{X} = \begin{bmatrix} 1 & 4.0 \\ 1 & 4.9 \\ 1 & 5.0 \\ 1 & 5.1 \\ 1 & 5.2 \\ \vdots & \vdots \\ 1 & 39.6 \\ 1 & 39.7 \end{bmatrix}, \qquad \boldsymbol{\theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}, \quad \text{and} \quad \mathbf{y} = \begin{bmatrix} 4.0 \\ 8.0 \\ 8.0 \\ 4.0 \\ 2.0 \\ \vdots \\ 134.0 \\ 110.0 \end{bmatrix}. \tag{3.6}$$

Altogether we can use this vector and matrix notation to describe the linear regression model for all training data points $\mathbf{x}_i$, $i = 1, \ldots, n$ in one equation as a matrix multiplication,

$$\mathbf{y} = \mathbf{X}\boldsymbol{\theta} + \boldsymbol{\epsilon}, \tag{3.7}$$

where $\boldsymbol{\epsilon}$ is a vector of errors/noise terms. Moreover, we can also define a vector of predicted outputs for the training data $\widehat{\mathbf{y}} = \begin{bmatrix} \widehat{y}(\mathbf{x}_1) & \widehat{y}(\mathbf{x}_2) & \ldots & \widehat{y}(\mathbf{x}_n) \end{bmatrix}^\mathsf{T}$, which also allows a compact matrix formulation,

$$\widehat{\mathbf{y}} = \mathbf{X}\boldsymbol{\theta}. \tag{3.8}$$

Note that whereas $\mathbf{y}$ is a vector of recorded training data values, $\widehat{\mathbf{y}}$ is a vector whose entries are functions of $\boldsymbol{\theta}$. Learning the unknown parameters $\boldsymbol{\theta}$ amounts to finding values such that $\widehat{\mathbf{y}}$ is similar to $\mathbf{y}$. That is, the predictions given by the model should fit the training data well. There are multiple ways to define what 'similar' or 'well' actually means, but it somehow amounts to finding $\boldsymbol{\theta}$ such that $\widehat{\mathbf{y}} - \mathbf{y} = \boldsymbol{\epsilon}$ is small. We will approach this by formulating a loss function, which gives a mathematical meaning to 'fitting the data well'. We will thereafter interpret the loss function from a statistical perspective, by understanding this as selecting the value of $\boldsymbol{\theta}$ which makes the observed training data $\mathbf{y}$ as likely as possible with respect to the model – the so-called *maximum likelihood* solution. Later, in Chapter 10, we will also introduce a conceptually different way of learning $\boldsymbol{\theta}$.

## Loss Functions and Cost Functions

A principled way to define the learning problem is to introduce a *loss function* $L(\widehat{y}, y)$ which measures how close the model's prediction $\widehat{y}$ is to the observed data $y$. If the model fits the data well, so that $\widehat{y} \approx y$, then the loss function should take a small value, and vice versa. Based on the chosen loss function, we also define the *cost function* as


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

the average loss over the training data. Training a model then amounts to finding the parameter values that minimise the cost

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \underbrace{\frac{1}{n}\sum_{i=1}^{n} \overbrace{L(\widehat{y}(\mathbf{x}_i;\boldsymbol{\theta}), y_i)}^{\text{loss function}}}_{\text{cost function } J(\boldsymbol{\theta})}. \tag{3.9}$$

Note that each term in the expression above corresponds to evaluating the loss function for the prediction $\widehat{y}(\mathbf{x}_i;\boldsymbol{\theta})$, given by (3.4), for the training point with index $i$ and the true output value $y_i$ at that point. To emphasise that the prediction depends on the parameters $\boldsymbol{\theta}$, we have included $\boldsymbol{\theta}$ as an argument to $\widehat{y}$ for clarity. The operator $\arg\min_{\boldsymbol{\theta}}$ means 'the value of $\boldsymbol{\theta}$ for which the cost function attains it minimum'. The relationship between loss and cost functions (3.9) is general for all cost functions in this book.

### Least Squares and the Normal Equations

For regression, a commonly used loss function is the *squared error* loss

$$L(\widehat{y}(\mathbf{x};\boldsymbol{\theta}), y) = \big(\widehat{y}(\mathbf{x};\boldsymbol{\theta}) - y\big)^2. \tag{3.10}$$

This loss function is 0 if $\widehat{y}(\mathbf{x};\boldsymbol{\theta}) = y$ and grows fast (quadratically) as the difference between $y$ and the prediction $\widehat{y}(\mathbf{x};\boldsymbol{\theta}) = \boldsymbol{\theta}^\mathsf{T}\mathbf{x}$ increases. The corresponding cost function for the linear regression model (3.7) can be written with matrix notation as

$$J(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n}(\widehat{y}(\mathbf{x}_i;\boldsymbol{\theta}) - y_i)^2 = \frac{1}{n}\|\widehat{\mathbf{y}} - \mathbf{y}\|_2^2 = \frac{1}{n}\|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|_2^2 = \frac{1}{n}\|\boldsymbol{\epsilon}\|_2^2, \tag{3.11}$$

where $\|\cdot\|_2$ denotes the usual Euclidean vector norm and $\|\cdot\|_2^2$ its square. Due to the square, this particular cost function is also commonly referred to as the *least squares* cost. It is illustrated in Figure 3.2. We will discuss other loss functions in Chapter 5.

When using the squared error loss for learning a linear regression model from $\mathcal{T}$, we thus need to solve the problem

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \frac{1}{n}\sum_{i=1}^{n}(\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i - y_i)^2 = \arg\min_{\boldsymbol{\theta}} \frac{1}{n}\|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|_2^2. \tag{3.12}$$

From a linear algebra point of view, this can be seen as the problem of finding the closest vector to $\mathbf{y}$ (in an Euclidean sense) in the subspace of $\mathbb{R}^n$ spanned by the columns of $\mathbf{X}$. The solution to this problem is the orthogonal projection of $\mathbf{y}$ onto this subspace, and the corresponding $\widehat{\boldsymbol{\theta}}$ can be shown (see Section 3.A) to fulfill

$$\mathbf{X}^\mathsf{T}\mathbf{X}\widehat{\boldsymbol{\theta}} = \mathbf{X}^\mathsf{T}\mathbf{y}. \tag{3.13}$$

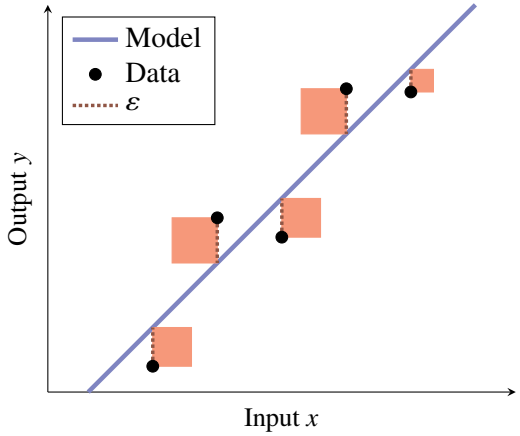



**Figure 3.2:** A graphical explanation of the squared error loss function: the goal is to choose the model (blue line) such that the sum of the squares (light red) of each error $\varepsilon$ is minimised. That is, the blue line is to be chosen so that the amount of red colour is minimised. This motivates the name *least squares*. The black dots, the training data, are fixed.

Equation (3.13) is often referred to as the *normal equations* and gives the solution to the least squares problem (3.12). If $\mathbf{X}^\mathsf{T}\mathbf{X}$ is invertible, which is often the case, then $\widehat{\boldsymbol{\theta}}$ has the closed form expression

$$\widehat{\boldsymbol{\theta}} = (\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}\mathbf{X}^\mathsf{T}\mathbf{y}. \tag{3.14}$$

The fact that this closed-form solution exists is important and is probably the reason for why the linear regression with squared error loss is so extremely common in practice. Other loss functions lead to optimisation problems that often lack closed-form solutions.

We now have everything in place for using linear regression, and we summarise it as Method 3.1 and illustrate it by Example 3.2.

**Learn** linear regression with squared error loss

**Data:** Training data $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^n$
**Result:** Learned parameter vector $\widehat{\boldsymbol{\theta}}$

1 Construct the matrix $\mathbf{X}$ and vector $\mathbf{y}$ according to (3.5).
2 Compute $\widehat{\boldsymbol{\theta}}$ by solving (3.13).

**Predict** with linear regression

**Data:** Learned parameter vector $\widehat{\boldsymbol{\theta}}$ and test input $\mathbf{x}_\star$
**Result:** Prediction $\widehat{y}(\mathbf{x}_\star)$

1 Compute $\widehat{y}(\mathbf{x}_\star) = \widehat{\boldsymbol{\theta}}^\mathsf{T}\mathbf{x}_\star$.

**Method 3.1:** Linear regression

***Time to reflect 3.1*** *What does it mean in practice if* $\mathbf{X}^\mathsf{T}\mathbf{X}$ *is not invertible?*

***Time to reflect 3.2*** *If the columns of* $\mathbf{X}$ *are linearly independent, and* $p = n - 1$, $\mathbf{X}$ *spans the entire* $\mathbb{R}^n$. *If that is the case,* $\mathbf{X}$ *is invertible, and* (3.14) *reduces to* $\boldsymbol{\theta} = \mathbf{X}^{-1}\mathbf{y}$. *Hence, a unique solution exists such that* $\mathbf{y} = \mathbf{X}\boldsymbol{\theta}$ *exactly, that is, the model fits the training data perfectly. Why would that not be a desired property in practice?*

**Example 3.2 Car stopping distances**

By inserting the matrices (3.6) from Example 3.1 into the normal equations (3.14), we obtain $\widehat{\theta}_0 = -20.1$ and $\widehat{\theta}_1 = 3.14$. Plotting the resulting model gives us Figure 3.3.

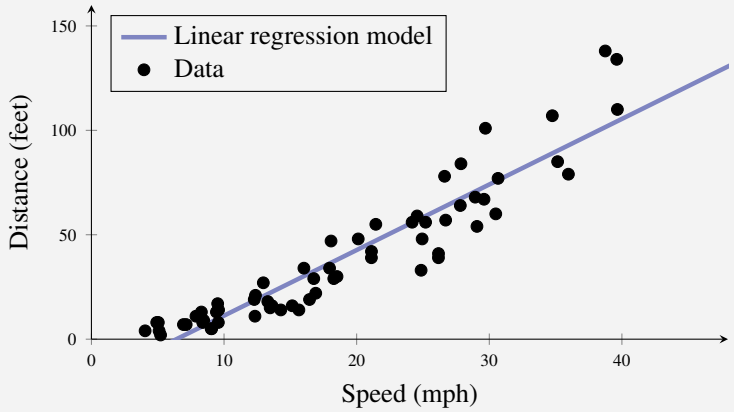



**Fig. 3.3**

This can be compared to how $k$-NN and decision trees solves the same problem (Figures 2.5 and 2.11). Clearly, the linear regression model behaves differently than these models; linear regression does not share the 'local' nature of $k$-NN and decision trees (only training data points close to $\mathbf{x}_\star$ affect $\widehat{y}(\mathbf{x}_\star)$), which is related to the fact that linear regression is a parametric model.

### The Maximum Likelihood Perspective

To get another perspective on the squared error loss, we will now reinterpret the least squares method above as a *maximum likelihood* solution. The word 'likelihood' refers to the statistical concept of the likelihood function, and maximising the likelihood function amounts to finding the value of $\boldsymbol{\theta}$ that makes observing $\mathbf{y}$ as likely as possible. That is, instead of (somewhat arbitrarily) selecting a loss function, we start with the

problem

$$\widehat{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta}} p(\mathbf{y} \mid \mathbf{X}; \boldsymbol{\theta}). \tag{3.15}$$

Here $p(\mathbf{y} \mid \mathbf{X}; \boldsymbol{\theta})$ is the probability density of all observed outputs $\mathbf{y}$ in the training data, given all inputs $\mathbf{X}$ and parameters $\boldsymbol{\theta}$. This determines mathematically what 'likely' means, but we need to specify it in more detail. We do that by considering the noise term $\varepsilon$ as a stochastic variable with a certain distribution. A common assumption is that the noise terms are independent, each with a Gaussian distribution (also known as a normal distribution) with mean zero and variance $\sigma_\varepsilon^2$,

$$\varepsilon \sim \mathcal{N}(0, \sigma_\varepsilon^2). \tag{3.16}$$

This implies that the $n$ observed training data points are independent, and $p(\mathbf{y} \mid \mathbf{X}; \boldsymbol{\theta})$ factorises as

$$p(\mathbf{y} \mid \mathbf{X}; \boldsymbol{\theta}) = \prod_{i=1}^{n} p(y_i \mid \mathbf{x}_i, \boldsymbol{\theta}). \tag{3.17}$$

Considering the linear regression model from (3.3), $y = \boldsymbol{\theta}^\mathsf{T}\mathbf{x} + \varepsilon$, together with the Gaussian noise assumption (3.16), we have

$$p(y_i \mid \mathbf{x}_i, \boldsymbol{\theta}) = \mathcal{N}\left(y_i; \boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i, \sigma_\varepsilon^2\right) = \frac{1}{\sqrt{2\pi\sigma_\varepsilon^2}} \exp\left(-\frac{1}{2\sigma_\varepsilon^2}\left(\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i - y_i\right)^2\right). \tag{3.18}$$

Recall that we want to maximise the likelihood with respect to $\boldsymbol{\theta}$. For numerical reasons, it is usually better to work with the logarithm of $p(\mathbf{y} \mid \mathbf{X}; \boldsymbol{\theta})$,

$$\ln p(\mathbf{y} \mid \mathbf{X}; \boldsymbol{\theta}) = \sum_{i=1}^{n} \ln p(y_i \mid \mathbf{x}_i, \boldsymbol{\theta}). \tag{3.19}$$

Since the logarithm is a monotonically increasing function, maximising the log-likelihood (3.19) is equivalent to maximising the likelihood itself. Putting (3.18) and (3.19) together, we get

$$\ln p(\mathbf{y} \mid \mathbf{X}; \boldsymbol{\theta}) = -\frac{n}{2}\ln(2\pi\sigma_\varepsilon^2) - \frac{1}{2\sigma_\varepsilon^2}\sum_{i=1}^{n}\left(\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i - y_i\right)^2. \tag{3.20}$$

Removing terms and factors independent of $\boldsymbol{\theta}$ does not change the maximising argument, and we see that we can rewrite (3.15) as

$$\widehat{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta}} p(\mathbf{y} \mid \mathbf{X}; \boldsymbol{\theta}) = \arg\max_{\boldsymbol{\theta}} -\sum_{i=1}^{n}\left(\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i - y_i\right)^2 = \arg\min_{\boldsymbol{\theta}} \frac{1}{n}\sum_{i=1}^{n}\left(\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i - y_i\right)^2. \tag{3.21}$$

This is indeed linear regression with the least squares cost (the cost function implied by the squared error loss function (3.10)). Hence, using the squared error loss is equivalent to assuming a Gaussian noise distribution in the maximum likelihood formulation. Other assumptions on $\varepsilon$ lead to other loss functions, as we will discuss further in Chapter 5.


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

### Categorical Input Variables

The regression problem is characterised by a numerical output $y$ and inputs $\mathbf{x}$ of arbitrary type. We have, however, only discussed the case of numerical inputs so far. To see how we can handle categorical inputs in the linear regression model, assume that we have an input variable that only takes two different values. We refer to those two values as `A` and `B`. We can then create a *dummy variable* $x$ as

$$x = \begin{cases} 0 & \text{if } \texttt{A}, \\ 1 & \text{if } \texttt{B}, \end{cases} \tag{3.22}$$

and use this variable in any supervised machine learning method as if it was numerical. For linear regression, this effectively gives us a model which looks like

$$y = \theta_0 + \theta_1 x + \varepsilon = \begin{cases} \theta_0 + \varepsilon & \text{if } \texttt{A}, \\ \theta_0 + \theta_1 + \varepsilon & \text{if } \texttt{B}. \end{cases} \tag{3.23}$$

The model is thus able to learn and predict two different values depending on whether the input is `A` or `B`.

If the categorical variable takes more than two values, let us say `A`, `B`, `C`, and `D`, we can make a so-called *one-hot encoding* by constructing a four-dimensional vector

$$\mathbf{x} = \begin{bmatrix} x_A & x_B & x_C & x_D \end{bmatrix}^\mathsf{T} \tag{3.24}$$

where $x_A = 1$ if `A`, $x_B = 1$ if `B`, and so on. That is, only one element of $\mathbf{x}$ will be 1, the rest are 0. Again, this construction can be used for any supervised machine learning method and is not restricted to linear regression.

## 3.2 Classification and Logistic Regression

After presenting a parametric method for solving the regression problem, we now turn our attention to classification. As we will see, with a modification of the linear regression model, we can apply it to the classification problem as well; however, this is the cost of not being able to use the convenient normal equations. Instead, we have to resort to numerical optimisation for learning the parameters of the model.

### A Statistical View of the Classification Problem

Supervised machine learning amounts to predicting the output from the input. From a statistical perspective, classification amounts to predicting the conditional class probabilities

$$p(y = m \mid \mathbf{x}), \tag{3.25}$$

where $y$ is the output $(1, 2, \ldots,$ or $M)$ and $\mathbf{x}$ is the input.[3] In words, $p(y = m \mid \mathbf{x})$ describes *the probability for class m given that we know the input* $\mathbf{x}$. Talking about

[3] We use the notation $p(y \mid \mathbf{x})$ to denote probability masses ($y$ discrete) as well as probability densities ($y$ continuous).

$p(y \mid \mathbf{x})$ implies that we think about the class label $y$ as a random variable. Why? Because we choose to model the real world, from where the data originates, as involving a certain amount of randomness (much like the random error $\varepsilon$ in regression). Let us illustrate this with an example.

**Example 3.3 Describing voting behavior using probabilities**

We want to construct a model that can predict voting preferences (= $y$, the categorical output) for different population groups (= $\mathbf{x}$, the input). However, we then have to face the fact that not everyone in a certain population group will vote for the same political party. We can therefore think of $y$ as a random variable which follows a certain probability distribution. If we know that the vote count in the group of 45 year old women (= $\mathbf{x}$) is 13% for the cerise party, 39% for the turquoise party, and 48% for the purple party (here we have $M = 3$), we could describe it as

$$\begin{aligned} p(y = \text{cerise party} \mid \mathbf{x} = \text{45 year old women}) &= 0.13, \\ p(y = \text{turqoise party} \mid \mathbf{x} = \text{45 year old women}) &= 0.39, \\ p(y = \text{purple party} \mid \mathbf{x} = \text{45 year old women}) &= 0.48. \end{aligned}$$

In this way, the probabilities $p(y \mid \mathbf{x})$ describe the non-trivial fact that

(a) all 45 year old women do not vote for the same party, but

(b) the choice of party does not appear to be completely random among 45 year old women either; the purple party is the most popular, and the cerise party is the least popular.

Thus, it can be useful to have a classifier which predicts not only a class $\widehat{y}$ (one party) but a distribution over classes $p(y \mid \mathbf{x})$.

We now aim to construct a classifier which can not only predict classes but also learn the class probabilities $p(y \mid \mathbf{x})$. More specifically, for binary classification problems ($M = 2$, and $y$ is either 1 or $-1$), we train a model $g(\mathbf{x})$ for which

$$\boxed{p(y = 1 \mid \mathbf{x}) \text{ is modelled by } g(\mathbf{x}).} \tag{3.26a}$$

By the laws of probabilities, it holds that $p(y = 1 \mid \mathbf{x}) + p(y = -1 \mid \mathbf{x}) = 1$, which means that

$$\boxed{p(y = -1 \mid \mathbf{x}) \text{ is modelled by } 1 - g(\mathbf{x}).} \tag{3.26b}$$

Since $g(\mathbf{x})$ is a model for a probability, it is natural to require that $0 \leq g(\mathbf{x}) \leq 1$ for any $\mathbf{x}$. We will see how this constraint can be enforced below.

For the multiclass problem, we instead let the classifier return a vector-valued function $\mathbf{g}(\mathbf{x})$, where

$$\boxed{\begin{bmatrix} p(y = 1 \mid \mathbf{x}) \\ p(y = 2 \mid \mathbf{x}) \\ \vdots \\ p(y = M \mid \mathbf{x}) \end{bmatrix} \text{ is modelled by } \begin{bmatrix} g_1(\mathbf{x}) \\ g_2(\mathbf{x}) \\ \vdots \\ g_M(\mathbf{x}) \end{bmatrix} = \mathbf{g}(\mathbf{x}).} \tag{3.27}$$


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

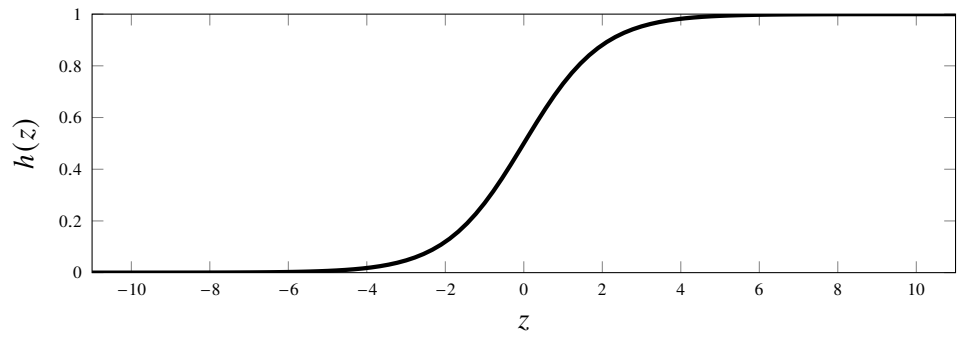



**Figure 3.4:** The logistic function $h(z) = \frac{e^z}{1+e^z}$.

In words, each element $g_m(\mathbf{x})$ of $\mathbf{g}(\mathbf{x})$ corresponds to the conditional class probability $p(y = m \,|\, \mathbf{x})$. Since $\mathbf{g}(\mathbf{x})$ models a probability vector, we require that each element $g_m(\mathbf{x}) \geq 0$ and that $\|\mathbf{g}(\mathbf{x})\|_1 = \sum_{m=1}^{M} |g_m(\mathbf{x})| = 1$ for any $\mathbf{x}$.

### The Logistic Regression Model for Binary Classification

We will now introduce the logistic regression model, which is one possible way of modelling conditional class probabilities. Logistic regression can be viewed as a modification of the linear regression model so that it fits the classification (instead of the regression) problem.

Let us start with binary classification. We wish to learn a function $g(\mathbf{x})$ that approximates the conditional probability of the positive class, see (3.26). To this end, we start with the linear regression model which, without the noise term, is given by

$$z = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_p x_p = \boldsymbol{\theta}^\mathsf{T} \mathbf{x}. \tag{3.28}$$

This is a mapping which takes $\mathbf{x}$ and returns $z$, which in this context is called the *logit*. Note that $z$ takes values on the entire real line, whereas we need a function which instead returns a value in the interval $[0, 1]$. The key idea of logistic regression is to 'squeeze' $z$ from (3.28) to the interval $[0, 1]$ by using the *logistic function* $h(z) = \frac{e^z}{1+e^z}$, see Figure 3.4. This results in

$$g(\mathbf{x}) = \frac{e^{\boldsymbol{\theta}^\mathsf{T}\mathbf{x}}}{1 + e^{\boldsymbol{\theta}^\mathsf{T}\mathbf{x}}}, \tag{3.29a}$$

which is restricted to $[0, 1]$ and hence can be interpreted as a probability. The function (3.29a) is the logistic regression model for $p(y = 1 \,|\, \mathbf{x})$. Note that this implicitly also gives a model for $p(y = -1 \,|\, \mathbf{x})$,

$$1 - g(\mathbf{x}) = 1 - \frac{e^{\boldsymbol{\theta}^\mathsf{T}\mathbf{x}}}{1 + e^{\boldsymbol{\theta}^\mathsf{T}\mathbf{x}}} = \frac{1}{1 + e^{\boldsymbol{\theta}^\mathsf{T}\mathbf{x}}} = \frac{e^{-\boldsymbol{\theta}^\mathsf{T}\mathbf{x}}}{1 + e^{-\boldsymbol{\theta}^\mathsf{T}\mathbf{x}}}. \tag{3.29b}$$

In a nutshell, the logistic regression model is linear regression appended with the logistic function. This is the reason for the (somewhat confusing) name, but despite

the name, logistic regression is a method for classification, not regression! The reason why there is no noise term $\varepsilon$ in (3.28), as we had in the linear regression model (3.3), is that the randomness in classification is statistically modelled by the class probability construction $p(y = m \mid \mathbf{x})$ instead of an additive noise $\varepsilon$.

As for linear regression, we have a model (3.29) which contains unknown parameters $\boldsymbol{\theta}$. Logistic regression is thereby also a parametric model, and we need to learn the parameters from training data. How this can be done is the topic for the next section.

### Training the Logistic Regression Model by Maximum Likelihood

By using the logistic function, we have transformed linear regression (a model for regression problems) into logistic regression (a model for classification problems). The price to pay is that we will not be able to use the convenient normal equations for learning $\boldsymbol{\theta}$ (as we could for linear regression if we used the squared error loss), due to the nonlinearity of the logistic function.

In order to derive a principled way of learning $\boldsymbol{\theta}$ in (3.29) from training data $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^n$, we start with the maximum likelihood approach. From a maximum likelihood perspective, learning a classifier amounts to solving

$$\widehat{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta}} p(\mathbf{y} \mid \mathbf{X}; \boldsymbol{\theta}) = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{n} \ln p(y_i \mid \mathbf{x}_i; \boldsymbol{\theta}), \tag{3.30}$$

where similarly to linear regression (3.19), we assume that the training data points are independent, and we consider the logarithm of the likelihood function for numerical reasons. We have also added $\boldsymbol{\theta}$ explicitly to the notation to emphasise the dependence on the model parameters. Remember that our model of $p(y = 1 \mid \mathbf{x}; \boldsymbol{\theta})$ is $g(\mathbf{x}; \boldsymbol{\theta})$, which gives

$$\ln p(y_i \mid \mathbf{x}_i; \boldsymbol{\theta}) = \begin{cases} \ln g(\mathbf{x}_i; \boldsymbol{\theta}) & \text{if } y_i = 1, \\ \ln\big(1 - g(\mathbf{x}_i; \boldsymbol{\theta})\big) & \text{if } y_i = -1. \end{cases} \tag{3.31}$$

It is common to turn the maximisation problem (3.30) into an equivalent minimisation problem by using the negative log-likelihood as cost function, $J(\boldsymbol{\theta}) = -\frac{1}{n}\sum \ln p(y_i \mid \mathbf{x}_i; \boldsymbol{\theta})$, that is,

$$J(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} \underbrace{\begin{cases} -\ln g(\mathbf{x}_i; \boldsymbol{\theta}) & \text{if } y_i = 1, \\ -\ln\big(1 - g(\mathbf{x}_i; \boldsymbol{\theta})\big) & \text{if } y_i = -1. \end{cases}}_{\text{Binary cross-entropy loss } L(g(\mathbf{x}_i;\boldsymbol{\theta}), y_i)} \tag{3.32}$$

The loss function in the expression above is called the *cross-entropy loss*. It is not specific to logistic regression but can be used for any binary classifier that predicts class probabilities $g(\mathbf{x}; \boldsymbol{\theta})$.

However, we will now consider specifically the logistic regression model, for which we can write out the cost function (3.32) in more detail. In doing so, the


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

particular choice of labelling $\{-1, 1\}$ turns out to be convenient. For $y_i = 1$, we can write

$$g(\mathbf{x}_i; \boldsymbol{\theta}) = \frac{e^{\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i}}{1 + e^{\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i}} = \frac{e^{y_i\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i}}{1 + e^{y_i\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i}}, \tag{3.33a}$$

and for $y_i = -1$,

$$1 - g(\mathbf{x}_i; \boldsymbol{\theta}) = \frac{e^{-\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i}}{1 + e^{-\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i}} = \frac{e^{y_i\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i}}{1 + e^{y_i\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i}}. \tag{3.33b}$$

Hence, we get the same expression in both cases and can write (3.32) compactly as

$$\begin{aligned} J(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} -\ln\frac{e^{y_i\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i}}{1 + e^{y_i\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i}} &= \frac{1}{n}\sum_{i=1}^{n} -\ln\frac{1}{1 + e^{-y_i\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i}} \\ &= \frac{1}{n}\sum_{i=1}^{n} \underbrace{\ln\left(1 + e^{-y_i\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i}\right)}_{\text{Logistic loss } L(\mathbf{x}_i, y_i, \boldsymbol{\theta})}. \end{aligned} \tag{3.34}$$

The loss function $L(\mathbf{x}, y_i, \boldsymbol{\theta})$ above, which is a special case of the cross-entropy loss, is called the *logistic loss* (or sometimes binomial deviance). Learning a logistic regression model thus amounts to solving

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \frac{1}{n}\sum_{i=1}^{n} \ln\left(1 + e^{-y_i\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i}\right). \tag{3.35}$$

Contrary to linear regression with squared error loss, the problem (3.35) has no closed-form solution, so we have to use numerical optimisation instead. Solving nonlinear optimisation problems numerically is central to the training of many machine learning models, not just logistic regression, and we will come back to this topic in Chapter 5. For now, however, it is enough to note that there exist efficient algorithms for solving (3.35) numerically to find $\widehat{\boldsymbol{\theta}}$.

### Predictions and Decision Boundaries

So far, we have discussed logistic regression as a method for predicting class probabilities for a test input $\mathbf{x}_\star$ by first learning $\boldsymbol{\theta}$ from training data and thereafter computing $g(\mathbf{x}_\star)$, our model for $p(y = 1 \mid \mathbf{x}_\star)$. However, sometimes we want to make a 'hard' prediction for the test input $\mathbf{x}_\star$, that is, predicting $\widehat{y}(\mathbf{x}_\star) = -1$ or $\widehat{y}(\mathbf{x}_\star) = 1$ in binary classification, just like with $k$-NN or decision trees. We then have to add a final step to the logistic regression model, in which the predicted probabilities are turned into a class prediction. The most common approach is to *let* $\widehat{y}(\mathbf{x}_\star)$ *be the most probable class*. For binary classification, we can express this as[4]

$$\widehat{y}(\mathbf{x}_\star) = \begin{cases} 1 & \text{if } g(\mathbf{x}) > r \\ -1 & \text{if } g(\mathbf{x}) \leq r \end{cases}, \tag{3.36}$$

[4]It is arbitrary what happens if $g(\mathbf{x}) = 0.5$.

**Learn** binary logistic regression

**Data:** Training data $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^n$ (with output classes $y = \{-1, 1\}$)
**Result:** Learned parameter vector $\widehat{\boldsymbol{\theta}}$

1 Compute $\widehat{\boldsymbol{\theta}}$ by solving (3.35) numerically.

**Predict** with binary logistic regression

**Data:** Learned parameter vector $\widehat{\boldsymbol{\theta}}$ and test input $\mathbf{x}_\star$
**Result:** Prediction $\widehat{y}(\mathbf{x}_\star)$

1 Compute $g(\mathbf{x}_\star)$ (3.29a).
2 If $g(\mathbf{x}_\star) > 0.5$, return $\widehat{y}(\mathbf{x}_\star) = 1$, otherwise return $\widehat{y}(\mathbf{x}_\star) = -1$.

**Method 3.2:** Logistic regression

with decision threshold $r = 0.5$, which is illustrated in Figure 3.5. We now have everything in place for summarising binary logistic regression in Method 3.2.

In some applications, however, it can be beneficial to explore different thresholds than $r = 0.5$. It can be shown that *if* $g(\mathbf{x}) = p(y = 1 \mid \mathbf{x})$, that is, the model provides a correct description of the real-world class probabilities, then the choice $r = 0.5$ will give the smallest possible number of misclassifications on average. In other words, $r = 0.5$ minimises the so-called *misclassification rate*. The misclassification rate is, however, not always the most important aspect of a classifier. Many classification problems are asymmetric (it is more important to correctly predict some classes than others) or imbalanced (the classes occur with very different frequencies). In a medical diagnosis application, for example, it can be more important not to falsely predict the negative class (that is, by mistake predict a sick patient being healthy) than to falsely predict the positive class (by mistake predict a healthy patient as sick). For such a problem, minimising the misclassification rate might not lead to the desired performance. Furthermore, the medical diagnosis problem could be imbalanced if the disorder is very rare, meaning that the vast majority of the data points (patients) belong to the negative class. By only considering the misclassification rate in such a situation, we implicitly value accurate predictions of the negative class higher than accurate predictions of the positive class, simply because the negative class is more common in the data. We will discuss how we can evaluate such situations more systematically in Section 4.5. In the end, however, the decision threshold $r$ is a choice that the user has to make.

The decision boundary for binary classification can be computed by solving the equation

$$g(\mathbf{x}) = 1 - g(\mathbf{x}). \tag{3.37}$$

The solutions to this equation are points in the input space for which the two classes are predicted to be equally probable. Therefore, these points lie on the decision boundary. For binary logistic regression, this means


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

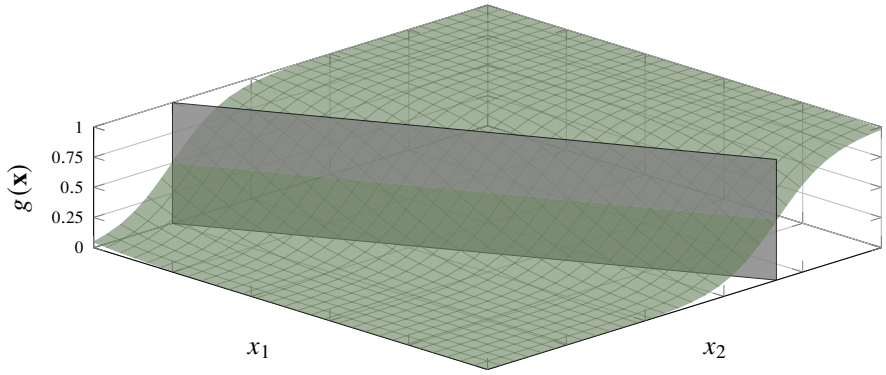



**Figure 3.5:** In binary classification ($y = -1$ or 1), logistic regression predicts $g(\mathbf{x}_\star)$ ($\mathbf{x}$ is two-dimensional here), which is an attempt to determine $p(y = 1 \mid \mathbf{x}_\star)$. This implicitly also gives a prediction for $p(y = -1 \mid \mathbf{x}_\star)$ as $1 - g(\mathbf{x}_\star)$. To turn these probabilities into actual class predictions ($\widehat{y}(\mathbf{x}_\star)$ is either $-1$ or 1), the class which is modelled to have the highest probability can be taken as the prediction, as in Equation (3.36). The point(s) where the prediction changes from from one class to another is the decision boundary (grey plane).

$$\frac{e^{\boldsymbol{\theta}^\mathsf{T}\mathbf{x}}}{1 + e^{\boldsymbol{\theta}^\mathsf{T}\mathbf{x}}} = \frac{1}{1 + e^{\boldsymbol{\theta}^\mathsf{T}\mathbf{x}}} \Leftrightarrow e^{\boldsymbol{\theta}^\mathsf{T}\mathbf{x}} = 1 \Leftrightarrow \boldsymbol{\theta}^\mathsf{T}\mathbf{x} = 0. \tag{3.38}$$

The equation $\boldsymbol{\theta}^\mathsf{T}\mathbf{x} = 0$ parameterises a (linear) hyperplane. Hence, the decision boundaries in logistic regression always have the shape of a (linear) hyperplane.

From the derivation above, it can be noted that the sign of the expression $\boldsymbol{\theta}^\mathsf{T}\mathbf{x}$ determines if we are predicting the positive or the negative class. Hence, we can compactly write (3.36), with the threshold $r = 0.5$, as

$$\widehat{y}(\mathbf{x}_\star) = \operatorname{sign}(\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_\star). \tag{3.39}$$

In general we distinguish between different types of classifiers by the shape of their decision boundaries.

A classifier whose decision boundaries are linear hyperplanes is a *linear classifier.*

All other classifiers are *non-linear classifiers*. Logistic regression is an example of a linear classifier, whereas $k$-NN and decision trees are non-linear classifiers. Note that the term 'linear' is used in a different sense for linear regression: linear regression is a model which is linear in its parameters, whereas a linear classifier is a model whose decision boundaries are linear.

The same arguments and constructions as above can be generalised to the multiclass setting. Predicting according to the most probable class then amounts to computing the prediction as

$$\widehat{y}(\mathbf{x}_\star) = \arg\max_m g_m(\mathbf{x}_\star). \tag{3.40}$$

As in the binary case, it is possible to modify this when working with an asymmetric or imbalanced problem. The decision boundaries will be given by a combination of $M - 1$ (linear) hyperplanes for a multiclass logistic regression model.

### Logistic Regression for More Than Two Classes

Logistic regression can be used also for the multiclass problem when there are more than two classes, $M > 2$. There are several ways of generalising logistic regression to this setting. We will follow one path using the so-called softmax function, which will also be useful later when introducing deep learning models in Chapter 6.

For the binary problem, we used the logistic function to design a model for $g(\mathbf{x})$, a scalar-valued function representing $p(y = 1 \mid \mathbf{x})$. For the multiclass problem, we instead have to design a vector-valued function $\mathbf{g}(\mathbf{x})$, whose elements should be non-negative and sum to one. For this purpose, we first use $M$ instances of (3.28), each denoted $z_m$ and each with a different set of parameters $\boldsymbol{\theta}_m$, $z_m = \boldsymbol{\theta}_m^\mathsf{T}\mathbf{x}$. We stack all $z_m$ into a vector of logits $\mathbf{z} = [z_1 \;\; z_2 \;\; \ldots \;\; z_M]^\mathsf{T}$ and use the *softmax function* as a vector-valued generalisation of the logistic function,

$$\text{softmax}(\mathbf{z}) \triangleq \frac{1}{\sum_{m=1}^{M} e^{z_m}} \begin{bmatrix} e^{z_1} \\ e^{z_2} \\ \vdots \\ e^{z_M} \end{bmatrix}. \tag{3.41}$$

Note that the argument $\mathbf{z}$ to the softmax function is an $M$-dimensional vector and that it also returns a vector of the same dimension. By construction, the output vector from the softmax function always sums to 1, and each element is always $\geq 0$. Similarly to how we combined linear regression and the logistic function for the binary classification problem (3.29), we have now combined linear regression and the softmax function to model the class probabilities:

$$\mathbf{g}(\mathbf{x}) = \text{softmax}(\mathbf{z}), \qquad \text{where } \mathbf{z} = \begin{bmatrix} \boldsymbol{\theta}_1^\mathsf{T}\mathbf{x} \\ \boldsymbol{\theta}_2^\mathsf{T}\mathbf{x} \\ \vdots \\ \boldsymbol{\theta}_M^\mathsf{T}\mathbf{x} \end{bmatrix}. \tag{3.42}$$

Equivalently, we can write out the individual class probabilities, that is, the elements of the vector $\mathbf{g}(\mathbf{x})$, as

$$g_m(\mathbf{x}) = \frac{e^{\boldsymbol{\theta}_m^\mathsf{T}\mathbf{x}}}{\sum_{j=1}^{M} e^{\boldsymbol{\theta}_j^\mathsf{T}\mathbf{x}}}, \qquad m = 1, \ldots, M. \tag{3.43}$$

This is the multiclass logistic regression model. Note that this construction uses $M$ parameter vectors $\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_M$ (one for each class), meaning that the number of parameters to learn grows with $M$. As for binary logistic regression, we can learn these parameters using the maximum likelihood method. We use $\boldsymbol{\theta}$ to denote *all* model


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

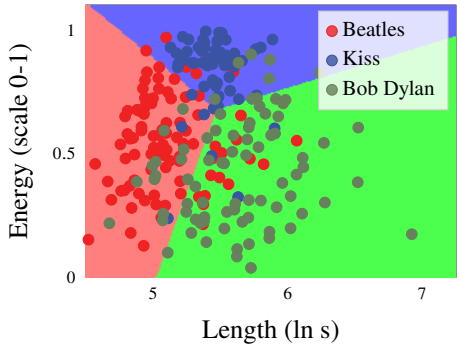



**Figure 3.6:** Logistic regression applied to the music classification problem from Example 2.1. The decision boundaries are linear, but unlike trees (Figure 2.11a), they are not perpendicular to the axes.

parameters, $\boldsymbol{\theta} = \{\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_M\}$. Since $g_m(\mathbf{x}_i; \boldsymbol{\theta})$ is our model for $p(y_i = m \mid \mathbf{x}_i)$, the cost function for the cross-entropy (equivalently, negative log-likelihood) loss for the multiclass problem is

$$J(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} \underbrace{-\ln g_{y_i}(\mathbf{x}_i; \boldsymbol{\theta})}_{\substack{\text{Multiclass cross-entropy} \\ \text{loss } L(\mathbf{g}(\mathbf{x}_i;\boldsymbol{\theta}), y_i)}} . \tag{3.44}$$

Note that we use the training data labels $y_i$ as index variables to select the correct conditional probability for the loss function. That is, the $i$th term of the sum is the negative logarithm of the $y_i$th element of the vector $\mathbf{g}(\mathbf{x}_i; \boldsymbol{\theta})$. We illustrate the meaning of this in Example 3.4.

Inserting the model (3.43) into the loss function (3.44) gives the cost function to optimise when learning multiclass logistic regression:

$$J(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} \left( -\boldsymbol{\theta}_{y_i}^{\mathsf{T}} \mathbf{x}_i + \ln \sum_{j=1}^{M} e^{\boldsymbol{\theta}_j^{\mathsf{T}} \mathbf{x}_i} \right). \tag{3.45}$$

We apply this to the music classification problem in Figure 3.6.

**Example 3.4 The cross-entropy loss for multiclass problems**

Consider the following (very small) data set with $n = 6$ data points, $p = 2$ input dimensions, and $M = 3$ classes, which we want to use to train a multiclass classifier:

$$\mathbf{X} = \begin{bmatrix} 0.20 & 0.86 \\ 0.41 & 0.18 \\ 0.96 & -1.84 \\ -0.25 & 1.57 \\ -0.82 & -1.53 \\ -0.31 & 0.58 \end{bmatrix}, \qquad \mathbf{y} = \begin{bmatrix} 2 \\ 3 \\ 1 \\ 2 \\ 1 \\ 3 \end{bmatrix}.$$

Multiclass logistic regression with softmax (or any other multiclass classifier which predicts a vector of conditional class probabilities) return a three-dimensional probability vector $\mathbf{g}(\mathbf{x};\boldsymbol{\theta})$ for any $\mathbf{x}$ and $\boldsymbol{\theta}$. If we stack the logarithms of the transpose of all vectors $\mathbf{g}(\mathbf{x}_i;\boldsymbol{\theta})$ for $i = 1,\ldots,6$, we obtain the matrix

$$\mathbf{G} = \begin{bmatrix} \ln g_1(\mathbf{x}_1;\boldsymbol{\theta}) & \ln g_2(\mathbf{x}_1;\boldsymbol{\theta}) & \ln g_3(\mathbf{x}_1;\boldsymbol{\theta}) \\ \ln g_1(\mathbf{x}_2;\boldsymbol{\theta}) & \ln g_2(\mathbf{x}_2;\boldsymbol{\theta}) & \ln g_3(\mathbf{x}_2;\boldsymbol{\theta}) \\ \ln g_1(\mathbf{x}_3;\boldsymbol{\theta}) & \ln g_2(\mathbf{x}_3;\boldsymbol{\theta}) & \ln g_3(\mathbf{x}_3;\boldsymbol{\theta}) \\ \ln g_1(\mathbf{x}_4;\boldsymbol{\theta}) & \ln g_2(\mathbf{x}_4;\boldsymbol{\theta}) & \ln g_3(\mathbf{x}_4;\boldsymbol{\theta}) \\ \ln g_1(\mathbf{x}_5;\boldsymbol{\theta}) & \ln g_2(\mathbf{x}_5;\boldsymbol{\theta}) & \ln g_3(\mathbf{x}_5;\boldsymbol{\theta}) \\ \ln g_1(\mathbf{x}_6;\boldsymbol{\theta}) & \ln g_2(\mathbf{x}_6;\boldsymbol{\theta}) & \ln g_3(\mathbf{x}_6;\boldsymbol{\theta}) \end{bmatrix}.$$

Computing the multi-class cross-entropy cost (3.44) now simply amounts to taking the average of all circled elements and multiplying that by $-1$. The element that we have circled in row $i$ is given by the training label $y_i$. Training the model amounts to finding $\boldsymbol{\theta}$ such that this negated average is minimised.

***Time to reflect 3.3*** *Can you derive* (3.32) *as a special case of* (3.44)*?*
*Hint: think of the binary classifier as a special case of the multiclass classifier with* $\mathbf{g}(\mathbf{x}) = \begin{bmatrix} g(\mathbf{x}) \\ 1 - g(\mathbf{x}) \end{bmatrix}$.

***Time to reflect 3.4*** *The softmax-based logistic regression is actually over-parameterised, in the sense that we can construct an equivalent model with fewer parameters. That is often not a problem in practice, but compare the multiclass model* (3.42) *for the case* $M = 2$ *with binary logistic regression* (3.29)*, and see if you can spot the over-parametrisation!*

## 3.3 Polynomial Regression and Regularisation

In comparison to $k$-NN and decision trees studied in Chapter 2, linear and logistic regression might appear to be rigid and non-flexible models with their straight lines (such as Figures 3.1 and 3.5). However, both models are able to adapt to the training data well if the input dimension $p$ is large relative to the number of data points $n$.

A common way of increasing the input dimension in linear and logistic regression, which we will discuss more thoroughly in Chapter 9, is to make a non-linear

transformation of the input. A simple non-linear transformation is to replace a one-dimensional input $x$ with itself raised to different powers, which makes the linear regression model a polynomial:

$$y = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \cdots + \varepsilon. \tag{3.46}$$

This is called *polynomial regression*. The same polynomial expansion can also be applied to the expression for the logit in logistic regression. Note that if we let $x_1 = x$, $x_2 = x^2$ and $x_3 = x^3$, this is still a linear model (3.2) with input $\mathbf{x} = [1 \;\; x \;\; x^2 \;\; x^3]$, but we have 'lifted' the input from being one-dimensional ($p = 1$) to three-dimensional ($p = 3$). Using non-linear input transformations can be very useful in practice, but it effectively increases $p$, and we can easily end up with *overfitting* the model to the noise – rather than the interesting patterns – in the training data, as in the example below.

**Example 3.5 Car stopping distances with polynomial regression**

We return to Example 2.2, but this time we also add the squared speed as an input and thereby use a second-order polynomial in linear regression. This gives the new matrices (compared to Example 3.1)

$$\mathbf{X} = \begin{bmatrix} 1 & 4.0 & 16.0 \\ 1 & 4.9 & 24.0 \\ 1 & 5.0 & 25.0 \\ \vdots & \vdots & \vdots \\ 1 & 39.6 & 1568.2 \\ 1 & 39.7 & 1576.1 \end{bmatrix}, \qquad \boldsymbol{\theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \end{bmatrix}, \qquad \mathbf{y} = \begin{bmatrix} 4.0 \\ 8.0 \\ 8.0 \\ \vdots \\ 134.0 \\ 110.0 \end{bmatrix}, \tag{3.47}$$

and when we insert these into the normal equations (3.13), the new parameter estimates are $\widehat{\theta}_0 = 1.58$, $\widehat{\theta}_1 = 0.42$, and $\widehat{\theta}_2 = 0.07$. (Note that also $\widehat{\theta}_0$ and $\widehat{\theta}_1$ change, compared to Example 3.2.)

In a completely analogous way, we also learn a 10th order polynomial, and we illustrate them all in Figure 3.7.

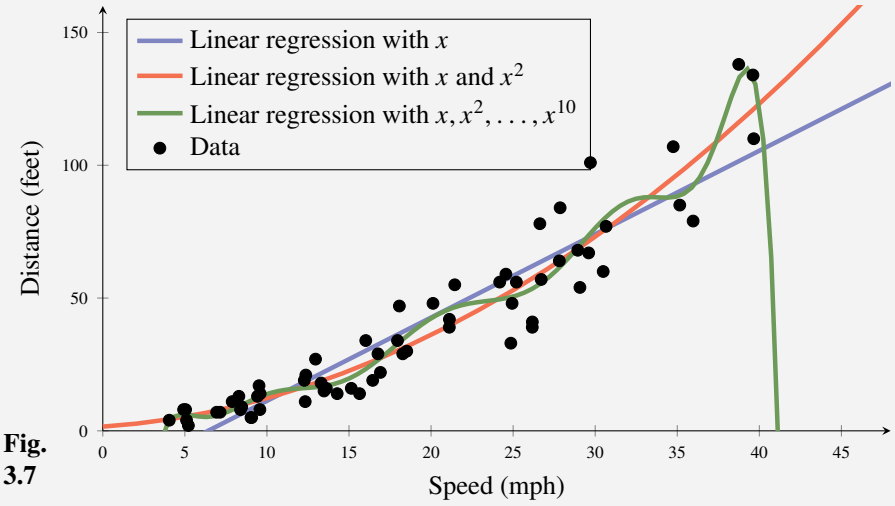



**Fig. 3.7**

The second-order polynomial (red line) appears sensible, and using a second-order polynomial seems to give some advantage compared to plain linear regression (blue line, from Example 3.2). However, using a tenth-order polynomial (green line) seems to make the model less useful than even plain linear regression due to overfitting. In conclusion, there is merit to the idea of polynomial regression, but it has to be applied carefully.

One way to avoid issues with overfitting when augmenting the input with non-linear transformation is to carefully select which inputs (transformations) to include. There are various strategies for doing this, for instance by adding inputs one at a time (forward selection) or by starting with a large number of inputs and then gradually removing the ones that are considered redundant (backward elimination). Different candidate models can be evaluated and compared using cross-validation, as we discuss in Chapter 4. See also Chapter 13, where we discuss input selection further.

An alternative approach to explicit input selection which can also be used to mitigate overfitting is *regularisation*. The idea of regularisation can be described as 'keeping the parameters $\widehat{\boldsymbol{\theta}}$ small unless the data really convinces us otherwise', or alternatively 'if a model with small parameter values $\widehat{\boldsymbol{\theta}}$ fits the data almost as well as a model with larger parameter values, the one with small parameter values should be preferred'. There are several ways to implement this idea mathematically, which lead to different regularisation methods. We will give a more complete treatment of this in Section 5.3 and only discuss the so-called $L^2$ regularisation for now. When paired with regularisation, the idea of using non-linear input transformations can be very powerful and enables a whole family of supervised machine learning methods that we will properly introduce and discuss in Chapter 9.

To keep $\widehat{\boldsymbol{\theta}}$ small, an extra penalty term $\lambda\|\boldsymbol{\theta}\|_2^2$ is added to the cost function when using $L^2$ regularisation. Here, $\lambda \geq 0$, referred to as the regularisation parameter, is a hyperparameter chosen by the user which controls the strength of the regularisation effect. The purpose of the penalty term is to prevent overfitting. Whereas the original cost function only rewards the fit to the training data (which encourages overfitting), the regularisation term prevents overly large parameter values at the cost of a slightly worse fit. It is therefore important to choose the regularisation parameter $\lambda$ wisely, to obtain the right amount of regularisation. With $\lambda = 0$, the regularisation has no effect, whereas $\lambda \to \infty$ will force all parameters $\widehat{\boldsymbol{\theta}}$ to 0. A common approach is to use cross-validation (see Chapter 4) to select $\lambda$.

If we add $L^2$ regularisation to the previously studied linear regression model with squared error loss (3.12), the resulting optimisation problem becomes[5]

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \frac{1}{n}\|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|_2^2 + \lambda\|\boldsymbol{\theta}\|_2^2. \tag{3.48}$$

[5] In practice, it can be wise to exclude $\theta_0$, the intercept, from the regularisation.


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

It turns out that, just like the non-regularised problem, (3.48) has a closed-form solution given by a modified version of the normal equations,

$$(\mathbf{X}^\mathsf{T}\mathbf{X} + n\lambda\mathbf{I}_{p+1})\widehat{\boldsymbol{\theta}} = \mathbf{X}^\mathsf{T}\mathbf{y}, \tag{3.49}$$

where $\mathbf{I}_{p+1}$ is the identity matrix of size $(p+1) \times (p+1)$. This particular application of $L^2$ regularisation is referred to as *ridge regression*.

Regularisation is not restricted to linear regression, however. The same $L^2$ penalty can be applied to any method that involves optimising a cost function. For instance, for logistic regression, we get the optimisation problem

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \frac{1}{n}\sum_{i=1}^{n} \ln\left(1 + \exp\left(-y_i\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i\right)\right) + \lambda\|\boldsymbol{\theta}\|_2^2. \tag{3.50}$$

It is common in practice to train logistic regression models using (3.50) instead of (3.29). One reason is to decrease possible issues with overfitting, as discussed above. Another reason is that for the non-regularised cost function, (3.29), the optimal $\widehat{\boldsymbol{\theta}}$ is not finite if the training data is linearly separable (meaning there exists a linear decision boundary which separates the classes perfectly). In practice, it this means that logistic regression training diverges with some datasets unless (3.50) (with $\lambda > 0$) is used instead of (3.29). Finally, the regularisation term implies that there is a unique solution to the optimisation problem, despite the fact that the softmax model is overparameterised as discussed above.

## 3.4 Generalised Linear Models

In this chapter we have introduced two basic parametric models for regression and classification: linear regression and logistic regression, respectively. The latter model was presented as a way of adapting linear regression to the categorical nature of the output $y$ encountered in a classification problem. This was done by passing the linear regression through a non-linear (in this case, logistic) function, allowing us to interpret the output as a class probability.

The same principle can be generalised to adapt the linear regression model to other properties of the output as well, resulting in what are referred to as *generalised linear models*. In the discussion above, we have focused on two specific problems corresponding to two different types of output data: real-valued regression ($y \in \mathbb{R}$) and classification ($y \in \{1, \ldots, M\}$). These are the most common instances of supervised learning problems, and, indeed, they will be central to most of the discussion and methods presented in this book.

However, in various applications, we might encounter data with other properties, not well described by either of the two standard problems. For instance, assume that the output $y$ corresponds to the count of some quantity, such as the number of earthquakes in a certain area during a fixed time interval, the number of persons diagnosed with a specific illness in a certain region, or the number of patents granted to a tech company.

In such cases, $y$ is a natural number taking one of the values 0, 1, 2, ... (formally, $y \in \mathbb{N}$). Such *count data*, despite being numerical in nature, is not well described by a linear regression model of the form (3.2).[6] The reason is that linear regression models are not restricted to discrete values or to being non-negative, even though we know that this is the case for the actual output $y$ that we are trying to model. Neither does this scenario correspond to a classification setting, since $y$ is numerical (that is, the values can be ordered), and there is no fixed upper limit on how large $y$ can be.

To address this issue, we will extend our notion of parametric models to encompass various probability distributions that can be used to model the conditional output distribution $p(y \mid \mathbf{x}; \boldsymbol{\theta})$. The first step is to choose a suitable *form* for the conditional distribution $p(y \mid \mathbf{x}; \boldsymbol{\theta})$. This is part of the model design, guided by the properties of the data. Specifically, we should select a distribution with support corresponding to that of the data (such as the natural numbers). Naturally, we still want to allow the distribution to depend on the input variable $\mathbf{x}$ – modelling the relationship between the input and the output variables is the fundamental task of supervised learning after all! However, this can be accomplished by first computing a linear regression term $z = \boldsymbol{\theta}^\mathsf{T}\mathbf{x}$ and then letting the conditional distribution $p(y \mid \mathbf{x}; \boldsymbol{\theta})$ depend on $z$ in some appropriate way. We illustrate with an example.

**Example 3.6 Poisson regression**

A simple model for count data is to use a Poisson likelihood. The Poisson distribution is supported on the natural numbers (including 0) and has the probability mass function

$$\text{Pois}(y; \lambda) = \frac{\lambda^y e^{-\lambda}}{y!}, \qquad y = 0, 1, 2, \ldots$$

The so-called rate parameter $\lambda$ controls the shape of the distribution and also corresponds to its mean value, $\mathbb{E}[y] = \lambda$. To use this likelihood in a regression model for count data, we can let $\lambda$ depend on the input variable $\mathbf{x}$ and the model parameters $\boldsymbol{\theta}$ through a linear regression $z = \boldsymbol{\theta}^\mathsf{T}\mathbf{x}$. However, the rate parameter $\lambda$ is restricted to being positive. To ensure that this constraint is satisfied, we model $\lambda$ according to

$$\lambda = \exp(\boldsymbol{\theta}^\mathsf{T}\mathbf{x}).$$

The exponential function maps the output from the linear regression component to the positive real line, resulting in a valid rate parameter for any $\mathbf{x}$ and $\boldsymbol{\theta}$. Thus, we get the model

$$p(y \mid \mathbf{x}; \boldsymbol{\theta}) = \text{Pois}\left(y; \exp(\boldsymbol{\theta}^\mathsf{T}\mathbf{x})\right),$$

referred to a *Poisson regression* model.

In the Poisson regression model, we can write the conditional mean of the

[6] Simply assuming that the distribution of the additive noise $\varepsilon$ is discrete is not enough, since the regression function itself $\boldsymbol{\theta}^\mathsf{T}\mathbf{x}$ can take arbitrary real values.


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

output as

$$\mathbb{E}[y \mid \mathbf{x}; \theta] = \phi^{-1}(z),$$

where $z = \boldsymbol{\theta}^\mathsf{T}\mathbf{x}$ and $\phi(\mu) \stackrel{\text{def}}{=} \log(\mu)$. The idea of providing an explicit link between the linear regression term and the conditional mean of the output in this way is what underlies the generic framework of generalised linear models. Specifically, a generalised linear model consists of:

(i) A choice of output distribution $p(y \mid \mathbf{x}; \boldsymbol{\theta})$ from the exponential family of distributions.[7]

(ii) A linear regression term $z = \boldsymbol{\theta}^\mathsf{T}\mathbf{x}$.

(iii) A strictly increasing, so-called *link function* $\phi$, such that $\mathbb{E}[y \mid \mathbf{x}; \theta] = \phi^{-1}(z)$.

By convention, we map the linear regression output through the *inverse* of the link function to obtain the mean of the output. Equivalently, if $\mu$ denotes the mean of $p(y \mid \mathbf{x}; \boldsymbol{\theta})$, we can express the model as $\phi(\mu) = \boldsymbol{\theta}^\mathsf{T}\mathbf{x}$.

Different choices of conditional distributions and link functions result in different models with varying properties. In fact, as hinted at above, we have already seen another example of a generalised linear model, namely the logistic regression model. In binary logistic regression, the output distribution $p(y \mid \mathbf{x}; \boldsymbol{\theta})$ is a Bernoulli distribution,[8] the logit is computed as $z = \boldsymbol{\theta}^\mathsf{T}\mathbf{x}$, and the link function $\phi$ is given by the inverse of the logistic function, $\phi(\mu) = \log(\mu/(1-\mu))$. Other examples include negative binomial regression (a more flexible model for count data than Poisson regression) and exponential regression (for non-negative real-valued outputs). Hence, the generalised linear model framework can be used to model output variables $y$ with many different properties, and it allows us to describe these models in a common language.

Since generalised linear models are defined in terms of the conditional distribution $p(y \mid \mathbf{x}; \boldsymbol{\theta})$, that is, the likelihood, it is natural to adopt the maximum likelihood formulation for training. That is, we train the model by finding the parameter values such that the negative log-likelihood of the training data is minimised:

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \left[ -\frac{1}{n} \sum_{i=1}^{n} \ln p(y_i \mid \mathbf{x}_i; \boldsymbol{\theta}) \right]. \tag{3.51}$$

A regularisation penalty can be added to the cost function, similarly to (3.50). Regularisation is discussed in more detail in Section 5.3.

[7] The exponential family is a class of probability distributions that can be written on a particular exponential form. It includes many of the commonly used probability distributions, such as Gaussian, Bernoulli, Poisson, exponential, gamma, etc.

[8] The generalised linear model interpretation of logistic regression is more straightforward if we encode the classes as $0/1$ instead of $-1/1$, in which case the output is modelled with a Bernoulli distribution with mean $\mathbb{E}[y \mid x; \boldsymbol{\theta}] = p(y = 1 \mid \mathbf{x}; \boldsymbol{\theta}) = g_{\boldsymbol{\theta}}(\mathbf{x})$.

In general, just as for logistic regression, the training objective (3.51) is a nonlinear optimisation problem without a closed-form solution. However, an important aspect of generalised linear models is that efficient numerical optimisation algorithms exist for solving the maximum likelihood problem. Specifically, under certain assumptions on the link function, the problem becomes convex, and Newton's method can be used to compute $\widehat{\boldsymbol{\theta}}$, efficiently. These are concepts that we will return to in Section 5.4 when we discuss numerical optimisation in more detail.

## 3.5 Generating outputs by sampling

So far we have introduced the probability distribution $p(y \mid \mathbf{x})$ with the motivation of being able to use the maximum likelihood framework. There are actually more reasons to consider $p(y \mid \mathbf{x})$.

So far we have had a *discriminative* perspective on the supervised learning problem: we are given an input $\mathbf{x}$, and we want to predict the corresponding output $y$ as accurately as possible. In some applications, however, we are less interested in a single best prediction and more in the variety of plausible outputs that could be associated with a given input. For example, a weather forecast is often more useful if it reflects not only the single most likely weather, but a family of likely scenarios. Also when using machine learning models in simulator environments, it might be needed to simulate different plausible outcomes rather than just predicting a single outcome. In such situations we are rather interested in *generating* new outputs $y$ given $\mathbf{x}$. We will refer to a model that can do so as a[9] *conditional generative model*.

The good news is that we already have encountered a few conditional generative models. In fact, linear and logistic regression, as well as generalized linear models, are models for the conditional distribution $p(y \mid \mathbf{x})$. While we so far have discussed how to use them in a discriminative way by reporting the or most likely output, we can also use them generatively by

| sample a new output $y$ from the predicted conditional distribution $p(y \mid \mathbf{x})$, |
|---|

instead of returning the most likely value. For the linear regression model this concretely amounts to drawing samples from $\mathcal{N}(\widehat{\boldsymbol{\theta}}^{\mathsf{T}}\mathbf{x}_\star, \sigma_\varepsilon^2)$ instead of just reporting the mean $\widehat{\boldsymbol{\theta}}^{\mathsf{T}}\mathbf{x}_\star$. Different draws produce different outputs, with a spread that is governed by the noise variance that can either be assumed known or learned from data. Using the multiclass logistic regression as a generative model amounts to sample random classes based on the predicted softmax probabilities (3.42), instead of returning the most probable class.

Several methods that will be introduced in later chapters will also be conditonal generative models, in particular deep neural networks (Chapter 6) and Gaussian

[9] We use the term *conditional* to emphasize that the generation is conditioned on a provided input $\mathbf{x}$. Models that are capable of also generating $\mathbf{x}$, which requires modelling either the joint distribution $p(\mathbf{x}, y)$ or the marginal $p(\mathbf{x})$, will be discussed in Chapter 11.

processes Chapter 10 .

## 3.6 Further Reading

Compared to the thousand-year-old $k$-NN idea (Chapter 2), the linear regression model with least squares cost is much younger and can 'only' be traced back a little over two hundred years. It was introduced by Adrien-Marie Legendre in his 1805 book *Nouvelles méthodes pour la détermination des orbites des cométes (New methods for the determination of the orbits of comets)* as well as Carl Friedrich Gauss in his 1809 book *Theoria Motus Corporum Coelestium in Sectionibus Conicis Solem Ambientium* (*Theory of the motion of the heavenly bodies moving about the sun in conic sections*; in that book he claims to have been using it since 1795). Gauss also made the interpretation of it as the maximum likelihood solution when the noise was assumed to have a Gaussian distribution (hence the name of the distribution), although the general maximum likelihood approach was introduced much later by the work of Ronald Fisher (Fisher 1922). The history of logistic regression is almost as old as linear regression and is described further by Cramer (2003). An in-depth account of generalised linear models is given in the classical textbook by McCullagh and Nelder (2018).

## 3.A Derivation of the Normal Equations

The normal equations (3.13)

$$\mathbf{X}^\mathsf{T}\mathbf{X}\widehat{\boldsymbol{\theta}} = \mathbf{X}^\mathsf{T}\mathbf{y}$$

can be derived from (3.12) (the scaling $\frac{1}{n}$ does not affect the minimising argument),

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \ \|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|_2^2,$$

in different ways. We will present one based on (matrix) calculus and one based on geometry and linear algebra.

No matter how (3.13) is derived, if $\mathbf{X}^\mathsf{T}\mathbf{X}$ is invertible, it (uniquely) gives

$$\widehat{\boldsymbol{\theta}} = (\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}\mathbf{X}^\mathsf{T}\mathbf{y}.$$

If $\mathbf{X}^\mathsf{T}\mathbf{X}$ is not invertible, then (3.13) has no or infinitely many solutions $\widehat{\boldsymbol{\theta}}$, which are all equally good solutions to the problem (3.12).

### A Calculus Approach

Let

$$V(\boldsymbol{\theta}) = \|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|_2^2 = (\mathbf{X}\boldsymbol{\theta} - \mathbf{y})^\mathsf{T}(\mathbf{X}\boldsymbol{\theta} - \mathbf{y}) = \mathbf{y}^\mathsf{T}\mathbf{y} - 2\mathbf{y}^\mathsf{T}\mathbf{X}\boldsymbol{\theta} + \boldsymbol{\theta}^\mathsf{T}\mathbf{X}^\mathsf{T}\mathbf{X}\boldsymbol{\theta}, \qquad (3.52)$$

and differentiate $V(\boldsymbol{\theta})$ with respect to the vector $\boldsymbol{\theta}$:

$$\frac{\partial}{\partial \boldsymbol{\theta}} V(\boldsymbol{\theta}) = -2\mathbf{X}^\mathsf{T}\mathbf{y} + 2\mathbf{X}^\mathsf{T}\mathbf{X}\boldsymbol{\theta}. \tag{3.53}$$

Since $V(\boldsymbol{\theta})$ is a positive quadratic form, its minimum must be attained at $\frac{\partial}{\partial \boldsymbol{\theta}} V(\boldsymbol{\theta}) = 0$, which characterises the solution $\widehat{\boldsymbol{\theta}}$ as

$$\frac{\partial}{\partial \boldsymbol{\theta}} V(\widehat{\boldsymbol{\theta}}) = 0 \Leftrightarrow -2\mathbf{X}^\mathsf{T}\mathbf{y} + 2\mathbf{X}^\mathsf{T}\mathbf{X}\boldsymbol{\theta} = 0 \Leftrightarrow \mathbf{X}^\mathsf{T}\mathbf{X}\widehat{\boldsymbol{\theta}} = \mathbf{X}^\mathsf{T}\mathbf{y}, \tag{3.54}$$

which are the normal equations.

### A Linear Algebra Approach

Denote the $p+1$ columns of $\mathbf{X}$ as $c_j$, $j = 1, \ldots, p+1$. We first show that $\|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|_2^2$ is minimised if $\boldsymbol{\theta}$ is chosen such that $\mathbf{X}\boldsymbol{\theta}$ is the orthogonal projection of $\mathbf{y}$ onto the (sub)space spanned by the columns $c_j$ of $\mathbf{X}$, and then show that the orthogonal projection is found by the normal equations.

Let us decompose $\mathbf{y}$ as $\mathbf{y}_\perp + \mathbf{y}_\parallel$, where $\mathbf{y}_\perp$ is orthogonal to the (sub)space spanned by all columns $c_i$, and $y_\parallel$ is in the (sub)space spanned by all columns $c_i$. Since $\mathbf{y}_\perp$ is orthogonal to both $\mathbf{y}_\parallel$ and $\mathbf{X}\boldsymbol{\theta}$, it follows that

$$\|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|_2^2 = \|\mathbf{X}\boldsymbol{\theta} - (\mathbf{y}_\perp + \mathbf{y}_\parallel)\|_2^2 = \|(\mathbf{X}\boldsymbol{\theta} - \mathbf{y}_\parallel) - \mathbf{y}_\perp\|_2^2 \geq \|\mathbf{y}_\perp\|_2^2, \tag{3.55}$$

and the triangle inequality also gives us

$$\|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|_2 = \|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}_\perp - \mathbf{y}_\parallel\|_2 \leq \|\mathbf{y}_\perp\|_2 + \|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}_\parallel\|_2. \tag{3.56}$$

This implies that if we choose $\boldsymbol{\theta}$ such that $\mathbf{X}\boldsymbol{\theta} = \mathbf{y}_\parallel$, the criterion $\|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|_2^2$ must have reached its minimum. Thus, our solution $\widehat{\boldsymbol{\theta}}$ must be such that $\mathbf{X}\widehat{\boldsymbol{\theta}} - \mathbf{y}$ is orthogonal to the (sub)space spanned by all columns $c_i$, meaning that

$$(\mathbf{y} - \mathbf{X}\widehat{\boldsymbol{\theta}})^\mathsf{T} c_j = 0, j = 1, \ldots, p+1 \tag{3.57}$$

(remember that two vectors $\mathbf{u}, \mathbf{v}$ are, by definition, orthogonal if their scalar product, $\mathbf{u}^\mathsf{T}\mathbf{v}$, is 0). Since the columns $c_j$ together form the matrix $\mathbf{X}$, we can write this compactly as

$$(\mathbf{y} - \mathbf{X}\widehat{\boldsymbol{\theta}})^\mathsf{T}\mathbf{X} = 0, \tag{3.58}$$

where the right hand side is the $p+1$-dimensional zero vector. This can equivalently be written as

$$\mathbf{X}^\mathsf{T}\mathbf{X}\widehat{\boldsymbol{\theta}} = \mathbf{X}^\mathsf{T}\mathbf{y},$$

which are the normal equations.

# 4 Understanding, Evaluating, and Improving Performance

So far we have encountered four methods for supervised machine learning, and more are to come. We train these models on training data and hope that they will also give us good predictions on new, previously unseen data. But can we really expect that to work? This question is central to whether machine learning is useful in practice. We discuss it in this chapter before we turn to more advanced methods. Along the way we will meet several useful concepts and practical tools for evaluating, improving, and choosing between supervised machine learning methods.

## 4.1 Expected New Data Error $E_{\text{new}}$: Performance in Production

We start with some concepts and notation. An error function $E(\widehat{y}, y)$ encodes what we mean by a good prediction: it compares a prediction $\widehat{y}(\mathbf{x})$ to a measured value $y$ and returns a small value (possibly zero) when $\widehat{y}(\mathbf{x})$ is close to $y$ and a larger value otherwise. Just as with loss functions for training, there are many possible error functions. Unless otherwise stated, our default choices are misclassification and squared error:

$$\text{Misclassification:} \quad E(\widehat{y}, y) \triangleq \mathbb{I}\{\widehat{y} \neq y\} = \begin{cases} 0 & \text{if } \widehat{y} = y \\ 1 & \text{if } \widehat{y} \neq y \end{cases} \quad \text{(classification)} \tag{4.1a}$$

$$\text{Squared error:} \quad E(\widehat{y}, y) \triangleq (\widehat{y} - y)^2 \quad \text{(regression).} \tag{4.1b}$$

The average misclassification (4.1a) is called the *misclassification rate*, and one minus the misclassification rate is the *accuracy*. The misclassification rate is a natural quantity in classification, but for imbalanced or asymmetric problems other aspects become more important. We return to this in Section 4.5.

The error function $E(\widehat{y}, y)$ resembles a loss function $L(\widehat{y}, y)$, but they play different roles. The loss function is used for *learning* (training) a model, whereas the error function is used to *analyse performance* of a model that has already been learned. We will soon see why the two should not always be chosen to be the same.

In the end, supervised machine learning is about designing methods that work well on an endless stream of new, unseen data. Think of all the new street-view images a self-driving car will process after it leaves the factory, or all the new patients a medical

diagnosis system will see once deployed in a clinic. Mathematically, performance on fresh data is the average of the error function – how often the classifier is right, or how accurately the regression method predicts. To reason about this endless stream, we introduce a *distribution over data* $p(\mathbf{x}, y)$. In most other chapters we treat only the output $y$ as random and the input $\mathbf{x}$ as fixed. Here we need $\mathbf{x}$ to be random as well. In any real-world problem $p(\mathbf{x}, y)$ is so complicated that we cannot write it down, but we can still use it to *reason* about machine learning methods.

Once a classification or regression method has been learned from training data $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^n$, it returns predictions $\widehat{y}(\mathbf{x}_\star)$ for any new input $\mathbf{x}_\star$. To emphasise that the model depends on the training data, we write $\widehat{y}(\mathbf{x}; \mathcal{T})$ throughout this chapter. A different training dataset $\mathcal{T}$ would give a different model, and thereby also different predictions.

Most of the time we consider what output a model predicts for one or a few test inputs $\mathbf{x}_\star$. Let us take the next step and average the error function (4.1) over *all* possible test data points $\mathbf{x}_\star$ from the distribution $p(\mathbf{x}, y)$. We call this the *expected new data error*,

$$\boxed{E_{\text{new}} \triangleq \mathbb{E}_\star \left[ E(\widehat{y}(\mathbf{x}_\star; \mathcal{T}), y_\star) \right],} \tag{4.2}$$

where the expectation $\mathbb{E}_\star$ is taken over test data points $(\mathbf{x}_\star, y_\star) \sim p(\mathbf{x}, y)$, that is,

$$\mathbb{E}_\star \left[ E(\widehat{y}(\mathbf{x}_\star; \mathcal{T}), y_\star) \right] = \int E(\widehat{y}(\mathbf{x}_\star; \mathcal{T}), y_\star) p(\mathbf{x}_\star, y_\star) \, d\mathbf{x}_\star dy_\star. \tag{4.3}$$

The model, no matter if it is linear regression, a classification tree, an ensemble, a neural network, or something else, is trained on the training dataset $\mathcal{T}$ and denoted $\widehat{y}(\cdot; \mathcal{T})$. Equation (4.2) then averages over possible test data points $(\mathbf{x}_\star, y_\star)$. In other words, $E_{\text{new}}$ describes how well the model *generalises* from the training data $\mathcal{T}$ to new situations in general.

We also introduce the *training error*,

$$\boxed{E_{\text{train}} \triangleq \frac{1}{n} \sum_{i=1}^{n} E(\widehat{y}(\mathbf{x}_i; \mathcal{T}), y_i),} \tag{4.4}$$

where $\{\mathbf{x}_i, y_i\}_{i=1}^n$ is the training data $\mathcal{T}$. $E_{\text{train}}$ tells us how well the method performs on the data it was trained on. It tells us, however, little about how the method will perform on new data.[1]

***Time to reflect 4.1*** *What is $E_{train}$ for k-NN with $k = 1$?*

[1] The term 'risk function' is used in some literature for the expected loss, which is the same as the new data error $E_{\text{new}}$ if the loss function and the error function are chosen to be the same. The training error $E_{\text{train}}$ is then referred to as 'empirical risk' and the idea of minimising the cost function as 'empirical risk minimisation'.


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

While $E_{\text{train}}$ describes how well the method 'reproduces' the data it was learned from, $E_{\text{new}}$ tells us how the method will perform 'in production'. For instance: what rates of false and missed pedestrian detections can we expect from the vision system in a self-driving car? What fraction of future patients will a medical diagnosis system misclassify?

The overall goal in supervised machine learning is to achieve as small an $E_{\text{new}}$ as possible.

This also sheds light on our earlier remark that the loss function $L(\widehat{y}, y)$ and the error function $E(\widehat{y}, y)$ need not be the same. A model that fits the training data well (small $E_{\text{train}}$) can still have a large $E_{\text{new}}$ on new, unseen data. The best strategy for a small $E_{\text{new}}$ is therefore *not* necessarily to minimise $E_{\text{train}}$ directly. For one reason, the misclassification error (4.1a) itself is a poor optimisation objective because it is discontinuous and has zero gradient almost everywhere. But on a more general level, a smarter choice of loss function can sometimes yield better generalisation properties and thus a smaller $E_{\text{new}}$ than what would be obtained by just optimising $E_{\text{train}}$. We will see this later for gradient boosting (Chapter 8) and support vector machines (Chapter 9).

Indeed, some methods are not trained by minimising a loss function at all, such as $k$-NN, but we can still evaluate them with an error function.

Unfortunately, we can never compute $E_{\text{new}}$ in a practical situation: its definition involves $p(\mathbf{x}, y)$, which we do not know. But $E_{\text{new}}$ is too important to abandon. The rest of the chapter is spent on how to *estimate* $E_{\text{new}}$ from data, and on how to understand what determines it.

It is important to remember that $E_{\text{new}}$ is a property of a specific trained model on a specific problem. It makes no sense to speak of '$E_{\text{new}}$ for logistic regression' in general. We must be specific, as in '$E_{\text{new}}$ for logistic regression trained on dataset X for task Y'.

## 4.2 Estimating $E_{\text{new}}$

A machine learning engineer may care about $E_{\text{new}}$ for several reasons:

- to judge whether the performance is good enough, or whether more work is needed and/or more training data should be collected;
- to choose between different methods;
- to choose hyperparameters (such as $k$ in $k$-NN, the regularisation parameter in ridge regression, or the number of hidden layers in deep learning) that minimise $E_{\text{new}}$;
- to report the expected performance to the customer.

Since we cannot compute $E_{\text{new}}$ directly, we need ways to *estimate* it. This leads us to the very useful idea of cross-validation.

### $E_{\text{train}} \not\approx E_{\text{new}}$: We Cannot Estimate $E_{\text{new}}$ from Training Data

We now have two quantities: the expected new data error $E_{\text{new}}$, which we cannot compute, and the training error $E_{\text{train}}$, which we can always compute.

Assume for now that $\mathcal{T}$ consists of samples from $p(\mathbf{x}, y)$. This just means that the training data was collected under the same conditions as the data the model will later face – a common and reasonable assumption.

When we cannot compute an expected value in closed form, one natural approach is to approximate it by a sample average, replacing the integral (the expected value) by a finite sum. A tempting idea is therefore to approximate the integral in $E_{\text{new}}$ by the sum in $E_{\text{train}}$:

$$E_{\text{new}} = \int E(\widehat{y}(\mathbf{x}; \mathcal{T}), y)) p(\mathbf{x}, y) d\mathbf{x} dy \overset{??}{\approx} \frac{1}{n} \sum_{i=1}^{n} E(\widehat{y}(\mathbf{x}_i; \mathcal{T}), y_i) = E_{\text{train}}. \tag{4.5}$$

In plain words: Can we expect a method to perform on new data as well as it did on the training data?

The answer is **no**.

> ***Time to reflect 4.2*** *Why can we not expect the performance on training data ($E_{train}$) to be a good approximation for how a method will perform on new, previously unseen data ($E_{new}$), even though the training data is drawn from the distribution $p(\mathbf{x}, y)$?*

Why does (4.5) fail? The samples used to approximate the integral are the training data. But those same data points were used to train the model. The integrand itself depends on the whole training set $\mathcal{T}$ through the first factor. We cannot use the same data to approximate an integral whose integrand depends on that data. Put differently, the expected value in (4.5) must be computed *conditionally* on a fixed training set $\mathcal{T}$.

As we will see, the typical behaviour is $E_{\text{train}} < E_{\text{new}}$. A method usually performs worse on new data than on training data. *The training error is therefore not a reliable estimate of $E_{new}$.*

### $E_{\text{hold-out}} \approx E_{\text{new}}$: We Can Estimate $E_{\text{new}}$ from Hold-Out Validation Data

The problem with (4.5) was that we used the training data twice: first to train the model, and then to evaluate the error. The fix is to set aside some *hold-out validation data* $\{\mathbf{x}'_j, y'_j\}_{j=1}^{n_v}$ that was not used for training, and use it only to estimate model


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`


**Figure 4.1:** The hold-out validation dataset approach: If we split the available data into two sets and train the model on the training set, we can compute $E_{\text{hold-out}}$ using the hold-out validation set. $E_{\text{hold-out}}$ is an unbiased estimate of $E_{\text{new}}$, and the more data there is in the hold-out validation dataset, the less variance there will be in $E_{\text{hold-out}}$, (better estimate) but the less data is left for training the model (larger $E_{\text{new}}$).

performance. We call this the *hold-out validation error*,

$$E_{\text{hold-out}} \triangleq \frac{1}{n_v} \sum_{j=1}^{n_v} E(\widehat{y}(\mathbf{x}'_j; \mathcal{T}), y'_j). \tag{4.6}$$

This procedure for estimating $E_{\text{new}}$ is illustrated in Figure 4.1. Some data goes to training; some is held aside purely for computing $E_{\text{hold-out}}$.

**Be aware!** *Always split the data randomly - for instance by shuffling before the training-validation split. Someone may have sorted the dataset for you, intentionally or not. Without a random split, your binary classification problem may end up with one class in training and the other in validation . . .*

If all (training and validation) data points are drawn from $p(\mathbf{x}, y)$, then $E_{\text{hold-out}}$ is an *unbiased* estimate of $E_{\text{new}}$: if we were to repeat the entire procedure many times with fresh data, the average $E_{\text{hold-out}}$ would equal $E_{\text{new}}$. That is reassuring on a theoretical level, but it says nothing about how close $E_{\text{hold-out}}$ is to $E_{\text{new}}$ in any single experiment. However, the law of large numbers tells us that the variance of $E_{\text{hold-out}}$ shrinks as the hold-out set grows. A small variance means $E_{\text{hold-out}}$ is likely close to $E_{\text{new}}$, so a large hold-out set gives a good estimate.

But there is a catch. Setting aside a large hold-out set means that less data is left for training. And less training data typically means a larger $E_{\text{new}}$ (we return to this in Section 4.3). That is bad, since a small $E_{\text{new}}$ is the whole point.

In some situations, there is a lot of data available. When it is, we can set aside a few percent for hold-out validation without a noticeable loss in $E_{\text{new}}$. *In data-rich situations, the hold-out validation approach is sufficient.* But with limited data available, we face a dilemma: *the better we want to know* $E_{\text{new}}$ (more hold-out data, less variance in $E_{\text{hold-out}}$), *the worse we make it* (less training data, larger $E_{\text{new}}$). This is unsatisfying, and we need a better approach.


**Figure 4.2:** Illustration of $k$-fold cross-validation. The data is split into $k$ batches of similar sizes. When looping over $\ell = 1, 2, \ldots, k$, batch $\ell$ is held out as validation data, and the model is trained on the remaining $k - 1$ data batches. Each time, the trained model is used to compute the average error $E^{(\ell)}_{k\text{-fold}}$ for the validation data. The final model is trained using all available data, and the estimate of $E_{\text{new}}$ for that model is $E_{k\text{-fold}}$, the average of all $E^{(\ell)}_{k\text{-fold}}$.

### *k*-Fold Cross-Validation: $E_{k\text{-fold}} \approx E_{\text{new}}$ Without Setting Aside Validation Data

To avoid setting aside validation data while still getting an estimate of $E_{\text{new}}$, one might try the following two-step procedure:

(i) split the data into a training and a hold-out validation set, train the model on the training data, and compute $E_{\text{hold-out}}$ on the hold-out set (as in Figure 4.1);

(ii) train the model again, this time on the full dataset.

This gives us a model trained on all data and an estimate of $E_{\text{new}}$. Not bad, but not quite right either. A low-variance estimate in step (i) requires a large hold-out set, which means the model trained in step (i) is quite different from the one in step (ii). And the estimate we get concerns the model from (i), not (ii). So this scheme does not give a good estimate of $E_{\text{new}}$ for the final model in step (ii). But the idea is close. Building on it leads to the *k-fold cross-validation* method.

Our goal is to use all the data for training and still get a good estimate of $E_{\text{new}}$ for that model. *k-fold cross-validation* gets us most of the way there. The idea is to repeat

the hold-out procedure several times, splitting the dataset into $k$ batches of similar size. The model is then trained $k$ separate times, each time using a different batch as a hold-out validation set and the remaining $k-1$ batches for training (so that every data point is used once for validation). The final performance estimate is taken as the average of the $k$ validation results:

(i) split the dataset into $k$ batches of similar size (see Figure 4.2), and let $\ell = 1$;

(ii) take batch $\ell$ as the hold-out validation data and the remaining batches as training data;

(iii) train the model on the training data, and compute $E^{(\ell)}_{\text{hold-out}}$ as the average error on the hold-out validation data, as in (4.6);

(iv) if $\ell < k$, set $\ell \leftarrow \ell + 1$ and return to (ii). If $\ell = k$, compute the *$k$-fold cross-validation error*

$$\boxed{E_{k\text{-fold}} \triangleq \frac{1}{k}\sum_{\ell=1}^{k} E^{(\ell)}_{\text{hold-out}}} \tag{4.7}$$

(v) train the model again, this time using the entire dataset.

The procedure is illustrated in Figure 4.2.

With $k$-fold cross-validation we obtain both a model trained on all data and an estimate $E_{k\text{-fold}}$ of $E_{\text{new}}$ for that model. However, unlike $E_{\text{hold-out}}$ (Section 4.2), $E_{k\text{-fold}}$ is *not* an unbiased estimate of $E_{\text{new}}$. But for large enough $k$ it is usually a good enough approximation. Let us see why.

We need to distinguish between the *final* model, trained on all data in step (v), and the *intermediate* models, each trained on a fraction $(k-1)/k$ of the data in step (iii). The key insight: for large enough $k$, the intermediate models are similar to the final model, since only a fraction $1/k$ of the data is missing when each model is trained. Each intermediate $E^{(\ell)}_{\text{hold-out}}$ is an unbiased (but high-variance) estimate of $E_{\text{new}}$ for its corresponding intermediate model. Because the intermediate and final models are similar, $E_{k\text{-fold}}$ (4.7) is approximately the average of $k$ high-variance estimates of $E_{\text{new}}$ for the final model. Averaging reduces variance, so $E_{k\text{-fold}}$ is a better estimate of $E_{\text{new}}$ than any single $E^{(\ell)}_{\text{hold-out}}$.

**Be aware!** *As with hold-out validation, always split the data randomly. The simplest approach: permute the dataset randomly before splitting into batches.*

We usually think of training as something done once. In $k$-fold cross-validation it happens $k$ (or even $k+1$) times. The special case $k = n$ is called *leave-one-out cross-validation*. For methods like linear regression, solving the normal equations takes milliseconds on a modern computer, so repeating it $n$ extra times may be fine in practice. However, for computationally demanding methods such as deep neural networks, this becomes cumbersome, and $k = 10$ is a more realistic choice. When data is plentiful, the cheaper hold-out approach is also an option.

### Using a Test Dataset

A very important use of $E_{k\text{-fold}}$ (or $E_{\text{hold-out}}$) in practice is to choose between methods and to select hyperparameters, such as choosing $k$ in $k$-NN, tree depths, regularisation parameters. The goal is to make $E_{k\text{-fold}}$ (or $E_{\text{hold-out}}$) as small as possible. However, just as $E_{\text{train}}$ cannot be used to estimate $E_{\text{new}}$, selecting models and hyperparameters based on $E_{k\text{-fold}}$ (or $E_{\text{hold-out}}$) invalidates its use as an estimator of $E_{\text{new}}$. If hyperparameters are tuned to minimise $E_{k\text{-fold}}$, we risk overfitting to the validation data, which makes $E_{k\text{-fold}}$ an overly optimistic estimate of the true new data error $E_{\text{new}}$. To get a reliable estimate of the final $E_{\text{new}}$, we should set aside another dataset – a *test dataset* – and use it only once, after all model and hyperparameter choices have been made.

When training data is expensive, it is common to enlarge the training set artificially, for instance by duplicating data points and adding noise, using simulated data, or using data from a related problem (we return to this in Chapter 13). When such techniques are used, $\mathcal{T}$ is no longer drawn from $p(\mathbf{x}, y)$. In the worst case the artificial data provides no information about $p(\mathbf{x}, y)$ at all, and the model cannot learn anything useful. Having a good estimate of $E_{\text{new}}$ then becomes especially valuable – but it can only be obtained from data we *know* is drawn from $p(\mathbf{x}, y)$ (collected under production-like conditions). If the training data is extended artificially, it is therefore extra important to set aside a test dataset *before* the data is artificially extended.

The error function evaluated on the test dataset could naturally be called 'test error'. We do, however, avoid this term to prevent confusion, since it is commonly used (ambiguously) both for the error on the test dataset and as another name for $E_{\text{new}}$.

## 4.3 The Training Error–Generalisation Gap Decomposition of $E_{\text{new}}$

Designing a method with small $E_{\text{new}}$ is the central goal of supervised machine learning, and cross-validation helps us estimate $E_{\text{new}}$. We can learn even more by analysing $E_{\text{new}}$ mathematically. To do so, we need another level of abstraction: the *training-data averaged* versions of $E_{\text{new}}$ and $E_{\text{train}}$. We write $E_{\text{new}}(\mathcal{T})$ and $E_{\text{train}}(\mathcal{T})$ to emphasise that both depend on a specific training set $\mathcal{T}$, and we define

$$\bar{E}_{\text{new}} \triangleq \mathbb{E}_{\mathcal{T}}\left[E_{\text{new}}(\mathcal{T})\right], \quad \text{and} \tag{4.8a}$$

$$\bar{E}_{\text{train}} \triangleq \mathbb{E}_{\mathcal{T}}\left[E_{\text{train}}(\mathcal{T})\right]. \tag{4.8b}$$

Here $\mathbb{E}_{\mathcal{T}}$ is the expectation over training sets $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^{n}$ of fixed size $n$, each drawn independently from $p(\mathbf{x}, y)$. So $\bar{E}_{\text{new}}$ is the *average* $E_{\text{new}}$ we would obtain if we retrained the model many times on different training sets of size $n$, and similarly for $\bar{E}_{\text{train}}$. The point is that the average behaviour $\bar{E}_{\text{new}}$ and $\bar{E}_{\text{train}}$ is easier to reason about than the quantities $E_{\text{new}}$ and $E_{\text{train}}$ for one particular $\mathcal{T}$. Although $E_{\text{new}}$ is what ultimately matters (the training data is usually fixed), the insights from $\bar{E}_{\text{new}}$ carry over.

***Time to reflect 4.3*** *$E_{new}(\mathcal{T})$ is the new data error when the model is trained on a specific training dataset $\mathcal{T}$, whereas $\bar{E}_{new}$ is averaged over all possible training datasets. Considering the fact that in the procedure of $k$-fold cross-validation, the model is trained each time on an (at least slightly) different training dataset, does $E_{k\text{-}fold}$ actually estimate $E_{new}$, or is it rather $\bar{E}_{new}$? And is that different for different values of $k$? How could $\bar{E}_{train}$ be estimated?*

We have seen that $E_{\text{train}}$ cannot be used to estimate $E_{\text{new}}$. In fact, on average,

$$\bar{E}_{\text{train}} < \bar{E}_{\text{new}}. \tag{4.9}$$

In words: a method usually performs worse on new data than on training data. The ability of a method to do well on unseen data is called its ability to *generalise* from the training data, and we call the difference between $\bar{E}_{\text{new}}$ and $\bar{E}_{\text{train}}$ the *generalisation gap*:[2]

$$\text{generalisation gap} \triangleq \bar{E}_{\text{new}} - \bar{E}_{\text{train}}. \tag{4.10}$$

The generalisation gap is the difference between expected performance on training data and expected performance 'in production'.

The decomposition

$$\bar{E}_{\text{new}} = \bar{E}_{\text{train}} + \text{generalisation gap} \tag{4.11}$$

opens the door to understanding what drives $\bar{E}_{\text{new}}$ in practice. We call (4.11) the *training error–generalisation gap decomposition of $\bar{E}_{new}$*.

### What Affects the Generalisation Gap?

The generalisation gap depends on both machine learning method as well as the problem at hand.

The behaviour of the generalisation gap can be divided into two regimes. The regime studied in traditional machine learning, before deep neural networks (Chapters 6 and 7) entered the stage, is characterised by *models that do **not** fit the training data perfectly*, or at the borderline have just enough capacity to exactly fit the training data (such as a $k$-NN with $k = 1$ or a fully grown decision tree). For the recurring car stopping distances or the song classification, such learned models or decision boundaries do not follow the training data points perfectly, but rather have a smoothing effect. This is the typical case for methods such as $k$-NN, trees, linear and logistic regression. We will discuss this traditional regime first.

In essence, *the more a model adapts to training data, the larger the generalisation gap*.

[2] Strictly speaking $\bar{E}_{\text{new}} - \bar{E}_{\text{train}}$ is the *expected* generalisation gap, whereas $E_{\text{new}} - E_{\text{train}}$ is the *conditional* generalisation gap. We use the same term for both.

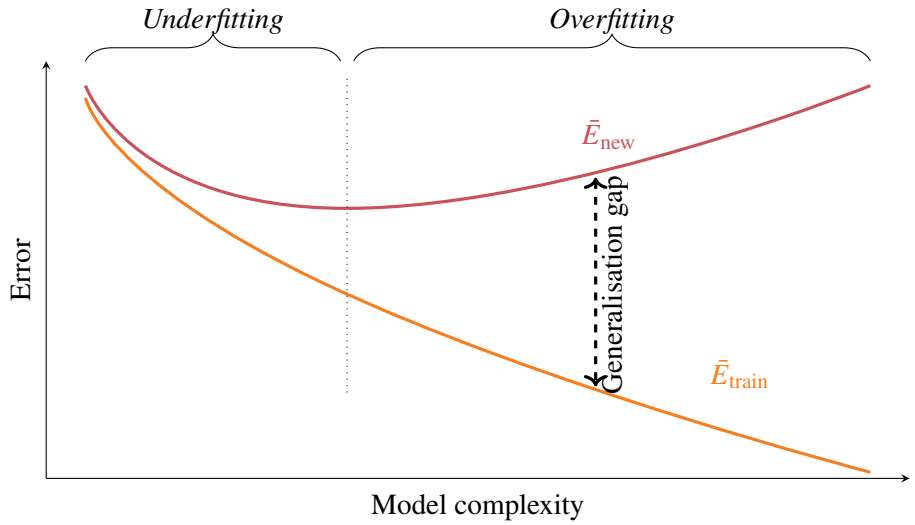



**Figure 4.3:** Behaviour of $\bar{E}_{\text{train}}$ and $\bar{E}_{\text{new}}$ for many supervised machine learning methods, as a function of model complexity. We have not made a formal definition of complexity, but a rough proxy is the number of parameters that are learned from the data. The difference between the two curves is the generalisation gap. The training error $\bar{E}_{\text{train}}$ decreases as the model complexity increases, whereas the new data error $\bar{E}_{\text{new}}$ typically has a U-shape. If the model is so complex that $\bar{E}_{\text{new}}$ is larger than it had been with a less complex model, the term *overfit* is commonly used. Somewhat less common is the term *underfitting*, used for the opposite situation. The level of model complexity which gives the minimum $\bar{E}_{\text{new}}$ (at the dotted line) could be called a balanced fit. When, for example, we use cross-validation to select hyperparameters (that is, tuning the model complexity), we are searching for a balanced fit.

Let us look a bit closer at this statement by first conceptually arrange all the methods on a scale according to how much they adapt to the data. There are theoretical frameworks for quantifying such a scale, for instance the Vapnik–Chervonenkis (VC) dimension or the Rademacher complexity. The details of those frameworks are however rather intricate and their practical use is often limited to very specific situations, so we will not pursue them any further here. Instead, we use the rather vague terms *model complexity* or *model flexibility* (used interchangeably) to mean a method's ability to adapt to patterns in the training data. A high-complexity model such as a deep tree, or $k$-NN with small $k$ can describe complicated input–output relationships. A low-complexity model, such as logistic regression, is more limited. For parametric models, complexity is roughly the number of learnable parameters, though regularisation matters too. Model complexity is a simplification and does not capture the full picture (we return to this), but it carries useful intuition.

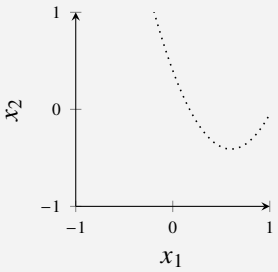
The typical pattern within this regime is: *higher model complexity implies a larger generalisation gap*. $\bar{E}_{\text{train}}$ decreases with complexity, while $\bar{E}_{\text{new}}$ follows a U-shape, reaching its minimum at some intermediate complexity. Either too low or too high complexity raises $\bar{E}_{\text{new}}$. Figure 4.3 shows this. When complexity is so high that $\bar{E}_{\text{new}}$ is larger than it would be with a simpler model, we call this *overfitting*. The opposite

case, when complexity is too low, is sometimes called *underfitting*. The point where $\bar{E}_{\text{new}}$ is minimised is a *balanced fit*, and that is the sweet spot we are after. Example 4.1 below illustrates all of this.

**Example 4.1 The training error–generalisation gap decomposition for $k$-NN**

We consider a simulated binary classification problem with two-dimensional input $\mathbf{x}$. Because the problem is simulated, we know $p(\mathbf{x}, y)$. Here $p(\mathbf{x})$ is uniform on the square $[-1, 1]^2$, and $p(y \mid \mathbf{x})$ is defined so that points above the dotted curve in Figure 4.4 are blue with probability 0.8, and points below are red with probability 0.8. The optimal classifier – the one with minimal $E_{\text{new}}$ – would use the dotted curve as its decision boundary and achieve $E_{\text{new}} = 0.2$.

**Fig. 4.4**




We take $n = 200$ training points and learn three classifiers: $k$-NN with $k = 70$, $k = 20$, and $k = 2$. In terms of model complexity, $k = 70$ gives the least flexible model and $k = 2$ the most flexible. Their decision boundaries, together with the training data, are shown in Figure 4.5.

**Fig. 4.5**

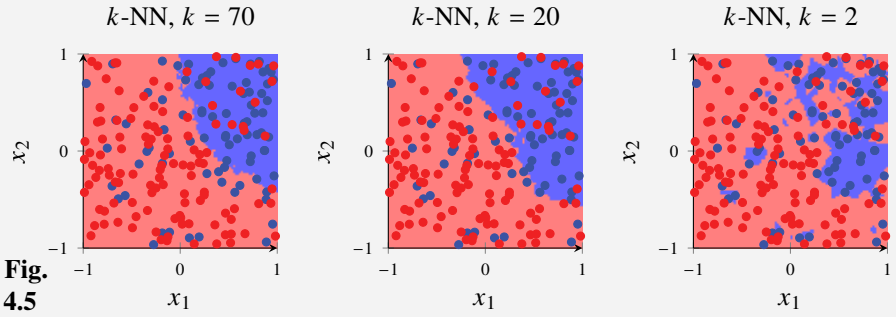
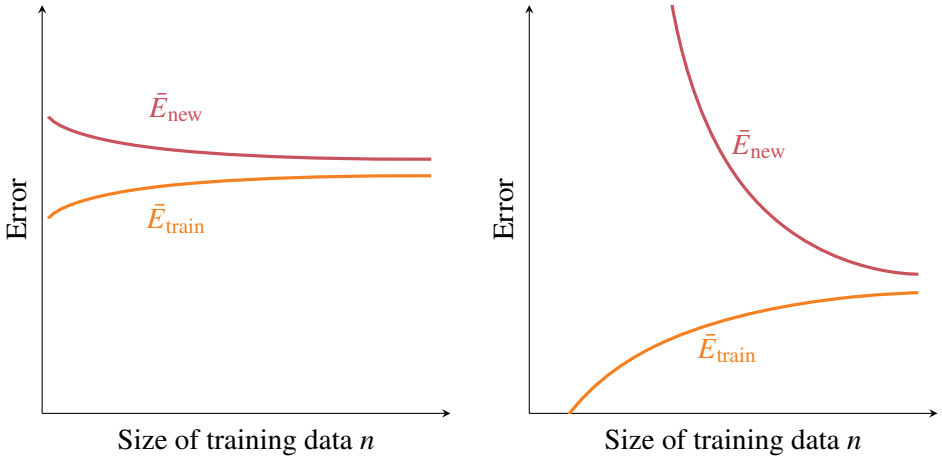



Figure 4.5 shows that $k = 2$ (right) adapts too much to the data. With $k = 70$ (left) the model is rigid enough not to chase the noise, but possibly too rigid to follow the true dotted line in Figure 4.4.

We can read $E_{\text{train}}$ off Figure 4.5 by counting misclassified training points. From left to right, $E_{\text{train}} = 0.27, 0.24, 0.22$. Since this is a simulated example, we can also estimate $E_{\text{new}}$ (by simulating lots of test data), which gives $E_{\text{new}} = 0.26, 0.23, 0.33$. This pattern resembles Figure 4.3, except that $E_{\text{new}}$ is actually *smaller* than $E_{\text{train}}$ for some values of $k$. This does not contradict the theory: the theory is about the *averages* $\bar{E}_{\text{new}}$ and $\bar{E}_{\text{train}}$, not the values $E_{\text{new}}$ and $E_{\text{train}}$ for one specific training set. To see $\bar{E}_{\text{new}}$ and $\bar{E}_{\text{train}}$, we repeat the whole experiment 100 times and average:

| | $k$-NN with $k = 70$ | $k$-NN with $k = 20$ | $k$-NN with $k = 2$ |
|---|---|---|---|
| $\bar{E}_{\text{train}}$ | 0.24 | 0.22 | 0.17 |
| $\bar{E}_{\text{new}}$ | 0.25 | 0.23 | 0.30 |




**(a)** Simple model

**(b)** More complex model

**Figure 4.6:** Typical relationship between $\bar{E}_{\text{new}}$, $\bar{E}_{\text{train}}$, and the number of data points $n$ in the training dataset for a simple model (low model flexibility, left) and a complex model (high model flexibility, right). The generalisation gap (difference between $\bar{E}_{\text{new}}$ and $\bar{E}_{\text{train}}$) decreases, at the same time as $\bar{E}_{\text{train}}$ increases. Typically, a more complex model (right panel) will for large enough $n$ attain a smaller $\bar{E}_{\text{new}}$ than a simpler model (left panel) would on the same problem (the figures should be thought of as having the same scales on the axes). However, the generalisation gap is typically larger for a more complex model, in particular when the training dataset is small.

This table matches Figure 4.3 well. The generalisation gap ($\bar{E}_{\text{new}} - \bar{E}_{\text{train}}$) is positive and grows with model complexity (smaller $k$ in $k$-NN), while $\bar{E}_{\text{train}}$ shrinks. Among these three values, $\bar{E}_{\text{new}}$ is smallest at $k = 20$. So $k = 2$ overfits and $k = 70$ underfits for this problem.

So far we have focused on the relationship between the generalisation gap and model complexity. Another key factor is the size $n$ of the training set. In general, *more training data means a smaller generalisation gap*.

In the regime we have been focusing on so far, $\bar{E}_{\text{train}}$ however typically *increases* with $n$, because the models in this regime can typically not fit all training points well when there are many of them. Figure 4.6 sketches that behaviour.

## Reducing $E_{\text{new}}$ in Practice

Our goal is a small error in production, a small $E_{\text{new}}$. By the decomposition $E_{\text{new}} = E_{\text{train}}$ + generalisation gap, both $E_{\text{train}}$ and the generalisation gap need to be small. Two practical conclusions follow:

- $E_{\text{new}}$ will, on average, not be smaller than $E_{\text{train}}$. So if $E_{\text{train}}$ is already much larger than the $E_{\text{new}}$ your application requires, there is no point in cross-validation. Rethink the problem and the method.

- The generalisation gap and $E_{\text{new}}$ typically shrink as $n$ grows. If collecting more training data is feasible, it is often a safe bet for reducing $E_{\text{new}}$.

More model flexibility reduces $E_{\text{train}}$ but may widen the generalisation gap, as long as we are in the traditional regime. The optimal tradeoff that minimises $E_{\text{new}}$ is usually reached with neither a zero training error nor a zero generalisation gap. Monitoring $E_{\text{train}}$ and estimating $E_{\text{new}}$ by cross-validation gives us the following rules of thumb:

- If $E_{\text{hold-out}} \approx E_{\text{train}}$ (small generalisation gap; possibly underfitting), try increasing the model flexibility: loosen the regularisation, add parameters, and so on.
- If $E_{\text{train}}$ is close to zero but $E_{\text{hold-out}}$ is not, you might be overfitting. Try reducing the model flexibility: tighten the regularisation, reduce the order, and so on.

### Shortcomings of the Model Complexity Scale

Figure 4.3 is a useful picture when there is only one hyperparameter to choose, such as choosing a regularisation parameter (more regularisation means less flexibility) or a polynomial degree (more degrees means more flexibility). However, already when considering only those two hyperparameters jointly, the one-dimensional model complexity scale is too simplistic, as Example 4.2 shows.

The shortcoming of the one-dimensional scale might become even more pronounced when choosing between different methods. For a given problem, one method can very well have a smaller generalisation gap than another *without* having a larger training error. Some methods are simply better suited to certain problems – an observation often framed by saying that the method has an *inductive bias* that aligns well with the data.

In Example 4.2 we can see how the relation between $\bar{E}_{\text{train}}$ and the generalisation gap can look quite different across methods, which contradicts the conceptual sketch in Figure 4.3 and confirms that the one-dimensional complexity scale is a simplified concept. (The theoretical concepts of VC dimension and Rademacher complexity circumvent this fact by only providing upper bounds, and not equalities, on certain quantities.) But in any real problem we cannot produce a plot like Example 4.2, since that requires a simulated example where we control the data-generating process. In practice we therefore rely on more limited information. Good practice is to pick models that are known to work well for the type of data at hand (well-aligned inductive bias), to draw on experience from similar problems, and to use cross-validation to compare models and tune hyperparameters. And despite its simplification, the mental picture of Figure 4.3 is still useful when deciding which method or hyperparameter value to try next.

## Example 4.2 Training error and generalisation gap for a regression problem

To see how the training error and generalisation gap behave, we use a simulated problem so that we can compute $E_{\text{new}}$. We generate $n = 10$ data points from $x \sim \mathcal{U}[-5, 10]$, $y = \min(0.1x^2, 3) + \varepsilon$, $\varepsilon \sim \mathcal{N}(0, 1)$, and consider these regression methods:

- Linear regression with $L^2$ regularisation
- Linear regression with a quadratic polynomial and $L^2$ regularisation
- Linear regression with a third order polynomial and $L^2$ regularisation
- Regression tree
- Random forest (Chapter 8) with 10 regression trees

For each method we try a few values of the hyperparameter (regularisation parameter or tree depth) and compute $\bar{E}_{\text{train}}$ and the generalisation gap.

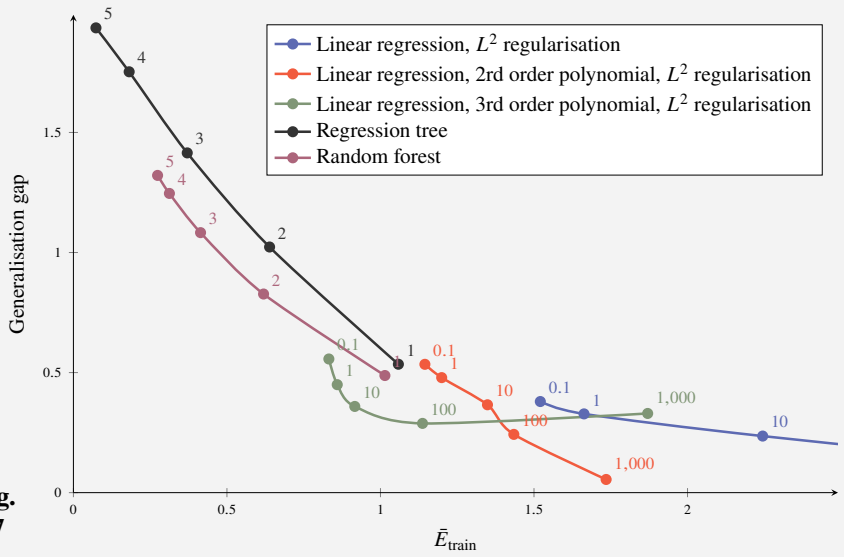



**Fig. 4.7**

For each method, the hyperparameter that minimises $\bar{E}_{\text{new}}$ is the one closest to the origin in the 1-norm sense in Figure 4.7, since $\bar{E}_{\text{new}} = \bar{E}_{\text{train}}$ + generalisation gap. Once we have chosen a method and are only tuning one hyperparameter, the situation matches Figure 4.3 well.

However, across *different* methods the picture reveals a more intricate situation than the one-dimensional model complexity scale in Figure 4.3. Compare the second-order (red) and third-order (green) polynomial regressions in Figure 4.7: for some regularisation values (like 0.1, 1 and 10), the training error decreases *without* any increase in the generalisation gap. Using a third instead of second order polynomial in those specific situations reduces the training error without increasing the generalisation gap, a phenomenon that goes beyond the the one-dimensional picture in Figure 4.3. Similarly, the random forest (purple) has a smaller generalisation gap than the tree (black) at tree depth 2, with the same training error. All of those relationships are intricate and problem-dependent, and are not captured by the simplified picture in


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

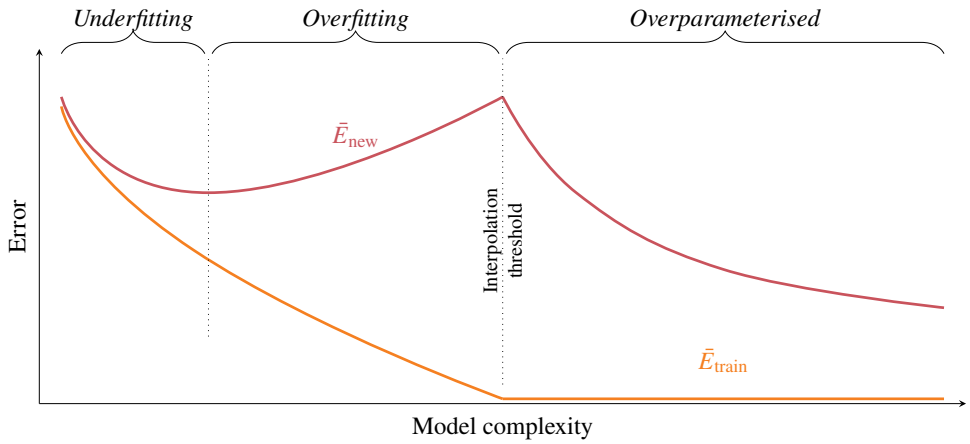



**Figure 4.8:** The double descent phenomenon. In the overparametrised regime past the interpolation threshold (the point where $\bar{E}_{\text{train}} = 0$ is first reached), increasing model complexity further can make $\bar{E}_{\text{new}}$ decrease again, contrary to the classical U-curve of Figure 4.3.

Figure 4.3. The picture will however clear up a bit once we introduce the bias–variance decomposition, in particular in Example 4.4.

### The Generalisation Gap in the Overparametrised Regime

As if the complications from Example 4.2 were not enough, there is also another completely different side of the story as well. Let us go back to conceptual sketch in Figure 4.3, where two typical models at the very far right are $k$-NN with $k = 1$ or a fully grown decision tree: they always fit the training data perfectly ($\bar{E}_{\text{train}} = 0$), yet they typically generalise quite badly. In a sense, this is the point where the flexibility of $k$-NN and the decision tree is stretched to its maximum.

But it is, in fact, possible to go even further in terms of model flexibility. With deep neural networks it has become practice to use models that are so flexible that their maximum expressivity goes well beyond fitting perfectly to the training data. Or, put differently, with those models there is not just a single parameter value that gives a perfect fit to the training data $E_{\text{train}} = 0$, but a full range of such parameter values. These models are usually referred to as *overparametrised*, meaning that they contain way more parameters than necessary to describe the training data.

A concrete example of what the ratio between parameter count and data points can look like is the ResNet models, Section 7.2, where models with about 60 million parameters are trained on about 1 million training data points. The overparametrisation in large language models is typically even more extreme. Counterintuitive as it may seem, such overparametrised models have been shown to perform amazingly well, and their generalisation gaps do not follow the same patterns as in the traditional machine learning regime.

The typical observed behaviour with these overparametrised models can be added as an extension of Figure 4.3. Going from left to right along the model complexity axis, we refer to the first point at which $\bar{E}_{\text{train}} = 0$ as the *interpolation threshold*. The interpolation threshold often corresponds to the worst case of overfitting – for example, a fully grown decision tree with a very poor generalisation gap. But when increasing the model complexity even further, one can observe that the generalisation gap may actually shrink for well-designed overparametrised models. This phenomenon is sometimes referred to as *double descent*, referring to the second descent of $\bar{E}_{\text{new}}$.

While the underlying theory is not as well understood as for the traditional regime, there is no doubt that the phenomenon exists in the overparametrised regime. The intuitive picture is that right at the interpolation threshold, the model is just flexible enough to exaclty interpolate the data, but it has no freedom to also generalize well. However, an even more flexible model that easily interpolates the training data has capacity left to also generalise well. This however assumes that the model training is designed such that it favors models that generalise well, by using well crafted methods where implicit and explicit regularisation (Section 5.3) typically plkays an important role.

The takeaway message is *not* that going beyond the interpolation threshold is automatically good for $\bar{E}_{\text{new}}$. That would be a stretch. In practice, it is a significant challenge to construct and train well-performing models in the overparametrised regime. The message is rather that *it is possible* for very well-designed overparametrised models to outperform the traditional regime in certain applications, but it requires a good deal of care and clever machine learning engineering. Two such successful examples are discussed in Chapter 7.

## 4.4 The Bias–Variance Decomposition of $E_{\text{new}}$

We now introduce a second decomposition of $\bar{E}_{\text{new}}$, this time into a (squared) *bias*, a *variance*, and an unavoidable *irreducible noise* term. This decomposition is more abstract than the training error–generalisation gap one, but it offers extra insight into how different models behave. We will however limit the bias–variance decomposition to the traditional regime, since the extension to the overparametrised regime is not straightforward.

Let us first recall the general concepts of bias and variance. Imagine an experiment with an unknown constant $z_0$ that we want to estimate, and a random variable $z$ that helps us do so. Think of $z_0$ as the true position of an object and $z$ as noisy GPS measurements of that position. Since $z$ is random, it has some mean $\mathbb{E}[z]$, which we denote $\bar{z}$. We define

$$\text{Bias: } \bar{z} - z_0 \tag{4.12a}$$

$$\text{Variance: } \mathbb{E}\big[(z - \bar{z})^2\big] = \mathbb{E}\big[z^2\big] - \bar{z}^2. \tag{4.12b}$$

The *variance* describes how much the experiment varies from trial to trial (noise in the GPS measurements). The *bias* describes the systematic error that remains no matter


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

how many times we repeat the experiment (a constant shift in the GPS measurements). By adding and removing $\bar{z}$, expanding the square, and using that $\bar{z}$ and $z_0$ are constants with respect to the expectation, we can decompose the squared error between $z$ and $z_0$ into variance and squared bias:

$$\begin{aligned}\mathbb{E}\big[(z - z_0)^2\big] &= \mathbb{E}\Big[\big((z - \bar{z}) + (\bar{z} - z_0)\big)^2\Big] = \\ &= \underbrace{\mathbb{E}\big[(z - \bar{z})^2\big]}_{\text{Variance}} + 2\underbrace{(\mathbb{E}[z] - \bar{z})}_{0,\text{ since } \mathbb{E}[z]=\bar{z}}(\bar{z} - z_0) + \underbrace{(\bar{z} - z_0)^2}_{\text{Bias}^2}. \end{aligned} \tag{4.13}$$

In words: the average squared error is the sum of the squared bias and the variance. To obtain a small expected squared error we must watch *both* terms. A small bias alone is not enough, and neither is a small variance alone.

We now apply the bias and variance concept to supervised machine learning. For mathematical simplicity we focus on regression with the squared error. The intuition carries over to classification. Here $z_0$ corresponds to the true relationship between input and output, and $z$ corresponds to the model we learn from training data. Since the training data itself is random, the learned model is random too.

We assume the true relationship between input $\mathbf{x}$ and output $y$ is some (possibly complicated) function $f_0(\mathbf{x})$ plus independent noise:

$$y = f_0(\mathbf{x}) + \varepsilon, \quad \text{with } \mathbb{E}[\varepsilon] = 0 \text{ and } \operatorname{var}(\varepsilon) = \sigma^2. \tag{4.14}$$

In our notation, $\widehat{y}(\mathbf{x}; \mathcal{T})$ is the model trained on $\mathcal{T}$ - the random variable, corresponding to $z$ above. We also introduce the *average trained model*, corresponding to $\bar{z}$:

$$\bar{f}(\mathbf{x}) \triangleq \mathbb{E}_{\mathcal{T}}\left[\widehat{y}(\mathbf{x}; \mathcal{T})\right]. \tag{4.15}$$

As before, $\mathbb{E}_{\mathcal{T}}$ is the expectation over $n$ training points drawn from $p(\mathbf{x}, y)$. So $\bar{f}(\mathbf{x})$ is the hypothetical average model we would get if we retrained the model on infinitely many training sets of size $n$ and averaged.

Remember that the definition of $\bar{E}_{\text{new}}$ (for regression with squared error) is

$$\bar{E}_{\text{new}} = \mathbb{E}_{\mathcal{T}}\left[\mathbb{E}_{\star}\left[(\widehat{y}(\mathbf{x}_\star; \mathcal{T}) - y_\star)^2\right]\right]. \tag{4.16}$$

We can change the order of integration and write (4.16) as

$$\bar{E}_{\text{new}} = \mathbb{E}_{\star}\left[\mathbb{E}_{\mathcal{T}}\left[(\widehat{y}(\mathbf{x}_\star; \mathcal{T}) - f_0(\mathbf{x}_\star) - \varepsilon)^2\right]\right]. \tag{4.17}$$

We can now apply the same idea as in (4.13), with the addition of a zero-mean noise $\varepsilon$ (independent of $\widehat{y}(\mathbf{x}_\star; \mathcal{T})$). The details become more intricate, but it turns out we can rewrite the integrand of $\mathbb{E}_\star$ in (4.17) as

$$\begin{aligned}&\mathbb{E}_{\mathcal{T}}\Big[\big(\underbrace{\widehat{y}(\mathbf{x}_\star; \mathcal{T})}_{\text{“}z\text{”}} - \underbrace{f_0(\mathbf{x}_\star)}_{\text{“}z_0\text{”}} - \varepsilon\big)^2\Big] \\ &\quad = \big(\bar{f}(\mathbf{x}_\star) - f_0(\mathbf{x}_\star)\big)^2 + \mathbb{E}_{\mathcal{T}}\Big[\big(\widehat{y}(\mathbf{x}_\star; \mathcal{T}) - \bar{f}(\mathbf{x}_\star)\big)^2\Big] - 2\varepsilon\big(\bar{f}(\mathbf{x}_\star) - f_0(\mathbf{x}_\star)\big) + \varepsilon^2.\end{aligned} \tag{4.18}$$

This is (4.13) applied to supervised machine learning. Taking the expectation $\mathbb{E}_\star$ over new data points removes the linear-in-$\varepsilon$ term, and we arrive at the decomposition

$$\bar{E}_{\text{new}} = \underbrace{\mathbb{E}_\star\left[\left(\bar{f}(\mathbf{x}_\star) - f_0(\mathbf{x}_\star)\right)^2\right]}_{\text{Bias}^2} + \underbrace{\mathbb{E}_\star\left[\mathbb{E}_\mathcal{T}\left[\left(\hat{y}(\mathbf{x}_\star;\mathcal{T}) - \bar{f}(\mathbf{x}_\star)\right)^2\right]\right]}_{\text{Variance}} + \underbrace{\sigma^2}_{\substack{\text{Irreducible} \\ \text{error}}} . \tag{4.19}$$

The squared bias term $\mathbb{E}_\star\left[\left(\bar{f}(\mathbf{x}_\star) - f_0(\mathbf{x}_\star)\right)^2\right]$ tells us how much the average trained model $\bar{f}(\mathbf{x}_\star)$ differs from the true $f_0(\mathbf{x}_\star)$, averaged over all possible test points. The variance term $\mathbb{E}_\star\left[\mathbb{E}_\mathcal{T}[\left(\hat{y}(\mathbf{x}_\star;\mathcal{T}) - \bar{f}(\mathbf{x}_\star)\right)^2]\right]$ tells us how much $\hat{y}(\mathbf{x};\mathcal{T})$ varies across different training sets. For the bias to be small, the model must be flexible enough for $\bar{f}(\mathbf{x})$ to match $f_0(\mathbf{x})$ - at least in regions where $p(\mathbf{x})$ is large. For the variance to be small, the model must not depend too strongly on which data points happened to be in the training set. The irreducible error $\sigma^2$ is a direct consequence of assumption (4.14): the noise $\varepsilon$ is random and cannot be predicted. There is little more to say about it, so we focus on bias and variance.

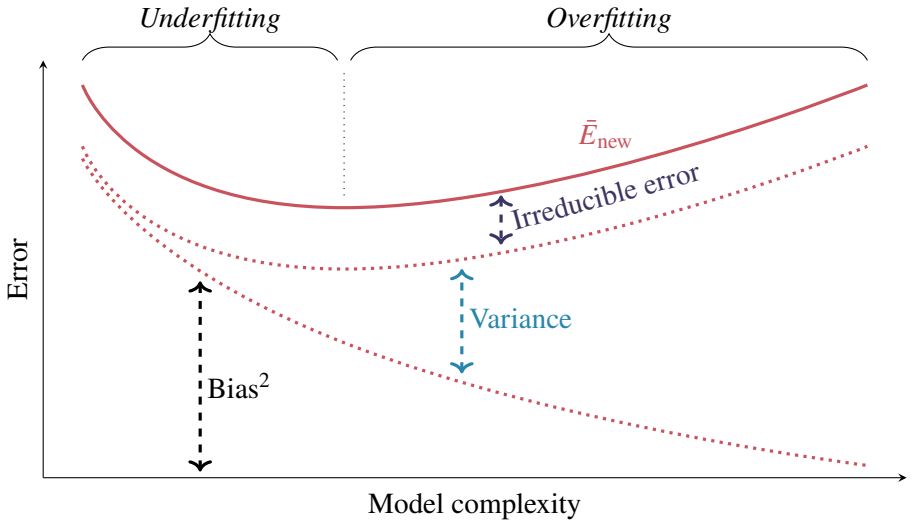
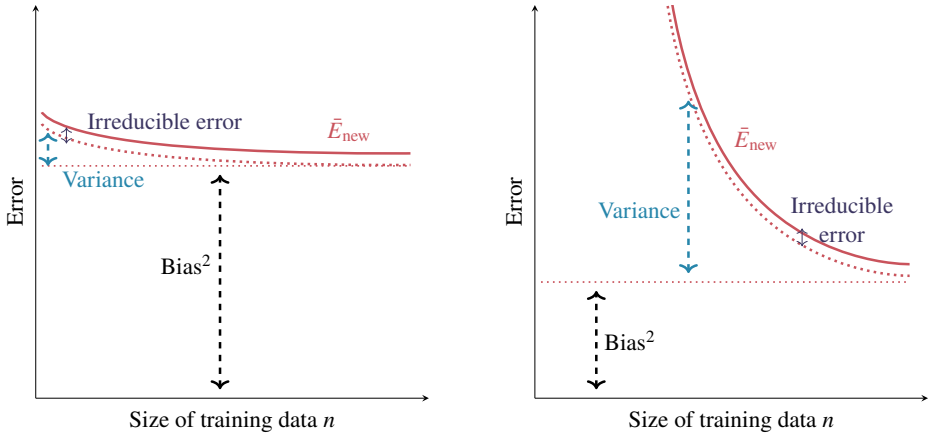



**Figure 4.9:** The bias–variance decomposition of $\bar{E}_{\text{new}}$, instead of the training error–generalisation gap decomposition in Figure 4.3. Low model complexity means high bias. The more complicated the model is, the more it adapts to (noise in) the training data, and the higher the variance. The irreducible error is independent of the particular choice of model and is therefore constant. The problem of achieving a small $E_{\text{new}}$ by selecting a suitable model complexity level is often called the bias–variance tradeoff.

## What Affects the Bias and Variance?

We have not defined model complexity formally, but bias and variance now give it a concrete meaning. High model complexity means low bias and high variance. Low model complexity means high bias and low variance. See Figure 4.9.




**(a)** Simple model **(b)** Complex model

**Figure 4.10:** The typical relationship between bias, variance, and the size $n$ of the training dataset. The bias is (approximately) constant, whereas the variance decreases as the size of the training dataset increases. This figure can be compared with Figure 4.6.

This matches intuition. The more flexible a model is, the more it adapts to $\mathcal{T}$ - not just to the interesting patterns but also to the particular data points and noise that happened to show up. That is exactly the variance. Conversely, a low-flexibility model can be too rigid to capture the true $f_0(\mathbf{x})$. That is the squared bias.

Figure 4.9 is the bias–variance counterpart of Figure 4.3, which used the training error–generalisation gap decomposition. In this picture, finding the right model complexity is called the *bias–variance tradeoff*. Example 4.3 gives an example.

The squared bias is more a property of the model than of the training set, and we may treat it as roughly independent of the number of training points $n$. The variance, in contrast, depends strongly on $n$.[3] We already know that $\bar{E}_{\text{new}}$ typically decreases as $n$ grows, and it is mostly the variance that does the shrinking. Intuitively, more data means more information about the parameters, which means a smaller variance. Figure 4.10 summarises this picture, which can be compared to Figure 4.6.

**Example 4.3 The bias–variance tradeoff for $L^2$ regularised linear regression**

Consider a simulated regression example with $p(x, y)$ given by $x \sim \mathcal{U}[0, 1]$ and

$$y = 5 - 2x + x^3 + \varepsilon, \quad \varepsilon \sim \mathcal{N}(0, 1)\,. \tag{4.20}$$

We use only $n = 10$ training points and try to model the data with a 4th-order

[3] This is not exactly true. The average model $\bar{f}$ might indeed differ for $n = 2$ and $n = 100\,000$, but we ignore that effect here.

polynomial,

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3 + \beta_4 x^4 + \varepsilon. \tag{4.21}$$

Since (4.20) is a special case of (4.21), and the squared error loss corresponds to Gaussian noise, the bias would actually be zero if we trained this model with squared error loss. But fitting 5 parameters from 10 points gives very high variance. We therefore train with squared error loss and $L^2$ regularisation, which reduces the variance at the cost of introducing some bias. More regularisation (larger $\lambda$) means more bias and less variance.

Since this is a simulated example, we can repeat the experiment and estimate the bias and variance (we can simulate as much training and test data as we need). Figure 4.11 plots them in the same style as Figures 4.3 and 4.9; note the reversed $x$-axis, where smaller $\lambda$ means higher model complexity. The optimal $\lambda$ is around 0.7, since $\bar{E}_{\text{new}}$ is minimised there. Finding this optimal $\lambda$ is a textbook bias–variance tradeoff.

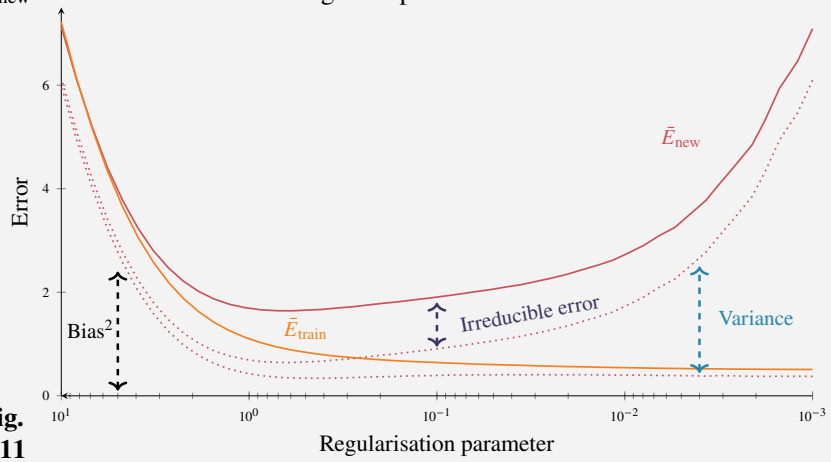



**Fig. 4.11**

## Connections Between Bias, Variance, and the Generalisation Gap

The bias and variance are well-defined in theory but hard to pin down in practice, since they are defined in terms of $p(\mathbf{x}, y)$. In practice we usually have an estimate of the generalisation gap (for example $E_{\text{hold-out}} - E_{\text{train}}$), while bias and variance require separate machinery to estimate.[4] It is therefore worth asking what $E_{\text{train}}$ and the generalisation gap tell us about bias and variance.

Consider the regression problem. Assume the squared error is used as both the

[4] Bias and variance can, to some extent, be estimated with the bootstrap, which we introduce in Chapter 8.

error function and the loss function, and that training finds the global minimum. Then

$$\sigma^2 + \text{bias}^2 = \mathbb{E}_\star\left[(\bar{f}(\mathbf{x}_\star) - y_\star)^2\right] \approx \frac{1}{n}\sum_{i=1}^{n}(\bar{f}(\mathbf{x}_i) - y_i)^2$$
$$\geq \frac{1}{n}\sum_{i=1}^{n}(\widehat{y}(\mathbf{x}_i;\mathcal{T}) - y_i)^2 = E_{\text{train}}. \tag{4.22}$$

The approximate equality is the sample-average approximation using the training data.[5] The inequality follows by assuming that $\widehat{y}$ could equal $\bar{f}$ and that training with squared-error loss always reaches the global minimum. Using $\bar{E}_{\text{new}} = \sigma^2 + \text{bias}^2 + \text{variance}$ and allowing the shorthand $\bar{E}_{\text{new}} - E_{\text{train}} = \text{generalisation gap}$, we obtain

$$\text{generalisation gap} \gtrapprox \text{variance}, \tag{4.23a}$$
$$E_{\text{train}} \lessapprox \text{bias}^2 + \sigma^2. \tag{4.23b}$$

These assumptions do not always hold in practice, but (4.23) gives a useful rough picture.

As we saw earlier, the choice of method is crucial for what $E_{\text{new}}$ we can achieve. The one-dimensional scale in Figure 4.9 and the notion of a bias–variance tradeoff are a simplification decreased bias does not *always* come with increased variance. But unlike the training error–generalisation gap decomposition, the bias–variance decomposition can shed light on *why* one method beats another: sometimes through lower bias, sometimes through lower variance.

A simple (and useless) way to increase variance in linear regression without reducing bias: solve the normal equations and add zero-mean noise to the parameters. The noise leaves the average model $\bar{f}$ unchanged, so the bias is unaffected, but the variance grows. (The training error and generalisation gap shift too, but in less obvious ways.) This is pointless in practice, since it increases $E_{\text{new}}$, but it shows that higher variance does *not* automatically mean lower bias.

A much more useful way to trade off bias and variance is the meta-method called *bagging* (Chapter 8). Bagging uses several copies, an *ensemble*, of a base model, each trained on a slightly different version of the training set. Averaging across the ensemble reduces variance while leaving the bias essentially unchanged. Switching from the base model to its bagged version thus reduces variance without much effect on bias, which at the bottom line often reduces $E_{\text{new}}$.

To sum up: the world is more complicated than the one-dimensional model complexity scale of Figure 4.3 and Figure 4.9, as Example 4.4 illustrates.

> ***Time to reflect 4.4*** *Can you modify linear regression such that the bias increases without decreasing the variance?*

[5] Since neither $\bar{f}(x_\star)$ nor $y_\star$ depends on the training data $\{\mathbf{x}_i, y_i\}_{i=1}^n$, we can use $\{\mathbf{x}_i, y_i\}_{i=1}^n$ to approximate the integral.

## Example 4.4 Bias and variance for a regression problem

We consider the exact same setting as in Example 4.2 but decompose $\bar{E}_{\text{new}}$ into bias and variance instead. This gives us Figure 4.12.

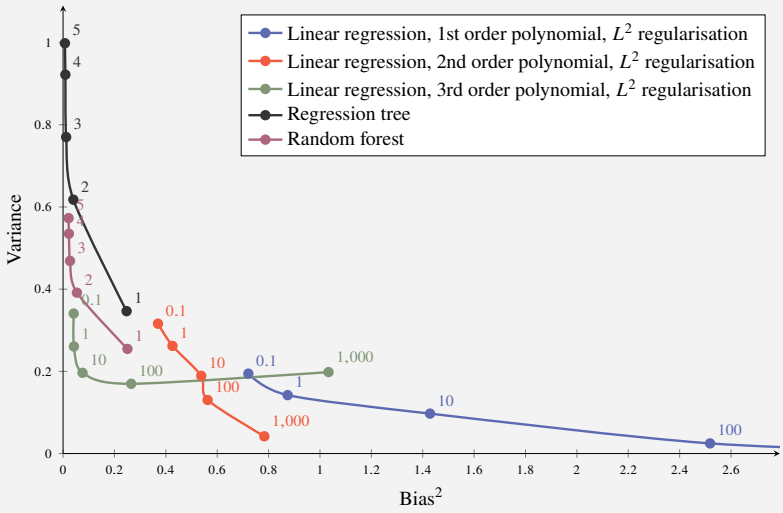
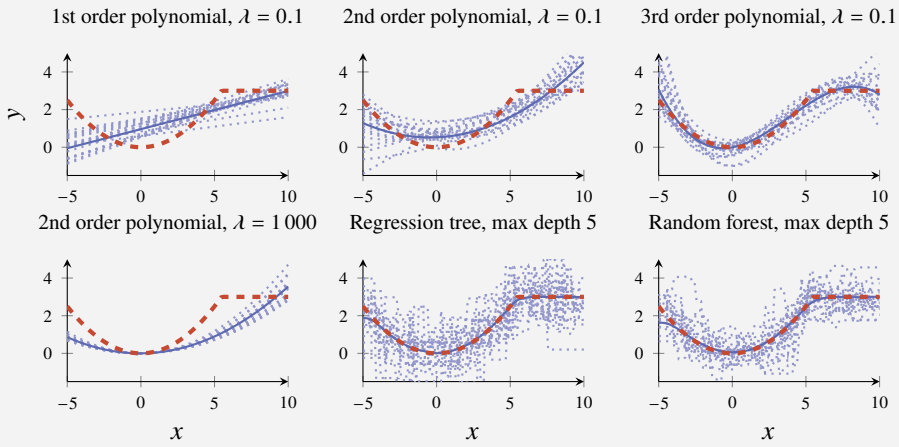



**Fig. 4.12**

The picture closely resembles Example 4.2, as expected from (4.23). The effect of bagging (in the random forest; see Chapter 8) is clearer here: it reduces the variance of the regression tree with no noteworthy increase in bias.

To further gain intuition on what bias and variance is, consider Figure 4.13. Dashed red is the true $f_0(\mathbf{x})$, dotted blue are the models $\widehat{y}(\mathbf{x}_\star; \mathcal{T})$ learned from different training sets $\mathcal{T}$, and solid blue is their mean $\bar{f}(\mathbf{x})$. Hence, bias is the gap between dashed red and solid blue, and variance is the spread of the dotted blue lines around the solid blue. The variance looks roughly the same across the three models linear regression models with same $\lambda$, maybe slightly smaller for the first-order polynomial, while the bias is clearly smaller for the higher-order polynomials. Compare to Figure 4.12.






*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September* 1, 2026, `http://smlbook.org`

**Fig. 4.13**

Comparing the second-order polynomial with light ($\lambda = 0.1$) and heavy ($\lambda = 1\,000$) regularisation, we see that regularisation reduces variance but also increases bias. The random forest has smaller variance than the regression tree, with no visible change in the solid line $\bar{f}(\mathbf{x})$, so indeed no change in bias there.

## 4.5 Additional Tools for Evaluating Binary Classifiers

For classification, and binary classification in particular, there are many ways to inspect performance beyond the misclassification rate. For simplicity we consider the binary case with a hold-out validation set. Several of the ideas extend to multiclass problems and to $k$-fold cross-validation.

Some of these tools matter especially for *imbalanced* and/or *asymmetric* problems, which we introduce later in the section. Recall that binary classification uses $y \in \{-1, 1\}$. When a binary classifier detects the presence of something (a disease, an object on the radar, and so on) the convention is that $y = 1$ (positive) means presence and $y = -1$ (negative) means absence. Much of the terminology below follows from this convention.

### The Confusion Matrix and the ROC Curve

A simple but useful way to inspect a binary classifier's performance, beyond just $E_{\text{hold-out}}$, is the *confusion matrix*. It splits the validation data into four groups according to the actual output $y$ and the prediction $\widehat{y}(\mathbf{x})$:

| | $y = -1$ | $y = 1$ | *total* |
|---|---|---|---|
| $\widehat{y}(\mathbf{x}) = -1$ | True neg (TN) | False neg (FN) | N* |
| $\widehat{y}(\mathbf{x}) = 1$ | False pos (FP) | True pos (TP) | P* |
| *total* | N | P | $n$ |

In an actual confusion matrix, TN, FN, FP, TP (and also N*, P*, N, P, $n$) are filled in with the numbers, as in Example 4.5. Here P and N are the total counts of positive and negative examples in the data, and P* and N* are the total counts of positive and negative predictions made by the model. The confusion matrix gives a quick, informative overview of a classifier's behaviour. For asymmetric problems, introduced below, we need to distinguish false positives (FP, or *type I error*) from false negatives (FN, or *type II error*). Ideally both are zero, but in practice there is a tradeoff, and the confusion matrix makes it easy to see. The tradeoff is often controlled by a *decision threshold* $r$, present in many binary classifiers (3.36).

The most natural way to summarise the performance in a single scalar value is the misclassification rate, which is the number of off-diagonal elements (that is, misclassified samples) in the confusion matrix. But there is a wide body of terminology

**Table 4.1:** Some common terms related to the quantities (TN, FN, FP, TP) in the confusion matrix. The terms written in italics are discussed in the text.

| Ratio | Name |
|---|---|
| FP/N | False positive rate, Fall-out, Probability of false alarm |
| TN/N | True negative rate, *Specificity*, Selectivity |
| TP/P | True positive rate, *Sensitivity*, Power, *Recall*, Probability of detection |
| FN/P | False negative rate, Miss rate |
| TP/P* | Positive predictive value (PPV), *Precision* |
| FP/P* | False discovery rate |
| TN/N* | Negative predictive value (NPV) |
| FN/N* | False omission rate |
| P/$n$ | Prevalence |
| (FN + FP)/$n$ | *Misclassification rate* |
| (TN + TP)/$n$ | Accuracy, 1 − misclassification rate |
| 2TP/(P* + P) | $F_1$*score* |
| $(1+\beta^2)$TP/$((1+\beta^2)$TP + $\beta^2$FN + FP) | $F_\beta$ *score* |

surrounding the confusion matrix, summarised in Table 4.1. Two particularly common terms are

$$\boxed{\textit{recall} = \frac{\text{TP}}{\text{P}} = \frac{\text{TP}}{\text{TP} + \text{FN}} \qquad \text{and the} \qquad \textit{precision} = \frac{\text{TP}}{\text{P*}} = \frac{\text{TP}}{\text{TP} + \text{FP}}.}$$

Recall is the fraction of positive data points that are correctly predicted as positive. High recall (close to 1) is good; low recall signals a problem with false negatives. Precision is the fraction of predicted-positive points that are in fact positive. High precision is good; low precision signals a problem with false positives.

Many classifiers contain a threshold $r$ (3.36). To compare different classifiers without committing to a particular $r$, the *ROC curve* is a useful tool. The name stands for 'receiver operating characteristics', a legacy of its roots in communications theory.

The ROC curve plots recall (true positive rate TP/P, larger is better) against the false positive rate (FP/N, smaller is better) for all values of $r \in [0, 1]$. A typical curve looks like Figure 4.14a. A perfect classifier (correct for all $r \in (0, 1)$) touches the upper left corner, while a classifier that just guesses randomly[6] gives the diagonal line.

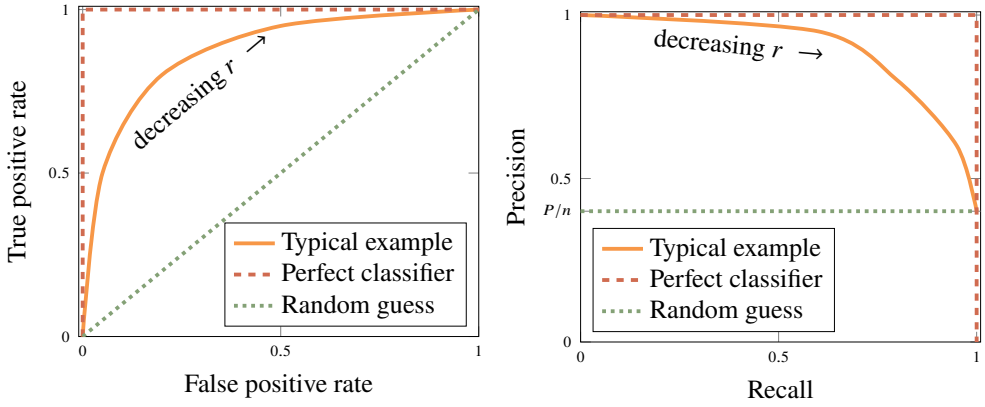
A compact summary of the ROC curve is the *area under the ROC curve, ROC-AUC*. From Figure 4.14a: a perfect classifier has ROC-AUC = 1, a random guess has ROC-AUC = 0.5. The ROC-AUC thus captures classifier performance across all decision thresholds $r$ in a single number.

### Imbalanced and Asymmetric Problems

Many binary classification problems have one or both of the following characteristics. A problem is

[6] That is, predicts $\widehat{y} = -1$ with probability $r$.


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`




**(a)** The ROC curve

**(b)** The precision–recall curve

**Figure 4.14:** The ROC (left) and the precision–recall (right) curves. Both plots summarise the performance of a classifier for *all* decision thresholds $r$ (see (3.36)), but the ROC curve is most relevant for balanced problems, whereas the precision–recall curve is more informative for imbalanced problems.

(i) *imbalanced* if the vast majority of data points belong to one class, let us say the negative class $y = -1$. A (useless) classifier that always predicts $\widehat{y}(\mathbf{x}) = -1$ then scores very well on misclassification rate (4.1a).

(ii) *asymmetric* if a false negative is considered more (or less) serious than a false positive. The misclassification rate (4.1a) ignores this asymmetry.

A typical imbalanced problem is detecting a rare disease. Most patients do not have it, so most data points are negative. This problem is also asymmetric if missing an infected patient is considered more serious than the opposite.

The imbalance adds to the complexity of evaluating models. One approach is to evaluate how good a model is at identifying the positive and negative class in isolation, which can be done by using the *specificity* and *sensitivity* (another name for recall). This is common in medicine, because it leaves the imbalance problem aside and makes it also possible to compare a method across different diseases which might have different levels of imbalance.

However, for ultra rare diseases, the number of patients that are falsely flagged as infected might easily become larger than the number of truly infected patients, even if the specificity is very high. For that reason, the positive predictive value (PPV), another name for precision, is also considered important in such cases.

As a single summary metric, the misclassification rate is typically problematic for imbalanced problems. Instead, the $F_1$ score is often preferred. It combines precision and recall into their harmonic mean,

$$F_1 = \frac{2 \cdot \mathit{precision} \cdot \mathit{recall}}{\mathit{precision} + \mathit{recall}}, \tag{4.24}$$

which lies between zero and one (higher is better). The effect of having the harmonic mean, instead of the arithmetic mean, is that the $F_1$ score is high only if both the precision/PPV *and* the recall/sensitivity both are high. If precision = 0.5 and recall = 0.5 then $F_1 = 0.5$, but precision = 0.1 and recall = 0.9 only gives $F_1 = 0.18$.

For asymmetric problems the $F_1$ score is not enough, since it does not factor in the relative severity of false negatives versus false positives. The $F_\beta$ score generalises $F_1$ by treating recall as $\beta$ times as important as precision:

$$F_\beta = \frac{(1+\beta^2)\mathit{precision} \cdot \mathit{recall}}{\beta^2 \cdot \mathit{precision} + \mathit{recall}}. \tag{4.25}$$

Just as the misclassification rate can mislead for imbalanced problems, so can the ROC curve. For imbalanced problems with $y = -1$ as the majority class, the *precision–recall curve* is often more informative. It plots precision (TP/P*, larger is better) against recall (TP/P, larger is better) for all values of $r \in [0, 1]$, analogously to the ROC curve. The precision–recall curve for a perfect classifier touches the upper right corner; a random-guess classifier gives a horizontal line at $P/n$, as shown in Figure 4.14b.

The *area under the precision–recall curve, PR-AUC* serves the same summarising role. The best possible PR-AUC is 1, and random guessing yields $P/n$.

We close this section with an example of a problem that is both imbalanced and asymmetric - a situation from medicine. Evaluating real-world classification problems is a rich topic with several dilemmas, which we revisit in Chapter 14.

**Example 4.5 The confusion matrix in thyroid disease detection**

The thyroid is an endocrine gland. Its hormones regulate the metabolic rate and protein synthesis, and thyroid disorders can be serious. We consider the problem of detecting thyroid disease using the dataset from the UCI Machine Learning Repository (Dheeru and Karra Taniskidou 2017). The dataset has 7 200 data points, each with 21 medical indicators (qualitative and quantitative) as inputs, plus the qualitative diagnosis {`normal`, `hyperthyroid`, `hypothyroid`}. For simplicity we reduce this to a binary problem with classes {`normal`, `abnormal`} and split the data into a training set of 3 772 points and a hold-out validation set of 3 428 points. Only 7% of data points are `abnormal`, so the problem is imbalanced. The naive predictor that always says `normal` therefore has a misclassification rate around 7%. The problem may also be asymmetric if missing the disease (a false negative) is considered more serious than a false alarm (a false positive). We train a logistic regression classifier and evaluate it on the validation set using the default threshold $r = 0.5$ (see (3.36)). The confusion matrix is

| | $y$ = `normal` | $y$ = `abnormal` |
|---|---|---|
| $\hat{y}(\mathbf{x})$ = `normal` | 3 177 | 237 |
| $\hat{y}(\mathbf{x})$ = `abnormal` | 1 | 13 |

Most validation points are correctly predicted as `normal`, but a large share of the `abnormal` data is also predicted as `normal`. In a medical application that is undesirable. The accuracy is 0.931 and the $F_1$ score is 0.106. (The useless always-`normal` predictor


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026, http://smlbook.org*

has a nearly identical accuracy of 0.927, but an $F_1$ score of 0.)

To change the picture, we lower the threshold to $r = 0.15$ in (3.36). That is, we predict `abnormal` whenever $g(\mathbf{x}) > 0.15$. The confusion matrix becomes

| | $y$ = `normal` | $y$ = `abnormal` |
|---|---|---|
| $\widehat{y}$ = `normal` | 3 067 | 165 |
| $\widehat{y}$ = `abnormal` | 111 | 85 |

We now have many more true positives (85 vs. 13 patients correctly flagged as `abnormal`) at the cost of more false positives (111 vs. 1 patients incorrectly flagged). Accuracy drops to 0.919, but the $F_1$ score rises to 0.381. The $F_1$ score captures the imbalance but not the asymmetry - whether this tradeoff between false negatives and false positives is good has to be judged by considering which type of error has the more serious consequences.

## 4.6 Further Reading

This chapter is partly inspired by the introductory textbook by Abu-Mostafa et al. (2012). Other textbooks on machine learning, including Vapnik (2000) and Mohri et al. (2018), make the generalisation gap their central theme, using formal notions of model flexibility like VC dimension or Rademacher complexity.

A friendly yet technical introduction to the double descent topic for the overparameterised regime is Schaeffer et al. (2024), and important parts of the unerlying research is found inis Chiyuan Zhang et al. (2017), Neyshabur et al. (2017), Belkin et al. (2019) and Nakkiran et al. (2020).

The bias–variance decomposition is typically presented for regression only (as in this chapter), but a generalisation to classification is given by Domingos (2000). An alternative to the precision–recall curve, the precision–recall-gain curve, is introduced by Flach and Kull (2015).

# 5 Learning Parametric Models

In this chapter, we elaborate on the concept of parametric modelling. We start by generalising the notion of a parametric model and outline basic principles for learning these models from data. The chapter then resolves around three central concepts, namely loss functions, regularisation, and optimisation. We have touched upon all of them already, mostly in connection with parametric models in Chapter 3, linear and logistic regression. These topics are, however, central to many supervised machine learning methods, in fact even beyond parametric models, and deserve a more elaborate discussion.

## 5.1 Principles of Parametric Modelling

In Chapter 3, we introduced two basic parametric models for regression and classification, linear regression and logistic regression, respectively. We also briefly discussed how generalised linear models could be used to handle different types of data. The concept of parametric modelling is, however, not restricted to these cases. We therefore start this chapter by introducing a general framework for parametric modelling and discuss basic principles for learning these models from data.

### Non-linear Parametric Functions

Consider the regression model (3.1), repeated here for convenience:

$$y = f_{\boldsymbol{\theta}}(\mathbf{x}) + \varepsilon. \tag{5.1}$$

Here we have introduced an explicit dependence on the parameters $\boldsymbol{\theta}$ in the notation to emphasise that the equation above should be viewed as our *model* of the true input–output relationship. To turn this model into a linear regression model, which could be trained using least squares with a closed form solution, we made two assumptions in Chapter 3. First, the function $f_{\boldsymbol{\theta}}$ was assumed to be linear in the model parameters, $f_{\boldsymbol{\theta}}(\mathbf{x}) = \boldsymbol{\theta}^{\mathsf{T}}\mathbf{x}$. Second, the noise term $\varepsilon$ was assumed to be Gaussian, $\varepsilon \sim \mathcal{N}(0, \sigma_\varepsilon^2)$. The latter assumption is sometimes implicit, but as we saw in Chapter 3, it makes the maximum likelihood formulation equivalent to least squares.

Both of these assumptions can be relaxed. Based on the expression above, the perhaps most obvious generalisation is to allow the function $f_{\boldsymbol{\theta}}$ to be some arbitrary non-linear function. Since we still want the function to be learnt from training data, we require that it is adaptable. Similarly to the linear case, this can be accomplished

by letting the function depend on some model parameters $\boldsymbol{\theta}$ that control the shape of the function. Different values of the model parameters will then result in different functions $f_{\boldsymbol{\theta}}(\cdot)$. Training the model amounts to finding a suitable value for the parameter vector $\boldsymbol{\theta}$, such that the function $f_{\boldsymbol{\theta}}$ accurately describes the true input–output relationship. In mathematical terms, we say that we have a *parametric family* of functions

$$\{f_{\boldsymbol{\theta}}(\cdot) : \boldsymbol{\theta} \in \Theta\},$$

where $\Theta$ is the space containing all possible parameter vectors. We illustrate with an example:

**Example 5.1 Michaelis–Menten kinetics**

A simple example of a non-linear parametric function is the Michaelis–Menten equation for modelling enzyme kinetics. The model is given by

$$y = \underbrace{\frac{\theta_1 x}{\theta_2 + x}}_{=f_{\boldsymbol{\theta}}(x)} + \varepsilon,$$

where $y$ corresponds to a reaction rate and $x$ a substrate concentration. The model is parameterised by the maximum reaction rate $\theta_1 > 0$ and the so-called Michaelis constant of the enzyme $\theta_2 > 0$. Note that $f_{\boldsymbol{\theta}}(x)$ depends non-linearly on the parameter $\theta_2$ appearing in the denominator.

Typically the model is written as a deterministic relationship without the noise term $\varepsilon$, but here we include the noise as an error term for consistency with our statistical regression framework.

In the example above, the parameters $\theta_1$ and $\theta_2$ have physical interpretations and are restricted to be positive. Thus, $\Theta$ corresponds to the positive quadrant in $\mathbb{R}^2$. However, in machine learning we typically lack such physical interpretations of the parameters. The model is more of a 'black box' which is adapted to fit the training data as well as possible. For simplicity, we will therefore assume that $\Theta = \mathbb{R}^d$, meaning that $\boldsymbol{\theta}$ is a $d$-dimensional vector of real-valued parameters. The archetypes of such non-linear black-box models are neural networks, which we will discuss in more detail in Chapter 6. If we need to restrict a parameter value in some way, for example to be positive, then this can be accomplished by a suitable transformation of that parameter. For instance, in the Michaelis–Menten equation, we can replace $\theta_1$ and $\theta_2$ with $\exp(\theta_1)$ and $\exp(\theta_2)$, respectively, where the parameters are now allowed to take arbitrary real values.

Note that the likelihood corresponding to the model (5.1) is governed by the noise term $\varepsilon$. As long as we stick with the assumption that the noise is additive and Gaussian with zero mean and variance $\sigma_\varepsilon^2$, we obtain a Gaussian likelihood function

$$p(y \mid \mathbf{x}; \boldsymbol{\theta}) = \mathcal{N}\left(f_{\boldsymbol{\theta}}(\mathbf{x}), \sigma_\varepsilon^2\right). \tag{5.2}$$


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

The only difference between this expression and the likelihood used in the linear regression model (3.18) is that the mean of the Gaussian distribution is now given by the arbitrary non-linear function $f_{\boldsymbol{\theta}}(\mathbf{x})$.

Non-linear classification models can be constructed in a very similar way, as a generalisation of the logistic regression model (3.29). In binary logistic regression, we first compute the logit $z = \boldsymbol{\theta}^\mathsf{T}\mathbf{x}$. The probability of the positive class, that is $p(y = 1 \mid \mathbf{x})$, is then obtained by mapping the logit value through the logistic function, $h(z) = \frac{e^z}{1+e^z}$. To turn this into a non-linear classification model, we can simply replace the expression for the logit with $z = f_{\boldsymbol{\theta}}(\mathbf{x})$ for some arbitrary real-valued non-linear function $f_{\boldsymbol{\theta}}$. Hence, the non-linear logistic regression model becomes

$$g(\mathbf{x}) = \frac{e^{f_{\boldsymbol{\theta}}(\mathbf{x})}}{1 + e^{f_{\boldsymbol{\theta}}(\mathbf{x})}}. \tag{5.3}$$

Analogously, we can construct a multiclass non-linear classifier by generalising the multiclass logistic regression model (3.42). That is, we compute a vector of logits $\mathbf{z} = [z_1 \;\; z_2 \;\; \ldots \;\; z_M]^\mathsf{T}$ according to $\mathbf{z} = \mathbf{f}_{\boldsymbol{\theta}}(\mathbf{x})$, where $\mathbf{f}_{\boldsymbol{\theta}}$ is some arbitrary function that maps $\mathbf{x}$ to an $M$-dimensional real-valued vector $\mathbf{z}$. Propagating this logit vector through the softmax function results in a non-linear model for the conditional class probabilities, $\mathbf{g}_{\boldsymbol{\theta}}(\mathbf{x}) = \text{softmax}(\mathbf{f}_{\boldsymbol{\theta}}(\mathbf{x}))$. We will return to non-linear classification models of this form in Chapter 6, where we use neural networks to construct the function $\mathbf{f}_{\boldsymbol{\theta}}$.

### Loss Minimisation as a Proxy for Generalisation

Having specified a certain model class – that is, a parametric family of functions defining the model – learning amounts to finding suitable values for the parameters so that the model as accurately as possible describes the actual input–output relationship. For parametric models, this learning objective is typically formulated as an optimisation problem, such as

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \underbrace{\frac{1}{n}\sum_{i=1}^{n} \overbrace{L(y_i, f_{\boldsymbol{\theta}}(\mathbf{x}_i))}^{\text{loss function}}}_{\text{cost function } J(\boldsymbol{\theta})}. \tag{5.4}$$

That is, we seek to minimise a cost function defined as the average of some (user-chosen) loss function $L$ evaluated on the training data. In some special cases (such as linear regression with squared loss), we can compute the the solution to this optimisation problem exactly. However, in most cases, and in particular when working with non-linear parametric models, this is not possible, and we need to resort to *numerical optimisation*. We will discuss such algorithms in more detail in Section 5.4, but it is useful to note already now that these optimisation algorithms are often iterative. That is, the algorithm is run over many iterations, and at each iteration, the current approximate solution to the optimisation problem (5.4) is updated to a new (hopefully better) approximate solution. This leads to a computational trade-off. The longer we

run the algorithm, the better solution we expect to find, but at the cost of a longer training time.

Finding the value of $\boldsymbol{\theta}$ which is such that the model fits the training data as well as possible is a natural idea. However, as we discussed in the previous chapter, the ultimate goal of machine learning is *not* to fit the training data as well as possible but rather to find a model that can *generalise to new data*, not used for training the model. Put differently, the problem that we are actually interested in solving is not (5.4) but rather

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} E_{\text{new}}(\boldsymbol{\theta}), \tag{5.5}$$

where $E_{\text{new}}(\boldsymbol{\theta}) = \mathbb{E}_{\star}\left[E(\widehat{y}(\mathbf{x}_{\star}; \boldsymbol{\theta}), y_{\star})\right]$ is the expected new data error (for some error function $E$ of interest; see Chapter 4). The issue is, of course, that the expected new data error is unknown to us. The 'true' data generating distribution is not available, and we thus cannot compute the expected error with respect to new data, nor can we optimise the objective (5.5) explicitly. However, this insight is still of practical importance, because it means that

| the training objective (5.4) is only a proxy for the actual objective of interest, (5.5). |
|---|

This view of the training objective as a proxy has implications for how we approach the optimisation problem (5.4) in practice. We make the following observations.

**Optimisation accuracy vs. statistical accuracy:** The cost function $J(\boldsymbol{\theta})$ is computed based on the training data and is thus subject to noise in the data. It can be viewed as a random approximation of the 'true' expected loss (obtained as $n \to \infty$). Hence, it is not meaningful to *optimise $J(\boldsymbol{\theta})$ with greater accuracy than the statistical error in the estimate.* This is particularly relevant in situations when we need to spend a lot of computational effort to obtain a very accurate solution to the optimisation problem. This is unnecessary as long as we are within the statistical accuracy of the estimate. In practice, however, it can be difficult to determine what the statistical accuracy is (and we will not elaborate on methods that can be used for estimating it in this book), but this trade-off between optimisation accuracy and statistical accuracy is still useful to have in the back of the mind.

**Loss function ≠ error function** As discussed in Chapter 4, we can use an error function $E$ for evaluating the performance of a model, which is different from the loss function $L$ used during training. In words, when training the model, we minimise an objective which is different from the one that we are actually interested in. This might seem counter-intuitive, but based on the proxy view on the training objective, it makes perfect sense and, in fact, provides the machine learning engineer with additional flexibility in designing a useful


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

training objective. There are many reasons why we might want to use a loss function that is different from the error function. First, it can result in a model which is expected to generalise better. The typical example is when evaluating a classification model based on accuracy (equivalently, misclassification error). If we were to train the model by minimising the misclassification loss, we would only care about placing the decision boundaries to get as many training data points as possible on the right side, but without taking the distances to the decision boundaries into account. However, due to the noise in the data, having some *margin* to the decision boundary can result in better generalisation, and there are loss functions that explicitly encourage this. Second, we can choose the loss function with the aim of making the optimisation problem (5.4) easier to solve, for instance by using a convex loss function (see Section 5.4). Third, certain loss functions can encourage other favorable properties of the final model, such as making the model less computationally demanding to use 'in production'.

**Early stopping** When optimising the objective (5.4) using an *iterative numerical optimisation method*, this can be thought of as generating a sequence of candidate models. At each iteration of the optimsation algorithm, we have access to the current estimate of the parameter vector, and we thus obtain a 'path' of parameter values. Interestingly, it is not necessarily the end point of this path that is closest to the solution of (5.5). Indeed, the path of parameter values can pass a useful solution (with good generalisation properties) before drifting off to a worse solution (for example due to overfitting). Based on this observation, there is another reason for stopping the optimisation algorithm prematurely, apart from the purely computational reason mentioned above. By *early stopping* of the algorithm, we can obtain a final model with superior performance to the one we would obtain if we ran the algorithm until convergence. We refer to this as *implicit regularisation* and discuss the details of how it can be implemented in practice in Section 5.3.

**Explicit regularisation** Another strategy is to explicitly modify the cost function (5.4) by adding a term independent of the training data. We refer to this technique as explicit regularisation. The aim is to make the final model generalise better – that is, we hope to make the solution to the modified problem closer to the solution of (5.5). We saw an example of this already in Section 3.3 where we introduced $L^2$ regularisation. The underlying idea is the law of parsimony: that the simplest explanation of an observed phenomenon is usually the right one. In the context of machine learning, this means that if both a 'simple' and a 'complicated' model fit the data (more or less) equally well, then the 'simple' one will typically have superior generalisation properties and should therefore be preferred. In explicit regularisation the vague notions of 'simple'

and 'complicated' are reduced to simply mean small and large parameter values, respectively. In order to favour a simple model, an extra term is added to the cost function that penalises large parameter values. We will discuss regularisation further in Section 5.3.

## 5.2 Loss Functions and Likelihood-Based Models

Which loss function $L$ to use in the training objective (5.4) is a design choice, and different loss functions will give rise to different solutions $\widehat{\boldsymbol{\theta}}$. This will in turn result in models with different characteristics. There is in general no 'right' or 'wrong' loss function, but for a given problem, one particular choice can be superior to another in terms of small $E_{\text{new}}$ (note that there is a similar design choice involved in $E_{\text{new}}$, namely how to choose the error function which defines how we measure the performance of the model). Certain combinations of models and loss functions have proven particularly useful and have historically been branded as specific methods. For example, the term 'linear regression' most often refers to the combination of a linear-in-the-parameter model and the squared error loss, whereas the term 'support vector classification' (see Chapter 9) refers to a linear-in-the-parameter model trained using the hinge loss. In this section, however, we provide a general discussion about different loss functions and their properties, without connections to a specific method.

One important aspect of a loss function is its *robustness*. Robustness is tightly connected to outliers, meaning spurious data points that do not describe the relationship we are interested in modelling. If outliers in the training data only have a minor impact on the learned model, we say that the loss function is robust. Conversely, a loss function is not robust if the outliers have a major impact on the learned model. Robustness is therefore a very important property in applications where the training data is contaminated with outliers. It is not a binary property, however, and loss functions can be robust to a greater or lesser extent. Some of the commonly used loss functions, including the squared error loss, are unfortunately not particularly robust, and it is therefore important for the user to make an active and informed decision before resorting to these 'default' options.

From a statistical perspective, we can link the loss function to statistical properties of the learnt model. First, the maximum likelihood approach provides a formal connection between the loss function and the probabilistic assumptions on the (noise in the) data. Second, even for loss functions that are not derived from a likelihood perspective, we can relate the so-called *asymptotic minimiser* of the loss function to the statistical properties of the model. The asymptotic minimiser refers to the model that minimises the expected loss when averaged over the true data-generating distribution. Equivalently, we can think about the asymptotic minimiser as the solution to the optimisation problem (5.4) as the number of training data points $n \to \infty$ (hence the name 'asymptotic'). If there is a unique asymptotic minimiser from which we can recover the true conditional distribution $p(y \mid \mathbf{x})$, then the loss function is said to be *strictly proper*. We will return to this concept below, specifically in the context of


Machine Learning – A First Course for Engineers and Scientists
Second Edition, online draft version September 1, 2026, http://smlbook.org

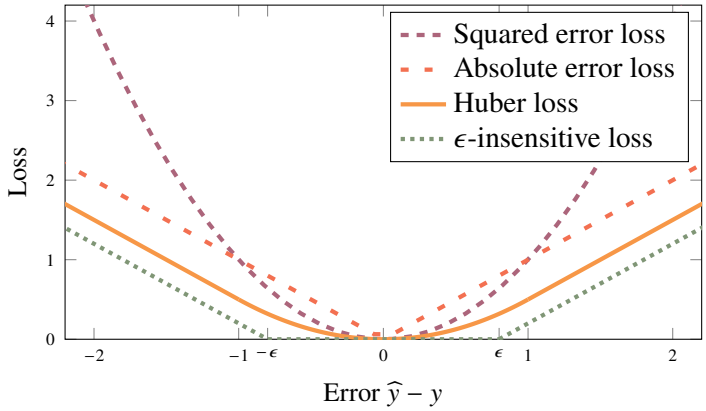



**Figure 5.1:** The loss functions for regression presented in the text, each as a function of the error $\widehat{y} - y$.

binary classification.

## Loss Functions for Regression

In Chapter 3 we introduced the *squared error loss*[1]

$$L(y, \widehat{y}) = \big(\widehat{y} - y\big)^2, \tag{5.6}$$

which is the default choice for linear regression since it simplifies the training to only solving the normal equations. The squared error loss is often also used for other regression models, such as neural networks. Another common choice is the *absolute error loss*,

$$L(y, \widehat{y}) = |\widehat{y} - y|. \tag{5.7}$$

The absolute error loss is more robust to outliers than the squared error loss since it grows more slowly for large errors; see Figure 5.1. In Chapter 3 we introduced the maximum likelihood motivation of the squared error loss by assuming that the output $y$ is measured with additive noise $\varepsilon$ from a Gaussian distribution, $\varepsilon \sim \mathcal{N}\big(0, \sigma_\varepsilon^2\big)$. We can similarly motivate the absolute error loss by instead assuming $\varepsilon$ to have a Laplace distribution, $\varepsilon \sim \mathcal{L}(0, b_\varepsilon)$. We elaborate and expand on the idea of deriving the loss function from the maximum likelihood objective and certain statistical modelling assumptions below. However, there are also some commonly used loss functions for regression which are not very natural to derive from a maximum likelihood perspective.

It is sometimes argued that the squared error loss is a good choice because of its quadratic shape, which penalises small errors ($\varepsilon < 1$) less than linearly. After all, the

[1] As you might already have noticed, the arguments to the loss function (here $y$ and $\widehat{y}$) vary with context. The reason for this is that different loss functions are most naturally expressed in terms of different quantities, for example the prediction $\widehat{y}$, the predicted conditional class probability $g(\mathbf{x})$, the classifier margin, etc.

Gaussian distribution appears (at least approximately) quite often in nature. However, the quadratic shape for large errors ($\varepsilon > 1$) is the reason for its non-robustness, and the *Huber loss* has therefore been suggested as a hybrid between the absolute loss and squared error loss:

$$L(y, \widehat{y}) = \begin{cases} \frac{1}{2}(\widehat{y} - y)^2 & \text{if } |\widehat{y} - y| < 1, \\ |\widehat{y} - y| - \frac{1}{2} & \text{otherwise.} \end{cases} \tag{5.8}$$

Another extension to the absolute error loss is the *$\epsilon$-insensitive loss*,

$$L(y, \widehat{y}) = \begin{cases} 0 & \text{if } |\widehat{y} - y| < \epsilon, \\ |\widehat{y} - y| - \epsilon & \text{otherwise,} \end{cases} \tag{5.9}$$

where $\epsilon$ is a user-chosen design parameter. This loss places a tolerance of width $2\epsilon$ around the observed $y$ and behaves like the absolute error loss outside this region. The robustness properties of the $\epsilon$-insensitive loss are very similar to those of the absolute error loss. The $\epsilon$-insensitive loss turns out to be useful for support vector regression in Chapter 9. We illustrate all these loss functions for regression in Figure 5.1.

### Loss Functions for Binary Classification

An intuitive loss function for binary classification is provided by the *misclassification loss*,

$$L(y, \widehat{y}) = \mathbb{I}\{\widehat{y} \neq y\} = \begin{cases} 0 & \text{if } \widehat{y} = y, \\ 1 & \text{if } \widehat{y} \neq y. \end{cases} \tag{5.10}$$

However, even though a small misclassification loss might be the ultimate goal in practice, this loss function is rarely used when training models. As mentioned in Section 5.1, there are at least two reasons for this. First, using a different loss function can result in a model that generalises better from the training data. This can be understood by noting that the final prediction $\widehat{y}$ does not reveal all aspects of the classifier. Intuitively, we may prefer to not have the decision boundary close to the training data points, even if they are correctly classified, but instead push the boundary further away to have some margin. To achieve this, we can formulate the loss function not just in terms of the hard class prediction $\widehat{y}$ but based on the predicted class probability $g(\mathbf{x})$ or some other continuous quantity used to compute the class prediction. The second reason for not using misclassification loss as the training objective, which is also important, is that it would result in a cost function that is piecewise constant. From a numerical optimisation perspective, this is a difficult objective since the gradient is zero everywhere, except where it is undefined.

For a binary classifier that predicts conditional class probabilities $p(y = 1 \,|\, \mathbf{x})$ in terms of a function $g(\mathbf{x})$, the *cross-entropy loss*, as introduced in Chapter 3, is a natural choice:

$$L(y, g(\mathbf{x})) = \begin{cases} \ln g(\mathbf{x}) & \text{if } y = 1, \\ \ln(1 - g(\mathbf{x})) & \text{if } y = -1. \end{cases} \tag{5.11}$$


Machine Learning – A First Course for Engineers and Scientists
Second Edition, online draft version September 1, 2026, http://smlbook.org

This loss was derived from a maximum likelihood perspective, but unlike regression (where we had to specify a distribution for $\varepsilon$), there are no user choices left in the cross-entropy loss, other than what model to use for $g(\mathbf{x})$. Indeed, for a binary classification problem, the model $g(\mathbf{x})$ provides a complete statistical description of the conditional distribution of the output given the input.

While cross entropy is commonly used in practice, there are also other loss functions for binary classification that are useful. To define an entire family of loss functions, let us first introduce the concept of *margins* in binary classification. Many binary classifiers $\widehat{y}(\mathbf{x})$ can be constructed by thresholding some real-valued function[2] $f(\mathbf{x})$ at 0. That is, we can write the class prediction

$$\widehat{y}(\mathbf{x}) = \operatorname{sign}\{f(\mathbf{x})\}. \tag{5.12}$$

Logistic regression, for example, can be brought into this form by simply using $f(\mathbf{x}) = \boldsymbol{\theta}^\mathsf{T}\mathbf{x}$, as shown in (3.39). More generally, for a non-linear generalisation of the logistic regression model where the probability of the positive class is modelled as in (5.3), the prediction (5.12) corresponds to the most likely class. Not all classifiers have a probabilistic interpretation, however, but often they can still be expressed as in (5.12) for some underlying function $f(\mathbf{x})$.

The decision boundary for any classifier of the form (5.12) is given by the values of $\mathbf{x}$ for which $f(\mathbf{x}) = 0$. To simplify our discussion, we will assume that none of the data points fall exactly on the decision boundary (which always gives rise to an ambiguity). This will imply that we can assume that $\widehat{y}(\mathbf{x})$ as defined above is always either $-1$ or $+1$. Based on the function $f(\mathbf{x})$, we say that

| the margin of a classifier for a data point $(\mathbf{x}, y)$ is $y \cdot f(\mathbf{x})$. |
|---|

It follows that if $y$ and $f(\mathbf{x})$ have the same sign, meaning that the classification is correct, then the margin is positive. Analogously, for an incorrect classification, $y$ and $f(\mathbf{x})$ will have different signs, and the margin will be negative. The margin can be viewed as a measure of certainty in a prediction, where data points with small margins are in some sense (not necessarily Euclidean) close to the decision boundary. The margin plays a similar role for binary classification as the prediction error $\widehat{y} - y$ does for regression.

We can now define loss functions for binary classification in terms of the margin, by assigning a small loss to positive margins (correct classifications) and a large loss to negative margins (misclassifications). We can, for instance, re-formulate the logistic loss (3.34) in terms of the margin as

$$L(y \cdot f(\mathbf{x})) = \ln\left(1 + \exp\left(-y \cdot f(\mathbf{x})\right)\right), \tag{5.13}$$

where, in line with the discussion above, the linear logistic regression model corresponds to $f(\mathbf{x}) = \theta^\mathsf{T}\mathbf{x}$. Analogously to the derivation of the logistic loss in Chapter 3,

[2] In general, the function $f(\mathbf{x})$ depends on the model parameters $\boldsymbol{\theta}$, but in the presentation below we will drop this dependence from the notation for brevity.

this is just another way of writing the cross entropy (or negative log-likelihood) loss (5.11), assuming that the probability of the positive class is modelled according to (5.3). However, from an alternative point of view, we can consider (5.13) as a generic margin-based loss, without linking it to the probabilistic model (5.3). That is, we simply postulate a classifier according to (5.12) and learn the parameters of $f(\mathbf{x})$ by minimising (5.13). This is, of course, equivalent to logistic regression, except for the fact that we have seemingly lost the notion of a conditional class probability estimate $g(\mathbf{x})$ and only have a 'hard' prediction $\widehat{y}(\mathbf{x})$. We will, however, recover the class probability estimate later when we discuss the asymptotic minimiser of the logistic loss.

We can also re-formulate the misclassification loss in terms of the margin,

$$L(y \cdot f(\mathbf{x})) = \begin{cases} 1 & \text{if } y \cdot f(\mathbf{x}) < 0, \\ 0 & \text{otherwise.} \end{cases} \tag{5.14}$$

More important, however, is that the margin view allows us to easily come up with other loss functions with possibly favourable properties. In principle, any decreasing function is a candidate loss function. However, most loss functions used in practice are also convex, which is useful when optimising the training loss numerically.

One example is the *exponential loss*, defined as

$$L(y \cdot f(\mathbf{x})) = \exp(-y \cdot f(\mathbf{x})), \tag{5.15}$$

which turns out to be a useful loss function when we derive the AdaBoost algorithm in Chapter 8. The downside of the exponential loss is that it is not particularly robust against outliers, due to its exponential growth for negative margins, compared to, for example, the linear asymptotic growth of the logistic loss.[3] We also have the *hinge loss*, which we will use for support vector classification in Chapter 9:

$$L(y \cdot f(\mathbf{x})) = \begin{cases} 1 - y \cdot f(\mathbf{x}) & \text{for } y \cdot f(\mathbf{x}) \leq 1, \\ 0 & \text{otherwise.} \end{cases} \tag{5.16}$$

As we will see in Chapter 9, the hinge loss has an attractive so-called support vector property. However, a downside of the hinge loss is that it is not a *strictly proper* loss function, which means that it is not possible to interpret the learnt classification model probabilistically when using this loss (we elaborate on this below). As a remedy, one may instead consider the *squared hinge loss*

$$L(y \cdot f(\mathbf{x})) = \begin{cases} (1 - y \cdot f(\mathbf{x}))^2 & \text{for } y \cdot f(\mathbf{x}) \leq 1, \\ 0 & \text{otherwise,} \end{cases} \tag{5.17}$$

[3] For $yf(\mathbf{x}) \ll 0$, it holds that $\ln(1 + \exp(-yf(\mathbf{x}))) \approx -yf(\mathbf{x})$.


Machine Learning – A First Course for Engineers and Scientists
Second Edition, online draft version September 1, 2026, http://smlbook.org

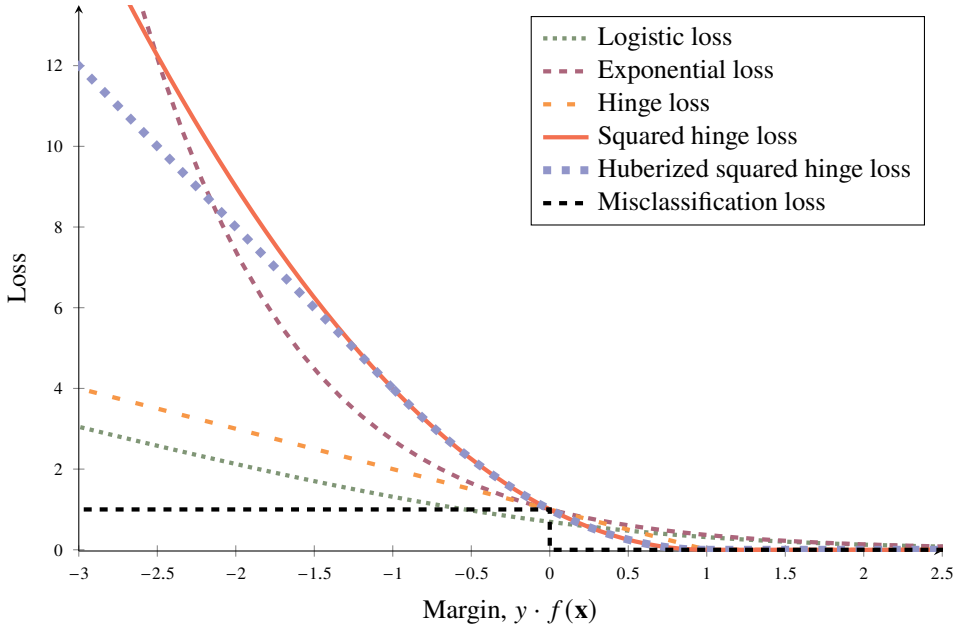



**Figure 5.2:** Comparison of some common loss functions for classification, plotted as a function of the margin.

which on the other hand is less robust than the hinge loss (quadratic instead of linear growth). A more elaborate alternative is therefore the *Huberised squared hinge* loss

$$L(y \cdot f(\mathbf{x})) = \begin{cases} -4y \cdot f(\mathbf{x}) & \text{for } y \cdot f(\mathbf{x}) \leq -1, \\ (1 - y \cdot f(\mathbf{x}))^2 & \text{for } -1 \leq y \cdot f(\mathbf{x}) \leq 1, \quad \text{(squared hinge loss)} \\ 0 & \text{otherwise,} \end{cases} \tag{5.18}$$

whose name refers to its similarities to the Huber loss for regression, namely that the quadratic function is replaced with a linear function for margins $< -1$. The three loss functions presented above are all particularly interesting for support vector classification, due to the fact that they are all exactly 0 for margins $>1$.

We summarise this cascade of loss functions for binary classification in Figure 5.2, which illustrates all these losses as a function of the margin.

When learning models for imbalanced or asymmetric problems, it is possible to modify the loss function to account for imbalance or asymmetry. For example, to reflect that not predicting $y = 1$ correctly is a '$C$ times more severe mistake' than not predicting $y = -1$ correctly, the misclassification loss can be modified into

$$L(y, \widehat{y}) = \begin{cases} 0 & \text{if } \widehat{y} = y, \\ 1 & \text{if } \widehat{y} \neq y \text{ and } y = -1, \\ C & \text{if } \widehat{y} \neq y \text{ and } y = 1. \end{cases} \tag{5.19}$$

The other loss functions can be modified in a similar fashion. A similar effect can also be achieved by, for this example, simply duplicating all positive training data points $C$ times in the training data, instead of modifying the loss function.

We have already made some claims about robustness. Let us motivate them using Figure 5.2. One characterisation of an outlier is as a data point on the wrong side of and far away from the decision boundary. From a margin perspective, that is equivalent to a large negative margin. The robustness of a loss function is therefore tightly connected to the shape of the loss function for large negative margins. The steeper the slope and heavier the penalisation of large negative margins, the more sensitive it is to outliers. We can therefore tell from Figure 5.2 that the exponential loss is expected to by sensitive to outliers, due to its exponential growth, whereas the squared hinge loss is somewhat more robust with a quadratic growth instead. However, even more robust are the Huberised squared hinge loss, the hinge loss, and the logistic loss which all have an asymptotic behavior which is linear. Most robust is the misclassification loss, but as already discussed, that loss has other disadvantages.

### Multiclass Classification

So far, we have only discussed the binary classification problem, with $M = 2$. The cross-entropy (equivalently, negative log-likelihood) loss is straightforward to generalise to the multiclass problem, that is, $M > 2$, as we did in Chapter 3 for logistic regression. This is a useful property of the likelihood-based loss since it allows us to systematically treat both binary and multiclass classification in the same coherent framework.

Generalising the other loss functions discussed above requires a generalisation of the margin to the multiclass problem. That is possible, but we do not elaborate on it in this book. Instead we mention a pragmatic approach, which is to reformulate the problem as several binary problems. This reformulation can be done using either a one-versus-rest or one-versus-one scheme.

The one-versus-rest (or one-versus-all or binary relevance) idea is to train $M$ binary classifiers. Each classifier in this scheme is trained for predicting one class against all the other classes. To make a prediction for a test data point, all $M$ classifiers are used, and the class which, for example, is predicted with the largest margin is taken as the predicted class. This approach is a pragmatic solution, which may turn out to work well for some problems.

The one-versus-one idea is instead to train one classifier for each pair of classes. If there are $M$ classes in total, there are $\frac{1}{2}M(M-1)$ such pairs. To make a prediction, each classifier predicts either of its two classes, and the class which overall obtains most 'votes' is chosen as the final prediction. The predicted margins can be used to break a tie if that happens. Compared to one-versus-rest, the one-versus-one approach has the disadvantage of involving $\frac{1}{2}M(M-1)$ classifiers, instead of only $M$. On the other hand, each of these classifiers is trained on much smaller datasets (only the data points that belong to either of the two classes), compared to one-versus-rest, which uses the entire original training dataset for all $M$ classifiers.

### Likelihood-Based Models and the Maximum Likelihood Approach

The maximum likelihood approach is a generic way of constructing a loss function based on a statistical model of the observed data. In general, maximising the data likelihood is equivalent to minimising a cost function based on the *negative log-likelihood loss*,

$$J(\boldsymbol{\theta}) = -\frac{1}{n}\sum_{i=1}^{n} \ln p(y_i \mid \mathbf{x}_i; \boldsymbol{\theta}).$$

Hence, in all cases where we have a probabilistic model of the conditional distribution $p(y \mid \mathbf{x})$, the negative log-likelihood is a plausible loss function. For classification problems, this takes a particularly simple form since $p(y \mid \mathbf{x})$ then corresponds to a probability vector over the $M$ classes, and the negative log-likelihood is then equivalent to the cross-entropy loss (3.44) (or (3.32) in the case of binary classification).

Also, in the regression case there is a duality between certain common loss functions and the maximum likelihood approach, as we have previously observed. For instance, in a regression model with additive noise as in (5.1), the squared error loss is equivalent to the negative log-likelihood if we assume a Gaussian noise distribution, $\varepsilon \sim \mathcal{N}(0, \sigma_\varepsilon^2)$. Similarly, we noted above that the absolute error loss corresponds to an implicit assumption of Laplace distributed noise, $\varepsilon \sim \mathcal{L}(0, b_\varepsilon)$.[4] This statistical perspective is one way to understand the fact that the absolute error loss is more robust (less sensitive to outliers) than the squared error loss, since the Laplace distribution has thicker tails compared to the Gaussian distribution. The Laplace distribution therefore encodes sporadic large noise values (that is, outliers) as more probable, compared to the Gaussian distribution.

Using the maximum likelihood approach, other assumptions about the noise or insights into its distribution can be incorporated in a similar way in the regression model (5.1). For instance, if we believe that the error is non-symmetric, in the sense that the probability of observing a large positive error is larger than the probability of observing a large negative error, then this can be modelled by a skewed noise

[4] This can be verified from the definition of the Laplace probability density function, which is an exponential of the negative absolute deviation from the mean.

distribution. Using the negative log-likelihood loss is then a systematic way of incorporating this skewness into the training objective.

Relaxing the Gaussian assumption in (5.1) gives additional flexibility to the model. However, the noise is still assumed to be additive and independent of the input $\mathbf{x}$. The real power of the likelihood perspective for designing a loss function comes when these basic assumptions are dropped. For instance, in Section 3.4, we introduced generalised linear models as a way to handle output variables with specific properties, such as count data (that is, $y$ takes values in the set of natural numbers 0, 1, 2, ...). In such situations, to build a model we often start from a specific form of the likelihood $p(y \mid \mathbf{x})$, which is chosen to capture the key properties of the data (for instance, having support only on the natural numbers). Hence, the likelihood becomes an integral part of the model, and this approach therefore lends itself naturally to training by maximum likelihood.

In generalised linear models (see Section 3.4), the likelihood is parameterised in a very particular way, but when working with non-linear parametric models, this is not strictly necessary. A more direct approach to (non-linear) likelihood-based parametric modelling is therefore to

> model the conditional distribution $p(y \mid \mathbf{x}; \boldsymbol{\theta})$ directly as a function parameterised by $\boldsymbol{\theta}$.

More specifically, once we have assumed a certain form for the likelihood (such as a Gaussian, a Poisson, or some other distribution), its shape will be controlled by some parameters (such as the mean and the variance of the Gaussian, or the rate of a Poisson distribution, not to be confused with $\boldsymbol{\theta}$). The idea is then to construct a parametric model $\mathbf{f}_{\boldsymbol{\theta}}(\mathbf{x})$, such that the output of this model is a *vector of parameters* controlling the shape of the distribution $p(y \mid \mathbf{x}; \boldsymbol{\theta})$.

As an example, assume that we are working with unbounded real-valued outputs and want to use a Gaussian likelihood, similarly to the regression model (5.2). However, the nature of the data is such that the noise variance, that is the magnitude of the errors that we expect to see, varies with the input $\mathbf{x}$. By directly working with the likelihood formulation, we can then hypothesise a model according to

$$p(y \mid \mathbf{x}; \boldsymbol{\theta}) = \mathcal{N}(f_{\boldsymbol{\theta}}(\mathbf{x}), \exp(h_{\boldsymbol{\theta}}(\mathbf{x}))) ,$$

where $f_{\boldsymbol{\theta}}$ and $h_{\boldsymbol{\theta}}$ are two arbitrary (linear or non-linear) real-valued regression functions parameterised by $\boldsymbol{\theta}$ (hence, following the notation above, $\mathbf{f}_{\boldsymbol{\theta}}(\mathbf{x}) = (f_{\boldsymbol{\theta}}(\mathbf{x}) \;\; h_{\boldsymbol{\theta}}(\mathbf{x}))^{\mathsf{T}}$). The exponential function is used to ensure that the variance is always positive without explicitly constraining the function $h_{\boldsymbol{\theta}}(\mathbf{x})$. The two functions can be learned simultaneously by minimising the negative log-likelihood over the training data. Note that in this case the problem does not simplify to a squared error loss, despite the fact that the likelihood is Gaussian, since we need to take the dependence on the variance into account. More precisely, the negative log-likelihood loss becomes


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

$$
\begin{aligned}
L(y, \boldsymbol{\theta}) &= -\ln \mathcal{N}(f_{\boldsymbol{\theta}}(\mathbf{x}), \exp(h_{\boldsymbol{\theta}}(\mathbf{x}))) \\
&\propto h_{\boldsymbol{\theta}}(\mathbf{x}) + \frac{(y - f_{\boldsymbol{\theta}}(\mathbf{x}))^2}{\exp(h_{\boldsymbol{\theta}}(\mathbf{x}))} + \text{const.}
\end{aligned}
$$

Once the parameters have been learned, the resulting model is capable of predicting a different mean *and a different variance* for the output $y$, depending on the value of the input variable $\mathbf{x}$.[5]

Other situations that can be modelled using a direct likelihood model in a similar way include multimodality, quantisation, and truncated data. As long as the modeller can come up with a reasonable likelihood – that is, a distribution that *could have* generated the data under study – the negative log-likelihood loss can be used to train the model in a systematic way.

### Strictly Proper Loss Functions and Asymptotic Minimisers

As mentioned earlier, the *asymptotic minimiser* of a loss function is an important theoretical concept for understanding its properties. The asymptotic minimiser is the model which minimises the cost function when the number of training data points $n \to \infty$ (hence the name asymptotic). To formalise this, assume that the model is expressed in terms of a function $f(\mathbf{x})$. As we have seen above, this captures not only regression but also classification, through the margin concept. The asymptotic minimiser $f^*(\mathbf{x})$ of a loss function $L(y, f(\mathbf{x}))$ is then defined as the function which minimises the expected loss:

$$
f^*(\cdot) = \arg\min_f \mathbb{E}[L(y, f(\mathbf{x}))] \,. \tag{5.20}
$$

There are a couple of things to note about this expression. First, we stated above that the asymptotic minimiser is obtained as the solution to the training objective (5.4) as $n \to \infty$, but this has now been replaced by an expected value. This is motivated by the law of large numbers stipulating that the cost function (5.4) will converge to the expected loss as $n \to \infty$, and the latter is more convenient to analyse mathematically. Note that the expected value is taken with respect to a ground truth data generating probability distribution $p(y, \mathbf{x})$, analogously to how we reasoned about a the new data error in Chapter 4. Second, when we talk about asymptotic minimisers, it is typically assumed that the model class is flexible enough to contain *any function* $f(\mathbf{x})$. Consequently, the minimisation in (5.20) is not with respect to a finite dimensional model parameter $\boldsymbol{\theta}$ but rather with respect to the function $f(\mathbf{x})$ itself. The reason for this, rather abstract, definition is that we want to derive the asymptotic minimiser as a property of the *loss function* itself, not of a particular combination of loss function and model class.

The expected value above is with respect to both inputs $\mathbf{x}$ and outputs $y$. However, by the law of total expectation, we can write $\mathbb{E}[L(y, f(\mathbf{x}))] = \mathbb{E}[\mathbb{E}[L(y, f(\mathbf{x})) \,|\, \mathbf{x}]]$,

[5] This property is referred to as *heteroskedasticity* (in contrast to the standard regression model (5.2) which is *homoskedastic* – that is, it has the same output variance for all possible inputs).

where the inner expectation is over $y$ (conditionally on $\mathbf{x}$), and the outer expectation is over $\mathbf{x}$. Now, since $f(\cdot)$ is free to be any function, minimising the total expectation is equivalent to minimising the inner expectation point-wise for each value of $\mathbf{x}$. Therefore, we can replace (5.20) with

$$f^*(\mathbf{x}) = \arg\min_{f(\mathbf{x})} \mathbb{E}[L(y, f(\mathbf{x})) \,|\, \mathbf{x}]\,, \tag{5.21}$$

where the minimisation is now done independently for any fixed value of $\mathbf{x}$.

By computing the asymptotic minimiser of a loss function, we obtain information about the expected behaviour or properties of a model that is trained using this loss function. Although the asymptotic minimiser is an idealised theoretical concept (assuming infinite data and infinite flexibility), it reveals, in some sense, what the training algorithm is striving to achieve when minimising a particular loss.

The concept is useful for understanding both regression and classification losses. A few notable examples in the regression setting are the asymptotic minimisers of the squared error loss and the absolute error loss, respectively. For the former, the asymptotic minimiser can be shown to be equal to the conditional mean, $f^*(\mathbf{x}) = \mathbb{E}[y \,|\, \mathbf{x}]$. That is, a regression model trained using squared error loss will strive to predict $y$ according to its true conditional mean under the data generating distribution $p(y, \mathbf{x})$ (although in practice this will be hampered by the limited flexibility of the model class and the limited amount of training data). For the absolute error loss, the asymptotic minimiser is given by the conditional median, $f^*(\mathbf{x}) = \text{Median}[y \,|\, \mathbf{x}]$. This is less sensitive to the tail probability of $p(y \,|\, \mathbf{x})$ than the conditional mean, providing yet another interpretation of the improved robustness of the absolute error loss.

Related to the concept of asymptotic minimisers is the notion of a *strictly proper* loss function. A loss function is said to be strictly proper[6] if its asymptotic minimiser is *(i)* unique and *(ii)* in one-to-one correspondence with the true conditional distribution $p(y \,|\, \mathbf{x})$. Put differently, for a strictly proper loss function, we can express $p(y \,|\, \mathbf{x})$ in terms of the asymptotic minimiser $f^*(\mathbf{x})$. Such a loss function will thus *strive to* recover a complete probabilistic characterisation of the true input–output relationship.

This requires a probabilistic interpretation of the model, in the sense that we can express $p(y \,|\, \mathbf{x})$ in terms of the model $f(\mathbf{x})$, which is not always obvious. One case which stands out in this respect is the maximum likelihood approach. Indeed, training by maximum likelihood requires a likelihood-based model since the corresponding loss is expressed directly in terms of the likelihood. As we have discussed above, the negative log-likelihood loss is a very generic loss function (it is applicable to regression, classification, and many other types of problems). We can now complement this by the theoretical statement that

| the negative-log likelihood loss is strictly proper. |
|---|

[6] A loss function that is *proper* but not *strictly proper* is minimised by the true conditional distribution $p(y \,|\, \mathbf{x})$, but the minimising argument is not unique.


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

Note that this applies to any type of problem where the negative log-likelihood loss can be used. As noted above, the concept of a loss function being strictly proper is related to its asymptotic minimiser, which in turn is derived under the assumption of infinite flexibility and infinite data. Hence, what our claim above says is that *if the likelihood-based model is flexible enough to describe the true conditional distribution* $p(y \mid \mathbf{x})$*, then the optimal solution to the maximum likelihood problem as* $n \to \infty$ *is to learn this true distribution.*

***Time to reflect 5.1*** *To express the expected negative log-likelihood loss mathematically, we need to distinguish between the likelihood according to the model, which we can denote by* $q(y \mid \mathbf{x})$ *for the time being, and the likelihood with respect to the true data generating distribution* $p(y \mid \mathbf{x})$*. The expected loss becomes*

$$\mathbb{E}_{p(y \mid \mathbf{x})} \left[ -\ln q(y \mid \mathbf{x}) \mid \mathbf{x} \right],$$

*which is referred to as the (conditional)* cross-entropy *of the distribution* $q(y \mid \mathbf{x})$ *with respect to the distribution* $p(y \mid \mathbf{x})$ *(which explains the alternative name* cross-entropy loss *commonly used in classification).*

*What does our claim, that the negative log-likelihood loss is strictly proper, imply regarding the cross entropy?*

That negative log-likelihood is strictly proper should not come as a surprise since it is tightly linked to the statistical properties of the data. What is perhaps less obvious is that there are other loss functions that are also strictly proper, as we will see next. To make the presentation below more concrete, we will focus on the case of binary classification for the remainder of this section. In binary classification, the conditional distribution $p(y \mid \mathbf{x})$ takes a particularly simple form, since it is completely characterised by a single number, namely the probability of the positive class, $p(y = 1 \mid \mathbf{x})$.

Returning to the margin-based loss functions discussed above, recall that any loss function that encourages positive margins can be used to train a classifier, which can then be used for making class predictions according to (5.12). However,

it is only when we use a strictly proper loss function that we can interpret the resulting classification model $g(\mathbf{x})$ as an estimate of the conditional class probability $p(y = 1 \mid \mathbf{x})$.

When choosing a loss function for classification, it is therefore instructive to consider its asymptotic minimiser, since this will determine whether or not the loss function is strictly proper. In turn, this will reveal whether or not it is sensible to use the resulting model to reason about conditional class probabilities.

We proceed by stating the asymptotic minimisers for some of the loss functions

presented above. Deriving the asymptotic minimiser is most often a straightforward calculation, but for brevity we do not include the derivations here. Starting with the binary cross-entropy loss (5.11), its asymptotic minimiser can be shown to be $g^*(\mathbf{x}) = p(y = 1 \mid \mathbf{x})$. In other words, when $n \to \infty$, the loss function (5.11) is uniquely minimised when $g(\mathbf{x})$ is equal to the true conditional class probability. This is in agreement with the discussion above, since the binary cross-entropy loss is just another name for the negative log-likelihood.

Similarly, the asymptotic minimiser of the logistic loss (5.13) is $f^*(\mathbf{x}) = \ln \frac{p(y=1 \mid \mathbf{x})}{1-p(y=1 \mid \mathbf{x})}$. This is an invertible function of $p(y = 1 \mid \mathbf{x})$ and hence the logistic loss is strictly proper. By inverting $f^*(\mathbf{x})$, we obtain $p(y = 1 \mid \mathbf{x}) = \frac{\exp f^*(\mathbf{x})}{1+\exp f^*(\mathbf{x})}$, which shows how conditional class probability predictions can be obtained from $f^*(\mathbf{x})$. With the 'margin formulation' of logistic regression, we seemingly lost the class probability predictions $g(\mathbf{x})$. We have now recovered them. Again, this is not surprising since the logistic loss is a special case of negative log-likelihood when using a logistic regression model.

For the exponential loss (5.15), the asymptotic minimiser is $f^*(\mathbf{x}) = \frac{1}{2} \ln \frac{p(y=1 \mid \mathbf{x})}{1-p(y=1 \mid \mathbf{x})}$, which is in fact the same expression as we got for the logistic loss apart from a constant factor $\frac{1}{2}$. The exponential loss is therefore also strictly proper, and $f^*(\mathbf{x})$ can be inverted and used for predicting conditional class probabilities.

Turning to the hinge loss (5.16), the asymptotic minimiser is

$$f^*(\mathbf{x}) = \begin{cases} 1 & \text{if } p(y = 1 \mid \mathbf{x}) > 0.5, \\ -1 & \text{if } p(y = 1 \mid \mathbf{x}) < 0.5. \end{cases}$$

This is a non-invertible transformation of $p(y = 1 \mid \mathbf{x})$, which means that it is not possible to recover $p(y = 1 \mid \mathbf{x})$ from the asymptotic minimiser $f^*(\mathbf{x})$. This implies that a classifier learned using hinge loss (such as support vector classification, Section 9.5) is *not able* to predict conditional class probabilities.

The squared hinge loss (5.17), on the other hand, is a strictly proper loss function, since its asymptotic minimiser is $f^*(\mathbf{x}) = 2p(y = 1 \mid \mathbf{x}) - 1$. This also holds for the Huberised square hinge loss (5.18). Recalling our robustness discussion, we see that by squaring the hinge loss, we make it strictly proper, but at the same time we impact its robustness. However, the 'Huberisation' (replacing the quadratic curve with a linear one for margins $< -1$) improves the robustness while keeping the property of being strictly proper.

We have now seen that some (but not all) loss functions are strictly proper, meaning they could potentially predict conditional class probabilities correctly. However, this is only under the assumption that the model is sufficiently flexible that $g(\mathbf{x})$ or $f(\mathbf{x})$ can actually take the shape of the asymptotic minimiser. This is possibly problematic; for instance, recall that $f(\mathbf{x})$ is a linear function in logistic regression, whereas $p(y = 1 \mid \mathbf{x})$ can be almost arbitrarily complicated in real world applications. It is therefore not sufficient to use a strictly proper loss function in order to accurately predict conditional class probabilities: our model also has to be flexible enough. This discussion is also only valid in the limit as $n \to \infty$. However, in practice $n$ is always


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

finite, and we may ask how large $n$ has to be for a flexible enough model to at least approximately learn the asymptotic minimiser? Unfortunately, we cannot give any general numbers, but following the same principles as the overfitting discussion in Chapter 4, the more flexible the model, the larger $n$ is required. If $n$ is not large enough, the predicted conditional class probabilities tend to 'overfit' to the training data. In summary, using a strictly proper loss function will *encourage* the training procedure to learn a model that is faithful to the true statistical properties of the data, but in itself it is not enough to guarantee that these properties are well described by the model.

In many practical applications, having access to reliable uncertainty estimates regarding a model's predictions is necessary for robust and well-informed decision making. In such cases, it is thus important to validate the model, not only in terms of accuracy or expected errors but also in terms of its statistical properties. One approach is to evaluate the so-called *calibration* of the model, which is, however, beyond the scope of this book.

## 5.3 Regularisation

We will now take a closer look at regularisation, which was briefly introduced in Section 3.3 as a useful tool for avoiding overfitting if the model was too flexible, such as a polynomial of high degree. We have also discussed thoroughly in Chapter 4 the need for tuning the model flexibility, which effectively is the purpose of regularisation. Finding the right level of flexibility, and thereby avoiding overfit, is very important in practice.

The idea of regularisation in a parametric model is to 'keep the parameters $\widehat{\boldsymbol{\theta}}$ small unless the data really convinces us otherwise', or alternatively 'if a model with small values of the parameters $\widehat{\boldsymbol{\theta}}$ fits the data almost as well as a model with larger parameter values, the one with small parameter values should be preferred'. There are, however, many different ways to implement this idea, and we distinguish between *explicit regularisation* and *implicit regularisation*. We will first discuss explicit regularisation, which amounts to modifying the cost function, and in particular so-called $L^2$ and $L^1$ regularisation.

### $L^2$ Regularisation

$L^2$ regularisation (also known as *Tikhonov regularisation*, *ridge regression*, and *weight decay*) amounts to adding an extra penalty term $\|\boldsymbol{\theta}\|_2^2$ to the cost function. Linear regression with squared error loss and $L^2$ regularisation, as an example, amounts to solving

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \frac{1}{n}\|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|_2^2 + \lambda\|\boldsymbol{\theta}\|_2^2. \tag{5.22}$$

By choosing the regularisation parameter $\lambda \geq 0$, a trade-off between the original cost function (fitting the training data as well as possible) and the regularisation term

(keeping the parameters $\widehat{\boldsymbol{\theta}}$ close to zero) is made. In setting $\lambda = 0$ we recover the original least squares problem (3.12), whereas $\lambda \to \infty$ will force all parameters $\widehat{\boldsymbol{\theta}}$ to 0. A good choice of lambda in practice is usually neither of those extremes but somewhere in between, and can be determined using cross-validation.

It is actually possible to derive a version of the normal equations for (5.22), namely

$$(\mathbf{X}^\mathsf{T}\mathbf{X} + n\lambda\mathbf{I}_{p+1})\widehat{\boldsymbol{\theta}} = \mathbf{X}^\mathsf{T}\mathbf{y}, \tag{5.23}$$

where $\mathbf{I}_{p+1}$ is the identity matrix of size $(p+1) \times (p+1)$. For $\lambda > 0$, the matrix $\mathbf{X}^\mathsf{T}\mathbf{X} + n\lambda\mathbf{I}_{p+1}$ is always invertible, and we have the closed form solution

$$\widehat{\boldsymbol{\theta}} = (\mathbf{X}^\mathsf{T}\mathbf{X} + n\lambda\mathbf{I}_{p+1})^{-1}\mathbf{X}^\mathsf{T}\mathbf{y}. \tag{5.24}$$

This also reveals another reason for using regularisation in linear regression, namely when $\mathbf{X}^\mathsf{T}\mathbf{X}$ is not invertible. When $\mathbf{X}^\mathsf{T}\mathbf{X}$ is not invertible, the ordinary normal equations (3.13) have no unique solution $\widehat{\boldsymbol{\theta}}$, whereas the $L^2$-regularised version always has the unique solution (5.24) if $\lambda > 0$.

### $L^1$ Regularisation

With $L^1$ regularisation (also called *LASSO*, an abbreviation for Least Absolute Shrinkage and Selection Operator), the penalty term $\|\boldsymbol{\theta}\|_1$ is added to the cost function. Here $\|\boldsymbol{\theta}\|_1$ is the 1-norm or 'taxicab norm' $\|\boldsymbol{\theta}\|_1 = |\theta_0| + |\theta_1| + \cdots + |\theta_p|$. The $L^1$ regularised cost function for linear regression (with squared error loss) then becomes

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \frac{1}{n}\|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|_2^2 + \lambda\|\boldsymbol{\theta}\|_1. \tag{5.25}$$

Contrary to linear regression with $L^2$ regularisation (3.48), there is no closed-form solution available for (5.25). However, as we will see in Section 5.4, it is possible to design an efficient numerical optimisation algorithm for solving (5.25).

As for $L^2$ regularisation, the regularisation parameter $\lambda$ has to be chosen by the user and has a similar meaning: $\lambda = 0$ gives the ordinary least squares solution and $\lambda \to \infty$ gives $\widehat{\boldsymbol{\theta}} = 0$. Between these extremes, however, $L^1$ and $L^2$ tend to give different solutions. Whereas $L^2$ regularisation pushes all parameters towards small values (but not necessarily exactly zero), $L^1$ tends to favour so-called *sparse* solutions, where only a few of the parameters are non-zero, and the rest are exactly zero. Thus, $L^1$ regularisation can effectively 'switch off' some inputs (by setting the corresponding parameter $\theta_k$ to zero), and it can therefore be used as an input (or feature) selection method.


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

**Example 5.2 Regularisation for car stopping distance**

Consider again Example 2.2 with the car stopping distance regression problem. We use the 10th order polynomial that was considered meaningless in Example 3.5 and apply $L^2$ and $L^1$ regularisation to it in turn. With manually chosen $\lambda$, we obtain the models shown in Figure 5.3.

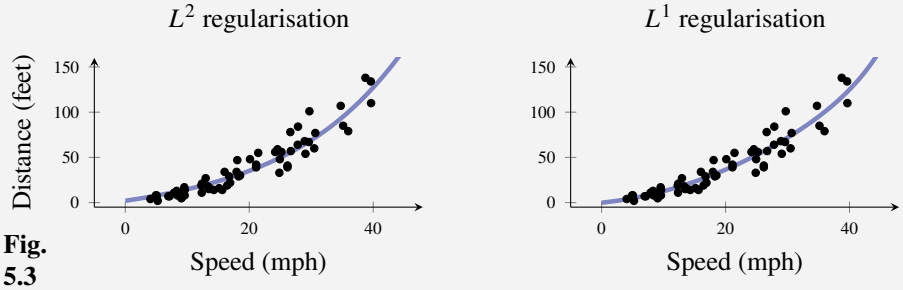



**Fig. 5.3**

Both models suffer less from overfitting than the non-regularised 10th order polynomial in Example 3.5. The two models here are, however, not identical. Whereas all parameters are relatively small but non-zero in the $L^2$-regularised model (left panel), only 4 (out of 11) parameters are non-zero in the $L^1$-regularised model (right panel). It is typical for $L^1$ regularisation to give sparse models, where some parameters are set exactly to zero.

## General Explicit Regularisation

$L^1$ and $L^2$ regularisation are two common examples of what we refer to as explicit regularisation since they are both formulated as modifications of the cost function. They suggest a general pattern on which explicit regularisation can be formulated:

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \underbrace{J(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y})}_{\text{(i)}} + \underbrace{\lambda}_{\text{(iii)}} \underbrace{R(\boldsymbol{\theta})}_{\text{(ii)}}. \tag{5.26}$$

This expression contains three important elements:

(i) the cost function, which encourages a good fit to the training data;

(ii) the regularisation term, which encourages small parameter values; and

(iii) the regularisation parameter $\lambda$, which determines the trade-off between (i) and (ii).

In this view, it is clear that explicit regularisation modifies the problem of fitting to the training data (minimising $E_{\text{train}}$) into something else, which hopefully minimises $E_{\text{new}}$ instead. The actual design of the regularisation term $R(\boldsymbol{\theta})$ can be done in many ways. As a combination of the $L^1$ and $L^2$ terms, one option is $R(\boldsymbol{\theta}) = \|\boldsymbol{\theta}\|_1 + \|\boldsymbol{\theta}\|_2^2$, which

often is referred to as elastic net regularisation. Regardless of the exact expression of the regularisation term, its purpose is to encourage small parameter values and thereby decrease the flexibility of the model, which might improve the performance and lower $E_{\text{new}}$.

### Implicit Regularisation

Any supervised machine learning method that is trained by minimising a cost function can be regularised using (5.26). There are, however, alternative ways to achieve a similar effect without explicitly modifying the cost function. One such example of implicit regularisation is early stopping. Early stopping is applicable to any method that is trained using *iterative* numerical optimisation, which is the topic of the next section. It amounts to aborting the optimisation before it has reached the minimum of the cost function. Although it may appear counter-intuitive to prematurely abort an optimisation procedure, it has proven useful in practice, and early stopping has been shown to be of practical importance to avoid overfitting for some models, most notably deep learning (Chapter 6). Early stopping can be implemented by setting aside some hold-out validation data and computing $E_{\text{hold-out}}$ as in (4.6) for $\boldsymbol{\theta}^{(t)}$ after each iteration $t$ of the numerical optimisation.[7] It is typically observed that $E_{\text{hold-out}}$ decreases initially but eventually reaches a minimum and thereafter starts to increase, even though the cost function (by design of the optimisation algorithm) decreases monotonically. The optimisation is then aborted at the point when $E_{\text{hold-out}}$ reached its minimum, as we will illustrate in Example 5.7.

Early stopping is a commonly used implicit regularisation technique, but not the only one. Another technique with a regularising effect is dropout for neural networks, which we discuss in Chapter 6, and data augmentation, which we discuss in Chapter 13. For decision trees, the splitting criterion can be seen as a type of implicit regularisation. It has also been argued that the randomness of the stochastic gradient optimisation algorithm in itself also has the effect of implicit regularisation.

## 5.4 Parameter Optimisation

Many supervised machine learning methods, linear and logistic regression included, involve one (or more) optimisation problems, such as (3.12), (3.35), or (5.25). A machine learning engineer therefore needs to be familiar with the main strategies for how to solve optimisation problems fast. Starting with the optimisation problems from linear and logistic regression, we will introduce the ideas behind some of the optimisation methods commonly used in supervised machine learning. This section only gives a brief introduction to optimisation theory and, for example, we will only discuss unconstrained optimisation problems.

[7] More practically, to reduce the computational overhead of early stopping, we can compute the validation error at regular intervals, for instance after each epoch.


Machine Learning – A First Course for Engineers and Scientists
Second Edition, online draft version September 1, 2026, http://smlbook.org

Optimisation is about finding the minimum or maximum of an *objective function*. Since the maximisation problem can be formulated as minimisation of the negative objective function, we can limit ourselves to minimisation without any loss of generality.

There are primarily two ways in which optimisation is used in machine learning:

1. For training a model by minimising the cost function with respect to the model parameters $\boldsymbol{\theta}$. In this case, the objective function corresponds to the cost function $J(\boldsymbol{\theta})$, and the optimisation variables correspond to the model parameters.

2. For tuning hyperparameters, such as the regularisation parameter $\lambda$. For instance, by using a held-out validation dataset (see Chapter 4), we can select $\lambda$ to minimise the hold-out validation error $E_{\text{hold-out}}$. In this case, the objective function is the validation error, and the optimisation variables correspond to the hyperparameters.

In the presentation below, we will use $\boldsymbol{\theta}$ to denote a general optimisation variable, but keep in mind that optimisation can also be used for selecting hyperparameters.

An important class of objective functions are *convex* functions. Optimisation is often easier to carry out for convex objective functions, and it is good practice to take some time to consider whether a non-convex optimisation problem can be re-formulated into a convex problem (which sometimes, but not always, is possible). The most important property of a convex function, for this discussion, is that a convex function has a unique minimum,[8] and no other local minima. Examples of convex functions are the cost functions for logistic regression, linear regression, and $L^1$-regularised linear regression. An example of a non-convex function is the cost function for a deep neural network. We illustrate this by Example 5.3.

**Example 5.3 Examples of objective functions**

Figure 5.4 contains examples of what an objective function can look like.

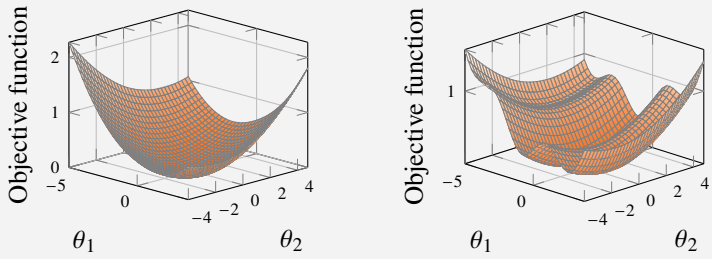



**Fig. 5.4**

Both examples are functions of a two-dimensional parameter vector $\boldsymbol{\theta} = [\theta_1\ \theta_2]^\mathsf{T}$. The left is convex and has a finite unique global minimum, whereas the right is non-convex

[8] The minimum does, however, not have to be finite. The exponential function, for example, is convex but attains its minimum at $-\infty$. Convexity is a relatively strong property, and also non-convex functions may have only one minimum.

and has three local minima (of which only one is the global minimum). We will in the following examples illustrate these objective functions using contour plots instead, as shown in Figure 5.5.

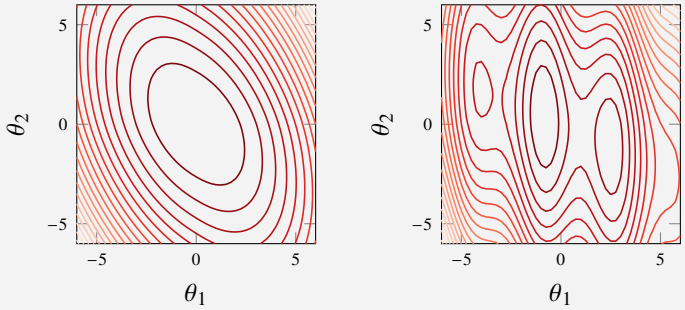



**Fig. 5.5**

***Time to reflect 5.2*** *After reading the rest of this book, return here and try to fill out this table, summarising how optimisation is used by the different methods.*

| ***Method*** | ***What is optimisation used for?*** | | | ***What type of optimisation?*** | | | |
|---|---|---|---|---|---|---|---|
| | *Training* | *Hyper-parameters* | *Nothing* | *Closed-form** | *Grid search* | *Gradient-based* | *Stochastic gradient descent* |
| *k-NN* | | | | | | | |
| *Trees* | | | | | | | |
| *Linear regression* | | | | | | | |
| *Linear regression with $L^2$-regularisation* | | | | | | | |
| *Linear regression with $L^1$-regularisation* | | | | | | | |
| *Logistic regression* | | | | | | | |
| *Deep learning* | | | | | | | |
| *Random forests* | | | | | | | |
| *AdaBoost* | | | | | | | |
| *Gradient boosting* | | | | | | | |
| *Gaussian processes* | | | | | | | |

**including coordinate descent*

## Optimisation Using Closed-Form Expressions

For linear regression with squared error loss, training the model amounts to solving the optimisation problem (3.12)

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \frac{1}{n} \|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|_2^2.$$

As we have discussed, and also proved in Appendix 3.A, the solution (3.14) to this problem can (under the assumption that $\mathbf{X}^\mathsf{T}\mathbf{X}$ is invertible) be derived analytically. If we just take the time to efficiently implement (3.14) once, for example using Cholesky or QR factorisation, we can use this every time we want to train a linear regression

model with squared error loss. Each time we use it, we know that we have found the optimal solution in a computationally efficient way.

If we instead want to learn the $L^1$-regularised version, we have to solve (5.25)

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \frac{1}{n}\|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|_2^2 + \lambda\|\boldsymbol{\theta}\|_1.$$

Unfortunately, this problem cannot be solved analytically. Instead we have to use computer power to solve it, by constructing an iterative procedure to find the solution. With a certain choice of such an optimisation algorithm, we can make use of some analytical expressions along the way, which turns out to offer an efficient way of solving it. Remember that $\boldsymbol{\theta}$ is a vector containing $p + 1$ parameters that we want to learn from the training data. As it turns out, if we seek the minimum for only one of these parameters, say $\theta_j$, while keeping the other parameters fixed, we can find the optimum as

$$\arg\min_{\theta_j} \frac{1}{n}\|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|_2^2 + \lambda\|\boldsymbol{\theta}\|_1 = \operatorname{sign}(t)(|t| - \lambda),$$

$$\text{where } t = \sum_{i=1}^{n} x_{ij}\left(y_i - \sum_{k\neq j} x_{ik}\theta_k\right). \tag{5.27}$$

It turns out that making repeated 'sweeps' through the vector $\boldsymbol{\theta}$ and updating one parameter at a time according to (5.27) is a good way to solve (5.25). This type of algorithm, where we update one parameter at a time, is referred to as *coordinate descent*, and we illustrate it in Example 5.4.

It can be shown that the cost function in (5.25) is convex. Convexity alone is not sufficient to guarantee that coordinate descent will find its (global) minimum, but for the $L^1$-regularised cost function (5.25), it can be shown that coordinate descent actually finds the (global) minimum. In practice we know that we have found the global minimum when no parameters have changed during a full 'sweep' of the parameter vector.

It turns out that coordinate descent is a very efficient method for $L^1$-regularised linear regression (5.25). The keys are that (i) (5.27) exists and is cheap to compute, and (ii) many updates will simply set $\theta_j = 0$ due to the sparsity of the optimal $\widehat{\boldsymbol{\theta}}$. This makes the algorithm fast. For most machine learning optimisation problems, however, it *cannot* be said that coordinate descent is the preferred method. We will now have a look at some more general families of optimisation methods that are widely used in machine learning.

**Example 5.4 Coordinate descent**

We apply coordinate descent to the objective functions from Example 5.3 and show the result in Figure 5.6. For coordinate descent to be an efficient alternative in practice, closed-form solutions for updating one parameter at a time, similar to (5.27), have to

be available.

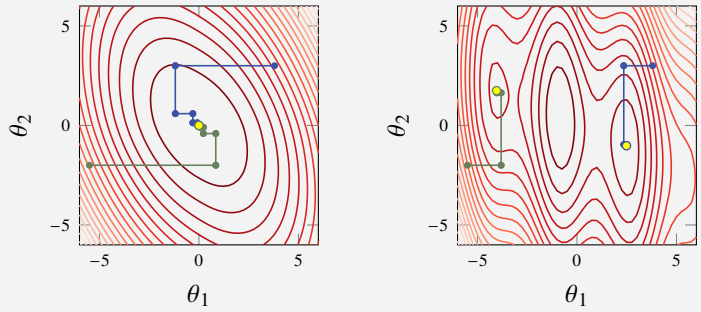



**Fig. 5.6**

Figure 5.6 shows how the parameters are updated in the coordinate descent algorithm, for two different initial parameter vectors (blue and green trajectory, respectively). It is clear from the figure that only one parameter is updated each time, which gives the trajectory a characteristic shape. The obtained minimum is marked with a yellow dot. Note how the different initialisations lead to different (local) minima in the non-convex case (right panel).

## Gradient Descent

In many situations, we cannot do closed-form manipulations, but we do have access to the value of the objective function as well as its derivative (or gradient). Sometimes we even have access to the second derivative (the Hessian). In those situations, it is often a good idea to use a *gradient descent* method, which we will introduce now, or even *Newton's method*, which we will discuss later.

Gradient descent can be used for learning parameter vectors $\boldsymbol{\theta}$ of high dimension when the objective function $J(\boldsymbol{\theta})$ is simple enough such that its gradient is possible to compute. Let us therefore consider the parameter learning problem

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \tag{5.28}$$

(even though gradient descent can potentially be used for hyperparameters as well). We will assume that the gradient of the cost function $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ exists for all $\boldsymbol{\theta}$. As an example, the gradient of the cost function for logistic regression (3.34) is[9]

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = -\frac{1}{n}\sum_{i=1}^{n}\left(\frac{1}{1+e^{y_i\boldsymbol{\theta}^{\mathsf{T}}\mathbf{x}_i}}\right) y_i\mathbf{x}_i. \tag{5.29}$$

[9]This assumption is primarily made for the theoretical discussion. In practice, there are successful examples of gradient descent being applied to objective functions not differentiable everywhere, such as neural networks with ReLu activation functions (Chapter 6).


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

Note that $\nabla_\theta J(\boldsymbol{\theta})$ is a vector of the same dimension as $\boldsymbol{\theta}$, which describes the direction in which $J(\boldsymbol{\theta})$ increases. Consequently, and more useful for us, $-\nabla_\theta J(\boldsymbol{\theta})$ describes the direction in which $J(\boldsymbol{\theta})$ decreases. That is, if we take a small step in the direction of the negative gradient, this will reduce the value of the cost function,

$$J\big(\boldsymbol{\theta} - \gamma \nabla_\theta J(\boldsymbol{\theta})\big) \leq J\big(\boldsymbol{\theta}\big) \tag{5.30}$$

for some (possibly very small) $\gamma > 0$. If $J(\boldsymbol{\theta})$ is convex, the inequality in (5.30) is strict except at the minimum (where $\nabla_\theta J(\boldsymbol{\theta})$ is zero). This suggests that if we have $\boldsymbol{\theta}^{(t)}$ and want to select $\boldsymbol{\theta}^{(t+1)}$ such that $J(\boldsymbol{\theta}^{(t+1)}) \leq J(\boldsymbol{\theta}^{(t)})$, we should

$$\boxed{\text{update } \boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \gamma \nabla_\theta J(\boldsymbol{\theta}^{(t)})} \tag{5.31}$$

with some positive $\gamma > 0$. Repeating (5.31) gives the gradient descent algorithm, Algorithm 5.1.

**Algorithm 5.1:** Gradient descent

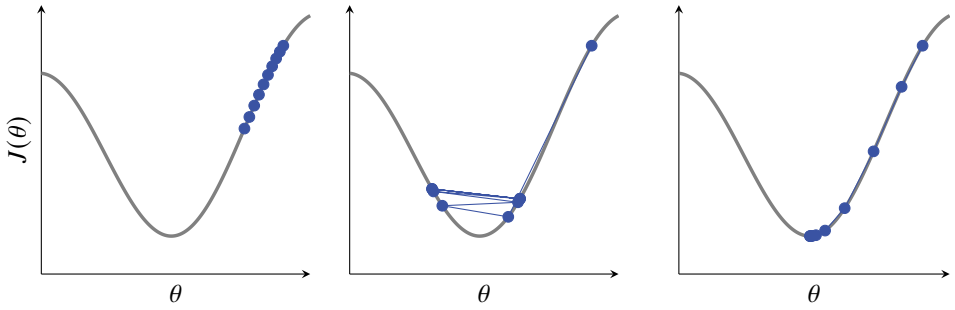
**Input:** Objective function $J(\boldsymbol{\theta})$, initial $\boldsymbol{\theta}^{(0)}$, learning rate $\gamma$
**Result:** $\widehat{\boldsymbol{\theta}}$

Set $t \leftarrow 0$
**do**
  Update $\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} - \gamma \nabla_\theta J(\boldsymbol{\theta}^{(t)})$
  Update $t \leftarrow t + 1$
**while** $\|\boldsymbol{\theta}^{(t)} - \boldsymbol{\theta}^{(t-1)}\|$ *not small enough*
**return** $\widehat{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta}^{(t-1)}$

In practice we do not know $\gamma$, which determines how big the $\boldsymbol{\theta}$-step is at each iteration. It is possible to formulate the selection of $\gamma$ as an internal optimisation problem that is solved at each iteration, a so-called *line-search problem*. This will result in a possibly different value for $\gamma$ at each iteration of the algorithm. Here we will consider the simpler solution where we leave the choice of $\gamma$ to the user, or more specifically view it as a hyperparameter.[10] In such cases, $\gamma$ is often referred to as the *learning rate* or step-size. Note that the gradient $\nabla_\theta J(\boldsymbol{\theta})$ will typically decrease and eventually attain 0 at a stationary point (possibly, but not necessarily, a minimum), so Algorithm 5.1 may converge if $\gamma$ is kept constant. This is in contrast to what we will discuss when we introduce the *stochastic* gradient algorithm.

The choice of learning rate $\gamma$ is important. Some typical situations with too small, too large and a good choice of learning rate are shown in Figure 5.7. With the intuition from these figures, we advise monitoring $J(\boldsymbol{\theta}^{(t)})$ during the optimisation, and to

[10] When viewed as a hyperparmeter, we can also optimise $\gamma$, for instance by using cross-validation, as discussed above. However, this is an 'external' optimisation problem, contrary to line-search which is an 'internal' optimisation problem.




**(a)** Low learning rate $\gamma = 0.05$ **(b)** High learning rate $\gamma = 1.2$ **(c)** Good learning rate $\gamma = 0.3$

**Figure 5.7:** Optimisation using gradient descent of a cost function $J(\theta)$ where $\theta$ is a scalar parameter. In the different subfigures we use a too low learning rate (a), a too high learning rate (b), and a good learning rate (c). Remember that a good value of $\gamma$ is very much related to the shape of the cost function; $\gamma = 0.3$ might be too small (or large) for a different $J(\theta)$.

- decrease the learning rate $\gamma$ if the cost function values $J(\theta^{(t)})$ are getting worse or oscillate widely (as in Figure 5.7b);
- increase the learning rate $\gamma$ if the cost function values $J(\theta^{(t)})$ are fairly constant and only slowly decreasing (as in Figure 5.7a).

No general convergence guarantees can be given for gradient descent, basically because a bad learning rate $\gamma$ may break the method. However, with the 'right' choice of $\gamma$, the value of $J(\boldsymbol{\theta})$ will decrease for each iteration (as suggested by (5.30)) until a point with zero gradient is found – that is, a stationary point. A stationary point is, however, not necessarily a minimum but can also be a maximum or a saddle-point of the objective function. In practice one typically monitors the value of $J(\boldsymbol{\theta})$ and terminates the algorithm when it seems not to be decreasing anymore, and hope it has arrived at a minimum.

In non-convex problems with multiple local minima, we cannot expect gradient descent to always find the global minimum. The initialisation is usually critical for determining which minimum (or stationary point) is found, as illustrated by Example 5.5. It can, therefore, be a good practice (if time and computational resources permit) to run the optimisation multiple times with different initialisations. For computationally heavy non-convex problems, such as training a deep neural network (Chapter 6), when we cannot afford to re-run the training, we usually employ method-specific heuristics and tricks to find a good initialisation point.

For convex problems, there is only one stationary point, which also is the global minimum. Hence, the initialisation for a convex problem can be done arbitrarily. However, by *warm-starting* the optimisation with a good initial guess, we may still save valuable computational time. Sometimes, such as when doing $k$-fold cross


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

validation (Chapter 4), we have to train $k$ models on similar (but not identical) datasets. In situations like this, we can typically make use of the situation by initialising Algorithm 5.1 with the parameters learned for the previous model.

For training a logistic regression model (3.35), gradient descent can be used. Since its cost function is convex, we know that once the gradient descent has converged to a minimum, it has reached the global minimum and we are done. For logistic regression there are, however, more advanced alternatives that usually perform better. We discuss these next.

**Example 5.5 Gradient descent**

We first consider the convex objective function from Example 5.3 and apply gradient descent to it with a seemingly reasonable learning rate. We show the result in Figure 5.8. Note that each step is perpendicular to the level curves at the point where it starts, which is a property of the gradient. As expected, we find the (global) minimum with both of the two different initialisations.

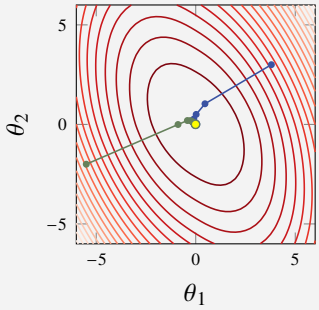



**Fig. 5.8**

For the non-convex objective function from Example 5.3, we apply gradient descent with two different learning rates and show the result in Figure 5.9. In the left plot, the learning rate seems well chosen and the optimisation converges nicely, albeit to different minima depending on the initialisation. Note that it *could* have converged also to one of the saddle points between the different minima. In the right plot, the learning rate is too big, and the procedure does not seem to converge.

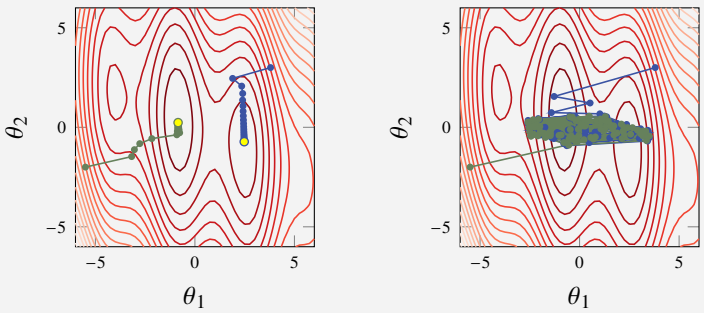



**Fig. 5.9**

### Second Order Gradient Methods

We can think of gradient descent as approximating $J(\boldsymbol{\theta})$ with a first order Taylor expansion around $\boldsymbol{\theta}^{(t)}$, that is, a (hyper-)plane. The next parameter $\boldsymbol{\theta}^{(t+1)}$ is selected by taking a step in the steepest direction of the (hyper-)plane. Let us now see what happen if we instead use a second order Taylor expansion,

$$J(\boldsymbol{\theta} + \mathbf{v}) \approx \underbrace{J(\boldsymbol{\theta}) + \mathbf{v}^{\mathsf{T}}[\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})] + \frac{1}{2}\mathbf{v}^{\mathsf{T}}[\nabla^2_{\boldsymbol{\theta}} J(\boldsymbol{\theta})]\mathbf{v}}_{\triangleq s(\boldsymbol{\theta}, \mathbf{v})}, \tag{5.32}$$

where $\mathbf{v}$ is a vector of the same dimension as $\boldsymbol{\theta}$. This expression contains not only the gradient of the cost function $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ but also the Hessian matrix of the cost function $\nabla^2_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. Remember that we are searching for the minimum of $J(\boldsymbol{\theta})$. We will compute this by iteratively minimising the second order approximation $s(\boldsymbol{\theta}, \mathbf{v})$. If the Hessian $\nabla^2_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ is positive definite, then the minimum of $s(\boldsymbol{\theta}, \mathbf{v})$ with respect to $\mathbf{v}$ is obtained where the derivative of $s(\boldsymbol{\theta}, \mathbf{v})$ is zero:

$$\frac{\partial}{\partial \mathbf{v}} s(\boldsymbol{\theta}, \mathbf{v}) = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) + [\nabla^2_{\boldsymbol{\theta}} J(\boldsymbol{\theta})]\mathbf{v} = 0 \Leftrightarrow \mathbf{v} = -[\nabla^2_{\boldsymbol{\theta}} J(\boldsymbol{\theta})]^{-1}[\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})]. \tag{5.33}$$

This suggests to update

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - [\nabla^2_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^{(t)})]^{-1}[\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^{(t)})], \tag{5.34}$$

which is *Netwton's method* for minimisation. Unfortunately, no general convergence guarantees can be given for Newton's method either. For certain cases, Newton's method can be much faster than gradient descent. In fact, *if* the cost function $J(\boldsymbol{\theta})$ is a quadratic function in $\boldsymbol{\theta}$, then (5.32) is exact, and Newton's method (5.34) will find the optimum in only one iteration! Quadratic objective functions are, however, rare in machine learning.[11] It is not even guaranteed that the Hessian $\nabla^2_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ is always positive definite in practice, which may result in rather strange parameter updates in (5.34). To still make use of the potentially valuable second order information but at the same time also have a robust and practically useful algorithm, we have to introduce some modification of Newton's method. There are multiple options, and we will look at so-called *trust regions*.

We derived Newton's method using the second order Taylor expansion (5.32) as a model for how $J(\boldsymbol{\theta})$ behaves around $\boldsymbol{\theta}^{(t)}$. We should perhaps not trust the Taylor expansion to be a good model for *all* values of $\boldsymbol{\theta}$ but only for those in the vicinity of $\boldsymbol{\theta}^{(t)}$. One natural restriction is, therefore, to trust the second order Taylor expansion (5.32)

[11] For regression, we often use the squared error loss $L(y, \widehat{y}) = (\widehat{y} - y)^2$, which is a quadratic function in $\widehat{y}$. That does *not* imply that $J(\boldsymbol{\theta})$ (the objective function) is necessarily a quadratic function in $\boldsymbol{\theta}$, since $\widehat{y}$ can depend non-linearly on $\boldsymbol{\theta}$. For linear regression with squared loss, however, the dependence is linear, and the cost function is indeed quadratic. This is why we can compute an explicit solution using the normal equations, which is of course the same solution that we would obtain after one iteration of Newton's method applied to this problem.


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

only within a ball of radius $D$ around $\boldsymbol{\theta}^{(t)}$, which we refer to as our trust region. This suggests that we could make a Newton update (5.34) of the parameters, unless the step is longer than $D$, in which case we downscale the step to never leave our trust region. In the next iteration, the trust region is moved to be centered around the updated $\boldsymbol{\theta}^{(t+1)}$, and another step is taken from there. We can express this as

$$\boxed{\text{update } \boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta[\nabla^2_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^{(t)})]^{-1}[\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^{(t)})],} \tag{5.35}$$

where $\eta \leq 1$ is chosen as large as possible such that $\|\boldsymbol{\theta}^{(t+1)} - \boldsymbol{\theta}^{(t)}\| \leq D$. The radius of the trust region $D$ can be updated and adapted as the optimisation proceeds, but for simplicity we will consider $D$ to be a user choice (much like the learning rate for gradient descent). We summarise this as Algorithm 5.2 and look at it in Example 5.6. The trust region Newton method, with a certain set of rules on how to update $D$, is actually one of the methods commonly used for training logistic regression in practice.

**Algorithm 5.2:** Trust region Newton's method

**Input:** Objective function $J(\boldsymbol{\theta})$, initial $\boldsymbol{\theta}^{(0)}$, trust region radius $D$
**Result:** $\widehat{\boldsymbol{\theta}}$

Set $t \leftarrow 0$
**do**
  Compute $\mathbf{v} \leftarrow [\nabla^2_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^{(t)})]^{-1}[\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^{(t)})]$
  Compute $\eta \leftarrow \frac{D}{\max(\|\mathbf{v}\|, D)}$
  Update $\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} - \eta\mathbf{v}$
  Update $t \leftarrow t + 1$
**while** $\|\boldsymbol{\theta}^{(t)} - \boldsymbol{\theta}^{(t-1)}\|$ *not small enough*
**return** $\widehat{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta}^{(t-1)}$

It can be computationally expensive or even impossible to compute the inverse of the Hessian matrix $[\nabla^2_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^{(t)})]^{-1}$. To this end, there is an entire class of methods called *quasi-Newton methods* that all use different ways to *approximate* the inverse of the Hessian matrix $[\nabla^2_{\boldsymbol{\theta}} J(\boldsymbol{\theta})]^{-1}$ in (5.34). This class includes, among others, the Broyden method and the BFGS method (an abbreviation of Broyden, Fletcher, Goldfarb, and Shanno). A further approximation of the latter, called limited-memory BFGS or L-BFGS, has proven to be another good choice for the logistic regression problem.

**Example 5.6 Newton's method**

We first apply Newton's method to the cost functions from Example 5.3 and show the result in Figure 5.10. Since the convex cost function (left) also happens to be close to a quadratic function, Newton's method works well and finds, for both initialisations, the minimum in only two iterations. For the non-convex problem (right), Newton's method diverges for both initialisations, since the second order Taylor expansion (5.32) is a poor approximation of this function and leads the method astray.

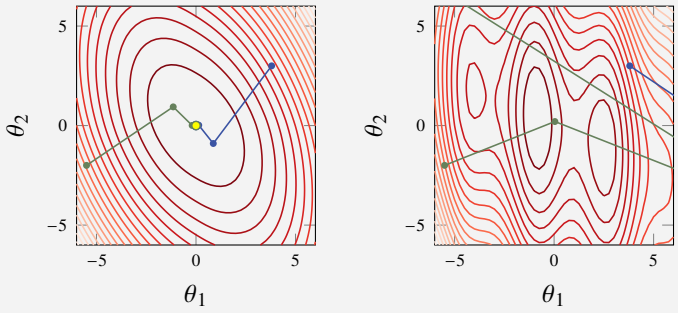



**Fig. 5.10**

We also apply the trust region Newton's method to both problems and show the result in Figure 5.11. Note that the first step direction is identical to the non-truncated version above, but the steps are now limited to stay within the trust region (here a circle of radius 2). This prevents the severe divergence problems for the non-convex case, and all cases converge nicely. Indeed, the convex case (left) requires more iterations than for the non-truncated version above, but that is a price we have to pay in order to have a robust method which also works for the non-convex case shown to the right.

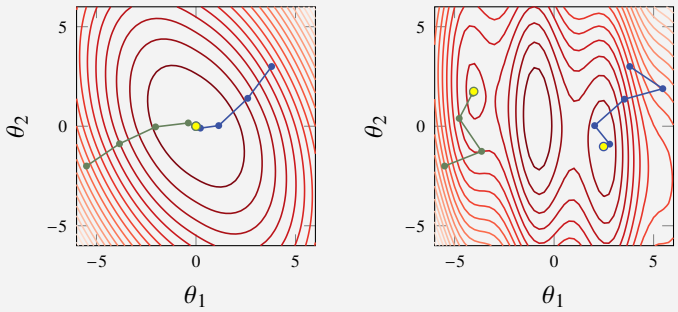



**Fig. 5.11**

Before we end this section, we will have a look at an example of early stopping for logistic regression when using a Newton-type of method. (It can in fact be shown that using early stopping when solving linear regression (with squared error loss) with gradient descent is equivalent to $L^2$ regularisation.[12]) Besides completing the discussion on early stopping from Section 5.3, this example also serves as a good reminder that it is actually not always the global optimum which is the goal when we use optimisation in machine learning.

[12] See, for example, Goodfellow, Bengio, et al. (2016, Section 7.8).


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September* 1, 2026, `http://smlbook.org`

**Example 5.7 Early stopping with logistic regression for the music classification example**

We consider again the music classification problem from Example 2.1. We apply multi-class logistic regression and, to exaggerate the point of this example, we apply a 20 degree polynomial input transformation. The polynomial transformation means that instead of having $\theta_0 + \theta_1 x_1 + \theta_2 x_2$ within the logistic regression model, we now have $\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_1 x_2 + \theta_5 x_2^2 + \cdots + \theta_{229} x_1 x_2^{19} + \theta_{230} x_2^{20}$. Such a setup with 231 parameters and a rather limited amount of data will most likely lead to overfitting if no regularisation is used.

Logistic regression is learned using a Newton-type numerical optimisation algorithm (Section 5.4). We can therefore apply early stopping to it. For this purpose we set aside some hold-out validation data and monitor how $E_{\text{hold-out}}$ (with misclassification error) evolves as the numerical optimisation proceeds.

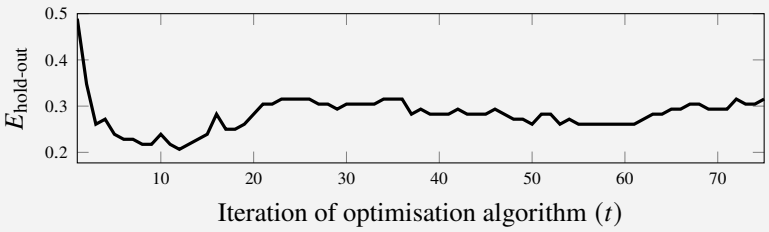



**Fig. 5.12**

As seen in Figure 5.12, $E_{\text{hold-out}}$ reaches a minimum for $t = 12$ and seems to increase thereafter. We do, however, know that the cost function, and thereby probably also $E_{\text{train}}$, decreases monotonically as $t$ increases. The picture, therefore, is that the model suffers from overfitting as $t$ becomes large. The best model (in terms of $E_{\text{hold-out}}$ and, hopefully, in the end also $E_{\text{new}}$) is for this case found after a only a few initial runs of the optimisation, long before it has reached the minimum of the cost function. To illustrate what is happening, we plot in Figure 5.13 the decision boundaries after $t = 1, 12, 75$, and 10 000 iterations, respectively.

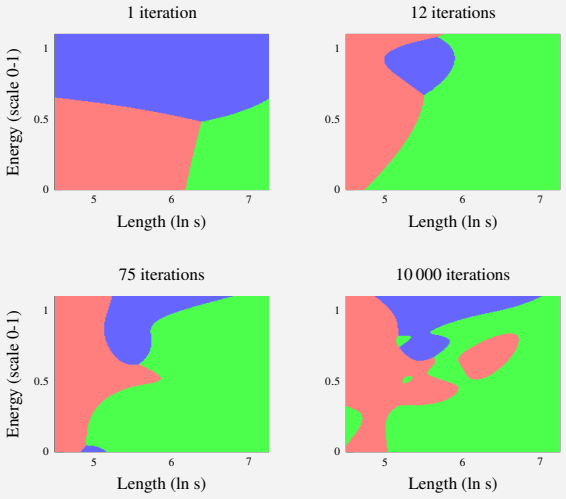


**Fig. 5.13**

It is clear that the shape of the decision boundaries becomes more complicated as $t$ increases, and the number of iterations $t$ can therefore, to some extent, be understood as a way to control the model flexibility. This example is indeed somewhat exaggerated, but the same effect can be seen in particular when training deep neural networks (Chapter 6).

## 5.5 Optimisation with Large Datasets

In machine learning, the training data may have $n$ = millions (or more) of data points. Computing, for example, the gradient of the cost function

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} \nabla_{\boldsymbol{\theta}} L(\mathbf{x}_i, y_i, \boldsymbol{\theta}) \tag{5.36}$$

thus can involve summing a million terms. Besides taking lot of time to sum, it can also be an issue to keep all data points in the computer memory at the same time. However, with that many data points, many of them are probably relatively similar, and in practice we might not need to consider all of them every time: looking at only a subset of them might give sufficient information. This is a general idea called *subsampling*, and we will have a closer look at how subsampling can be combined with gradient descent into a very useful optimisation method called *stochastic gradient descent*. It is, however, also possible to combine the subsampling idea with other methods.

### Stochastic Gradient Descent

With very large $n$, we can expect the gradient computed only for the first half of the dataset $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \approx \sum_{i=1}^{n/2} \nabla_{\boldsymbol{\theta}} L(\mathbf{x}_i, \mathbf{y}_i, \boldsymbol{\theta})$ to be almost identical to the gradient based on the second half of the dataset $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \approx \sum_{i=n/2+1}^{n} \nabla_{\boldsymbol{\theta}} L(\mathbf{x}_i, \mathbf{y}_i, \boldsymbol{\theta})$. Consequently, it might be a waste of time to compute the gradient based on the whole training dataset at each iteration of gradient descent. Instead, we could compute the gradient based on the first half of the training dataset, update the parameters according to the gradient descent method Algorithm 5.1, and then compute the gradient for the new parameters based on the second half of the training data:

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \gamma \frac{1}{n/2} \sum_{i=1}^{\frac{n}{2}} \nabla_{\boldsymbol{\theta}} L(\mathbf{x}_i, \mathbf{y}_i, \boldsymbol{\theta}^{(t)}), \tag{5.37a}$$

$$\boldsymbol{\theta}^{(t+2)} = \boldsymbol{\theta}^{(t+1)} - \gamma \frac{1}{n/2} \sum_{i=\frac{n}{2}+1}^{n} \nabla_{\boldsymbol{\theta}} L(\mathbf{x}_i, \mathbf{y}_i, \boldsymbol{\theta}^{(t+1)}). \tag{5.37b}$$

In other words, we use only a *subsample* of the training data when we compute the gradient. In this way we still make use of all the training data, but it is split


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

into two consecutive parameter updates. Hence, (5.37) requires roughly half the computational time compared to two parameter updates of normal gradient descent. This computational saving illustrates the benefit of the subsampling idea.

We can extend on this idea and consider subsampling with even fewer data points used in each gradient computation. The extreme version of subsampling would be to use only one single data point each time we compute the gradient. In practice it is most common to do something in between. We call a small subsample of data a *mini-batch*, which typically can contain $n_b = 10$, $n_b = 100$, or $n_b = 1\,000$ data points. One complete pass through the training data is called an *epoch*, and consequently consists of $n/n_b$ iterations.

When using mini-batches it is important to ensure that the different mini-batches are balanced and representative for the whole dataset. For example, if we have a big training dataset with a few different output classes, and the dataset is sorted with respect to the output, the mini-batch with the first $n_b$ data points would only include one class and hence not give a good approximation of the gradient for the full dataset. For this reason, the mini-batches should be formed randomly. One implementation of this is to first randomly shuffle the training data, and thereafter divide it into mini-batches in an ordered manner. When we have completed one epoch, we do another random reshuffling of the training data and do another pass through the dataset. We summarise gradient descent with mini-batches, often called *stochastic gradient descent*, as Algorithm 5.3.

Stochastic gradient descent is widely used in machine learning, and there are many extensions tailored to different methods. For training deep neural networks (Chapter 6),

**Algorithm 5.3:** Stochastic gradient descent

**Input:** Objective function $J(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} L(\mathbf{x}_i, y_i, \boldsymbol{\theta})$, initial $\boldsymbol{\theta}^{(0)}$, learning rate $\gamma^{(t)}$

**Result:** $\widehat{\boldsymbol{\theta}}$

Set $t \leftarrow 0$
**do**
  **for** $i = 1, 2, \ldots, E$ **do**
    Randomly shuffle the training data $\{\mathbf{x}_i, y_i\}_{i=1}^{n}$
    **for** $j = 1, 2, \ldots, \frac{n}{n_b}$ **do**
      Approximate the gradient using the mini-batch $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=(j-1)n_b+1}^{jn_b}$, $\widehat{\mathbf{d}}^{(t)} = \frac{1}{n_b}\sum_{i=(j-1)n_b+1}^{jn_b} \nabla_{\boldsymbol{\theta}} L(\mathbf{x}_i, y_i, \boldsymbol{\theta}^{(t)})$.
      Update $\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} - \gamma^{(t)}\widehat{\mathbf{d}}^{(t)}$
      Update $t \leftarrow t + 1$
    **end**
  **end**
**while** *convergence criteria not met*
**return** $\widehat{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta}^{(t-1)}$

some commonly used methods include automatic adaption of the learning rate and an idea called momentum to counteract the randomness caused by subsampling. The AdaGrad (short for adaptive gradient), RMSProp (short for root mean square propagation), and Adam (short for adaptive moments) methods are such examples. For logistic regression in the 'big data' setting, the stochastic average gradient (SAG) method, which averages over all previous gradient estimates, has proven useful, to mention but a few.

### Learning Rate and Convergence for Stochastic Gradient Descent

Standard gradient descent converges if the learning rate is wisely chosen and constant, since the gradient itself is zero at the minimum (or any other stationary point). For stochastic gradient descent, on the other hand, we *cannot* obtain convergence with a constant learning rate. The reason is that we only use an *estimate* of the true gradient, and this estimate will not necessarily be zero at the minimum of the objective function, but there might still be a considerable amount of 'noise' in the gradient estimate due to the subsampling. As a consequence, the stochastic gradient descent algorithm with a constant learning rate will not converge towards a point but will continue to 'wander around', somewhat randomly. For the algorithm to work properly, we also need the gradient estimate to be *unbiased*. The intuitive reason is that the unbiased gradient ensures that the algorithm will on average step in the right direction in its search for the optimum.

By not using a constant learning rate but instead decreasing it gradually towards zero, the parameter updates will be smaller and smaller, and eventually converge. We hence start at $t = 0$ with a fairly high learning rate $\gamma^{(t)}$ (meaning that we take big steps), and then decay $\gamma^{(t)}$ as $t$ increases. Under certain regularity conditions of the cost function and with a learning rate fulfilling the Robbins–Monro conditions $\sum_{t=0}^{\infty} \gamma^{(t)} = \infty$ and $\sum_{t=0}^{\infty} (\gamma^{(t)})^2 < \infty$, the stochastic gradient descent algorithm can be shown to almost surely converge to a local minimum. The Robbins–Monro conditions are, for example, fulfilled if using $\gamma^{(t)} = \frac{1}{t^{\alpha}}, \alpha \in (0.5, 1]$. For many machine learning problems, however, it has been found that better performance is often obtained in practice by not letting $\gamma^{(t)} \to 0$ but to cap it at some small value $\gamma_{\text{min}} > 0$. This will cause stochastic gradient descent not to exactly converge, and the Robbins–Monro conditions will not be fulfilled, but the algorithm will in fact walk around indefinitely (or until the algorithm is aborted by the user). For practical purposes, this seemingly undesirable property does not usually cause any major issue if $\gamma_{\text{min}}$ is small enough, and one heuristic for setting the learning rate in practice is

$$\gamma^{(t)} = \gamma_{\text{min}} + (\gamma_{\text{max}} - \gamma_{\text{min}}) e^{-\frac{t}{\tau}}. \tag{5.38}$$

Now the learning rate $\gamma^{(t)}$ starts at $\gamma_{\text{max}}$ and goes to $\gamma_{\text{min}}$ as $t \to \infty$. How to pick the parameters $\gamma_{\text{min}}$, $\gamma_{\text{max}}$, and $\tau$ is more art than science. As a rule of thumb, $\gamma_{\text{min}}$ can be chosen approximately as 1% of $\gamma_{\text{max}}$. The parameter $\tau$ depends on the size of the


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

dataset and the complexity of the problem, but it should be chosen such that multiple epochs have passed before we reach $\gamma_{\text{min}}$. The strategy for picking $\gamma_{\text{max}}$ can be chosen by monitoring the cost function as for standard gradient descent in Figure 5.7.

**Example 5.8 Stochastic gradient descent**

We apply the stochastic gradient descent method to the objective functions from Example 5.3. For the convex function in Figure 5.14, the choice of learning rate is not very crucial. Note, however, that the algorithm does not converge as nicely as, for example, gradient descent, due to the 'noise' in the gradient estimate caused by the subsampling. This is the price we have to pay for the substantial computational savings offered by the subsampling.

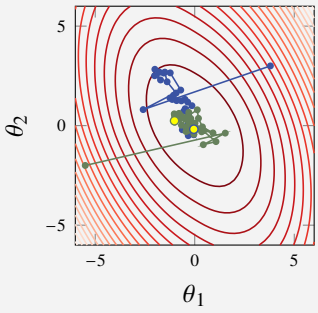



**Fig. 5.14**

For the objective function with multiple local minima, we apply stochastic gradient descent with two decaying learning rates but with different initial $\gamma^{(0)}$ in Figure 5.15. With a smaller learning rate, left, stochastic gradient descent converges to the closest minima, whereas a larger learning rate causes it to initially take larger steps, and it does not therefore necessarily converge to the closest minimum (right).

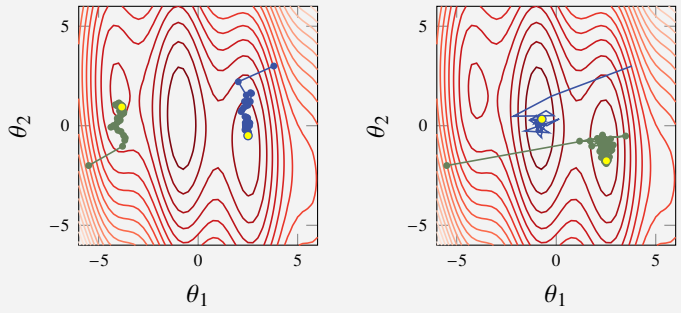



**Fig. 5.15**

## Stochastic Second Order Gradient Methods

The idea of improving the stochastic gradient method by exploiting second order information is natural in settings involving ill-conditioning and significant non-linearity. At the same time, this will add complexity and computational time to the algorithm, which needs to be traded off in the design of these algorithms.

A popular and rather natural strand of algorithms is those falling under the name *stochastic quasi-Newton methods*. As mentioned above, the idea underlying the deterministic quasi-Newton methods is to compute an approximation of the Hessian using information in the gradients. For large-scale problems, we make use of a receding history of gradients. These ideas can also be employed in the stochastic setting, albeit with new algorithms as the result.

### Adaptive Methods

The idea of using gradients from earlier steps is also exploited within the adaptive methods. By considering different ways of combining the gradients from earlier steps into suitable learning rates

$$\gamma_t = \gamma(\nabla J_t, \nabla J_{t-1}, \dots \nabla J_0) \tag{5.39}$$

and search directions

$$d_t = d(\nabla J_t, \nabla J_{t-1}, \dots \nabla J_0), \tag{5.40}$$

we obtain different members of this family of methods. In the basic stochastic gradient algorithms, $d_t$ only depends on the current gradient $\nabla J_t$. The resulting update rule for the adaptive stochastic gradient methods is

$$\theta_{(t+1)} = \theta_{(t)} - \gamma_t d_t. \tag{5.41}$$

The most popular member of this class of methods makes use of an *exponential moving average*, where recent gradients have higher weights than older gradients. Let $\beta_1 < 1$ and $\beta_2 < 1$ denote the exponential weights for the search direction and the learning rate, respectively. The ADAM optimiser then updates the search direction and the learning rate according to

$$d_t = (1 - \beta_1) \sum_{i=1}^{t} \beta_1^{t-i} \nabla J_i, \tag{5.42a}$$

$$\gamma_t = \frac{\eta}{\sqrt{t}} \left( (1 - \beta_2) \mathrm{diag} \left( \sum_{i=1}^{t} \beta_2^{t-i} \|\nabla J_i\|^2 \right) \right)^{1/2}. \tag{5.42b}$$

Both of the tuning parameters $\beta_1$ and $\beta_2$ are typically set to be close to 1, and common values are $\beta_1 = 0.9, \beta_2 = 0.999$. The reason is simply that too small values will effectively result in an exponential forgetting of the past information and remove the – often very valuable – memory effect inherent in this method.


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026, http://smlbook.org*

The first member of this adaptive family is called ADAGRAD, which makes use of the current gradient as its search direction $d_t = \nabla J_t$ and has a learning rate with a memory, but where all components are equally important:

$$\gamma_t = \frac{\eta}{\sqrt{t}} \left( \frac{1}{\sqrt{k}} \operatorname{diag} \left( \sum_{i=1}^{t} \|\nabla J_i\|^2 \right) \right)^{1/2}. \tag{5.43}$$

## 5.6 Hyperparameter Optimisation

Besides the learning parameters of a model, there are quite often also a set of hyperparameters that have to be optimised. As a concrete example, we will use the regularisation parameter $\lambda$, but the discussion below applies to all hyperparameters, as long as they are not of too high dimensions. We can usually estimate $E_{\text{new}}$ as, say, $E_{\text{hold-out}}$, and aim for minimising this.

Writing down an explicit form of $E_{\text{hold-out}}$ as a function of the hyperparameter $\lambda$ can be quite tedious, not to mention taking its derivate. In fact, $E_{\text{hold-out}}$ includes an optimisation problem itself – learning $\widehat{\boldsymbol{\theta}}$ for a given value of $\lambda$. However, we can nevertheless evaluate the objective function for any given $\lambda$, simply by running the entire learning procedure and computing the prediction errors on the validation dataset.

Perhaps the simplest way to solve such an optimisation problem is to 'try a few different parameter values and pick the one which works best'. That is the idea of *grid search* and its likes. The term 'grid' here refers to some (more or less arbitrarily chosen) set of different parameter values to try out, and we illustrate it in Example 5.9.

Although simple to implement, grid search can be computationally inefficient, in particular if the parameter vector has a high dimension. As an example, having a grid with a resolution of 10 grid points per dimension (which is a very coarse-grained grid) for a five-dimensional parameter vector requires $10^5 = 100\,000$ evaluations of the objective function. If possible one should avoid using grid search for this reason. However, with low-dimensional hyperparameters (in $L^1$ and $L^2$ regularisation, $\lambda$ is one-dimensional, for example), grid search can be feasible. We summarise grid search in Algorithm 5.4, where we use it to determine a regularisation parameter $\lambda$.

**Example 5.9 Grid search**

We apply grid search to the objective functions from Example 5.3, with an arbitrarily chosen grid indicated by the blue marks in Figure 5.16. The discovered minimum, which is the grid point with the smallest value of the objective functions, is marked with a yellow dot.

Due to the unfortunate selection of the grid, the global minimum is not found in the non-convex problem (right of Figure 5.16). This problem could be handled by increasing the resolution of the grid, which however requires more computations (more

evaluations of the objective function).

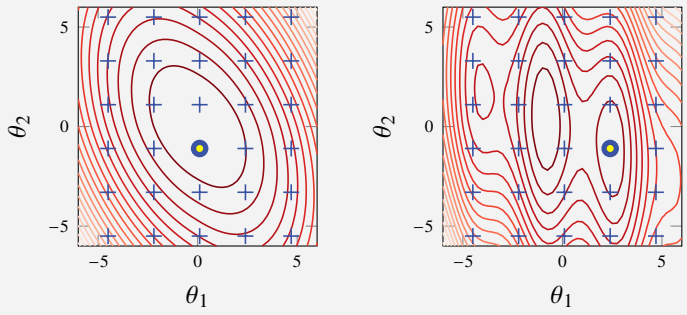



**Fig. 5.16**

**Algorithm 5.4:** Grid search for regularisation parameter $\lambda$

**Input:** Training data $\{\mathbf{x}_i, y_i\}_{i=1}^{n}$, validation data $\{\mathbf{x}_j, y_j\}_{j=1}^{n_v}$

**Result:** $\widehat{\lambda}$

**for** $\lambda = 10^{-3}, 10^{-2}, \ldots, 10^{3}$ *(as an example)* **do**

  Learn $\widehat{\boldsymbol{\theta}}$ with regularisation parameter $\lambda$ from training data

  Compute error on validation data $E_{\text{val}}(\lambda) \leftarrow \frac{1}{n_v} \sum_{j=1}^{n_v} (\widehat{y}(\mathbf{x}_j; \widehat{\boldsymbol{\theta}}) - y_j)^2$

**end**

**return** $\widehat{\lambda}$ as $\arg\min_{\lambda} E_{\text{val}}(\lambda)$

Some hyperparameters (for example $k$ in $k$-NN, Chapter 2) are integers, and sometimes it is feasible to simply try all reasonable integer values in grid search. However, most of the time the major challenge in grid search is to select a good grid. The grid used in Algorithm 5.4 is logarithmic between 0.001 and 1 000, but that is of course only an example. One could indeed do some manual work by first selecting a coarse grid to get an initial guess, and thereafter refine the grid only around the promising candidates, etc. In practice, if the problem has more than one dimension, it can also be beneficial to select the grid points randomly instead of using an equally spaced linear or logarithmic grid.

The manual procedure of choosing a grid might, however, become quite tedious, and one could wish for an automated method. That is, in fact, possible by treating the grid point selection problem as a machine learning problem itself. If we consider the points for which the objective function has already been evaluated as a training dataset, we can use a regression method to learn a model for the objective function. That model can, in turn, be used to answer questions on where to evaluate the objective function next, and thereby automatically select the next grid point. A concrete method built from this idea is the Gaussian process optimisation method, which uses Gaussian processes (Chapter 10) to train a model of the objective function.


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

## 5.7 Further Reading

A mathematically more thorough discussion on loss functions is provided by Gneiting and Raftery (2007). Some of the asymptotic minimisers, also referred to as population minimisers, are derived by Hastie et al. (2009, Section 10.5-10.6).

A standard reference for optimisation, covering much more than this chapter, is the book by Nocedal and Wright (2006). Stochastic gradient descent has its roots in the work on stochastic optimisation by Robbins and Monro (1951), and two overviews of its modern use in machine learning are given by Bottou et al. (2018) and Ruder (2017). For Gaussian process optimisation, see Frazier (2018) and Snoek et al. (2012).

$L^2$ regularisation was introduced independently in statistics by Hoerl and Kennard (1970) and earlier in numerical analysis by Andrey Nikolayevich Tikhonov. $L^1$ regularisation was first introduced by Tibshirani (1996). Early stopping has been used as a regulariser for long time in neural network practice and has been analysed by C. M. Bishop (1995) and Sjöberg and Ljung (1995). For the regularising effect of stochastic gradient descent, see Hardt et al. (2016) and Mandt et al. (2017). A lot has been written about adaptive methods, and many different algorithms are available. The ADAGRAD algorithm was introduced by Duchi et al. (2011), and ADAM was derived by D. P. Kingma and Ba (2015). Interesting insights about these algorithms are offered by Reddi et al. (2018).

# 6 Neural Networks and Deep Learning

In Chapter 3, we introduced linear regression and logistic regression as the two basic parametric models for solving the regression and classification problems. A neural network extends this by stacking multiple copies of these models to construct a hierarchical model that can describe more complicated relationships between inputs and outputs than linear or logistic regression models are able to.

We will start in Section 6.1 by generalising linear regression to a two-layer neural network (that is, a neural network with one hidden layer), and then generalise it further to a *deep* neural network. The use of deep neural networks is commonly referred to as *deep learning*. After introducing the deep neural network model, we will in Section 6.2 discuss how it is learned from training data, and in Section 6.3 and Section 6.4 we provide some methods that are important for building and training well-performing deep neural networks models: normalisation, residual connections and dropout.

In the next chapter, Chapter 7, we will continue the deep learning topic by introducing two important types of deep neural networks that go beyond the dense architecture considered in the present chapter: convolutional neural networks (for grid-like data such as images) and transformers (for sequential data such as text).

## 6.1 The Neural Network Model

In Section 5.1, we introduced the concept of non-linear parametric functions for modelling the relationship between the input variables $x_1, \ldots, x_p$ and the output $y$. We denote this non-linear relationship

$$\widehat{y} = f_{\boldsymbol{\theta}}(x_1, \ldots, x_p), \tag{6.1}$$

where the function $f$ is parametrised by $\boldsymbol{\theta}$. Such a non-linear function can be parametrised in many ways. In a neural network, the strategy is to use several *layers* of linear regression models and non-linear *activation functions*. We will explain carefully what that means step by step below.

### Generalised Linear Regression

We start the description of the neural network model with the linear regression model

$$\widehat{y} = W_1 x_1 + W_2 x_2 + \cdots + W_p x_p + b. \tag{6.2}$$

Here we denote the parameters by the weights $W_1, \ldots, W_p$ and the offset term $b$. In the context of neural networks we will however refer to $b$ as the *bias* term, since it is

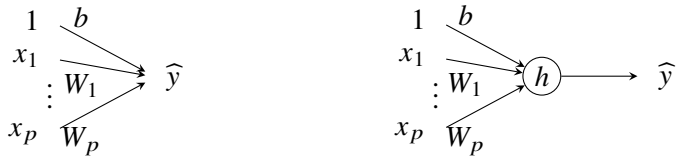



**Figure 6.1:** Graphical illustration of a linear regression model (Figure 6.1a) and a generalised linear regression model (Figure 6.1b). In Figure 6.1a, the output $\hat{y}$ is described as the sum of all terms, $b$ and $\{W_j x_j\}_{j=1}^{p}$, as in (6.2). In Figure 6.1b, the circle denotes addition and also transformation through the activation function $h$, as in (6.3).

the standard terminology. We choose to use this notation instead of the one used in (3.2) since we will later handle the weights slightly differently from the bias term. As before, $x_1, \ldots, x_p$ are the input variables. In Figure 6.1a, a graphical illustration of (6.2) is shown. Each input variable $x_j$ is represented by a node, and each parameter $W_j$ by a link. Furthermore, the output $\widehat{y}$ is described as the sum of all the terms, $W_j x_j$. Note that we use the constant value 1 as the input variable corresponding to the bias term $b$.

To describe *non-linear* relationships between $\mathbf{x} = [1 \;\; x_1 \;\; x_2 \;\; \ldots \;\; x_p]^\mathsf{T}$ and $\widehat{y}$, we introduce a non-linear scalar function called the *activation function* $h : \mathbb{R} \to \mathbb{R}$. The linear regression model (6.2) is now modified into a generalised linear regression model (see Section 3.4) where the linear combination of the inputs is transformed by the activation function

$$\widehat{y} = h\big(W_1 x_1 + W_2 x_2 + \cdots + W_p x_p + b\big). \tag{6.3}$$

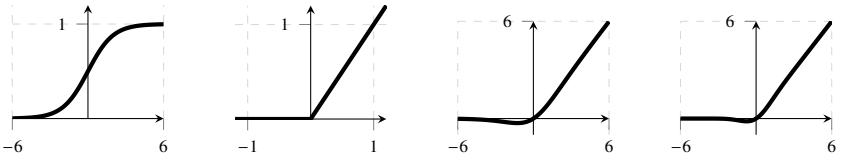
This extension to the generalised linear regression model is visualised in Figure 6.1b.

Common choices of activation function are the *logistic function*, the *rectified linear unit* (ReLU) function, the *Sigmoid linear unit* (SiLU) function or the Gaussian error linear unit (GELU) function.

$$\text{Logistic: } h(z) = \frac{1}{1+e^{-z}}, \quad \text{ReLU: } h(z) = \max(0, z)$$
$$\text{SiLU: } h(z) = \frac{z}{1+e^{-z}}, \quad \text{GELU: } h(z) = z\Phi(z),$$

where $\Phi(z) = \int_{-\infty}^{z} \frac{1}{\sqrt{2\pi}} e^{-\frac{t^2}{2}} dt$ is the cumulative distribution function for the Gaussian distribution. These are illustrated in Figure 6.2. The logistic (or sigmoid) function has already been used in the context of logistic regression (Section 3.2). The logistic function is linear close to $z = 0$ and saturates at 0 and 1 as $z$ decreases or increases. The ReLU is even simpler. The function is just equal to $z$ for positive inputs and equal to zero for negative inputs. The ReLU is possibly the most common choice in modern neural network models, despite (and partly due to) its simplicity.

The generalised linear regression model (6.3) is very simple and is itself not capable of describing particularly complicated relationships between the input $\mathbf{x}$ and




**(a)** Logistic: $\frac{1}{1+e^{-z}}$ **(b)** ReLU: $\max(0, z)$ **(c)** SiLU: $\frac{z}{1+e^{-z}}$ **(d)** GELU: $z\Phi(z)$

**Figure 6.2:** Four common activation functions used in neural networks. The logistic (or sigmoid) function (a), the rectified linear unit function (b), the sigmoid linear unit function (c) and the Gaussian error linear unit function (d).

the output $\widehat{y}$. Therefore, we make two further extensions to increase the generality of the model: We first make use of *several* parallel generalised linear regression models to build a layer (which will lead us to the *two-layer* neural network) and then stack these layers in a *sequential* construction (which will result in a *deep* neural network).

### Two-Layer Neural Network

In (6.3), the output $\widehat{y}$ is predicted using a regression model followed by an activation function. To increase its flexibility and turn it into a two-layer neural network, we instead let its output be a sum of $U$ such generalised linear regression models, each of which has its own set of parameters. The parameters for the $k$th regression model are $b_k, W_{k1}, \ldots, W_{kp}$, and we denote its output by $q_k$,

$$q_k = h\left(W_{k1}x_1 + W_{k2}x_2 + \cdots + W_{kp}x_p + b_k\right), \qquad k = 1, \ldots, U. \tag{6.4}$$

These intermediate outputs $q_k$ are so-called *hidden units*, since they will not be the final output of the model. The $U$ different hidden units $\{q_k\}_{k=1}^{U}$ instead act as input variables to another linear regression model

$$\widehat{y} = W_1 q_1 + W_2 q_2 + \cdots + W_U q_U + b. \tag{6.5}$$

To distinguish the parameters in (6.4) and (6.5), we add the superscripts (1) and (2), respectively. The equations describing this two-layer neural network are thus

$$\begin{aligned}
q_1 &= h\left(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + \cdots + W_{1p}^{(1)}x_p + b_1^{(1)}\right), \\
q_2 &= h\left(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + \cdots + W_{2p}^{(1)}x_p + b_2^{(1)}\right), \\
&\vdots \\
q_U &= h\left(W_{U1}^{(1)}x_1 + W_{U2}^{(1)}x_2 + \cdots + W_{Up}^{(1)}x_p + b_U^{(1)}\right),
\end{aligned} \tag{6.6a}$$

$$\widehat{y} = W_1^{(2)}q_1 + W_2^{(2)}q_2 + \cdots + W_U^{(2)}q_U + b^{(2)}. \tag{6.6b}$$

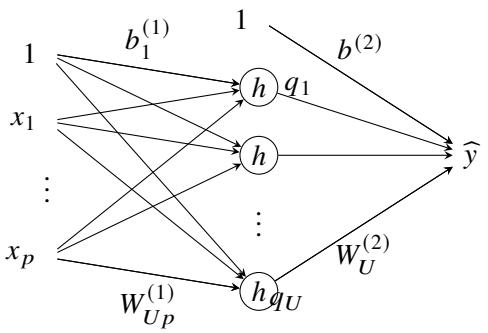



**Figure 6.3:** A two-layer neural network, or equivalently, a neural network with one intermediate layer of hidden units.

Extending the graphical illustration from Figure 6.1, this model can be depicted as a graph with two layers of links (illustrated using arrows); see Figure 6.3. As before, each link has a parameter associated with it. Note that we include an bias term not only in the input layer but also in the hidden layer.

### Vectorisation over Units

The two-layer neural network model in (6.6) can also be written more compactly using matrix notation, where the parameters in each layer are stacked in a *weight matrix* $\mathbf{W}$ and a *bias vector* $\mathbf{b}$ as

$$\mathbf{W}^{(1)} = \begin{bmatrix} W_{11}^{(1)} & \dots & W_{1p}^{(1)} \\ \vdots & & \vdots \\ W_{U1}^{(1)} & \dots & W_{Up}^{(1)} \end{bmatrix}, \quad \mathbf{b}^{(1)} = \begin{bmatrix} b_1^{(1)} \\ \vdots \\ b_U^{(1)} \end{bmatrix},$$
$$\mathbf{W}^{(2)} = \begin{bmatrix} W_1^{(2)} & \dots & W_U^{(2)} \end{bmatrix}, \quad \mathbf{b}^{(2)} = \begin{bmatrix} b^{(2)} \end{bmatrix}. \tag{6.7}$$

The full model can then be written as

$$\mathbf{q} = h\big(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\big), \tag{6.8a}$$
$$\widehat{y} = \mathbf{W}^{(2)}\mathbf{q} + \mathbf{b}^{(2)}, \tag{6.8b}$$

where we have also stacked the components in $\mathbf{x}$ and $\mathbf{q}$ as $\mathbf{x} = \begin{bmatrix} x_1 & \dots & x_p \end{bmatrix}^\mathsf{T}$ and $\mathbf{q} = \begin{bmatrix} q_1 & \dots & q_U \end{bmatrix}^\mathsf{T}$. Note that the activation function $h$ in (6.8a) acts element-wise on the input vector and results in an output vector of the same dimension. The two weight matrices and the two offset vectors are the parameters of the model, which can be written as

$$\boldsymbol{\theta} = \begin{bmatrix} \mathrm{vec}(\mathbf{W}^{(1)})^\mathsf{T} & \mathbf{b}^{(1)\mathsf{T}} & \mathrm{vec}(\mathbf{W}^{(2)})^\mathsf{T} & \mathbf{b}^{(2)\mathsf{T}} \end{bmatrix}^\mathsf{T}, \tag{6.9}$$


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

where the operator vec takes all elements in the matrix and puts them into a vector. Taken all together, (6.8) now describes a non-linear regression model of the form $\widehat{y} = f_{\boldsymbol{\theta}}(\mathbf{x})$.

### Deep Neural Network

The two-layer neural network is a useful model on its own, and a lot of research and analysis has been done on it. However, the real descriptive power of a neural network is achieved by stacking multiple such layers of generalised linear regression models and thereby achieve a *deep* neural network. Deep neural networks are capable of modelling very complicated relationships, such as the one between an image and a text describing its content, and is the flagship method of modern machine learning.

A deep neural network contains multiple layers. Let us denote the total number of layers as $L$ and enumerate the layers with index $l \in \{1, \ldots, L\}$. Each layer is parametrised with a weight matrix $\mathbf{W}^{(l)}$ and an bias vector $\mathbf{b}^{(l)}$, as for the two-layer case. For example, $\mathbf{W}^{(1)}$ and $\mathbf{b}^{(1)}$ belong to layer $l = 1$, $\mathbf{W}^{(2)}$ and $\mathbf{b}^{(2)}$ belong to layer $l = 2$, and so forth. We also have multiple *layers of hidden units* denoted by $\mathbf{q}^{(l)}$. Each such layer consists of $U_l$ hidden units $\mathbf{q}^{(l)} = [q_1^{(l)} \quad \ldots \quad q_{U_l}^{(l)}]^\mathsf{T}$, where the dimensions $U_1, \ldots, U_{L-1}$ can be different across the various layers.

Each layer maps a vector of hidden units $\mathbf{q}^{(l-1)}$ to the next vector of hidden units $\mathbf{q}^{(l)}$ according to

$$\mathbf{q}^{(l)} = h\big(\mathbf{W}^{(l)}\mathbf{q}^{(l-1)} + \mathbf{b}^{(l)}\big). \tag{6.10}$$

This means that the layers are stacked such that the output of the first layer of hidden units $\mathbf{q}^{(1)}$ is the input to the second layer, the output of the second layer $\mathbf{q}^{(2)}$ (the second layer of hidden units) is the input to the third layer, etc. By stacking multiple layers we have constructed a *deep* neural network. A deep neural network of $L$ layers can be described mathematically as

$$\begin{aligned}
\mathbf{q}^{(1)} &= h\big(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\big), \\
\mathbf{q}^{(2)} &= h\big(\mathbf{W}^{(2)}\mathbf{q}^{(1)} + \mathbf{b}^{(2)}\big), \\
&\vdots \\
\mathbf{q}^{(L-1)} &= h\big(\mathbf{W}^{(L-1)}\mathbf{q}^{(L-2)} + \mathbf{b}^{(L-1)}\big), \\
\widehat{y} &= \mathbf{W}^{(L)}\mathbf{q}^{(L-1)} + \mathbf{b}^{(L)}.
\end{aligned} \tag{6.11}$$

A graphical representation of this model is provided in Figure 6.4. The expression (6.11) for a deep neural network can be compared with the expression (6.8) for a two-layer neural network.

The weight matrix $\mathbf{W}^{(1)}$ for the first layer $l = 1$ has dimension $U_1 \times p$, and the corresponding bias vector $\mathbf{b}^{(1)}$ has dimension $U_1$. Since the output here is assumed to be scalar, in the last layer the weight matrix $\mathbf{W}^{(L)}$ has dimension $1 \times U_{L-1}$, and the bias vector $\mathbf{b}^{(L)}$ has dimension 1. For all intermediate layers $l = 2, \ldots, L-1$,

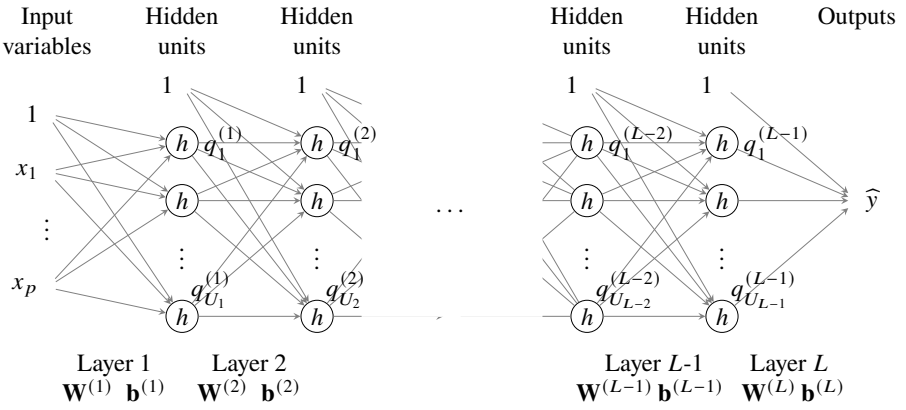



**Figure 6.4:** A deep neural network with $L$ layers. Each layer $l$ is parameterised by $\mathbf{W}^{(l)}$ and $\mathbf{b}^{(l)}$.

$\mathbf{W}^{(l)}$ has dimension $U_l \times U_{l-1}$, and the corresponding bias vector has dimension $U_l$. The number of inputs $p$ is given by the problem, but the number of layers $L$ and the dimensions $U_1, U_2, \ldots$ are user design choices that determine the flexibility of the model.

### Vectorisation over Data Points

During training, the neural network computes the predicted output, not only for one input $\mathbf{x}$ but for several inputs $\{\mathbf{x}_i\}_{i=1}^n$. For example, for the two-layer neural network presented in Section 6.1, we have

$$\mathbf{q}_i^\mathsf{T} = h\big(\mathbf{x}_i^\mathsf{T}\mathbf{W}^{(1)\mathsf{T}} + \mathbf{b}^{(1)\mathsf{T}}\big), \tag{6.12a}$$

$$\widehat{y}_i = \mathbf{q}_i^\mathsf{T}\mathbf{W}^{(2)\mathsf{T}} + \mathbf{b}^{(2)\mathsf{T}}, \qquad i = 1, \ldots, n. \tag{6.12b}$$

Similar to the vectorisation over units explained earlier, we also want to vectorise these equations over the data points to allow for efficient computation of the model. Note that the equations (6.12) are transposed in comparison to the model in (6.8). With this notation, we can, similarly to the linear regression model (3.5), stack all data points in matrices, where each data point represents one row:

$$\mathbf{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}, \qquad \mathbf{X} = \begin{bmatrix} \mathbf{x}_1^\mathsf{T} \\ \vdots \\ \mathbf{x}_n^\mathsf{T} \end{bmatrix}, \quad \widehat{\mathbf{y}} = \begin{bmatrix} \widehat{y}_1 \\ \vdots \\ \widehat{y}_n \end{bmatrix}, \quad \text{and} \quad \mathbf{Q} = \begin{bmatrix} \mathbf{q}_1^\mathsf{T} \\ \vdots \\ \mathbf{q}_n^\mathsf{T} \end{bmatrix}. \tag{6.13}$$

We can then conveniently write (6.12) as

$$\mathbf{Q} = h\left(\mathbf{X}\mathbf{W}^{(1)\mathsf{T}} + \mathbf{b}^{(1)\mathsf{T}}\right), \tag{6.14a}$$

$$\widehat{\mathbf{y}} = \mathbf{Q}\mathbf{W}^{(2)\mathsf{T}} + \mathbf{b}^{(2)\mathsf{T}}, \tag{6.14b}$$


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

where we have also stacked the predicted output and the hidden units in matrices. Note that the transposed bias vectors $\mathbf{b}^{(1)\mathsf{T}}$ and $\mathbf{b}^{(2)\mathsf{T}}$ are added to each row in this notation.

The vectorised equations in (6.14) are also how the model would typically be implemented in languages that support array programming. Such vectorised implementations are in practice necessary for implementing larger deep neural networks, because of the sheer amount of computations needed.

### Neural Networks for Classification

So far we have only formulated neural networks for regression, but they can also be used for classification, where we have categorical outputs $y \in \{1, \ldots, M\}$ instead of numerical. In Section 3.2, we extended linear regression to logistic regression by simply adding the logistic function to the output. In the same manner, we can extend the neural network for regression presented in the previous section to a neural network for classification. In doing so, we use the multiclass version of logistic regression presented in Section 3.2 and more specifically the softmax parametrisation (3.41), repeated here for convenience:

$$\operatorname{softmax}(\mathbf{z}) \triangleq \frac{1}{\sum_{j=1}^{M} e^{z_j}} \begin{bmatrix} e^{z_1} \\ e^{z_2} \\ \vdots \\ e^{z_M} \end{bmatrix}. \tag{6.15}$$

The model is constructed as in (6.11), but with the output now being of dimension $M$. The softmax function now acts like an additional activation function on the final layer of the neural network:

$$\mathbf{q}^{(1)} = h\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right), \tag{6.16a}$$

$$\vdots$$

$$\mathbf{q}^{(L-1)} = h\left(\mathbf{W}^{(L-1)}\mathbf{q}^{(L-2)} + \mathbf{b}^{(L-1)}\right), \tag{6.16b}$$

$$\mathbf{z} = \mathbf{W}^{(L)}\mathbf{q}^{(L-1)} + \mathbf{b}^{(L)}, \tag{6.16c}$$

$$\mathbf{g} = \operatorname{softmax}(\mathbf{z}). \tag{6.16d}$$

The softmax function maps the output of the last layer $\mathbf{z} = [z_1, \ldots, z_M]^\mathsf{T}$ to $\mathbf{g} = [g_1, \ldots, g_M]^\mathsf{T}$, where $g_m$ is a model of the class probability $p(y = m \mid \mathbf{x})$. The input variables $z_1, \ldots, z_M$ to the softmax function are referred to as *logits*. Note that the softmax function does not come as a layer with additional parameters; it merely acts as a transformation of the output into the modelled class probabilities. By construction, the outputs of the softmax function will always be in the interval $g_m \in [0, 1]$ and sum to $\sum_{m=1}^{M} g_m = 1$, otherwise they could not be interpreted as probabilities. Since the output now has dimension $M$, the last layer of the weight matrix $\mathbf{W}^{(L)}$ has dimension $M \times U_{L-1}$, and the bias vector $\mathbf{b}^{(L)}$ has dimension $M$.

**Example 6.1 Classification of hand-written digits – problem formulation**

We consider the so-called MNIST dataset,[a] which used to be one of the most well studied image classification datasets in the early days of the neural network models. The dataset has 60 000 training data points and 10 000 validation data points. Each data point consists of a $28 \times 28$ pixel greyscale image of a handwritten digit. The digit has been size-normalised and centred within a fixed-sized image. Each image is also labelled with the digit 0, 1, . . . , 8, or 9 that it is depicting. Figure 6.5 shows a batch of 20 data points from this dataset.

**Fig. 6.5**

In this classification task, we consider the image as our input $\mathbf{x} = [x_1, \ldots x_p]^\mathsf{T}$. Each input variable $x_j$ corresponds to a pixel in the image. In total we have $p = 28 \times 28 = 784$ input variables which we flatten out into one long vector.[b] The value of each $x_j$ represents the intensity of that pixel. The intensity-value is within the interval $[0, 1]$, where $x_j = 0$ corresponds to a black pixel and $x_j = 1$ to a white pixel. Anything between 0 and 1 is a grey pixel with corresponding intensity. The output is the class $y \in \{0, \ldots, 9\}$. This means that we have in total 10 classes representing the 10 digits. Based on a set of training data $\{\mathbf{x}_i, y_i\}_{i=1}^n$ with images and labels, the problem is to find a good model for the class probabilities

$$p(y = m \,|\, \mathbf{x}), \qquad m = 0, \ldots, 9,$$

in other words, the probabilities that an unseen image $\mathbf{x}$ belongs to each of the $M = 10$ classes. Assume that we would like to use logistic regression to solve this problem with a softmax output. This is identical to a neural network with just one layer – that is, (6.16) where $L = 1$. The parameters of that model would be

$$\mathbf{W}^{(1)} \in \mathbb{R}^{784 \times 10} \quad \mathbf{b}^{(1)} \in \mathbb{R}^{10},$$

which gives in total $784 \cdot 10 + 10 = 7\,850$ parameters. If we would like to extend this model with a two-layer neural network with $U = 200$ hidden units, that would require two sets of weight matrices and bias vectors:

$$\mathbf{W}^{(1)} \in \mathbb{R}^{784 \times 200} \quad \mathbf{b}^{(1)} \in \mathbb{R}^{200}, \quad \mathbf{W}^{(2)} \in \mathbb{R}^{200 \times 10} \quad \mathbf{b}^{(2)} \in \mathbb{R}^{10},$$

which is a model with $784 \cdot 200 + 200 + 200 \cdot 10 + 10 = 159\,010$ parameters. In the next section, we will learn how to fit all of these parameters to the training data.

[a] `http://yann.lecun.com/exdb/mnist/`

[b] By flattening, we disregard information about the structure of the image. In Section 7.1, we will look at another neural network model where this spatial information is preserved.



*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

### Different Types of Neural Network Models

Before we dive into the important topic of how to learn a neural network model from training data, let us add a few notes on the terminology for neural network models.

First of all, the design of the neural network model is often referred to as its *architecture*. The two-layer neural network model (6.8) is for historical reasons also referred to as a Multi-Layer Perceptron, or just *MLP* for short. The entire family of models we have described so far are so-called *dense* or *fully connected*, meaning there is a connection from each incoming unit to each outgoing unit in each layer. Such architectures are arguably the most flexible type, which is not always a desired property because of the risk of overfitting, poor generalisation properties and poor scaling properties when increasing, for example, the input dimension. In the next chapter we will introduce two other architectures that are not fully connected. For certain types of data, both of these architectures reduce the number of parameters while also helping the model to generalise better.

Finally, all neural networks introduced here are so-called *feedforward* neural networks, in the sense that the computational flow is going strictly from input to output without any internal loops or feedback. This is in contrast to *recurrent* neural networks, which are out of scope for this book.

## 6.2 Training a Neural Network

A neural network is a parametric model, meaning that we have to learn its parameters from training data using the methods explained in Chapter 5. The parameters in the neural network model are all the weight matrices and all the bias vectors:

$$\boldsymbol{\theta} = \begin{bmatrix} \operatorname{vec}(\mathbf{W}^{(1)})^\mathsf{T} & \mathbf{b}^{(1)\mathsf{T}} & \cdots & \operatorname{vec}(\mathbf{W}^{(L)})^\mathsf{T} & \mathbf{b}^{(L)\mathsf{T}} \end{bmatrix}^\mathsf{T}. \tag{6.17}$$

To find parameter values $\boldsymbol{\theta}$ that fit the training data, we solve an optimisation problem of the form

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}), \qquad \text{where } J(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} L(\mathbf{x}_i, y_i, \boldsymbol{\theta}), \tag{6.18}$$

where $L(\mathbf{x}_i, \mathbf{y}_i, \boldsymbol{\theta})$ is the loss function and $J(\boldsymbol{\theta})$ is the resulting cost function to optimise. What particular loss function to use depends on the problem at hand, but the typical choice for a regression problem is (as for linear regression) the squared error loss (5.6),

$$L(\mathbf{x}, y, \boldsymbol{\theta}) = \big(y - f(\mathbf{x}; \boldsymbol{\theta})\big)^2, \tag{6.19}$$

(where $f(\mathbf{x}; \boldsymbol{\theta})$ is the output of the neural network), and the cross-entropy loss function (3.44) for multiclass classification (just like multiclass logistic regression),

$$L(\mathbf{x}, y, \boldsymbol{\theta}) = -\ln g_y\big(\mathbf{f}(\mathbf{x}; \boldsymbol{\theta})\big) = -z_y + \ln \sum_{j=1}^{M} e^{z_j}. \tag{6.20}$$

Here $z_j = f_j(\mathbf{x}; \boldsymbol{\theta})$ is the $j$th logit – that is, the $j$th output of the last layer before the softmax function $\mathbf{g}(\mathbf{z})$. Also, similar to the notation in (3.44), we use the training data label $y$ as an index variable to select the correct logit for the loss function.

These are just typical choices, but following the discussion in Section 5.2 we could use also other loss functions that are relevant for the problem at hand.

Optimising the cost function cannot be done in closed form for neural networks, so numerical optimisation has to be used. Optimisation was discussed in Chapter 5, and in short it amounts to

1. Pick an initialisation $\boldsymbol{\theta}_0$.
2. Update the parameters as $\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \gamma \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)$ for $t = 0, 1, 2, \ldots$. (6.21)
3. Terminate when some criterion is fulfilled, and take the last $\boldsymbol{\theta}_t$ as $\widehat{\boldsymbol{\theta}}$.

For many deep learning applications, there are typically two simultaneous challenges:

- $n$ **is large.** The first computational challenge is the big data problem. For many deep learning applications, the number of data points $n$ is very large, making the computation of the cost function and its gradient very costly since it requires sums over all data points. As a consequence, we cannot afford to compute the exact gradient $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)$ at each iteration. For this reason, stochastic gradient descent, as explained in Section 5.5, is used. In short, we compute only an approximation of the gradient by just considering a random subset of the training data at each iteration, which we refer to as a mini-batch.

- $\dim(\boldsymbol{\theta})$ **is large.** The second computational challenge is that the number of parameters $\dim(\boldsymbol{\theta})$ is also very large for deep learning problems. To efficiently compute the gradient $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)$, we apply the chain rule of calculus and reuse the partial derivatives needed to compute this gradient. This is called the backpropagation algorithm, which is explained in the next section.

### Backpropagation

The backpropagation algorithm is a way to programmatically compute the gradient $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)$ of a neural network, where the parameters in $\boldsymbol{\theta}$ typically have complicated dependencies on each other because of the stacking of multiple layers. On a mathematical level, it is an algorithmic formulation of the chain rule for derivatives when applied to a deep neural network.

The parameters $\boldsymbol{\theta}$ are again all the weight matrices and all the bias vectors. Hence, at each iteration in our gradient-based search algorithm (6.21), we also need to find the gradient of the cost function with respect to all elements in these matrices and

vectors. To summarise, we want to find

$$d\mathbf{W}^{(l)} \triangleq \nabla_{\partial \mathbf{W}^{(l)}} J(\boldsymbol{\theta}) = \begin{bmatrix} \frac{\partial J(\boldsymbol{\theta})}{\partial W_{11}^{(l)}} & \cdots & \frac{\partial J(\boldsymbol{\theta})}{\partial W_{1,U_{l-1}}^{(l)}} \\ \vdots & & \vdots \\ \frac{\partial J(\boldsymbol{\theta})}{\partial W_{U_l,1}^{(l)}} & \cdots & \frac{\partial J(\boldsymbol{\theta})}{\partial W_{U_l,U_{l-1}}^{(l)}} \end{bmatrix} \quad \text{and}$$

$$d\mathbf{b}^{(l)} \triangleq \nabla_{\partial \mathbf{b}^{(l)}} J(\boldsymbol{\theta}) = \begin{bmatrix} \frac{\partial J(\boldsymbol{\theta})}{\partial b_1^{(l)}} \\ \vdots \\ \frac{\partial J(\boldsymbol{\theta})}{\partial b_{U_l}^{(l)}} \end{bmatrix}. \tag{6.22}$$

for all $l = 1, \ldots, L$. Note that the cost function $J(\boldsymbol{\theta})$ here only includes the losses in the current mini-batch. When these gradients are computed, we can update the weight matrices and bias vectors accordingly:

$$\mathbf{W}_{t+1}^{(l)} \leftarrow \mathbf{W}_t^{(l)} - \gamma d\mathbf{W}_t^{(l)}, \tag{6.23a}$$

$$\mathbf{b}_{t+1}^{(l)} \leftarrow \mathbf{b}_t^{(l)} - \gamma d\mathbf{b}_t^{(l)}. \tag{6.23b}$$

To compute all these gradients efficiently, backpropagation exploits the structure of the model instead of naively computing the derivatives with respect to each single parameter separately. To do that, backpropagation uses the chain rule of calculus.

First, we describe how backpropagation works for one single data point $(\mathbf{x}, y)$. In Algorithm 6.1, we later generalise this to multiple data points. The backpropagation algorithm consists of two steps, the forward propagation and the backward propagation. In the forward propagation, we simply evaluate the cost function using the neural network model we presented in Section 6.1. We start with the input $\mathbf{x}$ and evaluate each layer sequentially from layer 1 to the last layer $L$: hence, we propagate forwards:

$$\mathbf{q}^{(0)} = \mathbf{x}, \tag{6.24a}$$

$$\begin{cases} \mathbf{z}^{(l)} = \mathbf{W}^{(l)}\mathbf{q}^{(l-1)} + \mathbf{b}^{(l)}, \\ \mathbf{q}^{(l)} = h(\mathbf{z}^{(l)}), \end{cases} \qquad \text{for } l = 1, \ldots, L-1 \tag{6.24b}$$

$$\mathbf{z}^{(L)} = \mathbf{W}^{(L)}\mathbf{q}^{(L-1)} + \mathbf{b}^{(L)}, \tag{6.24c}$$

$$J(\boldsymbol{\theta}) = \begin{cases} \left(y - z^{(L)}\right)^2, & \text{if regression problem,} \\ -z_y^{(L)} + \ln \sum_{j=1}^{M} e^{z_j^{(L)}}, & \text{if classification problem.} \end{cases} \tag{6.24d}$$

Note that, since we only consider one data point, the cost function $J(\boldsymbol{\theta})$ will only include one loss term.

When it comes to backward propagation, we need to introduce some additional notation, namely the gradient of the cost $J$ with respect to the hidden units $\mathbf{z}^{(l)}$ and

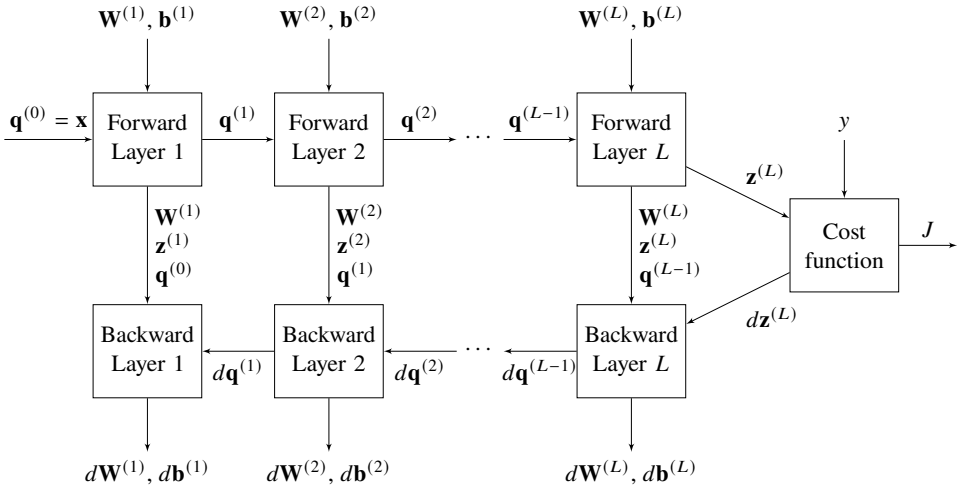



**Figure 6.6:** A computational graph of the backpropagation algorithm. We start with the input in the upper left corner and propagate forward and evaluate the cost function. Along the way, we also cache the values for the hidden units $\mathbf{q}^{(l)}$ and $\mathbf{z}^{(l)}$. We then propagate the gradients $d\mathbf{q}^{(l)}$ and $d\mathbf{z}^{(l)}$ backwards and compute the gradients for the parameters $\mathbf{W}^{(l)}$, $\mathbf{b}^{(l)}$. The equations behind this computational graph are given in (6.24), (6.26), and (6.27).

$\mathbf{q}^{(l)}$, given by

$$d\mathbf{z}^{(l)} \triangleq \nabla_{\mathbf{z}^{(l)}} J(\boldsymbol{\theta}) = \begin{bmatrix} \frac{\partial J(\boldsymbol{\theta})}{\partial z_1^{(l)}} \\ \vdots \\ \frac{\partial J(\boldsymbol{\theta})}{\partial z_{U^{(l)}}^{(l)}} \end{bmatrix} \quad \text{and} \quad d\mathbf{q}^{(l)} \triangleq \nabla_{\mathbf{q}^{(l)}} J(\boldsymbol{\theta}) = \begin{bmatrix} \frac{\partial J(\boldsymbol{\theta})}{\partial q_1^{(l)}} \\ \vdots \\ \frac{\partial J(\boldsymbol{\theta})}{\partial q_{U^{(l)}}^{(l)}} \end{bmatrix}. \tag{6.25}$$

In the backward propagation, we compute the gradients $d\mathbf{z}^{(l)}$ and $d\mathbf{q}^{(l)}$ for all layers. We do this recursively, in the opposite direction to what we did in the forward propagation – that is, starting from the last layer $L$, we propagate back to the first layer. To start these recursions, we first need to compute $d\mathbf{z}^{(L)}$, the gradient of the cost function with respect to the hidden units in the last hidden layer. This depends on our choice of loss function (6.24d). If we have a regression problem and choose the squared error loss, we get

$$dz^{(L)} = \frac{\partial J(\boldsymbol{\theta})}{\partial z^{(L)}} = \frac{\partial}{\partial z^{(L)}} \big(y - z^{(L)}\big)^2 = -2\big(y - z^{(L)}\big). \tag{6.26a}$$

For a multiclass classification problem, we instead use the cross-entropy loss (6.20), and we get

$$dz_j^{(L)} = \frac{\partial J(\boldsymbol{\theta})}{\partial z_j^{(L)}} = \frac{\partial}{\partial z_j^{(l)}} \left( -z_y^{(L)} + \ln \sum_{k=1}^{M} e^{z_k^{(L)}} \right) = -\mathbb{I}\{y = j\} + \frac{e^{z_j^{(L)}}}{\sum_{k=1}^{M} e^{z_k^{(L)}}}. \tag{6.26b}$$


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

The backwards propagation now proceeds with the following recursions:

$$d\mathbf{z}^{(l)} = d\mathbf{q}^{(l)} \odot h'(\mathbf{z}^{(l)}), \tag{6.27a}$$

$$d\mathbf{q}^{(l-1)} = \mathbf{W}^{(l)\mathsf{T}} d\mathbf{z}^{(l)}, \tag{6.27b}$$

where $\odot$ denotes the element-wise product and where $h'(z)$ is the derivative of the activation function $h(z)$. Similar to the notation in (6.24b), $h'(\mathbf{z})$ acts element-wise on the vector $\mathbf{z}$. Note that the first line (6.27a) is not applied to layer $L$ since that layer does not have an activation function; see (6.24c).

With $d\mathbf{z}^{(l)}$, we can now compute the gradients of the weight matrices and bias vectors as

$$d\mathbf{W}^{(l)} = d\mathbf{z}^{(l)} \mathbf{q}^{(l-1)\mathsf{T}}, \tag{6.27c}$$

$$d\mathbf{b}^{(l)} = d\mathbf{z}^{(l)}. \tag{6.27d}$$

All equations for the backward propagation (6.27) can be derived from the chain rule of calculus; for further details see Appendix 6.A. A computational graph of the backpropagation algorithm is summarised in Figure 6.6.

So far, we have only considered backpropagation for one data point $(\mathbf{x}, y)$. However, we do want to compute the cost function $J(\boldsymbol{\theta})$ and its gradients $d\mathbf{W}^{(l)}$ and $d\mathbf{b}^{(l)}$ for the whole mini-batch $\{\mathbf{x}_i, y_i\}_{i=(j-1)n_b+1}^{jn_b}$. Therefore, we run the equations (6.24), (6.26), and (6.27) for all data points in the current mini-batch and average their results for $J$, $d\mathbf{W}^{(l)}$, and $d\mathbf{b}^{(l)}$. To do this in a computationally efficient manner, we process all $n_b$ data points in the mini-batch simultaneously by stacking them in matrices as we did in (6.14), where each row represents one data point (with the indices in the mini batch starting at 1 to get a not too cluttered notation):

$$\mathbf{Q} = \begin{bmatrix} \mathbf{q}_1^\mathsf{T} \\ \vdots \\ \mathbf{q}_{n_b}^\mathsf{T} \end{bmatrix}, \qquad \mathbf{Z} = \begin{bmatrix} \mathbf{z}_1^\mathsf{T} \\ \vdots \\ \mathbf{z}_{n_b}^\mathsf{T} \end{bmatrix}, \qquad d\mathbf{Q} = \begin{bmatrix} d\mathbf{q}_1^\mathsf{T} \\ \vdots \\ d\mathbf{q}_{n_b}^\mathsf{T} \end{bmatrix} \quad \text{and} \quad d\mathbf{Z} = \begin{bmatrix} d\mathbf{z}_1^\mathsf{T} \\ \vdots \\ d\mathbf{z}_{n_b}^\mathsf{T} \end{bmatrix}. \tag{6.28}$$

The complete algorithm we then get is summarised as Algorithm 6.1.

### Initialisation

Most of the optimisation problems for simpler models (such as $L^1$-regularised regression or logistic regression) that we have encountered have been convex. That implies that an optimisation algorithm will be able to find the optimum regardless of the initial parameter values $\boldsymbol{\theta}_0$ that are used when starting the optimisation. In contrast, the cost functions for training neural networks are usually non-convex, and

[1] One might want to consider normalising the logits $Z_{ij}^{(L)} \leftarrow Z_{ij}^{(L)} - \max_j Z_{ij}^{(L)}$ before computing the cost to avoid potential overflow while computing $\exp\left(Z_{ij}^{(L)}\right)$. Note that this normalisation does not change the value of the cost function.

**Algorithm 6.1:** Backpropagation

**Input:** Parameters $\boldsymbol{\theta} = \{\mathbf{W}^{(l)}, \mathbf{b}^{(l)}\}_{l=1}^{L}$, activation function $h$, and data $\mathbf{X}$, $\mathbf{y}$, with $n_b$ rows, where each row corresponds to one data point in the current mini-batch.

**Result:** $J(\boldsymbol{\theta})$, $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ of the current mini-batch.

*Forward propagation*
Set $\mathbf{Q}^0 \leftarrow \mathbf{X}$
**for** $l = 1, \ldots, L$ **do**
  $\mathbf{Z}^{(l)} = \mathbf{Q}^{(l-1)}\mathbf{W}^{(l)\mathsf{T}} + \mathbf{b}^{(l)\mathsf{T}}$
  $\mathbf{Q}^{(l)} = h(\mathbf{Z}^{(l)})$  *Do not execute this line for last layer $l = L$*
**end**
*Evaluate cost function*
**if** *Regression problem* **then**
  $J(\boldsymbol{\theta}) = \frac{1}{n_b}\sum_{i=1}^{n_b}\left(y_i - Z_i^{(L)}\right)^2$
  $d\mathbf{Z}^{(L)} = -2\left(\mathbf{y} - \mathbf{Z}^{(L)}\right)$
**else if** *Classification problem*[1] **then**
  $J(\boldsymbol{\theta}) = \frac{1}{n_b}\sum_{i=1}^{n_b}\left(-Z_{i,y_i}^{(L)} + \ln\left(\sum_{j=1}^{M}\exp\left(Z_{ij}^{(L)}\right)\right)\right)$
  $dZ_{ij}^{(L)} = -\mathbb{I}\{y_i = j\} + \frac{\exp\left(Z_{ij}^{(L)}\right)}{\sum_{j=1}^{M}\exp\left(Z_{ij}^{(L)}\right)}$  $\forall i, j$
*Backward propagation*
**for** $l = L, \ldots, 1$ **do**
  $d\mathbf{Z}^{(l)} = d\mathbf{Q}^{(l)} \odot h'(\mathbf{Z}^{(l)})$  *Do not execute this line for last layer $l = L$*
  $d\mathbf{Q}^{(l-1)} = d\mathbf{Z}^{(l)}\mathbf{W}^{(l)}$
  $d\mathbf{W}^{(l)} = \frac{1}{n_b} d\mathbf{Z}^{(l)\mathsf{T}}\mathbf{Q}^{(l-1)}$
  $db_j^{(l)} = \frac{1}{n_b}\sum_{i=1}^{n_b} dZ_{ij}^{(l)}$  $\forall j$
**end**
$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \begin{bmatrix}\mathrm{vec}(d\mathbf{W}^{(1)})^\mathsf{T} & d\mathbf{b}^{(1)\mathsf{T}} & \cdots & \mathrm{vec}(d\mathbf{W}^{(L)})^\mathsf{T} & d\mathbf{b}^{(L)\mathsf{T}}\end{bmatrix}$
**return** $J(\boldsymbol{\theta})$, $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$

the training of those models is sensitive to the value of the initial parameters. If training a deep neural network from scratch, one typically initialises all the parameters to small random numbers. There are principled methods for achieving such random initialisations that fit well to the activation function used, so that the optimization problem starts in a point where the gradients of the activation functions are interesting and non-zero.

However, with huge modern deep neural network models it is common to obtain a good initialisation by copying the weights from another model that has a similar architecture and has been trained on a similar problem. This practice is called *transfer*

*learning*.

**Example 6.2 Classification of hand-written digits – first attempt**

We consider the example of classifying hand-written digits, introduced in Example 6.1. Based on the presented data, we train the two models mentioned at the end of that example: a logistic regression model (or equivalently a one-layer neural network) and a two-layer neural network.

We use stochastic gradient descent, explained in Algorithm 5.3, with learning rate $\gamma = 0.5$ and a mini-batch size of $n_b = 100$ data points. Since we have $n = 60\,000$ training data points in total, one epoch is completed after $6\,000/100 = 600$ iterations. We run the algorithm for 15 epochs, i.e., 9 000 iterations.

As explained earlier, at each iteration of the algorithm, the cost function is evaluated for the current mini-batch of the training data. The value for this cost function is illustrated in blue in the left parts of Figures 6.7 and 6.8. In addition, we also compute the misclassification rate for the 100 data points in the current mini-batch. This is illustrated in the right parts of Figures 6.7 and 6.8. Since the current mini-batch consists of 100 randomly selected training data points, the performance fluctuates depending on which mini-batch that is considered.

However, the important measure is how we perform on data which has not been used during training. Therefore, at every 100th iteration, we also evaluate the cost function and the misclassification error on the whole validation dataset of 10 000 data points. This performance is illustrated in red in Figures 6.7 and 6.8.

**Logistic regression model**

Figure 6.7 illustrates the training of the logistic regression model.

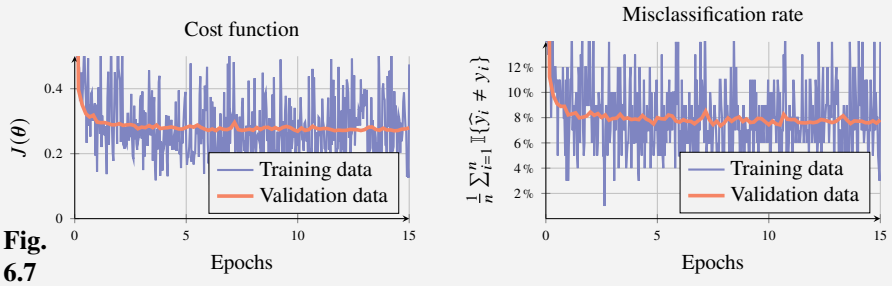



**Fig. 6.7**

We can see that the performance on validation data improves, and already after a few epochs we do not see any additional improvements, and we get a misclassification rate of approximately 8% on the validation data.

**Two-layer neural network**
In Figure 6.8, the training of the two-layer neural network with $U = 200$ hidden units is illustrated. We use the ReLU as activation function in this model.

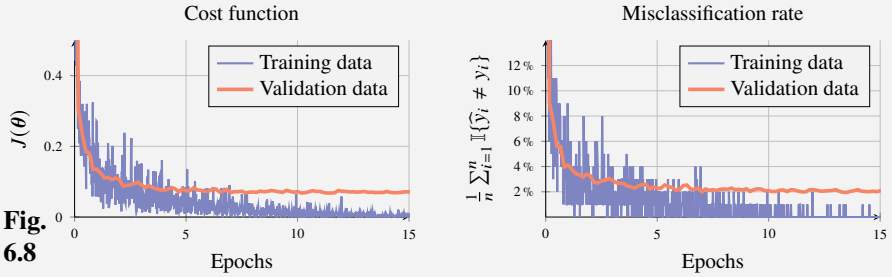



**Fig. 6.8**

Adding this layer of hidden units significantly reduces the misclassification rate down to 2% on the validation data. We can also see that during the later epochs, we often get all 100 data points in the current mini-batch correct. The difference between training error and validation error, the generalisation gap, is noticeable. Referring to the discussion in Section 4.3, we might be in the overparameterised regime where there is a chance that a well-designed even more flexible model can perform even better. Such a deep neural network is the convoluational neural network, in the next chapter.

## 6.3 Going Deeper: Normalisation and Residual Connections

When designing a deep neural network like Figure 6.4 there are many decisions to make, but a fundamental choice is the width of the layers $U$ and the numbers of layers $L$. While increasing either of them certainly makes the model more flexible, experience has proven that an increased depth $L$ makes it particularly capable for most applications. State-of-the-art neural network models typically use hundreds, if not thousands, of stacked layers, as we will see in Chapter 7.

However, those really deep neural network models would not be possible without a few certain techniques. It turns out that stacking more than a handful of layers in a deep neural network typically leads to numerical challenges in the training, unless some extra care is taken. During backpropagation, the gradients for the early layers often become either very small or very large, the so-called vanishing or exploding gradient problem. This is a mathematically expected consequence for derivatives of nested functions (like stacked layers in a neural network), but it has the negative effect that the training procedure easily becomes unstable. There are, however, two important techniques that can circumvent this issue: batch normalisation and residual

connections.

### Batch and Layer Normalisation

As discussed in Chapter 2, it is common to normalise the inputs to a machine learning model according to (2.2). Noting that the hidden units in a neural network acts as the inputs to the next layer, a simple yet powerful idea is to apply the input normalisation approach (2.2) also to the hidden units in a deep neural network. That is, a normalisation function is added between the neural network layers (or, alternatively, right before the activation function in a layer). This prevents the hidden units, and hence the inputs to each layer, from shrinking or growing uncontrollably as the information propagates through the network. This has proven to stabilise the optimisation process, ultimately making the training of very deep architectures faster and less sensitive to initial weight settings.

There are two main ways to apply the normalisation. Let us first look at *batch normalisation*.

If a deep neural network has dimension $U$ at a certain layer, and the input to the network $\mathbf{x}_i$ gives the value of the hidden units $\mathbf{q}_i = [q_i^1, q_i^2, \ldots, q_i^U]^\mathsf{T}$ (that is, $i$ is referring to the input data point index), normalisation can be applied as in (2.2) if we only know the mean $\bar{q}^j$ and variance $\sigma_j^2$ (for $j = 1, \ldots, U$) of the hidden unit. However, for knowing that, we would have to propagate not only the input data sample $\mathbf{x}_i$ but the entire training data $\{\mathbf{x}_i\}_{i=1}^n$ through the model, not only once but after each parameter update. That would defeat the purpose of the lightweight stochastic gradient approach, where only a mini-batch of the training data is used at a time. A mini-batch is, however, usually still big enough to give a reasonable estimate of $\bar{q}^j$ and $\sigma_j^2$, and the idea in batch normalisation is to normalise using the mean and variance computed from the current mini-batch only. Since $\sigma_j^2$ will appear in the denominator, it is common to safeguard against divisions by zero by forcing it to be at least a small $\epsilon$ in numerical estimations. In a way, an estimate of the mean and variance is used, much like stochastic gradient descent uses an estimate of the gradient.

To see why this stabilizes gradients, consider the activation function $\sigma(\cdot)$. The gradient during backpropagation involves its derivative $\sigma'$ and specifically the term $\sigma'(q_i^j)$. If the hidden units $q_i^j$ are allowed to grow uncontrollably, all of them may land in the "saturated regions" of the activation function where the derivative $\sigma'$ is zero or nearly zero, such as $|q_i^j| \gg 0$ for the logistic function or $q_i^j < 0$ for the ReLU function, leading to vanishing gradients. By keeping the distribution of $q$ centered and scaled, normalisation ensures that the activations remain in the most informative, non-saturated range of the activation function.

However, always forcing zero mean and unit variance of the hidden units may limit the expressive power of a layer too much. To address this, batch normalisation also contains two learnable parameters, a scale $\gamma_j$ and a shift $\beta_j$, which allow the network to adapt the distribution of the normalized hidden units to the needs of the subsequent layers. If the normalised hidden unit is denoted with $\widehat{q}_i^j$, the batch normalisation

concept can be summarized as

$$\widehat{q}_i^j = \frac{q_i^j - \bar{q}^j}{\sqrt{\sigma_j^2}}, \tag{6.29a}$$

$$z_i^j = \gamma_j \widehat{q}_i^j + \beta_j, \tag{6.29b}$$

where $\bar{q}^j$ and $\sigma_j^2$ are estimated from the current mini-batch, and $\gamma_j$ and $\beta_j$ are learned during training. This does not add much complexity to the training, and improves its numerical stability significantly.

For later doing prediction when there is only a single $\mathbf{x}_\star$ available, estimates of $\bar{q}^j$ and $\sigma_j^2$ are usually computed and saved using a running mean and variance during training.

Another way to apply the normalisation is *layer normalisation*. The difference between the two is in the direction of the averaging. In batch normalisation, for each fixed dimension $j$ of the hidden unit layer, the values $q_i^j$ are normalised by computing statistics across all data points $i = 1, \ldots, n_b$ in the mini-batch. In layer normalisation, the opposite is done: for each fixed data point $i$, the values $q_i^j$ are normalised by computing statistics across all dimensions $j = 1, \ldots, p$. In other words, batch normalisation amounts to normalising such that $\sum_{i=1}^{n_b} q_i^j = 0$ for each $j$, whereas layer normalisation amounts to normalising such that $\sum_{j=1}^{p} q_i^j = 0$ for each $i$ (and similarly for the variances but summing to 1). As with batch normalisation, a scale $\gamma_j$ and a shift $\beta_j$ are also used.

The choice between these techniques often depends on the batch size and the structural nature of the data. Neither approach is more or less correct, but batch normalisation is particularly effective when the mini-batch is large enough to provide stable statistical estimates, making it a natural fit for architectures where samples are processed in fixed, parallel blocks. In contrast, layer normalisation is more robust when batch sizes are small or when the data has a strong internal dependency, as its statistics are computed independently for each individual sample. We will see example of both situations in Chapter 7.

### Residual Connections

The idea of residual connections is simple but perhaps also surprising at first glance. An intuitive way of understanding it is to start with the following thought experiment: If you have a deep neural network and add one extra layer, you would probably expect the extended model to not perform worse than the original one. Your argument would be that, if nothing else, the extra layer could learn to just be the identity mapping, meaning that the new layer would "do nothing", and that the worst case implication of an additional layer would just be that you get the original model back.

Yet that is not what happens, at least not in some cases. Instead, adding one extra layer might actually derail the training and give you a much worse model in return. The reason, intuitively speaking, is that it is quite hard for a neural network model to "learn"

the identity mapping. Although conceptually simple for a human, the non-linear activation function can make it hard, or even impossible, for the optimisation to find even an approximation of the identity mapping.

On an intuitive level, the residual connections makes the identity mapping becoming the default mode for a layer. If we use $q^{(l)}$ and $q^{(l+1)}$ to denote the input and output of a certain layer in a neural network, respectively, and $H(\,\cdot\,)$ for the mapping that happens in that layer, we would write $q^{(l+1)} = H(q^{(l)})$ if there are no residual connections. With a residual connection, we would instead let

$$q^{(l+1)} = H(q^{(l)}) + q^{(l)}, \tag{6.30}$$

that is, we would let the nonlinear function H compute the "residual" and add this to the input $q^{l}$ to get the output $q^{l+1}$. This is also illustrated in Figure 6.9. By doing so, the trainable parts of the layer, $H(\,\cdot\,)$, will no longer describe the mapping from $q^{(l)}$ to $q^{(l+1)}$, but rather what changes are made when turning $q^{(l)}$ into $q^{(l+1)}$. The identity mapping, the "do nothing" alternative, is now $H(q^{(l)}) = 0$ instead of $H(q^{(l)}) = q^{(l)}$, which is easily attained by the optimisation algorithm by setting all trainable weights to zero.

On a slightly more mathematical level, we can see the advantage of residual connections during backpropagation as follows: Consider the gradient of the output with respect to the input in (6.30): $\frac{\partial q^{(l+1)}}{\partial q^{(l)}} = \frac{\partial H}{\partial q^{(l)}} + 1$. This +1 term ensures that the gradient can flow through the network even if the weights in the trainable block $H$ are very small, effectively solving the vanishing gradient problem. However, it is worth noting that while this "shortcut" prevents the gradient from disappearing, it does not inherently prevent it from growing. In fact, by repeatedly adding signals together, residual connections can potentially amplify the exploding gradient problem or lead to uncontrolled growth in activation variance. For this reason, residual connections are almost always used in conjunction with the normalisation techniques discussed in the previous section, which keep the signal magnitude in check.

The residual connection is probably the simplest and most used example of a more general family of techniques called *skip connections*. All skip connection methods build on the same idea of preserving $q^{(l)}$, but the exact operation in (6.30) differs.

While normalisation ensures that the signal in each layer remains numerically stable, residual connections provide a direct path for that signal to travel through the entire depth of the architecture. Together, they allow for the training of networks with hundreds of layers, a feat that would be impossible with either technique alone.

## 6.4 Dropout

A method that has proven to improve the generalisation properties of neural networks is the dropout method. It can be motivated from different perspectives, and we will here introduce it as a clever adoption of bagging (Chapter 8) to neural networks. In brief the idea is to train multiple independent copies of the model, *ensemble members*, and use them together as an *ensemble*.

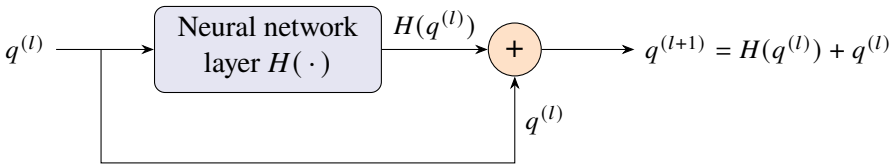
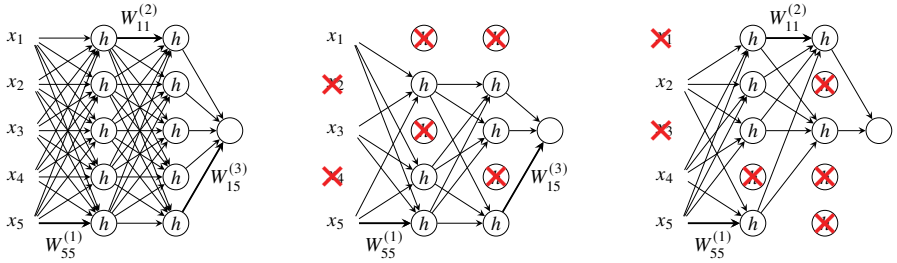



**Figure 6.9:** The residual connection: a sort of shortcut that allows the information to "skip" a layer and directly be added to that layer's output. This way, the trainable parts of the layer, $H(\cdot)$ in the figure, describe the transformation from $q^{(l)}$ to $q^{(l+1)}$ in terms of changes, rather than describing the full mapping from $q^{(l)}$ to $q^{(l+1)}$ itself. This has proven useful in practice because the identity mapping, or "do nothing" option, is now $H(\cdot) = 0$, which is a more stable solution during training.

It is possible to apply bagging, as described in Chapter 8, directly to neural networks. However, it comes with some practical problems, since neural networks are typically large and time-consuming to train. *Dropout*, however, can be understood as a bagging-like technique that allows us to combine many neural networks without the need to train them separately. The trick is to let the different models share parameters with each other, which reduces the computational cost and memory requirement.




**(a)** A standard neural network

**(b)** Two sub-networks

**Figure 6.10:** A neural network with two hidden layers (a) and two sub-networks with dropped units (b). The collection of units that have been dropped are independent between the two sub-networks.

## Ensemble of Sub-networks

Consider a dense neural network like the one in Figure 6.10a. In dropout, we construct the counterpart to an ensemble member by randomly removing some of the hidden units. We say that we drop the units, hence the name dropout. When a unit is removed, we also remove all of its incoming and outgoing connections. With this procedure, we obtain a sub-network that only contains a subset of the units and parameters present in the original network. Two such sub-networks are displayed in Figure 6.10b.

Mathematically, we can write this as sampling a *mask* $\mathbf{m}^{(l-1)} = [m_1^{(l-1)} \ \ldots \ m_{U_{l-1}}^{(l-1)}]$ for each layer, multiplying that mask element-wise with the hidden units $\mathbf{q}^{(l-1)}$, and


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

then feeding the masked hidden units $\tilde{\mathbf{q}}^{(l-1)}$ to the next layer:

$$m_j^{(l-1)} = \begin{cases} 1 & \text{with probability} \quad r \\ 0 & \text{with probability} \quad 1-r, \end{cases} \qquad \text{for all} \quad j = 1, \ldots, U_{l-1}, \tag{6.31a}$$

$$\tilde{\mathbf{q}}^{(l-1)} = \mathbf{m}^{(l-1)} \odot \mathbf{q}^{(l-1)}, \tag{6.31b}$$

$$\mathbf{q}^{(l)} = h(\mathbf{W}^{(l)}\tilde{\mathbf{q}}^{(l-1)} + \mathbf{b}^{(l)}). \tag{6.31c}$$

The probability $r$ of keeping a unit is a hyperparameter set by the user. We can choose to apply dropout to all layers or only some of them and can also use different probabilities $r$ for the different layers. We can also apply dropout to the input variables as we do in Figure 6.10b. However, we do not apply dropout on the output units. Since all sub-networks stem from the very same original network, the different sub-networks share some parameters with each other. For example, in Figure 6.10b, the parameter $W_{55}^{(1)}$ is present in both sub-networks. The fact that they share parameters with each other allows us to train the ensemble of sub-networks in an efficient manner.

### Training with Dropout

To train with dropout, we use stochastic gradient descent as described in Algorithm 5.3. In each gradient step, a mini-batch of data is used to compute an approximation of the gradient, as normally done in stochastic gradient descent. However, instead of computing the gradient for the full network, we generate a random sub-network by randomly dropping units as described above. We compute the gradient for that sub-network and then do a gradient step. This gradient step only updates the parameters present in the sub-network. The parameters that are associated with the dropped units do not affect the output of that sub-network and are hence left untouched. In the next gradient step, we use another mini-batch of data, remove another randomly selected collection of units, and update the parameters present in that sub-network. We proceed in this manner until some terminal condition is fulfilled.

### Prediction at Test Time

After we have trained the sub-networks, we want to make a prediction based on an unseen input data point $\mathbf{x}_\star$. If this was an ensemble method, we would evaluate all independent sub-networks and average their predictions. But with dropout we do not have independent sub-networks, but only one network that was trained as a different sub-network at each iteration. Hence, instead of evaluating all sub-networks, we simply evaluate the full network containing all the parameters, but to compensate for the fact that the model was trained with dropout, we multiply each estimated parameter

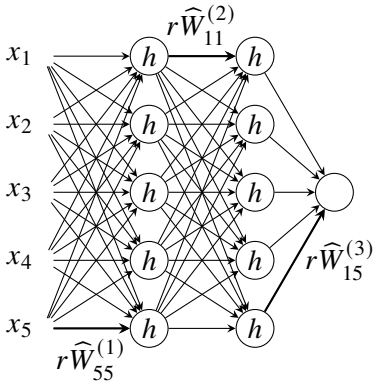



**Figure 6.11:** The network used for prediction after being trained with dropout. All units and links are present (no dropout), but the weights going out from a certain unit are multiplied by the probability of that unit being included during training. This is to compensate for the fact that some of them were dropped during training. Here all units have been kept with probability $r$ during training (and consequently dropped with probability $1 - r$).

going out from a unit by the probability of that unit being kept during training. This ensures that the expected value of the input to the next unit is the same during training and testing. If during training, we kept a unit in layer $l - 1$ with probability $r$, then during prediction we multiply the following weight matrix $\mathbf{W}^{(l)}$ with $r$, that is

$$\tilde{\mathbf{W}}^{(l)} = r\mathbf{W}^{(l)}, \tag{6.32a}$$

$$\mathbf{q}^{(l)} = h(\tilde{\mathbf{W}}^{(l)}\mathbf{q}^{(l-1)} + \mathbf{b}^{(l)}). \tag{6.32b}$$

This is also illustrated in Figure 6.11, where we have kept a unit with probability $r$ in all layers during training.

### Dropout vs. Bagging

As already pointed out, dropout has similarities with bagging. If you have already learned about bagging in Chapter 8, there are a few important differences to point out:

- In bagging, all models are independent in the sense that they have their own parameters. In dropout, the different models (the sub-networks) share parameters.
- In bagging, each ensemble member is trained until convergence. In dropout, the counterpart is the sub-networks, but each of them have most likely only been trained for one single gradient step. However, since they share parameters, all models will also be updated when the other sub-networks are trained.
- Similar to bagging, in dropout, we train each model on a dataset that has been randomly selected from our training data. However, in bagging, we usually do it

on a bootstrapped version of the whole dataset, whereas in dropout, each model is trained on a randomly selected mini-batch of data.

Even though dropout differs from bagging in some aspects, it has empirically been shown to enjoy similar properties as bagging in terms of avoiding overfitting and reducing the variance of the model.

### Regularisation in Neural Networks

Dropout can be seen as one regularisation method for neural networks, with the purpose of reducing the variance and avoiding overfitting. There are also plenty of other regularisation methods for neural networks, including explicit regularisation (like $L^1$ and $L^2$ regularisation; see Chapter 5), early stopping (the training is stopped before the parameters have converged; see Chapter 5), and various non-dense architectures (for example, CNNs in Chapter 7 can be seen as a regularisation method where most parameters are forced to be zero), just to mention a few. Interestingly, it has been shown that the stochastic gradient descent method itself also has a regularising effect due to its stochasticity.

## 6.5 Further Reading

Although the first conceptual ideas of neural networks date back to the 1940s (McCulloch and Pitts 1943), they had their first main success stories in the late 1980s and early 1990s with the use of the backpropagation algorithm from Rumelhart et al. (1986). At that stage, neural networks could, for example, be used to classify handwritten digits from low-resolution images (LeCun, Boser, et al. 1989). However, in the late 1990s, neural networks were largely forsaken in favor of general methods such as support vector machines. In areas such as computer vision and speech recognition, neural networks were not able to compete with hand-crafted solutions based on domain-specific prior knowledge.

This situation has changed dramatically since the late 2000s, with the advent of *deep* neural networks. Progress in software, hardware, and algorithm parallelisation made it possible to address more complicated problems, which were unthinkable only a couple of decades ago, which also has been acknowledged by awarding it the Nobel Prize in physics 2024. There are now several good textbooks available on the topic, including Prince (2023) and C. M. Bishop and H. Bishop (2024).

## 6.A Derivation of the Backpropagation Equations

To derive the backpropagation equations (6.27), consider the non-vectorised version of layer $l$:

$$\begin{cases} z_j^{(l)} = \sum_k W_{jk}^{(l)} q_k^{(l-1)} + b_j^{(l)} \\ q_j^{(l)} = h(z_j^{(l)}) \end{cases}, \qquad \forall j = 1, \dots U^{(l)}. \tag{6.33}$$

Assume that we want to compute the derivatives of the cost function with respect to the parameters $W_{jk}^{(l)}$ and $b_j^{(l)}$. Note that the cost function $J(\boldsymbol{\theta})$ depends on both $W_{jk}^{(l)}$ and $b_j^{(l)}$ only via the hidden unit $z_j^{(l)}$ (and none of the other hidden units in that layer). We can use the chain rule of calculus to write

$$\frac{\partial J}{\partial b_j^{(l)}} = \frac{\partial J}{\partial z_j^{(l)}} \underbrace{\frac{\partial z_j^{(l)}}{\partial b_j^{(l)}}}_{=1} = \frac{\partial J}{\partial z_j^{(l)}},$$

$$\frac{\partial J}{\partial W_{jk}^{(l)}} = \frac{\partial J}{\partial z_j^{(l)}} \underbrace{\frac{\partial z_j^{(l)}}{\partial W_{jk}^{(l)}}}_{=q_k^{(l-1)}} = \frac{\partial J}{\partial z_j^{(l)}} q_k^{(l-1)}, \qquad \forall j = 1, \dots U^{(l)}, \tag{6.34}$$

where the partial derivatives of $z_j^{(l)}$ with respect to $W_{jk}^{(l)}$ and $b_j^{(l)}$ can be directly derived from (6.33).

Similarly, we can also use the chain rule to compute the partial derivative of $J(\boldsymbol{\theta})$ with respect to the post-activation hidden unit $q_k^{(l-1)}$ for layer $l-1$. Note, $J(\boldsymbol{\theta})$ depends on $q_k^{(l-1)}$ via all of the pre-activation hidden units $\{z_j^{(l)}\}_{j=1}^{U^{(l)}}$ in layer $l$; hence we get

$$\frac{\partial J}{\partial q_k^{(l-1)}} = \sum_j \frac{\partial J}{\partial z_j^{(l)}} \underbrace{\frac{\partial z_j^{(l)}}{\partial q_k^{(l-1)}}}_{W_{jk}^{(l)}} = \sum_j \frac{\partial J}{\partial z_j^{(l)}} W_{jk}^{(l)}, \qquad \forall k = 1, \dots U^{(l-1)}. \tag{6.35}$$

Finally, we can also use the chain rule to express

$$\frac{\partial J}{\partial z_j^{(l)}} = \frac{\partial J}{\partial q_j^{(l)}} \frac{\partial q_j^{(l)}}{\partial z_j^{(l)}} = \frac{\partial J}{\partial q_j^{(l)}} h'(z_j^{(l)}), \qquad \forall j = 1, \dots U^{(l)}, \tag{6.36}$$

where $h'$ is the derivative of the activation function. With the vectorised notation for $d\mathbf{W}$, $d\mathbf{b}$, $d\mathbf{z}$, and $d\mathbf{q}$ introduced in (6.2) and (6.25), we get that Equation (6.36) gives (6.27a), Equation (6.35) gives (6.27b), and Section 6.A gives (6.27c)–(6.27d).

# 7 Deep Learning Architectures: CNN and Transformers

Chapter 6 introduced the concept of neural networks and how the layers can be stacked on top of each other to create *deep neural networks*, or *deep learning* for short. Deep learning has been the most successful machine learning paradigm for more than a decade now. One of the success factors has been its flexibility that allows it to be adapted to different types of data, by constructing specific neural network models that fit the inherent structure of the data. These different types of neural networks are often referred to as *architectures*.

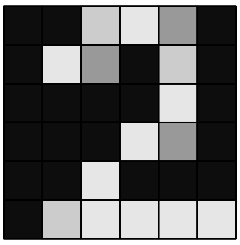
In this chapter, we will look at two of the most important building blocks in deep learning today: convolutional neural networks (CNN) and transformer models. CNNs opened the door to modern image processing, while transformers have changed the way we handle natural language. Although they are famous for these specific uses, both models are actually quite flexible. They can be used for many different tasks, as long as the input data has the right structure.

A key idea that runs through both architectures, and that distinguishes them from the fully connected networks of Chapter 6, is *parameter sharing*. In a fully connected layer, every connection between two units has its own weight. This gives the layer a lot of flexibility, but it also means that the number of parameters grows quickly with the size of the input, and that the model has to learn from scratch that, say, an edge in the top-left corner of an image looks just the same as an edge in the bottom-right corner. Both CNNs and transformers instead reuse a small set of parameters across many positions in the input: across pixel locations in a CNN, and across positions in a sequence in a transformer. Beyond the practical benefit of keeping the parameter count manageable as we build deeper models, this is a way of baking useful prior knowledge about the structure of the data directly into the architecture, and it turns out to be one of the main reasons these models generalise so well.

We will start by describing image data and the building blocks of an CNN in Section 7.1, and then assemble the full model in Section 7.2 and arrive at the very popular CNN architecture called ResNet.

We will then turn our attention to text data and describe how text can be used as input to a neural network in Section 7.3, and thereafter introduce the transformer architecture itself in Section 7.4. We will conclude in Section 7.5 by discussing how the transformer models can be used as language models, and how they are turned into large language models (LLM).

Image



Data representation

| | | | | | |
|---|---|---|---|---|---|
| 0.0 | 0.0 | 0.8 | 0.9 | 0.6 | 0.0 |
| 0.0 | 0.9 | 0.6 | 0.0 | 0.8 | 0.0 |
| 0.0 | 0.0 | 0.0 | 0.0 | 0.9 | 0.0 |
| 0.0 | 0.0 | 0.0 | 0.9 | 0.6 | 0.0 |
| 0.0 | 0.0 | 0.9 | 0.0 | 0.0 | 0.0 |
| 0.0 | 0.8 | 0.9 | 0.9 | 0.9 | 0.9 |

Input variables

| | | | | | |
|---|---|---|---|---|---|
| $x_{1,1}$ | $x_{1,2}$ | $x_{1,3}$ | $x_{1,4}$ | $x_{1,5}$ | $x_{1,6}$ |
| $x_{2,1}$ | $x_{2,2}$ | $x_{2,3}$ | $x_{2,4}$ | $x_{2,5}$ | $x_{2,6}$ |
| $x_{3,1}$ | $x_{3,2}$ | $x_{3,3}$ | $x_{3,4}$ | $x_{3,5}$ | $x_{3,6}$ |
| $x_{4,1}$ | $x_{4,2}$ | $x_{4,3}$ | $x_{4,4}$ | $x_{4,5}$ | $x_{4,6}$ |
| $x_{5,1}$ | $x_{5,2}$ | $x_{5,3}$ | $x_{5,4}$ | $x_{5,5}$ | $x_{5,6}$ |
| $x_{6,1}$ | $x_{6,2}$ | $x_{6,3}$ | $x_{6,4}$ | $x_{6,5}$ | $x_{6,6}$ |

**Figure 7.1:** Data representation of a greyscale image with $6 \times 6$ pixels. Each pixel is represented by a number encoding the intensity of that pixel. These pixel values are stored in a matrix with the elements $x_{j,k}$.

## 7.1 The Building Blocks for the Convolutional Neural Network

*Convolutional neural networks* (CNN) are a neural network architecture developed for problems where the input data has a grid-like structure, such as images which we will focus on here. Images are indeed the most common application for CNNs, but they can also be used for any input data with a grid-structure, also in one dimension (e.g. audio waveform data) or three dimensions (volumetric data, such as computer tomography (CT) scans or lidar data). We will focus on greyscale images here, but the approach can easily be extended to colour images as well by adding the colours as additional input channels.

### Data Representation of an Image

Digital greyscale images consist of pixels ordered in a matrix. Each pixel can be represented as a range from 0 (total absence, black) to 1 (total presence, white), and values between 0 and 1 represent different shades of grey. In Figure 7.1, this is illustrated for an image with $6 \times 6$ pixels. In image classification, the input $\mathbf{x}$ is an image consisting of the pixels $x_{1,1}$, $x_{1,2}$, $\ldots$, $x_{6,6}$. The two indices determine the position of the pixel in the image-grid, as illustrated in Figure 7.1.

If we put all input variables representing the image pixels in a long vector, as we did in Example 6.1 and Example 6.2, we could use the dense neural network as presented in Section 6.1. However, by doing that, a lot of the structure present in the image data would be lost. The knowledge about whether two pixels are close or distant to each other is important information, and it seems like a waste to remove that information by vectorising the image and let a fully connected network re-discover that information during its training. In contrast, CNNs preserve this information by representing both the input variables and the hidden layers as grid-like data structures.


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026, http://smlbook.org*

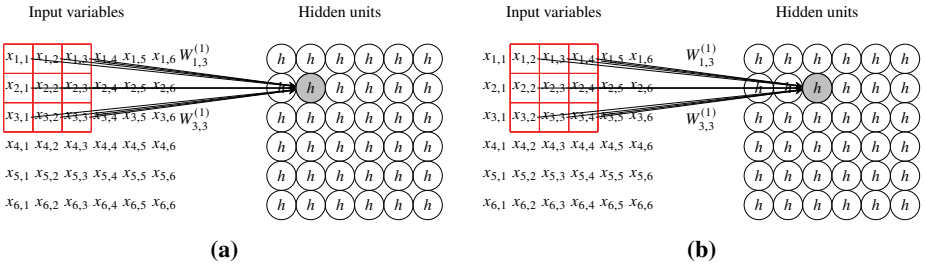



**Figure 7.2:** An illustration of the interactions in a convolutional layer: Each hidden unit (circle) is only dependent on the pixels in a small region of the image (red boxes), here of size $3 \times 3$ pixels. The location of the hidden unit corresponds to the location of the region in the image: if we move to a hidden unit one step to the right, the corresponding region in the image also moves one step to the right – compare Figure 7.2a and b. Furthermore, the nine parameters $W^{(1)}_{1,1}, W^{(1)}_{1,2}, \ldots, W^{(1)}_{3,3}$ are the *same* for all hidden units in the layer.

## The Convolutional Layer

In a convolutional layer, which is the core component of a CNN, both the input and the output of the layer is a matrix. Let us say that the input is a greyscale image with $6 \times 6$ pixels. A straightforward choice is then to also let the first layer of hidden units be $6 \times 6 = 36$ hidden units. We order the hidden units in a $6 \times 6$ matrix, in the same manner as we did for the input variables; see Figure 7.2a.

In contrast to the dense layers in the previous chapter (see for example Figure 6.3), the unique feature in a convolutional layer is to restrict the connections between the input and output to be only a fraction of the full flexibility of a dense layer. There are two important concepts in play here: *sparse interactions* and *parameter sharing*, explained in detail below. These together will give our convolutional layer here only 10 instead of 1332 (as would have been the case with a fully connected layer) trainable parameters. It turns out this has a huge effect on improving how the model generalises to unseen data. In addition, having so (relatively) few parameters per layer will also make it practically feasible to build even deeper models, which turns out to make the model even more powerful without losing the ability to generalise well.

### Sparse Interactions

In the first convolutional layer, each unit only depends on a small part of the input image, called its *receptive field*. For example, in Figure 7.2, this area is $3 \times 3$ pixels.

As we move from one hidden unit to the next, this window slides across the image. If we move one step to the right in the output, the window also moves one step to the right on the input image, see Figure 7.2. For units near the edges, we use zero-padding to fill in the gaps, as shown in Figure 7.3.

This is very different from a dense layer, where every unit connects to every single pixel. You can think of a convolutional layer as a "lite" version of a dense layer where most connections are set to zero. This concept is known as *sparse interactions*.

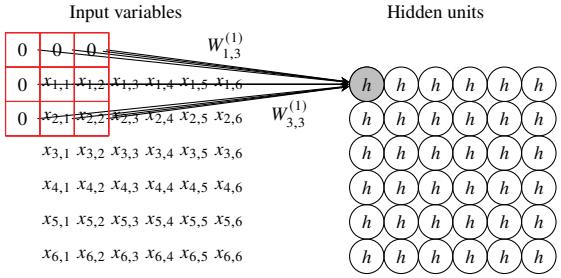



**Figure 7.3:** An illustration of zero-padding used when the region is partly outside the image. With zero-padding, the size of the image can be preserved in the following layer.

### Parameter Sharing

In a dense layer, each link between an input variable and a hidden unit has its own unique parameter. With parameter sharing, we instead let the same parameter be present in multiple places in the network. In a convolutional layer, the set of parameters for the different hidden units are all *the same*. For example, in Figure 7.2a, we use the same set of parameters to map the $3 \times 3$ region of pixels to the hidden unit as we do in Figure 7.2b. Instead of learning separate sets of parameters for every position, we only learn one set of a few parameters, and use it for all links between the input layer and the hidden units. We call this set of parameters a *filter*. The mapping between the input variables and the hidden units can be interpreted as a convolution between the input variables and the filter, hence the name convolutional neural network. Mathematically, this convolution can be written as

$$q_{ij} = h\left(\sum_{k=1}^{F}\sum_{l=1}^{F} x_{i+k-1,j+l-1} W_{k,l} + b_{ij}\right), \tag{7.1}$$

where $x_{i,j}$ denotes the zero-padded input to the layer, $q_{ij}$ the output of the layer, and $W_{k,l}$ the filter with $F$ rows and $F$ columns.

The sparse interactions and parameter sharing in a convolutional layer makes the CNN invariant (save for boundary effects) to translations of objects in the image. After training, the parameters in a filter typically become sensitive to a certain detail (such as a corner, an edge, etc.), and the filter will react to the presence or absence of this detail regardless of the absolute position in the image: because of the parameter sharing the same filter is applied to all locations in the image.

Thanks to the sparse interactions and the parameter sharing, a convolutional layer uses significantly fewer parameters compared to a corresponding dense layer. In Figure 7.2, only $3 \cdot 3 + 1 = 10$ parameters are required (including the bias parameter). If we instead had used a dense layer, $(36 + 1) \cdot 36 = 1\,332$ parameters would have been needed.


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September* 1, 2026, `http://smlbook.org`

### Stacking Convolutional Layers

In each convolutional layer, the output from the filter is typically passed through an activation function and becomes the hidden units. That output from a convolutional layer, after the activation function, is often called a *feature map*. Both the ReLU and the SiLU activation function are common in modern CNNs.

It is common practice to build deep CNNs by stacking convolutional layers after each other, where the output from one layer becomes the input to the next, etc. By doing so, the concept of the filters become even more powerful as they will typically learn to build up more advanced features.

This hierarchical feature construction is closely tied to the concept of the *receptive field*, which refers to the specific region of the original input image that a particular hidden unit "sees" or is influenced by. In the first layer, the receptive field is equal to the filter size. However, as layers are stacked, the receptive field grows for the hidden units in the subsequent layers. A unit in the second layer takes inputs from several units in the first layer, each of which "saw" a different part of the original image, and so on. Consequently, units in deeper layers have a larger global view, allowing them to detect complex, large-scale structures, while the initial layers focus on fine, local details.

### Convolutional Layer with Strides

In the convolutional layer presented above, we have as many hidden units as we have pixels in the image. However, when we add more layers, we might want to reduce the number of hidden units and thereby only store the most important information computed in the previous layers. One way of doing this is by not applying the filter to every pixel but to, say, every two pixels. If we apply the filter to every two pixels, both row-wise and column-wise, the hidden units will only have half as many rows and half as many columns. For a $6 \times 6$ image we get $3 \times 3$ hidden units. This concept is illustrated in Figure 7.4.

The *stride* controls how many pixels the filter shifts over the image at each step. In Figure 7.2, the stride is $s = 1$ since the filter moves by one pixel both row- and column-wise. In Figure 7.4, the stride is $s = 2$ since it moves by two pixels row- and column-wise. Note that the convolutional layer in Figure 7.4 still requires 10 parameters, just as the convolutional layer in Figure 7.2 does.

### Pooling Layer

Another way of summarising the information in previous layers is achieved by using *pooling*. A pooling layer acts as an additional layer after the convolutional layer. Similar to the convolutional layer, it only depends on a region of pixels. However, in contrast to convolutional layers, the pooling layer does not come with any extra trainable parameters. In pooling layers, we can also use strides to condense the information, meaning that region is shifted by $s \geq 1$ pixels. Two common versions of

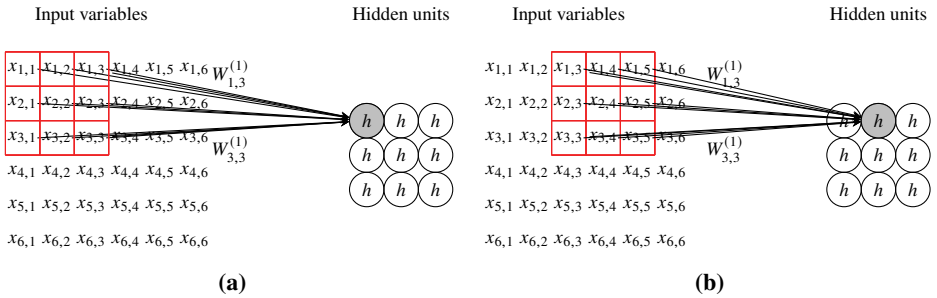



**Figure 7.4:** A convolutional layer with stride 2 and filter size $3 \times 3$.

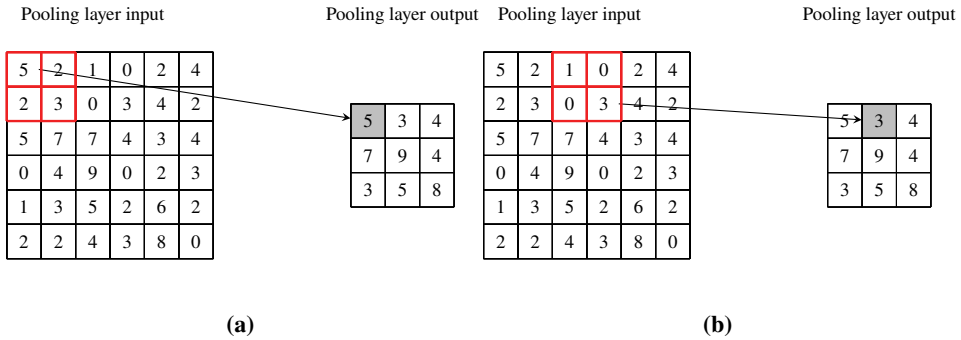



**Figure 7.5:** A max pooling layer with stride 2 and pooling filter size $2 \times 2$.

pooling are average pooling and max pooling. In average pooling, the average of the units in the corresponding region is computed. Mathematically, this means

$$\tilde{q}_{ij} = \frac{1}{F^2} \sum_{k=1}^{F} \sum_{l=1}^{F} q_{s(i-1)+k, s(j-1)+l}, \tag{7.2}$$

where $q_{i,j}$ is the input to the pooling layer, $\tilde{q}_{ij}$ is the output, $F$ is the pooling size, and $s$ is the stride used in the pooling layer. In max pooling, we instead take the maximum of the pixels. Max pooling is illustrated in Figure 7.5.

Like convolution with strides, pooling layers can make the model more invariant to small translations of the input, meaning that if we translate the input by a small amount, many of the pooling outputs do not change. For example, in Figure 7.5, if we shift the input units one step to the right (and replace the first column with 0's), the output of the pooling layer would be the same except for the 7 and 3 in the first column, which would become a 5 and a 2.

> ***Time to reflect 7.1*** *In what sense is average pooling a special case of a convolution? For a $3 \times 3$ average pooling filter, what weights would that correspond to in a $3 \times 3$ convolutional filter? Is also max pooling possible to understand as a convolutional filter, or is it a different type of operation?*


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

### Multiple Channels

The networks presented in Figures 7.2 and 7.4 only have 10 parameters each. Even though this parameterisation comes with several important advantages, one filter in each layer is for most applications not sufficient to catch all interesting properties of the images. To extend the network we add multiple filters, each with its own set of parameters. Each filter produces its own set of hidden units, referred to as *channel*, using the same convolution operation as explained in Section 7.1. Hence, each layer of hidden units in a CNN is organised into a so-called tensor with dimension[1] (rows × columns × channels). In Figure 7.6, the first layer of hidden units has four channels, and that hidden layer consequently has dimension $6 \times 6 \times 4$.

When we continue to stack convolutional layers, each filter depends not only on one channel but on all the channels in the previous layer. This is displayed in the second convolutional layer in Figure 7.6. As a consequence, each filter is a tensor of dimension (filter rows × filter columns × input channels). For example, each filter in the second convolutional layer in Figure 7.6 is of size $3 \times 3 \times 4$. If we collect all filter parameters in one weight tensor $\mathbf{W}$, that tensor will be of dimension (filter rows × filter columns × input channels × output channels). In the second convolutional layer in Figure 7.6, the corresponding weight matrix $\mathbf{W}^{(2)}$ is a tensor of dimension $3 \times 3 \times 4 \times 6$. With multiple filters in each convolutional layer, each of them might during training get sensitive to different features in the image, such as certain edges, lines, or circles. This way of adding flexibility and more parameters to the model has proved very helpful.

The convolutional layer $\ell$ with multiple input channels and output channels can be described mathematically as

$$q_{i,j,n}^{(\ell)} = h\left(\sum_{k=1}^{F_l}\sum_{l=1}^{F_l}\sum_{m=1}^{U_{l-1}} q_{s_l(i-1)+k-1,s_l(j-1)+l,m}^{(\ell-1)} W_{k,l,m,n}^{(\ell)} + b_{i,j,n}^{\ell}\right), \tag{7.3}$$

where $q_{i,j,m}^{(\ell-1)}$ is the input to layer $\ell$, $q_{i,j,n}^{(\ell)}$ is the output from layer $l$, $U_{\ell-1}$ is the number of input channels, $F_l$ is the filter rows/columns, $s_l$ is the stride, and $W_{k,l,m,n}^{(\ell)}$ is the weight tensor.

## 7.2 The Full Convolutional Neural Network

Having introduced the convolutional layer and its variants, we now turn to how complete convolutional networks are assembled from these building blocks. We first describe the general anatomy of a CNN classifier and illustrate it on a handwritten-digit task. We then turn to a more specific and very successful CNN architecture, ResNet,

[1]The index ordering is not standardized, and some software packages use a different ordering.

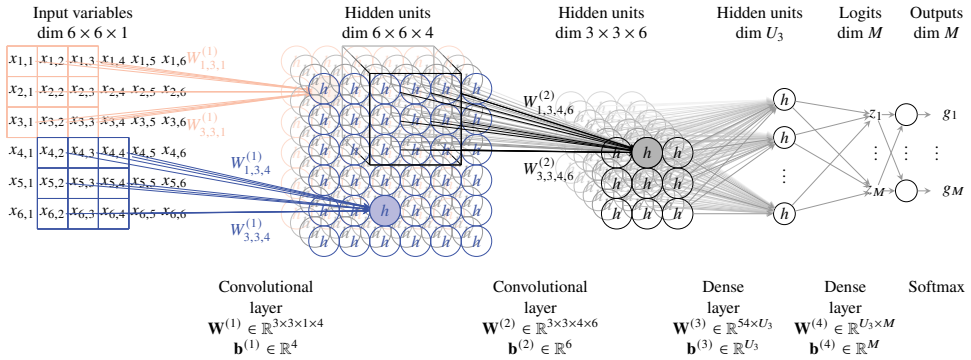



**Figure 7.6:** A full CNN architecture for classification of greyscale $6 \times 6$ images. In the first convolutional layer, four filters, each of size $3 \times 3$, produce a hidden layer with four channels. The first channel (in the back) is visualised in red, and the fourth channel (in the front) is visualised in blue. We use stride 1, which maintains the number of rows and columns. In the second convolutional layer, six filters of size $3 \times 3 \times 4$ and stride 2 are used. They produce a hidden layer with 3 rows, 3 columns, and 6 channels. After the two convolutional layers follows a dense layer where all $3 \cdot 3 \cdot 6 = 54$ hidden units in the second hidden layer are densely connected to all $U_3$ hidden units in the third layer, where all links have their unique parameters. We add an additional dense layer mapping down to the $M$ logits. The network ends with a softmax function to provide predicted class probabilities as output. Note that the arrows corresponding to the bias parameters are not included here in order to make the figure less cluttered.

that combines convolutional layers with batch normalisation and residual connections to scale CNNs to dozens or hundreds of layers.

## Full CNN Architecture

A full CNN architecture consists of multiple convolutional layers. How to design the full network is, in the end, more of an art than a science. A good practice is to take inspiration from the architecture of already published models that are known to perform well on certain tasks.

With that said, a common practice is to decrease the number of rows and columns in the hidden layers as we proceed through the network, while simultaneously increasing the number of channels to enable the network to encode more high level features. After some convolutional layers we usually end the network with one or more dense layers. If we have an image classification task, we place a softmax layer at the very end to get outputs in the range $[0, 1]$. The loss function when training a CNN will be the same as for the dense networks in Chapter 6. Figure 7.6 shows a small example of a full CNN architecture.


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September* 1, 2026, `http://smlbook.org`

**Example 7.1 Classification of hand-written digits – convolutional neural network**

In the previous models explained in Examples 6.1 and 6.2, we placed all the $28 \times 28$ pixels of each image into a long vector with 784 elements. With this action, we did not exploit the information that two neighbouring pixels are more likely to be correlated than two pixels further apart. Instead, we now keep the matrix structure of the data and use a CNN with three convolutional layers and two dense layers. The settings for the layers are given in the table below.

| | Convolutional layers | | | Dense layers | |
|---|---|---|---|---|---|
| | **Layer 1** | **Layer 2** | **Layer 3** | **Layer 4** | **Layer 5** |
| Number of filters/output channels | 4 | 8 | 12 | – | – |
| Filter rows and columns | $(5 \times 5)$ | $(5 \times 5)$ | $(4 \times 4)$ | – | – |
| Stride | 1 | 2 | 2 | – | – |
| Number of hidden units | 3 136 | 1 568 | 588 | 200 | 10 |
| Number of parameters (including bias vector) | 104 | 808 | 1 548 | 117 800 | 2 010 |

In a high-dimensional parameter space, saddle points in the cost function are frequent. Since the gradients are zero also at these saddle points, stochastic gradient descent might get stuck there. Therefore, we train this model using an extension of stochastic gradient descent called Adam (short for adaptive moment estimation). Adam uses running averages on the gradients and their second order moments and can pass these saddle points more easily. For the Adam optimiser, we use a learning rate of $\gamma = 0.002$. In Figure 7.7, the cost function and the misclassification rate on the current training mini-batch and the validation data are displayed. The shaded red line is the performance on the validation data for the two-layer network that was presented in Example 6.2.

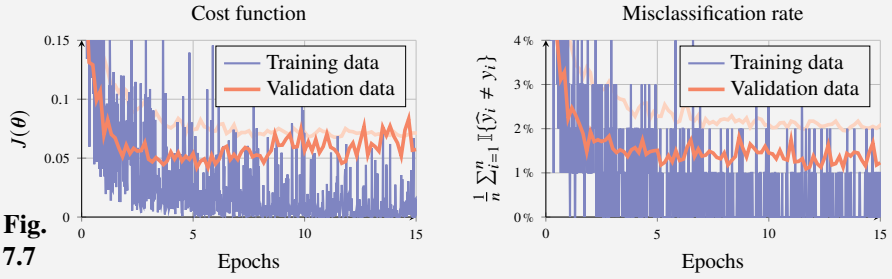



**Fig. 7.7**

In comparison to the result on the dense two-layer network, we can see an improvement going from 2% down to just over 1% misclassification on the validation data. From Figure 7.7 we can also see that the performance on validation data does not quite settle but is fluctuating in the span 1–1.5%. As explained in Section 5.5, this is due to the randomness introduced by stochastic gradient

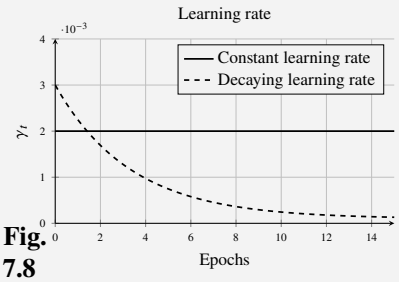



**Fig. 7.8**

descent itself. To circumvent this effect, we use decaying learning rate. We use the scheme suggested in (5.38), with $\gamma_{\max} = 0.003$, $\gamma_{\min} = 0.0001$, and $\tau = 2\,000$ (Figure 7.8).

After employing the adaptive learning rate in Figure 7.9, the misclassification rate on validation data settles around 1% rather than only sometimes bouncing down to 1% as it did before we applied a decaying learning rate. However, we can do more. In the last epochs, we get almost all data points correct in the current mini-batch. In addition, looking at the plot for the cost function evaluated for the validation data, it starts increasing after five epochs. Hence, we see signs of overfitting. To circumvent this overfitting, we can add regularisation, such as dropout from Section 6.4.

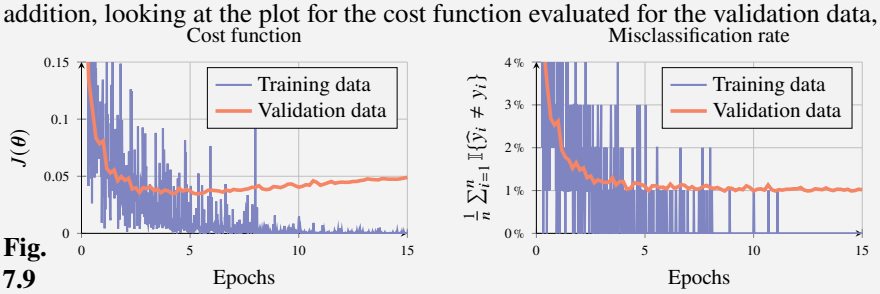



**Fig. 7.9**

Let us use dropout in the final hidden layer and keep only $r = 75\%$ of the 200 hidden units in that layer at each iteration. The result for this regularised model is presented in Figure 7.10. In shaded red, we also present the performance on validation data for the non-regularised version from Figure 7.9.

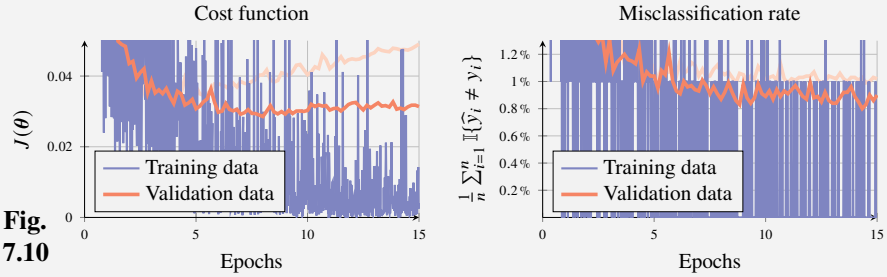



**Fig. 7.10**

In contrast to the non-regularised version, the cost function evaluated for the validation data is (almost) no longer increasing, and we also reduce the misclassification rate by an additional 0.1% to 0.2% thanks to dropout.

***Time to reflect 7.2*** *In the table in Example 7.1, the number of parameters for all five layers, as well as the number of hidden units for the three convolutional layers, can all be computed from the remaining numbers in that table and previously stated information. Can you do this computation?*


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September* 1, 2026, `http://smlbook.org`

### The deep CNN: ResNet

The real strength of CNNs can be achieved when combined with techniques such as the batch normalisation and residual connections from Section 6.3. Their combination is referred to as Residual Network, or the ResNet family of CNN architectures. Without those additional techniques, simply adding more layers to a CNN does eventually hurt the performance. Not because of overfitting, but because the optimisation itself becomes unstable, due to the vanishing and exploding gradient problems. However, the combination with residual connections and batch normalisation allows the barrier to be broken: when it was originally presented in 2015, the 152-layer ResNet outperformed all previous known CNN models and started the era of large-scale deep neural networks.

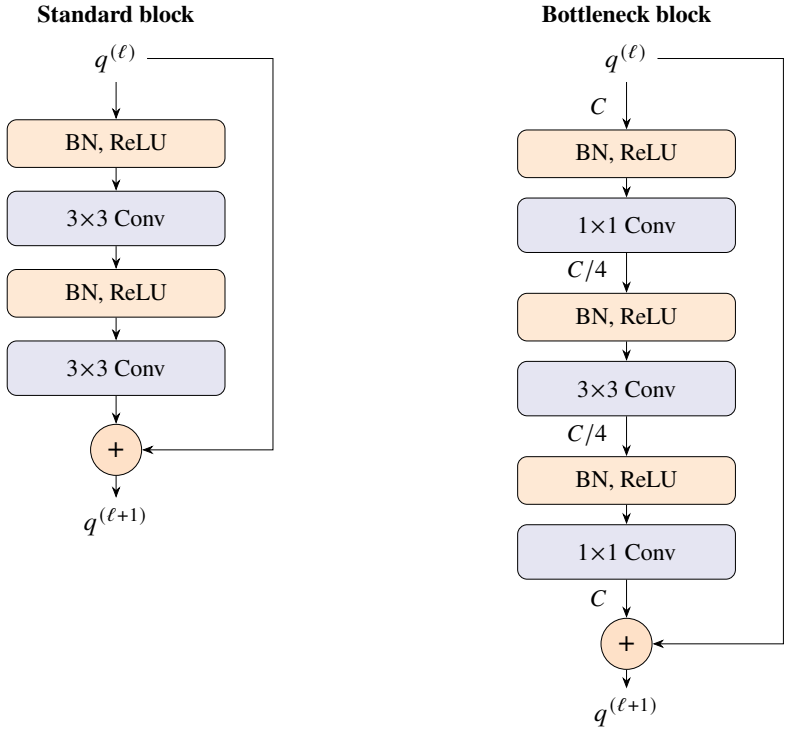



**Figure 7.11:** The two building blocks of the ResNet family, both implementing the residual connection $q^{(\ell+1)} = H(q^{(\ell)}) + q^{(\ell)}$ of Equation (6.30). **Left:** The standard residual block used in ResNet-18 and ResNet-34, in which $H$ consists of two $3 \times 3$ convolutional layers, each preceded by batch normalisation (BN) and a ReLU activation. **Right:** The bottleneck block used in ResNet-50 and deeper variants, in which $H$ surrounds a single $3 \times 3$ convolution with two $1 \times 1$ convolutions. The first $1 \times 1$ convolution compresses the number of channels from $C$ down to $C/4$, so the computationally expensive $3 \times 3$ convolution operates on a much smaller tensor; the second $1 \times 1$ convolution then restores the channel count to $C$. The labels on the arrows show the number of channels at each stage of the block. In both designs the residual connection adds the block's input directly to its output, with no further nonlinearity applied to the sum. This is sometimes called a pre-activation residual block.

The core building block of ResNet is the residual block, illustrated in Figure 7.11.

Each block implements the residual connection of Equation (6.30), where the function $H$ is built from two (or three, in deeper variants) convolutional layers with $3 \times 3$ filters, each preceded by batch normalisation and a ReLU activation. The block's input is then added directly to its output, with no further activation applied to the sum, so that the skip path is a true identity mapping. This arrangement, sometimes called the *pre-activation* residual block, was introduced shortly after the original ResNet and was shown to make very deep networks easier to optimise. The original ResNet architecture as presented in 2015 instead placed the final ReLU *after* the addition.

The choice of $3 \times 3$ filters is deliberate: two stacked $3 \times 3$ convolutional layers have the same receptive field as a single $5 \times 5$ layer, but require fewer parameters and introduce an extra nonlinearity.

Several standard variants of ResNet exist, differing primarily in depth: ResNet-18, ResNet-34, ResNet-50, ResNet-101, and ResNet-152, where the number indicates the total count of layers with trainable parameters. The shallower variants (ResNet-18 and ResNet-34) use simple two-layer residual blocks as described above, whereas the deeper variants (ResNet-50 and above) use a somewhat more intricate structure with bottleneck blocks (see Figure 7.11) to keep the parameter count and computational demand at a reasonable level even with as many as 152 layers. The full network ends with global average pooling, rather than one or more dense layers, which averages each feature map down to a single number, before a final linear classification layer. This design choice substantially reduces the number of parameters in the latter part and has become standard in modern CNN architectures.

One of the most practically important consequences of ResNet's success is its role as a foundation for downstream applications. In practice the models are often used by starting with a ResNet model already trained on a standard labelled dataset, and often only the final classification layer is re-trained for the problem at hand, so-called *transfer learning*. This works well because the convolutional layers tend to act as general feature-extractors for images that generalises remarkably well to new tasks.

## 7.3 Working with Text

We will now introduce another important deep learning architecture, namely the *transformer*. The transformer is a general architecture that can be applied to different types of data, but it was originally developed for machine translation, and problems such as text generation and translation remain among the most important application areas for these models.

While CNNs exploit the grid-like structure of images to extract local features hierarchically, transformers treat the input as a set of elements and use their attention mechanisms to capture long-range dependencies in the input. This is a key feature, since the meaning of a word in a text can depend on another word that appears many sentences earlier or later. The key component for this is the so-called *attention mechanism*, which is another example of a parameter sharing idea that matches the text data structure well, and in addition also will enable the transformer model to be


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September* 1, 2026, `http://smlbook.org`

trained in parallel on massive amounts of input data.

We will start by laying the foundation of how to process natural text in a neural network, through the use of tokens and embeddings. Thereafter, in Section 7.4, we will introduce the design of the attention mechanism and the full transformer model, and in Section 7.5 we will look at how it can be used as a language model. Our goal is to end up with a language model which is able to predict the next word in a sentence. For a partial sentence like

*"the overfitted model learns the"*

we would expect a good language model to assign high probability to words such as *pattern* and *noise* (or perhaps *typos*?) as the continuation of the sentence, while assigning a very small probability to most other words.

## Tokens

Before any neural network can process text, the text must be converted into numbers. Unlike image pixels, which are straightforward to convert to numbers, words and characters have no obvious numerical representation. In fact the choice of representation turns out to play a critical role.

The first step is to split the raw text into discrete units called *tokens*. The set of all tokens is called a vocabulary. If each character were used as a token, the vocabulary would simply be the alphabet plus a few extra symbols for special characters and spaces. If instead each full word were used as a token, the vocabulary would be hundreds of thousands of distinct word forms, including all possible inflections of every word that could ever be used.

Neither of these extremes has proven to be the most useful in practice. Instead, the most common tokenisation strategies in modern language models use tokens that are a compromise between single letters and full words. A common strategy is *byte-pair encoding*, which builds the vocabulary by starting at the character level and iteratively merging the most frequently adjacent pair of tokens into a new token, repeating until a target vocabulary of size $V$ is reached. Typical vocabulary sizes in modern models range from tens of thousands to hundreds of thousands of tokens. Smaller vocabularies are typically sufficient if all text is forced to lowercase and is written in a single language (such as English), whereas larger vocabularies are needed for handling texts with both lower- and uppercase letters, and in multiple languages. There are also some design choices to be made on how to handle punctuation, spaces, end-of-sequence tokens, and so on.

In practice, the effect of a tokeniser is typically that common words like `the` or `model` become single tokens, while rare words are split into smaller pieces; for example *overfitted* might be split into `over` and `fitted`. Each token is assigned a unique dictionary integer index from 0 to $V - 1$, so after tokenisation every piece of text becomes a sequence of integers.

For our partial example, a tokeniser might produce the token sequence

`[the,over,fitted,model,learns,the]` $\rightarrow$ `[891,600,1435,1520,1067,891]`,

where the integers are the dictionary indices. The word *the* appears twice and is mapped to the same index `891` both times, as expected.

These indices are not yet suitable as input to a neural network, because they carry no notion of similarity: the fact that index 891 and index 892 are numerically adjacent says nothing about the relationship between the corresponding tokens. Transforming the indices to one-hot encoded vectors $\mathbf{e}_t$ is not a full solution either; it only moves the problem to a $V$-dimensional space where all tokens are equally distant from each other. We will, however, use the one-hot encoded vectors as a convenient notation for arriving at the actual solution: *token embeddings*.

## Embeddings

Token embeddings are a learned linear transformation that maps each one-hot vector to a dense real-valued vector of much smaller dimension $d_{\text{emb}} \ll V$. Concretely, we introduce an *embedding matrix* $\mathbf{W}^{\text{emb}} \in \mathbb{R}^{d_{\text{emb}} \times V}$, and the embedding of a token with vocabulary index $t$ is simply the $t$-th column of this matrix, which we can write as $\mathbf{W}^{\text{emb}}\mathbf{e}_t \in \mathbb{R}^{d_{\text{emb}}}$. Because $\mathbf{e}_t$ is a one-hot vector, this multiplication is just a column lookup: $\mathbf{W}^{\text{emb}}$ acts as a lookup table, and the embedding $\mathbf{x}$ of each token is one column of that table. The key point is that $\mathbf{W}^{\text{emb}}$ is not fixed in advance, instead it is *learned jointly with the rest of the model* during training. During the model training it simultaneously learns to place tokens with similar meanings close together in the embedding space, since words that are interchangeable tend to occur in similar contexts, so the model can lower its loss by treating them simlilarly and mapping them to similar embeddings. Vectors for tokens like `book` and `text` will therefore be much closer to each other than to, say, `dumplings`. Typical values for $d_{\text{emb}}$ in modern language models range from 512 to 12 288.

> ***Time to reflect 7.3*** *The embedding matrix $\mathbf{W}^{\text{emb}} \in \mathbb{R}^{d_{\text{emb}} \times V}$ can have a very large number of parameters. For $V = 50\,000$ and a small $d_{\text{emb}} = 24$, how many parameters does $\mathbf{W}^{\text{emb}}$ contain? How does this compare to a single dense layer from a $d_{\text{emb}}$-dimensional input to a $d_{\text{emb}}$-dimensional output?*

For our partial sentence the sequence of six tokens, with vocabulary indices $t_1, \ldots, t_6$, corresponds to the sequence of six embedding vectors

$$\mathbf{x}_i = \mathbf{W}^{\text{emb}}\mathbf{e}_{t_i} \in \mathbb{R}^{d_{\text{emb}}}, \quad i = 1, \ldots, 6, \tag{7.4}$$

one embedding vector $\mathbf{x}_i$ for each token `[the, over, fitted, model, learns, the]`. Since both token one and token six are the word `the` ($t_1 = t_6$), the embedding vectors $\mathbf{x}_1$ and $\mathbf{x}_6$ are identical. The vectors in (7.4) will be the input to the transformer model. Note that it is possible to also define a mapping in the opposite direction, that maps any given vector in embedding space to a token and thereafter a text. Such a mapping is, for instance, learned in language models so they can suggest the continuation of a sentence, a process we will describe in Section 7.5.

## 7.4 The Transformer Architecture

With a sequence of numerical vectors – the token embeddings – in place, we are now ready to introduce the neural network architecture itself. Let us start by introducing the attention mechanism, which is arguably the most distinctive feature of a transformer compared to other types of neural networks. We will thereafter add the positional encoding and compile the transformer model itself.

### The Attention Mechanism

The input to an attention head is a sequence of token embeddings $\mathbf{x}_1, \ldots, \mathbf{x}_d$, and the output is another sequence, of equal length, of so-called *contextualised* token embeddings $\mathbf{z}_1, \ldots, \mathbf{z}_d$. In our sentence example $d = 6$, but the attention mechanism can handle sequences that are much longer than that. In concrete terms, each output embedding $\mathbf{z}_i$ is a combination of all the different input embeddings that the attention mechanism has considered relevant for token $i$.

The attention mechanism is, in a sense, quite similar to the convolutional operation in a CNN. Whereas a convolution processes grid-like data (the input image) into another grid-like data (the feature map), the attention mechanism processes a sequence of vectors (token embeddings) into another sequence of vectors (contextualised token embeddings). And just like the convolutional operations in a CNN can take place in parallel (in different channels) as well as in series (in subsequent layers), also the attention mechanism is often used in multiple instances in a transformer model. The established terminology is that each such instance of the attention mechanism, with its own learned parameters, is referred to as an *attention head*.

Consider for instance the token `model`, which can refer to a machine learning model, but also to a fashion model, an organisational model, a role model or a miniature model. The static embedding $\mathbf{x}$ produced by the lookup in $\mathbf{W}^{\text{emb}}$ is the same vector in all of these cases, since it depends only on the token itself. A good *contextualised* embedding, by contrast, should differ between sentences such as *"the overfitted model learns the noise"* and *"the architect built a wooden model of the cathedral"*, placing the first close to vectors like `network` or `classifier`, and the second close to vectors like `miniature` or `replica`. The fact that the sentences differ in their other content, besides the word "model", is what makes the contextualised embeddings differ. Distinguishing two sentences that contain the same words in a different order is a separate problem, which we will return to when we introduce positional encodings.

To sum up, the meaning of any token is shaped by its context, and the attention head will learn what the relevant context is for any combination of embedding vectors. This is in contrast to a CNN, where the counterpart to context, the receptive field, is hard-coded to be the local neighbourhood defined by the filter.

### Queries, Keys, and Values

An attention head begins by projecting each token embedding into three distinct vectors using three learned linear transformations:

$$\mathbf{q}_i = \mathbf{W}^Q \mathbf{x}_i, \ \textit{query} \tag{7.5a}$$

$$\mathbf{k}_i = \mathbf{W}^K \mathbf{x}_i, \ \textit{key} \tag{7.5b}$$

$$\mathbf{v}_i = \mathbf{W}^V \mathbf{x}_i, \ \textit{value} \tag{7.5c}$$

where $\mathbf{W}^Q, \mathbf{W}^K \in \mathbb{R}^{d_k \times d_{\text{emb}}}$ and $\mathbf{W}^V \in \mathbb{R}^{d_v \times d_{\text{emb}}}$ are learned weight matrices independent of the sequence index $i$, that is, the same three matrices are applied at every position. This is itself a form of parameter sharing: just as a convolutional filter is reused at every pixel, the same attention head is reused at every position in the sequence, and the parameter count does not grow with the sequence length $d$. The $d_k$ and $d_v$ are the dimensions of the query, key, and value vectors, respectively, and are design choices in the architecture. In a straightforward model with consecutive layers of single attention heads, $d_k = d_v = d_{\text{emb}}$ is a natural choice.

The role of the query $\mathbf{q}_i$ and key $\mathbf{k}_j$ vectors is to identify tokens that are relevant to each other, by computing the *attention score*

$$a_{ij} = \frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_k}} \tag{7.6}$$

between every pair of $i, j$ in the input sequence. Recall that the inner product depends on the absolute length and angle between the two vectors, meaning that the attention score is largely driven by how well aligned the query and key vectors are. The scaling factor $1/\sqrt{d_k}$ is a pragmatic solution that keeps the values of the attention score $a_{ij}$ away from near-saturated regimes of the softmax function. A higher attention score suggests that the tokens are more relevant to each other than a lower attention score. Note that the attention score is asymmetrical since the query $\mathbf{q}_i$ comes from token $i$ and the key $\mathbf{k}_j$ from token $j$, meaning that $a_{ij}$ might be large and $a_{ji}$ small, or vice versa.

A simplified but perhaps useful mental model is to think of the query as saying "Hey! I'm an adjective, and I'm looking for a noun that I could modify", whereas the key says either "I'm not a noun" or "I'm a noun". This would be the behaviour of an attention head whose parameters $\mathbf{W}^Q, \mathbf{W}^K$ have specialised on identifying adjective–noun relationships. Attention-head specialisation of this kind is illustrated in Figure 7.12, using actual attention heads from a real language model.

> ***Time to reflect 7.4*** *The attention score is in general asymmetric, meaning that $a_{ij} \neq a_{ji}$. Can you think of a concrete situation in a natural language where this asymmetry makes sense? And what would happen with the asymmetry if $\mathbf{W}^Q = \mathbf{W}^K$?*


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September* 1, 2026, `http://smlbook.org`

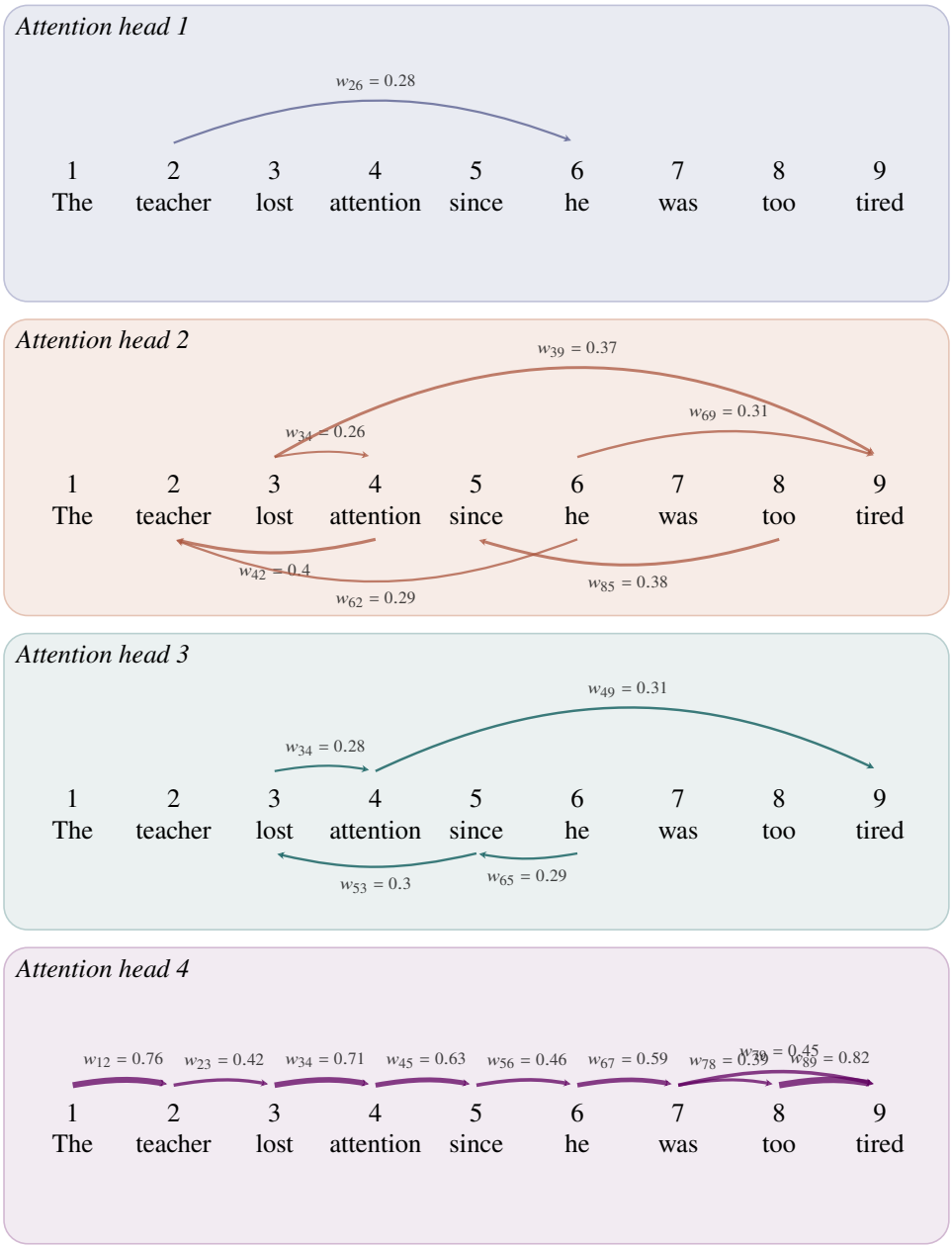



**Figure 7.12:** A visualisation of four attention heads in one of the BERT transformer models released by Google. The full model has 12 different attention heads that process the input in parallel, and the model has a depth of 12 such layers, so all in all there are 144 attention heads in that model. Here we show four of the heads from the first layer, for the sentence "The teacher lost attention since he was too tired". We illustrate the most dominant attention weights with arrows, using the threshold $w_{ij} > 0.25$, and the "ego-attention" for $i = j$ is not shown in the figure. The individual attention heads should not be overinterpreted, since it is the combined effect of all the attention heads that gives the model its power. However, we can see that the first and second heads make the connection between "teacher" and "he", whereas the third head makes the connection between "lost", "attention" and "tired". The fourth head seems to be more focused on local connections, making very strong connections between "the" and "teacher", "lost" and "attention", as well as "too" and "tired".

### Attention Weights and Contextualised Embeddings

The attention scores will be used to combine the different token embeddings into contextualised token embeddings. To this end, the attention scores are transformed such that they sum to 1 using the softmax function applied across all keys $j$ for each fixed query $i$,

$$[w_{i1} \ \cdots \ w_{id}] = \text{softmax}([a_{i1} \ \cdots \ a_{id}]) \,. \tag{7.7}$$

The resulting weights $w_{ij} \geq 0$, $\sum_j w_{ij} = 1$ are called the *attention weights*. They quantify how much token $i$ attends to the other positions in the sequence.

The final output from the attention head is then a weighted average of all the value vectors $\mathbf{v}_j$, $j = 1, \ldots, d$,

$$\mathbf{z}_i = \sum_{j=1}^{d} w_{ij} \, \mathbf{v}_j, \tag{7.8}$$

the so-called contextualised embedding for token $i$. Intuitively, $\mathbf{z}_i$ is a blend of the information carried by every token in the sequence, where the blend is shaped by how relevant each token is to the current one.

**Example 7.2 A toy attention computation**

Consider the three-token phrase "the ripe tomato", and suppose the tokens have already been projected to the following query, key and value vectors, with $d_k = d_v = 2$:

| | `the` | `ripe` | `tomato` |
|---|---|---|---|
| query $\mathbf{q}_j$ | $[1, 0]^\mathsf{T}$ | $[2, 0]^\mathsf{T}$ | $[1, 0]^\mathsf{T}$ |
| key $\mathbf{k}_j$ | $[0, 2]^\mathsf{T}$ | $[1, 1]^\mathsf{T}$ | $[3, 0]^\mathsf{T}$ |
| value $\mathbf{v}_j$ | $[1, 0]^\mathsf{T}$ | $[0, 1]^\mathsf{T}$ | $[2, 2]^\mathsf{T}$ |

Now let us focus on the adjective `ripe` at position 2 and see how the attention mechanism interprets it. Its query vector $\mathbf{q}_2 = [2, 0]^\mathsf{T}$ points along the first coordinate, the direction in which the key $\mathbf{k}_3$ for `tomato` also lies. We could read this as the query for `ripe` saying "I'm an adjective, looking for a noun", whereas the key for `tomato` says "I'm that noun". The (scaled) attention scores $a_{2j} = \mathbf{q}_2^\mathsf{T}\mathbf{k}_j / \sqrt{d_k}$ become

$$a_{2,\texttt{the}} = \frac{0}{\sqrt{2}} = 0, \qquad a_{2,\texttt{ripe}} = \frac{2}{\sqrt{2}} \approx 1.41, \qquad a_{2,\texttt{tomato}} = \frac{6}{\sqrt{2}} \approx 4.24,$$

and passing through the softmax gives the attention weights

$$\begin{bmatrix} w_{2,\texttt{the}} & w_{2,\texttt{ripe}} & w_{2,\texttt{tomato}} \end{bmatrix} = \text{softmax}([\,0 \ \ 1.41 \ \ 4.24\,]) \approx [\,0.01 \ \ 0.06 \ \ 0.93\,],$$

so `ripe` attends almost entirely to `tomato`. The contextualised embedding is the corresponding weighted average of the value vectors,

$$\mathbf{z}_2 = \sum_{j=1}^{3} w_{2j} \, \mathbf{v}_j \approx 0.01\,[1, 0]^\mathsf{T} + 0.06\,[0, 1]^\mathsf{T} + 0.93\,[2, 2]^\mathsf{T} \approx [1.88, 1.92]^\mathsf{T},$$

which is almost exactly the value vector of `tomato`. The static embedding of `ripe` has thus been pulled towards the token it modifies: in context, it has become "ripe,


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026, `http://smlbook.org`*

as applied to a tomato". Repeating the same three steps with $\mathbf{q}_1$ and $\mathbf{q}_3$ as the query produces the contextualised embeddings of the other two tokens.

Note the role of the scaling factor $1/\sqrt{d_k}$. Without it, the raw scores $[0, 2, 6]$ would give weights $\approx [0.00, 0.02, 0.98]$ – an even sharper distribution. With many dimensions the unscaled scores grow large enough to push the softmax into a near-saturated regime where almost all weight collapses onto a single token; dividing by $\sqrt{d_k}$ keeps the distribution in a usable range.

Attention head

**Input:** Parameters $\mathbf{W}^Q, \mathbf{W}^K, \mathbf{W}^V$ and a sequence of length $d$ with token embeddings $\{\mathbf{x}_i\}_{i=1}^d$

**Result:** A sequence of length $d$ with contextualised embeddings $\{\mathbf{z}_i\}_{i=1}^d$

**for** *each token embedding* $\mathbf{x}_i$, $i = 1, \ldots, d$ **do**
  Compute query vector: $\mathbf{q}_i = \mathbf{W}^Q \mathbf{x}_i$
  Compute key vector: $\mathbf{k}_i = \mathbf{W}^K \mathbf{x}_i$
  Compute value vector: $\mathbf{v}_i = \mathbf{W}^V \mathbf{x}_i$
**end**
**for** *each position* $i = 1, \ldots, d$ **do**
  **for** *each position* $j = 1, \ldots, d$ **do**
    Compute attention score: $a_{ij} = \frac{\mathbf{q}_i^\mathsf{T} \mathbf{k}_j}{\sqrt{d_k}}$
  **end**
  Compute attention weights: $[w_{i1} \ \cdots \ w_{id}] = \text{softmax}([a_{i1} \ \cdots \ a_{id}])$
  Compute contextualised embedding for token $i$: $\mathbf{z}_i = \sum_{j=1}^d w_{ij} \mathbf{v}_j$
**end**

**Method 7.1:** The core mechanism of attention, a so-called attention head.

The full procedure is summarised in Method 7.1. Note that although it is described position by position, all positions can be processed in parallel by stacking the queries, keys, and values into matrices. Letting $\mathbf{Q}, \mathbf{K} \in \mathbb{R}^{d \times d_k}$ and $\mathbf{V} \in \mathbb{R}^{d \times d_v}$ collect the corresponding vectors as rows, so that row $i$ of $\mathbf{Q}$ is $\mathbf{q}_i^\mathsf{T}$, and similarly for $\mathbf{K}$ and $\mathbf{V}$, the entire attention head can be expressed compactly as

$$\mathbf{Z} = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\mathsf{T}}{\sqrt{d_k}}\right)\mathbf{V}, \tag{7.9}$$

where the softmax is applied row-wise to the $d \times d$ matrix $\mathbf{Q}\mathbf{K}^\mathsf{T}$, and row $i$ of $\mathbf{Z} \in \mathbb{R}^{d \times d_v}$ is the output embedding $\mathbf{z}_i^\mathsf{T}$. This matrix formulation makes it possible to process the entire input sequence in a single step, with no sequential dependence. This is a key property that makes transformers very efficient on modern parallel hardware. However, the $d \times d$ matrix $\mathbf{Q}\mathbf{K}^\mathsf{T}$ implies that compute and memory will still grow quadratically with sequence length, even though the parameter count only grows linearly with $d$.

### Positional Encoding

One subtlety of the attention mechanism as described so far is that $a_{ij}$ depends only on the content of tokens $i$ and $j$ – their query and key vectors – and not on their positions in the sequence. The same attention weights would be produced if the tokens were shuffled into a different order, which means that a pure attention head has no sense of word order. This is a significant limitation: word order matters, and the sentence *"I think, therefore I am"* has a very different meaning from *"I am, therefore I think"*.

To remedy this, *positional encodings* are used. One type of positional encoding is to add a vector to each token embedding before the attention mechanism is applied, like $\mathbf{x}_i + \mathbf{p}_i$ where the positional encoding is a vector $\mathbf{p}_i \in \mathbb{R}^{d_{\text{emb}}}$ that depends on the token position index $i$ but not on the token itself. This implies that two tokens at different positions in the sequence will produce different queries and keys even if they share the same token embedding. The result is that the attention scores pick up both content similarity (from the token embeddings) and proximity (from the positional encodings), giving the model access to both kinds of information.

There are also more elaborate ways to include a positional encoding. A now widely used alternative is *rotary positional encoding* (RoPE), which applies a position-dependent rotation to the query and key vectors rather than adding a vector to the embeddings. The rotation angle is proportional to the position index, and it is constructed so that the dot product $\mathbf{q}_i^\mathsf{T}\mathbf{k}_j$ depends on the token contents and on the *relative* offset $i - j$ between the two positions, rather than on the absolute positions $i$ and $j$ separately. This relative-position property is the main motivation for RoPE: a relationship the model has learned for one pair of positions applies equally to any other pair the same distance apart, which also tends to make the model generalise better to sequences longer than those seen during training.

> ***Time to reflect 7.5*** *Without positional encoding, we claim that the attention mechanism would produce the same output embeddings for the sentences "dog bites man" and "man bites dog". Convince yourself that this is true by tracing through what would happen in an attention head for these two sentences.*

### Multi-Head Attention

The attention head described in the previous section is the central building block of the transformer, playing a role analogous to a single filter in a convolutional layer in a CNN. Just as a single convolutional filter can detect only one type of local feature, a single attention head can specialise in only one type of relationship between tokens. To build a richer representation, transformers use several attention heads in parallel, a design called *multi-head attention*, and then stack multiple such layers on top of each other.

In multi-head attention, $H$ attention heads are run in parallel, each with its own set of weight matrices $\mathbf{W}_h^Q, \mathbf{W}_h^K, \mathbf{W}_h^V$ for $h = 1, \ldots, H$. Each head produces its own

output sequence, collected row-wise into a matrix $\mathbf{Z}_h \in \mathbb{R}^{d \times d_v}$ by (7.9). For each position $i$, the $H$ output vectors from the different heads are concatenated into a single vector of length $Hd_v$, and then projected back to dimension $d_{\text{emb}}$ via a learned matrix $\mathbf{W}^O \in \mathbb{R}^{Hd_v \times d_{\text{emb}}}$. In matrix form,

$$\text{MultiHead}(\mathbf{X}) = \left[\mathbf{Z}_1 \;\middle|\; \cdots \;\middle|\; \mathbf{Z}_H\right] \mathbf{W}^O \tag{7.10}$$

where the matrices $\mathbf{Z}_1, \ldots, \mathbf{Z}_H$ are placed side by side, so that $[\mathbf{Z}_1 | \cdots | \mathbf{Z}_H] \in \mathbb{R}^{d \times Hd_v}$ and the final output lies in $\mathbb{R}^{d \times d_{\text{emb}}}$. It is common to set $d_v = d_k = d_{\text{emb}}/H$ so that the total computation per layer is comparable to that of a single head operating at full dimension.

Different heads learn to specialise in different types of relationships. In a trained language model, one head might learn to link pronouns to their antecedents (connecting *he* to *teacher* in *"The teacher lost attention since he was too tired"*, as in Figure 7.12), while another might capture syntactic dependencies between verbs and their objects, and a third might focus on adjacent-token relationships. As with CNN filters, the specialisations emerge when learning from training data, and are not crafted by hand.

### The Transformer Block

A full transformer model is typically constructed by stacking several identical transformer blocks. As shown in Figure 7.13, each block consists of two primary sub-layers: a multi-head attention mechanism and a position-wise MLP. These sub-layers are wrapped in a specific "Add & Norm" structure, where the input to the sub-layer is added back to its output via a residual connection, followed by a layer normalisation step. Concretely, with input $\mathbf{X} \in \mathbb{R}^{d \times d_{\text{emb}}}$ (the embeddings collected as rows, as in (7.9)), the standard transformer block computes

$$\mathbf{X}' = \text{LayerNorm}(\mathbf{X} + \text{MultiHead}(\mathbf{X})), \tag{7.11a}$$

$$\mathbf{X}'' = \text{LayerNorm}(\mathbf{X}' + \text{MLP}(\mathbf{X}')), \tag{7.11b}$$

where the MLP is a two-layer dense network applied independently and identically to each token position:

$$\text{MLP}(\mathbf{x}) = \mathbf{W}_2\, h(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2, \tag{7.12}$$

(where $h$ is an activation function, GELU is a common choice here). This design ensures that while the attention mechanism allows tokens to "talk" to each other, the MLP provides the necessary non-linear processing at each individual position.

Recall from Section 6.3 that residual connections (the additive terms $+\mathbf{X}$) solve the vanishing gradient problem by providing a gradient path through the stack and making the identity mapping the "default behaviour" of any layer.

Layer normalisation, on the other hand, serves as a numerical stabiliser by ensuring that numbers are placed on an equal numerical footing before moving to the next layer. Unlike batch normalisation, which computes statistics across different data

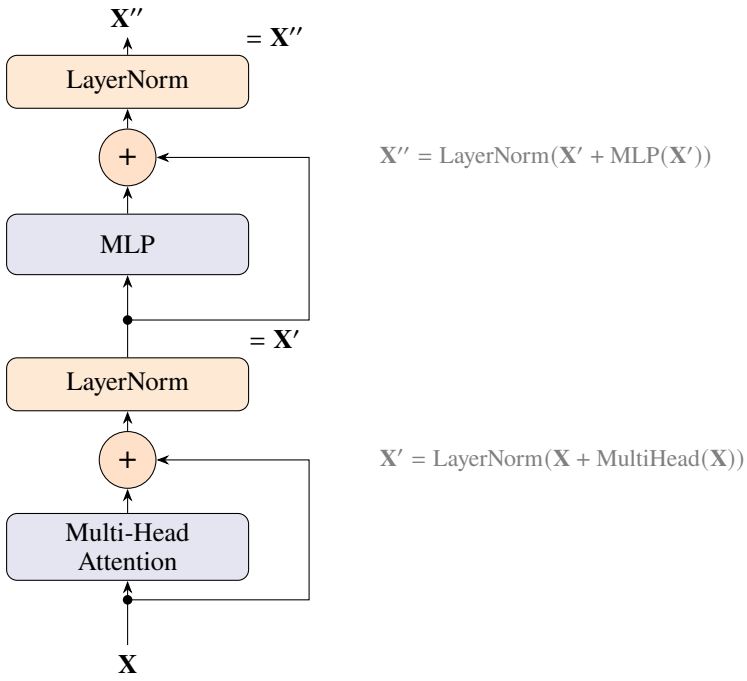



**Figure 7.13:** A transformer block. The input $\mathbf{X}$ is first processed by multi-head attention, with the result added to the original input via a residual connection and normalised to produce $\mathbf{X}'$. A position-wise MLP then processes $\mathbf{X}'$ in the same fashion to produce the block output $\mathbf{X}''$. Layer normalisation and residual connections appear in both sub-layers to stabilise training.

points, layer normalisation is here applied to each embedding vector independently. It calculates the mean and variance across the elements of each embedding vector independently, which makes it particularly well-suited for transformers working on text since it is invariant to the length of the input sequence.

The transformer block is summarised in Figure 7.13. The specific layer ordering in the figure is known as Post-LN, whereas the Pre-LN version places the normalisation before the sub-layer and has gained more attention recently. There is indeed no single "correct" design; instead, these choices reflect an ongoing process of trial and refinement, where machine learning engineers balance different configurations in search of the ones that perform best.

> ***Time to reflect 7.6*** *The MLP in the transformer block is described as "position-wise": the same small network is applied independently at each position. Why not use a single large MLP operating on the entire sequence of embeddings at once?*

### Stacking Transformer Blocks

The full transformer model consists of $L$ transformer blocks stacked sequentially. The output of block $\ell$ is fed as input to block $\ell + 1$, and each block has its own independent set of parameters. Across the layers, the sequence of representations progresses from raw token embeddings in the first layer to increasingly abstract, contextualised representations in deeper layers. Early layers tend to capture local syntactic patterns, while deeper layers encode higher-level semantic relationships – a hierarchy closely analogous to the one observed in deep CNNs. Typical values of $L$ range from 6 in small models to 96 or more in the largest language models. Similarly, the number of heads $H$ typically ranges from 8 to 128, and the embedding dimension $d_{\text{emb}}$ from 512 to 12 288. Together, these choices determine the total number of parameters in the model, which ranges from tens of millions in compact models to hundreds of billions in the largest deployed systems.

All together, the transformer model takes a sequence of token embeddings and returns another sequence of contextualised embeddings of the same length. It is, in this sense, a generic sequence-to-sequence map: nothing in the architecture as described so far ties it specifically to language. What we ultimately do with the output sequence depends entirely on the task at hand. We could pool it into a single vector and feed that to a classifier (for sentiment analysis, say), or attach a small network to each output position (for tagging tasks), or, as we will see in the next section, use it to predict the next token in a sequence. It is the latter use that has turned the transformer into the workhorse of modern natural language processing, and that we will now develop in detail.

It is however worth remembering that transformers are far from limited to text data. They have been successfully used in applications such as time-series forecasting, protein structure prediction, image recognition (by treating image patches as tokens), and reinforcement learning, among many other domains. Wherever the relevant relationships in the data are long-range and not easily captured by local operations, the attention mechanism provides an interesting and powerful alternative to the convolutions used in a CNN.

## 7.5 Large Language Models

The perhaps most important use of the transformer architecture is as a *language model*: a model that, given a sequence of tokens, predicts the next one. This is a particular way of using the generic embedding-to-embedding map provided by a transformer. We now compile a language model for that task, which is built with a transformer and some additional token handling at the input and output ends.

### From Transformer to Language Model

A transformer block, or a stack of them, takes a sequence of token embeddings as input and produces a sequence of contextualised embeddings of the same length as

output. To turn this into a language model that assigns probabilities to the next token in a sequence, we need to add a tokeniser and embedding layer at the input, as well as a final linear layer that maps the output embedding at the last position to a vector of logits of size $V$ (the vocabulary size), followed by a softmax that converts the logits to a probability distribution over all tokens,

$$p(\text{next token} \mid \text{input sequence}) = \text{softmax}\left(\mathbf{W}^{\text{out}}\mathbf{z}_d + \mathbf{b}^{\text{out}}\right), \tag{7.13}$$

where $\mathbf{W}^{\text{out}} \in \mathbb{R}^{V \times d_{\text{emb}}}$ and $\mathbf{b}^{\text{out}} \in \mathbb{R}^{V}$ also are learned during training. The full pipeline, from raw text to a probability distribution over the next token, is illustrated in Figure 7.14. This $p(\text{next token} \mid \text{input sequence})$ is a conditional distribution over the vocabulary; sampling from it to produce text uses the model as a conditional generative model (Section 3.5), as we detail below.

The input sentence *"the overfitted model learns the"* is tokenised, embedded, and passed through $L$ transformer blocks. The output distribution over the vocabulary will, if the model is trained properly, assign high probabilities to tokens such as *noise*, *patterns*, or maybe *typos*, and low probability to most other tokens. By sampling from this distribution and appending the sampled token to the sequence, the model can then generate the token after that, and so on, producing text one token at a time. This autoregressive generation process is summarised in Method 7.2.

Language model for text generation

**Input:** A prompt (partial sentence) and a maximum length $T$
**Result:** A generated continuation of the prompt

Tokenise the prompt to obtain token indices $t_1, \ldots, t_d$
Look up embeddings: $\mathbf{x}_i = \mathbf{W}^{\text{emb}}\mathbf{e}_{t_i}$ for $i = 1, \ldots, d$
**for** $\tau = 1, 2, \ldots, T$ **do**
    Add positional encodings: $\tilde{\mathbf{x}}_i = \mathbf{x}_i + \mathbf{p}_i$
    Pass $\tilde{\mathbf{x}}_1, \ldots, \tilde{\mathbf{x}}_{d+\tau-1}$ through the $L$ transformer blocks to obtain $\mathbf{z}_{d+\tau-1}$
    Compute logits and distribution: $\mathbf{s} = \text{softmax}(\mathbf{W}^{\text{out}}\mathbf{z}_{d+\tau-1} + \mathbf{b}^{\text{out}})$
    Sample next token index $t_{d+\tau}$ from $\mathbf{s}$
    Look up embedding $\mathbf{x}_{d+\tau} = \mathbf{W}^{\text{emb}}\mathbf{e}_{t_{d+\tau}}$
    **if** $t_{d+\tau}$ *is the end-of-sequence token* **then**
        **break**
    **end**
**end**
Convert token indices $t_{d+1}, \ldots$ back to text and return

**Method 7.2:** Autoregressive text generation with a language model.

Sampling the next token from the predicted distribution $p(\text{next token} \mid \text{input sequence})$ rather than reporting its most probable value is what makes a language model a *conditional generative model* in the sense of Section 3.5: the softmax output – a multiclass

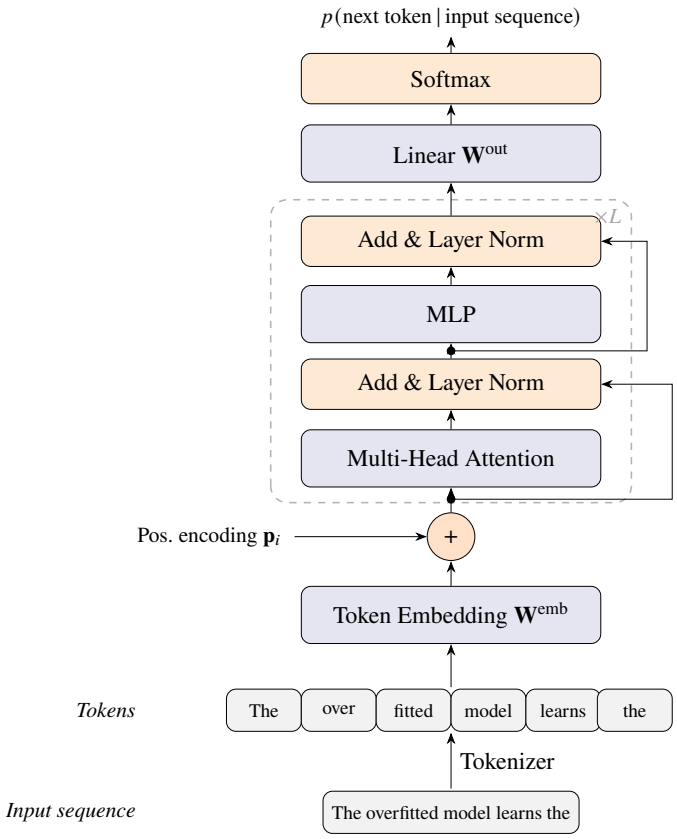



**Figure 7.14:** Architecture of a language model using transformers. **Bottom:** The input sequence is tokenised, and the input tokens are looked up in the token embedding matrix $\mathbf{W}^{\text{emb}}$ to produce dense vectors, and a positional encoding $\mathbf{p}_i$ is added at each position to encode word order. **Middle:** The resulting sequence passes through $L$ transformer blocks (dashed rectangle, repeated $L$ times). Each block contains two sub-layers: multi-head attention, and a position-wise dense network applied independently at each position. Both sub-layers are wrapped in an Add & Layer Norm step: the sub-layer input is added ("Add") back to its output via a residual connection (the arrows routed to the right of the block) before layer normalisation ("Layer Norm") is applied. All positions in the sequence are processed in parallel within each block. **Top:** The output embedding at the final position is projected to a vector of length $V$ (the vocabulary size) by the linear layer $\mathbf{W}^{\text{out}}$, and a softmax converts the logits to a probability distribution over all tokens in the vocabulary, from which the next token can be sampled.

distribution over the $V$ tokens – is used *generatively* (by sampling) rather than *discriminatively* (by taking the most likely token). It is worth noting that this sampling procedure is stochastic, which in practice means the model might return different results every time. That is indeed the case in many applications where language models are used as chatbots. Several elaborate sampling strategies exist for making the generated text "feel" more like natural human language, for example by scaling the probabilities and avoiding word repetitions. Always taking the token with the highest predicted probability, rather than sampling, often turns out to give a rather unnatural

"feel" to the generated text.

### Training in Parallel

The language model Method 7.2 generates text *sequentially*: each token must be sampled before the next can be predicted, since the sampled token becomes part of the input at the following step. Training, by contrast, can be carried out fully *in parallel*, and it is this asymmetry that makes it feasible to train a model on massive amounts of texts.

To see why, consider how training data is constructed. Suppose the token sequence "We choose hope over fear" appears in the training text. For the task of predicting the next token, this short sentence already contains four distinct prediction problems:

| Input token sequence | Output token |
|---:|---|
| `We` | `choose` |
| `We choose` | `hope` |
| `We choose hope` | `over` |
| `We choose hope over` | `fear` |

These input–output pairs are exactly the *self-supervised* construction of Section 2.4: an unlabelled sequence is turned into supervised pairs by letting the input be the tokens seen so far and the target be the next one. Where Example 2.7 did this for a numerical time series, here we do the same for a sequence of text tokens. The target sequence is simply the input sequence shifted one step to the left.

Treated literally, these are four separate inputs of growing length, requiring four forward passes – and a training data of billions of tokens would demand a hopeless number of them. Here the parameter sharing of the attention mechanism pays off. We have already seen that sharing the matrices $\mathbf{W}^Q, \mathbf{W}^K, \mathbf{W}^V$ across positions keeps the parameter count independent of the sequence length $d$. The same property means that the computation at every position is not only identical but *independent of the others*, so that (recall the matrix form (7.9)) the whole sequence passes through a block in a single parallel step rather than one position at a time. We can therefore feed the entire sequence "We choose hope over fear" through the model *once* and read off, at every position $i$ at the same time, the predicted distribution for token $i + 1$. A single forward pass yields all four training pairs simultaneously.

For this to be legitimate, the prediction at position $i$ must depend only on tokens $1, \ldots, i$; otherwise the model could simply read the answer off the later position it is meant to predict. The remedy is straightforward: before the softmax, set the attention scores $a_{ij} = -\infty$ for every $j > i$, which forces the corresponding weights $w_{ij}$ to zero, so that each position attends only to itself and to earlier positions. In matrix form this masks out the upper triangle of $\mathbf{QK}^\mathsf{T}$ in (7.9). This constraint is called *causal masking* (or autoregressive masking). It introduces no new parameters and changes nothing conceptually; it merely enforces, during the parallel pass, the same look-back-only restriction that sequential generation obeys automatically.

With causal masking in place the entire sequence is trained in one pass, and the objective is the average negative log-likelihood over all positions,

$$J(\boldsymbol{\theta}) = -\frac{1}{d-1}\sum_{i=1}^{d-1} \log p(t_{i+1} \mid t_1, \ldots, t_i; \boldsymbol{\theta}), \tag{7.14}$$

where $\boldsymbol{\theta}$ collects all trainable parameters (the embedding matrix, the attention weight matrices, the fully connected weights, and the output projection). The self-supervision and parallelism are key enablers for allowing virtually the entire written record of humanity to serve as training material, which makes it possible to scale language models into so-called *large language models* (LLM).

### Encoders, Decoders, and Cross-Attention

Predicting the next token is by far the most common use of a transformer today, but not the only one. A transformer block on its own simply maps a sequence of embeddings to a contextualised sequence of the same length, and what we attach to that output depends on the task. The causal mask is what specialises the architecture for generation: models built around it, attending only backwards, are called *decoders*, and the large majority of modern large language models are decoders in this sense.

When the task is instead to produce some output from a *complete* input sequence – classifying the sentiment of a sentence, labelling each token with its part-of-speech tag, or producing a single vector embedding of the whole sentence – there is no reason to forbid a position from also looking forwards. Dropping the mask gives a *bidirectional* model, in which every position attends to the entire sequence. Such models are called *encoders*, and are built from exactly the same transformer blocks, only without the causal mask.

Encoders and decoders can also be combined, classically for language translation, through a third use of attention called *cross-attention*. Mechanically nothing changes from the self-attention of (7.9) except one small but important twist: the queries $\mathbf{q}_i$ are computed from one sequence, while the keys $\mathbf{k}_j$ and values $\mathbf{v}_j$ come from another. In a Chinese-to-English translation model, this is how the decoder generating the English output "looks at" the encoded Chinese sentence. The same pattern is what lets a text decoder attend to image-patch embeddings in image captioning, or to audio embeddings in speech-to-text, and more generally wherever one sequence is generated under the influence of another. It is a key building block of so-called *multimodal* models, capable of jointly processing inputs of different kinds, such as text and images.

This lets us sharpen the term *decoder*: a decoder generates an output sequence one token at a time, using causally masked self-attention and possibly also cross-attention to whatever other information is relevant. A plain next-token language model is a decoder with no such additional input and hence no cross-attention, and that is the case we return to for the remainder of the chapter.

### Scaling to Large Language Models

A language model of the type described above becomes a *large language model* when it is scaled up along all relevant dimensions simultaneously: more transformer blocks, more attention heads, a larger embedding dimension, and crucially a much larger training data set. Empirical research has shown that the performance of language models improves remarkably predictably as a function of the number of parameters and the amount of training data, following so-called *scaling laws*. This predictability has encouraged the development of progressively larger models, from hundreds of millions of parameters in early transformer language models to hundreds of billions of parameters in the largest systems today.

It turns out that by training a language model on vast amounts of text data, the model learns not only the structure of the language (grammar, and so on) but also the content of what is being discussed or described in that text. Such a model is therefore able not only to produce human-like written language, but also to discuss facts about the real world.

A very important application of large language models is as conversational assistants, or *chatbots*. To make a language model better suited to a chatbot setting, transfer learning is often applied. A straightforward supervised fine-tuning method is to train an already pre-trained model on a much smaller but well-prepared dataset of question-and-answer pairs, teaching the model to answer questions rather than just continuing the provided text input. Such a question-and-answer dataset is however more expensive to collect than just large amounts of plain texts, since it needs to be sourced from humans. A method that is typically added in addition to supervised fine-tuning is therefore *reinforcement learning from human feedback*, in which different responses are sampled from a model and a human is ranking the different responses based on how good they are. Letting humans rank different responses is typically more efficient than writing the responses from scratch. These rankings are used to train a separate reward model, with the purpose of predicting how a human would rank any given answer, and the reward model is in turn used for training the language model in a reinforcement learning framework that goes beyond the scope of this book.

Fine-tuning on question-and-answer pairs has proven vital for building well-performing LLM-chatbots, but the vast majority of what an LLM-chatbot "knows" (the factual knowledge and the understanding of language) is, however, still acquired during the initial training on huge text corpora. Fine-tuning, which requires far less data, largely teaches the model *how* to format its answers, but not primarily the semantic content of those answers.

> ***Time to reflect 7.7*** *Large language models have hundreds of billions of parameters, yet in practice they do not overfit in the ways one might expect. Why do you think that is the case?*


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September* 1, 2026, `http://smlbook.org`

### Beyond Text Generation: Context, Hallucinations, and Agents

The LLM-chatbot application is arguably one of the societally most impactful applications of machine learning, and a new world of methods, technology and terminology is being built upon it.

A key term is the *prompt*, which is another word for the input to an LLM, and the art of formulating good prompts has become an emerging topic among practitioners. And while the underlying model is optimized for generating text that is likely, which arguably gives the model a vast amount of "understanding" of the real world in topics such as politics, physics and art, LLMs may also produce grammatically correct and well-argued texts that yet are factually ungrounded, so called *hallucinations*. While this phenomenon is expected from a machine learning point of view (again, the model is optimised for generating texts that appear similar to text seen during training), it is a considerable challenge for practitioners.

One way to adapt LLM-chatbots to specific situations, without requiring transfer learning, is to append relevant information to the prompt. The so-called *retrieval-augmented generation* is a method to first search for information in certain sources (such as documents that were not available during training), and then add relevant parts of that to the prompt. This has also proven to reduce the amount of hallucinations.

Beyond natural languages, LLMs can be trained also on programming languages, and have proven capable in both understanding and generating programming code. Since the syntax of a programming language follows strict formal rules (such as the indentation in Python), it is possible to constrain the sampling of the next token such that the model is guaranteed to always generate syntactically valid programming code. Beyond acting as a code-generating chatbot, this capability allows LLMs to also work as agents. In an LLM-agent, the model is instructed to follow a schema that can be parsed and executed by an external program, with outputs like:

```
{
  "action": "send_email",
  "parameters": {
    "recipient": "colleague@example.com",
    "subject": "Project Update",
    "body": "The analysis is complete. The results look good."
  }
}
```

Here the LLM does not merely "talk" about sending an email, but it generates a command that triggers a mail server. The system can then feed the execution result (success or failure) back to the LLM-agent, allowing it to further react to the outcome of its own actions.

## 7.6 Further Reading

The original CNN idea was formulated by LeCun, Bottou, et al. (1998), and the modern CNN architecture was developed actively during the 2010s, prior to the rise of transformer models. A recent view on CNNs, applying the latest best practice for neural networks, is found in Liu et al. (2022). The batch normalisation was suggested by Ioffe and Szegedy (2015), and the original ResNet paper was written by He et al. (2016).

For transformer and language models, a more detailed and hands-on introduction is given by Burkov (2025). The landmark paper that established the transformer model, 'Attention is All you Need', was written by Vaswani et al. (2017). Two highly influential papers that laid the foundation for the modern LLM era are probably Brown et al. (2020), which introduced the first language model with more than 100 billion parameters, and Ouyang et al. (2022), which outlined the very successful strategy of reinforcement learning from human feedback that enabled ChatGPT.

# 8 Ensemble Methods: Bagging and Boosting

In the preceding chapters, we have seen several examples of different machine learning models, from $k$-NN to deep neural networks. In this chapter, we will introduce a new way of constructing models, by combining multiple instances of some basic model. We refer to this as an *ensemble of base models*, and the resulting methods are consequently referred to as *ensemble methods*. The key idea behind ensemble methods is the 'wisdom of crowds': by training each base model in a slightly different way, they can all contribute to learning the input–output relationship. Specifically, to obtain a prediction from an ensemble, we let each base model make its own prediction and then use a (possibly weighted) average or majority vote to obtain the final prediction. With a carefully constructed ensemble, the prediction obtained in this way is better than the predictions of the individual base models.

We start in Section 8.1 by introducing a general technique referred to as bootstrap aggregating, or *bagging* for short. The bagging idea is to first create multiple slightly different 'versions' of the training data by, essentially, randomly sampling overlapping subsets of the training data (the so-called bootstrap). Thereafter, one base model is trained from each such 'version' of the training data. In this way, an ensemble of similar, but not identical, base models is obtained. With this procedure it is possible to *reduce the variance* (without any notable increase in bias) compared to using only a single base model trained on the entire training dataset. In practice, this means that by using bagging, the risk of overfitting decreases compared to using the base model itself. In Section 8.2 we introduce an extension to bagging only applicable when the base model is a classification or regression tree, which results in a powerful off-the-shelf method called random forests. In random forests, each tree is randomly perturbed in order to obtain additional variance reduction, beyond what is already obtained by the bagging procedure itself.

In Sections 8.3–8.4 we introduce another ensemble method known as *boosting*. Boosting is different from bagging and random forests, since its base models are trained sequentially, one after the other, where each model tries to 'correct' for the 'mistakes' made by the previous ones. Contrary to bagging, the main effect of boosting is *bias reduction* compared to the base model. Thus, boosting is able to turn an ensemble of 'weak' base models (e.g., linear classifiers) into one 'strong' ensemble model (e.g., a heavily non-linear classifier).

## 8.1 Bagging

As already discussed in Chapter 4, a central concept in machine learning is the bias–variance trade-off. Roughly speaking, the more flexible a model is, the lower its bias will be. That is, a flexible model is capable of representing complicated input–output relationships. Examples of simple yet flexible models are $k$-NN with a small value of $k$ and a classification tree that is grown deep. Such highly flexible models are sometimes needed for solving real-world machine learning problems, where relationships are far from linear. The downside, however, is the risk of overfitting, or equivalently, high model variance. Despite their high variance, those models are not useless. By using them as base models in bootstrap aggregating, or *bagging*, we can

reduce the variance of the base model without increasing its bias.

We outline the main idea of bagging with the example below.

**Example 8.1 Using bagging for a regression problem**

Consider the data (black dots) that are drawn from a function (dashed line) plus noise in Figure 8.1a. As always in supervised machine learning, we want to train a model from the data which is able to predict new data points well. Being able to predict new data points well means, among other things, that the model should predict the dotted line at $x_\star$ (the empty blue circle) well.

To solve this problem, we could use any regression method. Here, we use a regression tree which is grown until each leaf node only contains one data point, whose prediction is shown to the lower left in Figure 8.1c (blue line and dot). This is a typical low-bias high-variance model, and the overfit to the training data is apparent from the figure. We could decrease its variance, and hence the overfitting, by using a shallower tree, but that would on the other hand increase the bias. Instead, we lower the variance (without increasing the bias much) by using bagging with the regression tree as base model.

The rationale behind bagging goes as follows: Because of the noise in the training data, we may think of the prediction $\hat{y}(x_\star)$ (the blue dot) as a random variable. In bagging, we learn an ensemble of base models (Figure 8.1b), where each base model is trained on a different 'version' of the training data, obtained using the bootstrap. We may therefore think of each base model as a different realisation of the random variable $\hat{y}(x_\star)$. The average of multiple realisations of a random variable has a lower variance than the random variable itself, which means that by taking the average (Figure 8.1d) of all the base models, we obtain a prediction with less variance than the base model itself. That is, the bagged regression tree (Figure 8.1d) has lower variance than a single prediction tree (Figure 8.1c). Since the base model itself also has low bias, the averaged prediction will have low bias *and* low variance. We can visually confirm that the prediction is better (blue dot and circle are closer to each other) for bagging than for the single regression tree.

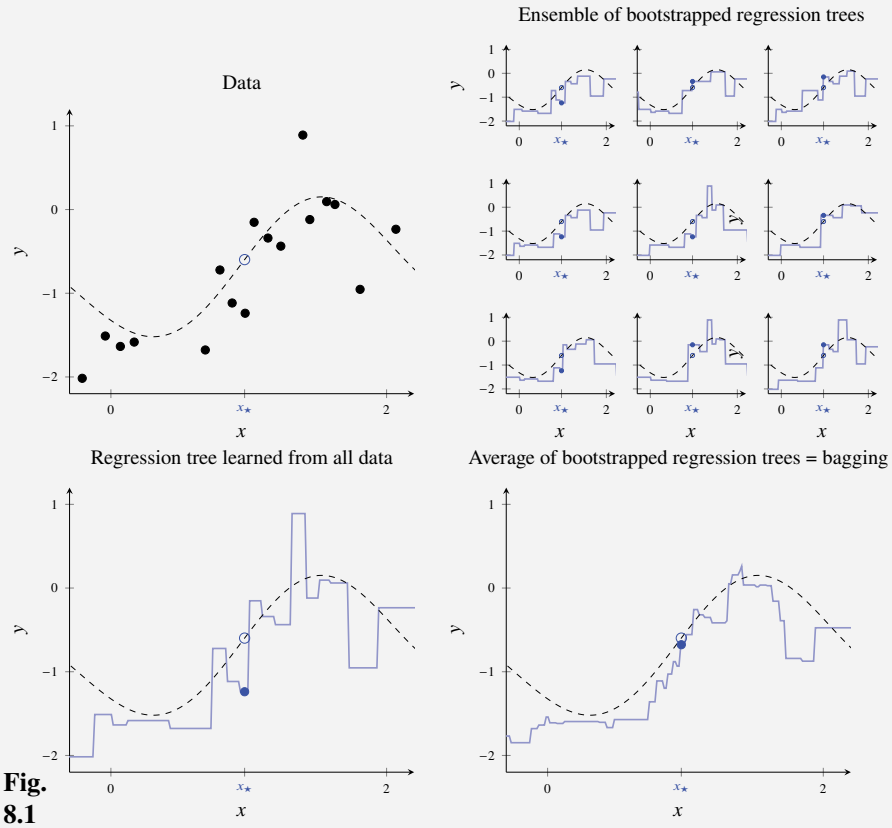



**Fig. 8.1**

## The Bootstrap

As outlined in Example 8.1, the idea of bagging is to average over multiple base models, each learned from a different training dataset. Therefore, we first have to construct different training datasets. In the best of worlds, we would just collect multiple datasets, but most often we cannot do that and instead we have to make the most of the limited data available. For this purpose, the bootstrap is useful.

The bootstrap is a method for artificially creating multiple datasets (of size $n$) out of one dataset (also of size $n$). The traditional usage of the bootstrap is to quantify uncertainties in statistical estimators (such as confidence intervals), but it turns out that it can also be used to construct machine learning models. We denote the original dataset $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^n$ and assume that $\mathcal{T}$ provides a good representation of the real-world data generating process, in the sense that *if* we were to collect more training data, these data points would likely be similar to the training data points already contained in $\mathcal{T}$. We can thus argue that randomly picking data points from $\mathcal{T}$ is a

reasonable way to simulate a 'new' training dataset. In statistical terms, instead of sampling from the population (collecting more data), we sample from the available training data, which is assumed to provide a good representation of the population.

The bootstrap is stated in Algorithm 8.1 and illustrated in Example 8.2 below. Note that the sampling is done with replacement, meaning that the resulting bootstrapped dataset may contain multiple copies of some of the original training data points, whereas other data points may not be included at all.

**Algorithm 8.1:** The bootstrap.

**Data:** Training dataset $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^n$

**Result:** Bootstrapped data $\widetilde{\mathcal{T}} = \{\widetilde{\mathbf{x}}_i, \widetilde{y}_i\}_{i=1}^n$

**for** $i = 1, \ldots, n$ **do**
  Sample $\ell$ uniformly on the set of integers $\{1, \ldots, n\}$
  Set $\widetilde{\mathbf{x}}_i = \mathbf{x}_\ell$ and $\widetilde{y}_i = y_\ell$
**end**

***Time to reflect 8.1*** *What would happen if the sampling was done without replacement in the bootstrap?*

## Example 8.2 The bootstrap

We have a small training dataset with $n = 10$ data points, with a two-dimensional input $\mathbf{x} = [x_1\ x_2]$ and a binary output $y \in \{\text{Blue}, \text{Red}\}$ (see Figure 8.2).

Original training data, $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^{10}$

| Index | $x_1$ | $x_2$ | $y$ |
|---|---|---|---|
| 1 | 9.0 | 2.0 | Blue |
| 2 | 1.0 | 4.0 | Blue |
| 3 | 4.0 | 6.0 | Blue |
| 4 | 4.0 | 1.0 | Blue |
| 5 | 1.0 | 2.0 | Blue |
| 6 | 1.0 | 8.0 | Red |
| 7 | 6.0 | 4.0 | Red |
| 8 | 7.0 | 9.0 | Red |
| 9 | 9.0 | 8.0 | Red |
| 10 | 9.0 | 6.0 | Red |

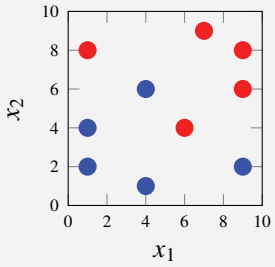



**Fig. 8.2**

To generate a bootstrapped dataset $\widetilde{\mathcal{T}} = \{\widetilde{\mathbf{x}}_i, \widetilde{y}_i\}_{i=1}^{10}$, we simulate 10 times with replacement from the index set $\{1, \ldots, 10\}$, resulting in the indices $\{2, 10, 10, 5, 9, 2, 5, 10, 8, 10\}$. Thus, $(\widetilde{\mathbf{x}}_1, \widetilde{y}_1) = (\mathbf{x}_2, y_2)$, $(\widetilde{\mathbf{x}}_2, \widetilde{y}_2) = (\mathbf{x}_{10}, y_{10})$, etc. We end up with the dataset in Figure 8.3, where the numbers in parentheses in the right panel indicate that there are multiple copies of some of the original data points in the bootstrapped data.


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September* 1, 2026, `http://smlbook.org`

Bootstrapped data, $\widetilde{\mathcal{T}} = \{\widetilde{\mathbf{x}}_i, \widetilde{y}_i\}_{i=1}^{10}$

| Index | $\widetilde{x}_1$ | $\widetilde{x}_2$ | $\widetilde{y}$ |
|---|---|---|---|
| 2 | 1.0 | 4.0 | Blue |
| 10 | 9.0 | 6.0 | Red |
| 10 | 9.0 | 6.0 | Red |
| 5 | 1.0 | 2.0 | Blue |
| 9 | 9.0 | 8.0 | Red |
| 2 | 1.0 | 4.0 | Blue |
| 5 | 1.0 | 2.0 | Blue |
| 10 | 9.0 | 6.0 | Red |
| 8 | 7.0 | 9.0 | Red |
| 10 | 9.0 | 6.0 | Red |

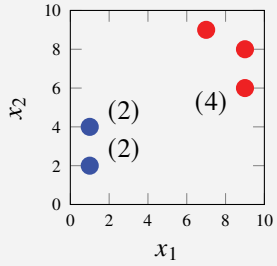



**Fig. 8.3**

## Variance Reduction by Averaging

By running the bootstrap (Algorithm 8.1) repeatedly $B$ times, we obtain $B$ random but *identically distributed* bootstrapped datasets $\widetilde{\mathcal{T}}^{(1)}, \ldots, \widetilde{\mathcal{T}}^{(B)}$. We can then use those bootstrapped datasets to train an ensemble of $B$ base models. We thereafter average their predictions

$$\widehat{y}_{\text{bag}}(\mathbf{x}_\star) = \frac{1}{B}\sum_{b=1}^{B} \widetilde{y}^{(b)}(\mathbf{x}_\star) \quad \text{or} \quad \mathbf{g}_{\text{bag}}(\mathbf{x}_\star) = \frac{1}{B}\sum_{b=1}^{B} \widetilde{\mathbf{g}}^{(b)}(\mathbf{x}_\star), \tag{8.1}$$

depending on whether we are concerned with regression (predicting an output value $\widehat{y}_{\text{bag}}(\mathbf{x}_\star)$) or classification (predicting class probabilities $\mathbf{g}_{\text{bag}}(\mathbf{x}_\star)$). The latter expression assumes that each base classifier outputs a vector of class probabilities. If this is not the case, we can instead obtain a 'hard' class prediction by taking a majority vote among the ensemble members. Note that it is natural to take a plain average across the ensemble (each member is weighted equally) in (8.1) due to the fact that all ensemble members are constructed in the same way, that is, they are identically distributed.

In (8.1), $\widetilde{y}^{(1)}(\mathbf{x}_\star), \ldots, \widetilde{y}^{(B)}(\mathbf{x}_\star)$ and $\widetilde{\mathbf{g}}^{(1)}(\mathbf{x}_\star), \ldots, \widetilde{\mathbf{g}}^{(B)}(\mathbf{x}_\star)$ denote the predictions from the individual ensemble members. The averaged prediction, denoted $\widehat{y}_{\text{bag}}(\mathbf{x}_\star)$ or $\mathbf{g}_{\text{bag}}(\mathbf{x}_\star)$, is the final prediction obtained from bagging. We summarise this by Method 8.1. (For classification, the prediction could alternatively be decided by majority vote among the ensemble members, but that typically degrades the performance slightly compared to averaging the predicted class probabilities.)

We will now give some more details on the variance reduction that happens in (8.1), which is the entire point of bagging. We focus on regression, but the intuition also works for classification.

First we make a basic observation regarding random variables, namely that averaging reduces variance. To formalise this, let $z_1, \ldots, z_B$ be a collection of identically distributed (but possibly dependent) random variables with mean value $\mathbb{E}[z_b] = \mu$ and variance $\text{Var}[z_b] = \sigma^2$ for $b = 1, \ldots, B$. Furthermore, assume that the average

**Learn** all base models

**Data:** Training dataset $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^n$
**Result:** $B$ base models

**for** $b = 1, \ldots, B$ **do**
  Run Algorithm 8.1 to obtain a bootstrapped training dataset $\widetilde{\mathcal{T}}^{(b)}$
  Learn a base model from $\widetilde{\mathcal{T}}^{(b)}$
**end**
Obtain $\widehat{y}_{\text{bag}}(\mathbf{x}_\star)$ or $\mathbf{g}_{\text{bag}}(\mathbf{x}_\star)$ by averaging (8.1).

**Predict** with the base models

**Data:** $B$ base models and test input $\mathbf{x}_\star$
**Result:** A prediction $\widehat{y}_{\text{bag}}(\mathbf{x}_\star)$ or $\mathbf{g}_{\text{bag}}(\mathbf{x}_\star)$

**for** $b = 1, \ldots, B$ **do**
  Use base model $b$ to predict $\widetilde{y}^{(b)}(\mathbf{x}_\star)$ or $\widetilde{\mathbf{g}}^{(b)}(\mathbf{x}_\star)$
**end**
Obtain $\widehat{y}_{\text{bag}}(\mathbf{x}_\star)$ or $\mathbf{g}_{\text{bag}}(\mathbf{x}_\star)$ by averaging (8.1).

**Method 8.1:** Bagging

correlation[1] between any pair of variables is $\rho$. Then, computing the mean and the variance of *the average* $\frac{1}{B}\sum_{b=1}^B z_b$ of these variables, we get

$$\mathbb{E}\left[\frac{1}{B}\sum_{b=1}^{B} z_b\right] = \mu, \tag{8.2a}$$

$$\operatorname{Var}\left[\frac{1}{B}\sum_{b=1}^{B} z_b\right] = \frac{1-\rho}{B}\sigma^2 + \rho\sigma^2. \tag{8.2b}$$

The first equation (8.2a) tells us that the mean is unaltered by averaging a number of identically distributed random variables. Furthermore, the second equation (8.2b) tells us that the variance is reduced by averaging if the correlation $\rho < 1$. The first term in the variance expression (8.2b) can be made arbitrarily small by increasing $B$, whereas the second term is only determined by the correlation $\rho$ and variance $\sigma^2$.

To make a connection between bagging and (8.2), consider the predictions $\widetilde{y}^{(b)}(\mathbf{x}_\star)$ from the base models as random variables. All base models, and hence their predictions, originate from the same data $\mathcal{T}$ (via the bootstrap), and $\widetilde{y}^{(b)}(\mathbf{x}_\star)$ are therefore identically distributed but correlated. By averaging the predictions, we decrease the variance, according to (8.2b). If we choose $B$ large enough, the achieved variance reduction will be limited by the correlation $\rho$. Experience has shown that $\rho$

[1] That is, $\frac{1}{B(B-1)}\sum_{b\neq c}\mathbb{E}[(z_b-\mu)(z_c-\mu)] = \rho\sigma^2$.


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

is often small enough such that the computational complexity of bagging (compared to only using the base model itself) pays off well in terms of decreased variance. To summarise, by averaging the identically distributed predictions from several base models as in (8.1), each with a low bias, the *bias remains low*[2] (according to (8.2a)), and *the variance is reduced* (according to (8.2b)).

At first glance, one might think that a bagging model (8.1) becomes more 'complex' as the number of ensemble members $B$ increases, and that we therefore run a risk of overfitting if we use many ensemble members $B$. However, there is nothing in (8.2) which indicates any such problem (bias remains low; variance decreases), and we confirm this by Example 8.3.

**Example 8.3 Bagging for regression (continued)**

We consider the problem from Example 8.1 again and explore how the number of base models $B$ affects the result. We measure the squared error between the 'true' function value at $x_\star$ and the predicted $\widehat{y}^{\text{bag}}(x_\star)$ when using different values of $B$. (Because of the bootstrap, there is a certain amount of randomness in the bagging algorithm itself. To avoid that 'noise', we average the result over multiple runs of the bagging algorithm.)

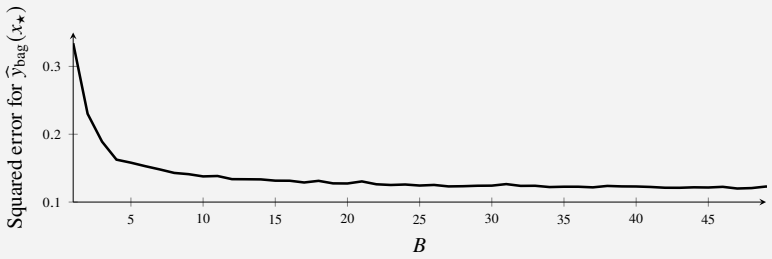



**Fig. 8.4**

What we see in Figure 8.4 is that the squared error eventually reaches a plateau as $B \to \infty$. Had there been an overfitting issue with $B \to \infty$, the squared error would have started to increase again for some large value of $B$.

Despite the fact that the number of parameters in the model increases as $B$ increases, the lack of overfitting as $B \to \infty$ according to Example 8.3 is the expected (and intended) behaviour. It is important to understand that from the construction of bagging, *more ensemble members does not make the resulting model more flexible* but only reduces the variance. This can be understood by noting that the addition of an ensemble member to the bagging model is *not* done in order to obtain a better fit to the training data. On the contrary, if each ensemble member overfits to its own perturbed version of the training data, the averaging across the ensemble will result in

[2] Strictly speaking, (8.2a) implies that the bias is identical for a single ensemble member and the ensemble average. The use of the bootstrap might, however, affect the bias, in that a base model trained on the original data might have a smaller bias than a base model trained on a bootstrapped version of the training data. Most often, this is not an issue in practice.

a smoothing effect which typically results in a larger training error (compared to the individual ensemble members' training errors) but also better generalisation. We can also understand this by considering the limiting behavior as $B \to \infty$. By the law of large numbers and the fact that the ensemble members are identically distributed, the bagging model becomes

$$\widehat{y}_{\text{bag}}(x_\star) = \frac{1}{B}\sum_{b=1}^{B} \widetilde{y}^{(b)}(x_\star) \overset{B\to\infty}{\longrightarrow} \mathbb{E}\big[\widetilde{y}^{(b)}(x_\star) \,|\, \mathcal{T}\big], \tag{8.3}$$

where the expectation is with respect to the randomness of the bootstrapping algorithm. As $B$ increases, we expect the bagging model to converge to the hypothetical (limited flexibility) model on the right hand side. With this in mind, in practice the choice of $B$ is mainly guided by computational constraints. The larger $B$ is, the better, but increasing $B$ when there is no further reduction in test error is computationally wasteful.

**Be aware!** *Bagging can still suffer from overfitting since each individual ensemble member can overfit. The only claim we have made is that the overfitting is not caused by (or made worse by) using too many ensemble members and, conceptually, there is no problem with taking $B \to \infty$.*

### Out-of-Bag Error Estimation

When using bagging (or random forests, which we discuss below), it turns out that there is a way to estimate the expected new data error $E_{\text{new}}$ *without* using cross-validation. The first observation we have to make is that not all data points from the original dataset $\mathcal{T}$ will have been used for training all ensemble members. It can be shown that with the bootstrap, on average only 63% of the original training data points in $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^n$ will be present in a bootstrapped training dataset $\widetilde{\mathcal{T}} = \{\widetilde{\mathbf{x}}_i, \widetilde{y}_i\}_{i=1}^n$. Roughly speaking, this means that for any given $\{\mathbf{x}_i, y_i\}$ in $\mathcal{T}$, about one third of the ensemble members will not have seen that data point during training. We refer to these (roughly $B/3$) ensemble members as being out-of-bag for data point $i$, and we let them form their own ensemble: the $i$th out-of-bag-ensemble. Note that the out-of-bag-ensemble is different for each data point $\{\mathbf{x}_i, y_i\}$.

The next key insight is that for the out-of-bag-ensemble $i$, the data point $\{\mathbf{x}_i, y_i\}$ can act as a test data point since it has not yet been seen by any of its ensemble members. By computing the (e.g., squared or misclassification) error when the out-of-bag-ensemble $i$ predicts $\{\mathbf{x}_i, y_i\}$, we thus get an estimate of $E_{\text{new}}$ for this out-of-bag-ensemble, which we denote $E_{\text{OOB}}^{(i)}$. Since $E_{\text{OOB}}^{(i)}$ is based on only one data point, it will be a fairly poor estimate of $E_{\text{new}}$. However, if we repeat this for all data points $\{\mathbf{x}_i, y_i\}$ in the training data $\mathcal{T}$ and average $E_{\text{OOB}} = \frac{1}{n}\sum_{i=1}^n E_{\text{OOB}}^{(i)}$, we get a better estimate of $E_{\text{new}}$. Indeed, $E_{\text{OOB}}$ will be an estimate of $E_{\text{new}}$ for an ensemble with only $B/3$ (and not $B$) members, but as we have seen (Example 8.3), the performance of bagging plateaus after a certain number of ensemble members. Hence, if $B$ is large enough so that ensembles with $B$


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

and $B/3$ members perform similarly, $E_{\text{OOB}}$ provides an estimate of $E_{\text{new}}$ which can be at least as good as the estimate $E_{k\text{-fold}}$ from $k$-fold cross-validation. Most importantly, however, $E_{\text{OOB}}$ comes almost for free in bagging, whereas $E_{k\text{-fold}}$ requires much more computation when re-training $k$ times.

## 8.2 Random Forests

In bagging, we reduce the variance by averaging over an ensemble of models. Unfortunately, the variance reduction is limited by the correlation between the individual ensemble members (compare this with the dependence on the average correlation $\rho$ in (8.2b)). However, using a simple trick, it is possible to reduce the correlation beyond what is achieved by the bootstrap, resulting in a method referred to as *random forests*.

While bagging is a general technique that in principle can be used to reduce the variance of any base model, random forests assume that these base models are classification or regression trees. The idea is to inject additional randomness when constructing each tree, in order to further reduce the correlation among the base models. At first this might seem like a silly idea: randomly perturbing the training of a model should intuitively degrade its performance. There is a rationale for this perturbation, however, which we will discuss below, but first we present the details of the algorithm.

Let $\widetilde{\mathcal{T}}^{(b)}$ be one of the $B$ bootstrapped datasets in bagging. To train a classification or regression tree on this data, we proceed as usual (see Section 2.3) but with one difference. Throughout the training, whenever we are about to split a node, we do not consider all possible input variables $x_1, \ldots, x_p$ as splitting variables. Instead, we pick a random subset consisting of $q \leq p$ inputs and only consider these $q$ variables as possible splitting variables. At the next splitting point, we draw a new random subset of $q$ inputs to use as possible splitting variables, and so on. Naturally, this random subset selection is done independently for each of the $B$ ensemble members, so that we (with high probability) end up using different subsets for the different trees. This additional random constraint when training is what turns bagging into *random forests*. This will cause the $B$ trees to be less correlated, and averaging their predictions can therefore result in a larger variance reduction compared to bagging. It should be noted, however, that this random perturbation of the training procedure will increase the variance[3] of each *individual tree*. In the notation of equation (8.2b), the random forest decreases $\rho$ (good) but increases $\sigma^2$ (bad) compared to bagging. Experience has, however, shown that the reduction in correlation is the

[3] And possibly also the bias, in a similar manner as the bootstrap might increase the bias; see footnote 2, page 197.

dominant effect, so that the averaged prediction variance is often reduced. We illustrate this in Example 8.4 below.

To understand why it can be a good idea to only consider a subset of inputs as splitting variables, recall that tree-building is based on recursive binary splitting, which is a greedy algorithm. This means that the algorithm can make choices early on that appear to be good but which nevertheless turn out to be suboptimal further down the splitting procedure. For instance, consider the case when there is one dominant input variable. If we construct an ensemble of trees using plain bagging, it is then very likely that all of the ensemble members will pick this dominant variable as the first splitting variable, making all trees identical (that is, perfectly correlated) after the first split. If we instead apply a random forest, some of the ensemble members will not even have access to this dominant variable at the first split, since it most likely will not be present in the random subset of $q$ inputs selected at the first split for some of the ensemble members. This will force those members to split according to some other variable. While there is no reason for why this would improve the performance of the individual tree, it *could* prove to be useful further down the splitting process, and since we average over many ensemble members, the overall performance could therefore be improved.

**Example 8.4 Random forests and bagging for a binary classification problem**

Consider the binary classification with $p = 2$ using the data in Figure 8.5. The different classes are blue and red. The $n = 200$ input values were randomly sampled from $[0, 2] \times [0, 2]$ and labelled red with probability 0.98 if above the dotted line, and vice versa. We use two different classifiers: bagging with classification trees (which is equivalent to a random forest with $q = p = 2$) and a random forest with $q = 1$, each with $B = 9$ ensemble members. In Figure 8.5, we plot the decision boundary for each ensemble member as well as the majority-voted final decision boundary.

The most apparent difference is the higher individual variation of the random forest ensemble members compared to bagging. Roughly half of the random forest ensemble members have been forced to make the first split along the horizontal axis, which has led to an increased variance and a decreased correlation compared to bagging, where all ensemble members make the first split along the vertical axis.

Since it is hard to visually compare the final decision boundaries for bagging and random forest (top right), we also compute $E_{\text{new}}$ for different numbers of ensemble members $B$. Since the learning itself has a certain amount of randomness, we average over multiple learned models to avoid being confused by that random effect. Indeed, we see that the random forest performs better than bagging, except for very small $B$, and we conclude that the positive effect of the reduced correlation between the ensemble members outweighs the negative effect of additional variance. The poor performance of random forest with only one ensemble member is expected, since this single model has higher variance, and no averaging is taking place when $B = 1$.

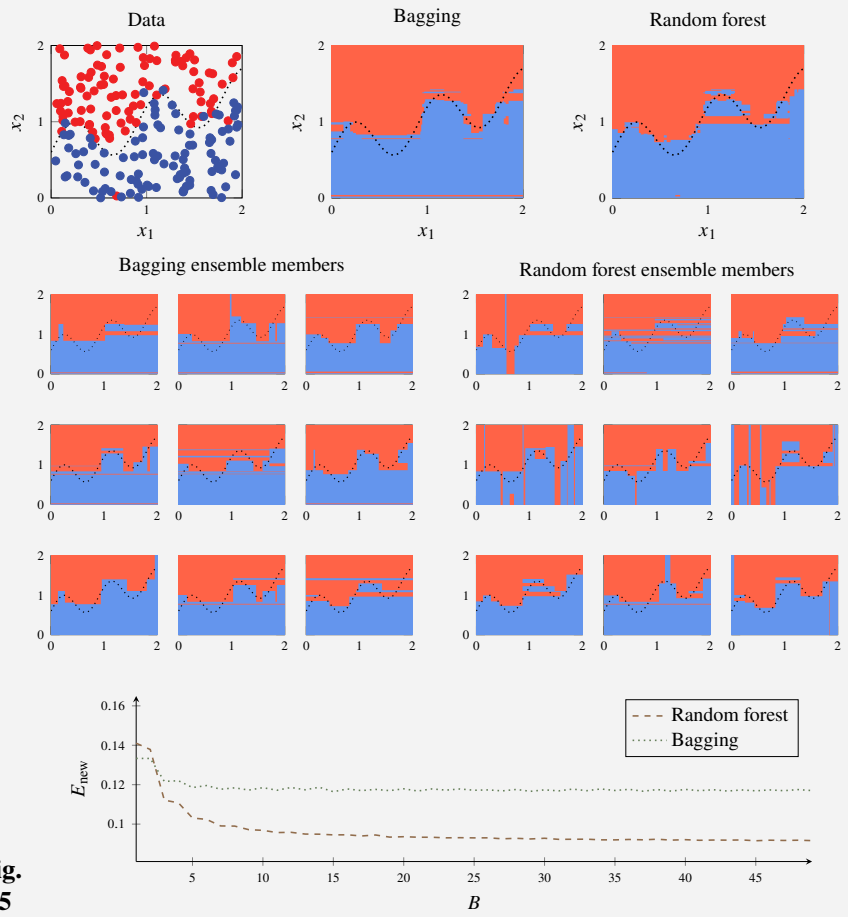



**Fig. 8.5**

Since the random forest is a bagging method, the tools and properties from Section 8.1 apply also to random forests, such as out-of-bag error estimation. As for bagging, taking $B \to \infty$ does not lead to overfitting in random forests. Hence, the only reason to choose $B$ small is to reduce the computational cost. Compared to using a single tree, a random forest requires approximately $B$ times as much computation. Since all trees are identically distributed, it is, however, possible to parallelise the implementation of random forest learning.

The choice of $q$ is a tuning parameter, where for $q = p$, we recover the basic bagging method described previously. As a rule-of-thumb, we can set $q = \sqrt{p}$ for classification problems and $q = p/3$ for regression problems (values rounded down to closest integer). A more systematic way of selecting $q$ is to use out-of-bag error estimation or cross-validation and select $q$ such that $E_{\text{OOB}}$ or $E_{k\text{-fold}}$ is minimised.

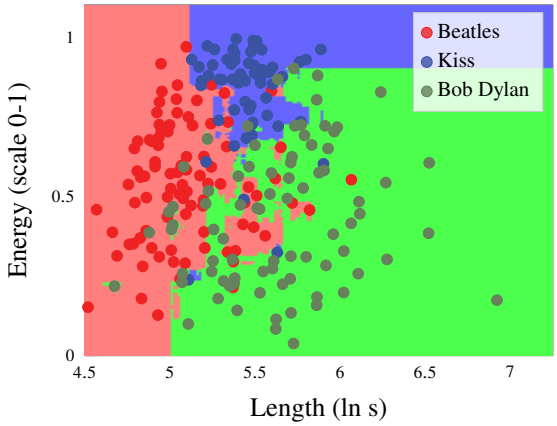



**Figure 8.6:** Random forest applied to the music classification problem from Example 2.1. This figure can be compared to Figure 2.11a, which is the decision boundary from a single tree.

We finally apply random forests to the music classification problem from Example 2.1 in Figure 8.6.

## 8.3 Boosting and AdaBoost

As we have seen above, bagging is an ensemble method for reducing the variance in high-variance base models. Boosting is another ensemble method, which is primarily used for reducing bias in high-bias base models. A typical example of a simple (or, in other words, weak) high-bias model is a classification tree of depth one (sometimes called a classification stump). Boosting is built on the idea that even a weak high-bias model can often capture *some* of the relationship between the inputs and the output. Thus, by training multiple weak models, each describing part of the input–output relationship, it might be possible to combine the predictions of these models into an overall better prediction. Hence, the intention is to *reduce the bias* by turning an ensemble of weak models into one strong model.

Boosting shares some similarities with bagging. They are both ensemble methods, in the sense that they are based on combining the predictions from multiple models (an ensemble). Both bagging and boosting can also be viewed as meta-algorithms, in the sense that they can be used to combine essentially any regression or classification algorithm – they are algorithms built on top of other algorithms. However, there are also important differences between boosting and bagging which we will discuss below.

The main difference is how the base models are trained. In bagging, we train $B$ identically distributed models in parallel. Boosting, on the other hand, uses a *sequential* construction of the ensemble members. Informally, this is done in such a way that each model tries to correct the mistakes made by the previous one. This is accomplished by modifying the training dataset at each iteration in order to put more


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September* 1, 2026, `http://smlbook.org`

emphasis on the data points for which the model (so far) has performed poorly. The final prediction is obtained from a weighted average or a weighted majority vote among the models. We look at the simple Example 8.5 to illustrate this idea.

### Example 8.5 Boosting illustration

We consider a binary classification problem with a two-dimensional input $\mathbf{x} = [x_1 \ x_2]$. The training data consists of $n = 10$ data points, 5 from each of the two classes. We use a decision stump, a classification tree of depth one, as a simple (weak) base classifier. A decision stump amounts to selecting one of the input variables, $x_1$ or $x_2$, and splitting the input space into two half spaces to minimise the training error. The first panel in Figure 8.7 shows the training data, illustrated by red and blue dots for the two classes. In the panel below that in Figure 8.7, the coloured regions show the decision boundary for a decision stump $\widehat{y}^{(1)}(\mathbf{x})$ trained on this data.

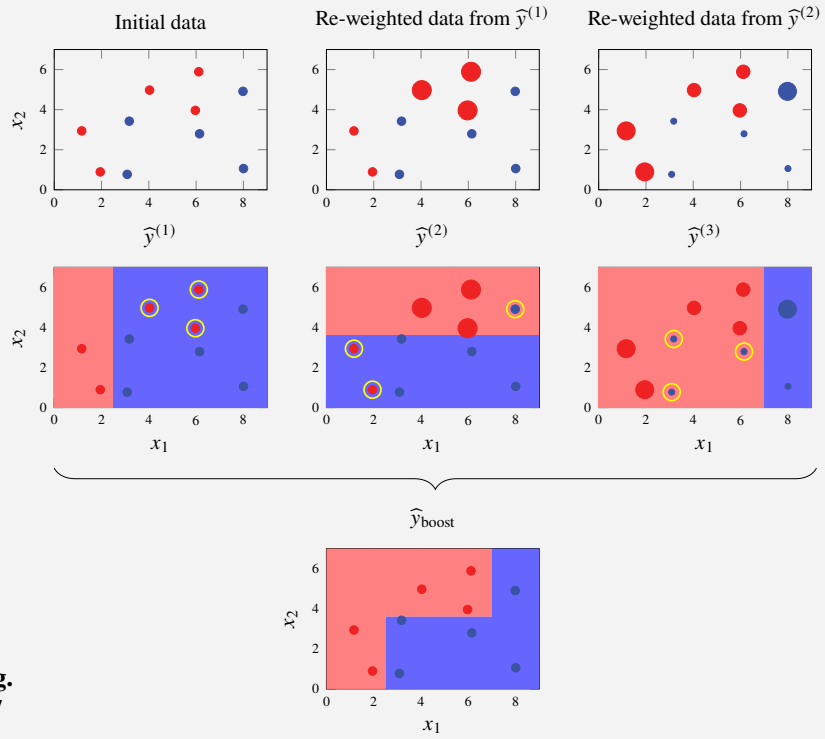



**Fig. 8.7**

The model $\widehat{y}^{(1)}(\mathbf{x})$ incorrectly classifies three data points (red dots falling in the blue region), which are circled in the figure. To improve the performance of the classifier, we want to find a model that can distinguish these three points from the blue class. To put emphasis on the three misclassified points when training the next decision stump, we assign *weights* $\{w_i^{(2)}\}_{i=1}^n$ to the data, which are shown in the upper middle panel in Figure 8.7 (larger radius = higher weight). The points correctly classified by $\widehat{y}^{(1)}(\mathbf{x})$ are down-weighted, whereas the three points misclassified by $\widehat{y}^{(1)}(\mathbf{x})$ are up-weighted. We train another decision stump, $\widehat{y}^{(2)}(\mathbf{x})$, on the weighted data. The classifier $\widehat{y}^{(2)}(\mathbf{x})$ is found by minimising the *weighted* misclassification error, $\frac{1}{n}\sum_{i=1}^n w_i^{(2)} \mathbb{I}\{\widehat{y}^{(2)}(\mathbf{x}_i) \neq y_i\}$, resulting in the decision boundary shown in the

lower middle panel. This procedure is repeated for a third and final iteration: we update the weights based on the hits and misses of $\widehat{y}^{(2)}(\mathbf{x})$ and train a third decision stump $\widehat{y}^{(3)}(\mathbf{x})$ shown in the lower right panel in Figure 8.7.

The final classifier $\widehat{y}_{\text{boost}}(\mathbf{x})$, in the bottom panel in Figure 8.7, is then obtained as a weighted majority vote of the three decision stumps. Note that its decision boundary is non-linear, whereas the decision boundary for each ensemble member is linear. This illustrates the concept of turning an ensemble of three weak (high-bias) base models into a stronger (low-bias) model.

The example illustrates the idea of boosting, but there are still important details left to be specified in order to have a complete algorithm. Specifically, how to compute the weights of the training data points at each iteration and how to combine the ensemble members to get the final model. Next, we will have a look at the AdaBoost algorithm, which is one approach for filling in these missing details. AdaBoost was the first successful implementation of the boosting idea, so it is interesting in its own right but also because it is simple enough to allow for closed form derivations. However, there are also more modern approaches to boosting, so after discussing AdaBoost, we will introduce the more general framework of gradient boosting.

Throughout this section, we will restrict our attention to binary classification, but boosting is also applicable to multiclass classification and regression problems.

### AdaBoost

What we have discussed so far is a general idea, but there are still a few technical design choices left. Let us now derive an actual boosting method, the AdaBoost (Adaptive Boosting) algorithm for binary classification. AdaBoost was the first successful practical implementation of the boosting idea and led the way for its popularity.

As we outlined in Example 8.5, boosting attempts to construct a sequence of $B$ (weak) binary classifiers $\widehat{y}^{(1)}(\mathbf{x}), \widehat{y}^{(2)}(\mathbf{x}), \ldots, \widehat{y}^{(B)}(\mathbf{x})$. In this procedure, we will only consider the final 'hard' prediction $\widehat{y}(\mathbf{x})$ from the base models and not their predicted class probabilities $g(\mathbf{x})$. Any classification model can, in principle, be used as base classifier – shallow classification trees are common in practice. The individual predictions of the $B$ ensemble members are then combined into a final prediction. Unlike bagging, all ensemble members are not treated equally. Instead, we assign some positive coefficients $\{\alpha^{(b)}\}_{b=1}^{B}$ and construct the boosted classifier using a *weighted* majority vote:

$$\widehat{y}_{\text{boost}}^{(B)}(\mathbf{x}) = \operatorname{sign}\left\{\sum_{b=1}^{B} \alpha^{(b)} \widehat{y}^{(b)}(\mathbf{x})\right\}. \tag{8.4}$$

Each ensemble member votes either $-1$ or $+1$, and the output from the boosted classifier is $+1$ if the weighted sum of the individual votes is positive and $-1$ if it is negative. The coefficient $\alpha^{(b)}$ can be thought of as a degree of confidence in the predictions made by the $b$th ensemble member.


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

The construction of the AdaBoost classifier in (8.4) follows the general form of a binary classifier from (5.12). That is, we obtain the class prediction by thresholding a real-valued function $f(\mathbf{x})$ at zero, where in this case the function is given by the weighted sum of predictions made by all the ensemble members. In AdaBoost, the ensemble members and their coefficients $\alpha^{(b)}$ are trained greedily by minimising the *exponential loss* of the boosted classifier at each iteration. Recall from (5.15) that the exponential loss is given by

$$L(y \cdot f(\mathbf{x})) = \exp(-y \cdot f(\mathbf{x})), \tag{8.5}$$

where $y \cdot f(\mathbf{x})$ is the *margin* of the classifier. The ensemble members are added one at a time, and when member $b$ is added, this is done to minimise the exponential loss (8.5) of the entire ensemble constructed so far (that is, the boosted classifier consisting of the first $b$ members). The main reason for choosing the exponential loss, and not one of the other loss functions discussed in Section 5.2, is that it results in convenient closed form expressions (much like the squared error loss in linear regression), as we will see when deriving the AdaBoost procedure below.

Let us write the boosted classifier after $b$ iterations as $\widehat{y}^{(b)}_{\text{boost}}(\mathbf{x}) = \text{sign}\{f^{(b)}(\mathbf{x})\}$, where $f^{(b)}(\mathbf{x}) = \sum_{j=1}^{b} \alpha^{(j)} \widehat{y}^{(j)}(\mathbf{x})$. We can express $f^{(b)}(\mathbf{x})$ iteratively as

$$f^{(b)}(\mathbf{x}) = f^{(b-1)}(\mathbf{x}) + \alpha^{(b)} \widehat{y}^{(b)}(\mathbf{x}), \tag{8.6}$$

initialised with $f^{0}(\mathbf{x}) = 0$. The ensemble members (as well as the coefficients $\alpha^{(b)[j]}$) are constructed sequentially, meaning that at iteration $b$ of the procedure, the function $f^{(b-1)}(\mathbf{x})$ is known and fixed. This is what makes this construction 'greedy'. Consequently, what remains to be learned at iteration $b$ is the ensemble member $\widehat{y}^{(b)}(\mathbf{x})$ and its coefficient $\alpha^{(b)}$. We do this by minimising the exponential loss of the training data,

$$\begin{aligned}
(\alpha^{(b)}, \widehat{y}^{(b)}) &= \arg\min_{(\alpha, \widehat{y})} \sum_{i=1}^{n} L(y_i \cdot f^{(b)}(\mathbf{x}_i)) && \text{(8.7a)} \\
&= \arg\min_{(\alpha, \widehat{y})} \sum_{i=1}^{n} \exp\left(-y_i \left(f^{(b-1)}(\mathbf{x}_i) + \alpha \widehat{y}(\mathbf{x}_i)\right)\right) && \text{(8.7b)} \\
&= \arg\min_{(\alpha, \widehat{y})} \sum_{i=1}^{n} \underbrace{\exp\left(-y_i f^{(b-1)}(\mathbf{x}_i)\right)}_{= w_i^{(b)}} \exp\left(-y_i \alpha \widehat{y}(\mathbf{x}_i)\right), && \text{(8.7c)}
\end{aligned}$$

where for the first equality, we have used the definition of the exponential loss function (8.5) and the sequential structure of the boosted classifier (8.6). The last equality is where the convenience of the exponential loss appears, namely the fact that $\exp(a + b) = \exp(a)\exp(b)$. This allows us to define the quantities

$$w_i^{(b)} \overset{\text{def}}{=} \exp\big(-y_i f^{(b-1)}(\mathbf{x}_i)\big), \tag{8.8}$$

which can be interpreted as *weights* for the individual data points in the training dataset. Note that the weights $w_i^{(b)}$ are independent of $\alpha$ and $\widehat{y}$. That is, when learning $\widehat{y}^{(b)}(\mathbf{x})$ and its coefficient $\alpha^{(b)}$ by solving (8.7c), we can regard $\{w_i^{(b)}\}_{i=1}^n$ as constants.

To solve (8.7), we start by rewriting the objective function as

$$\sum_{i=1}^{n} w_i^{(b)} \exp\left(-y_i \alpha \widehat{y}(\mathbf{x}_i)\right) = e^{-\alpha} \underbrace{\sum_{i=1}^{n} w_i^{(b)} \mathbb{I}\{y_i = \widehat{y}(\mathbf{x}_i)\}}_{=W_c} + e^{\alpha} \underbrace{\sum_{i=1}^{n} w_i^{(b)} \mathbb{I}\{y_i \neq \widehat{y}(\mathbf{x}_i)\}}_{=W_e}, \tag{8.9}$$

where we have used the indicator function to split the sum into two parts: the first ranging over all training data points correctly classified by $\widehat{y}$ and the second ranging over all points misclassified by $\widehat{y}$. (Remember that $\widehat{y}$ is the ensemble member we are to learn at this step.) Furthermore, for notational simplicity, we define $W_c$ and $W_e$ as the sums of weights of correctly classified and erroneously classified data points, respectively. Furthermore, let $W = W_c + W_e$ be the total weight sum, $W = \sum_{i=1}^{n} w_i^{(b)}$.

Minimising (8.9) is done in two stages: first with respect to $\widehat{y}$ and then with respect to $\alpha$. This is possible since the minimising argument in $\widehat{y}$ turns out to be independent of the actual value of $\alpha > 0$, another convenient effect of using the exponential loss function. To see this, note that we can write the objective function (8.9) as

$$e^{-\alpha} W + (e^{\alpha} - e^{-\alpha}) W_e. \tag{8.10}$$

Since the total weight sum $W$ is independent of $\widehat{y}$, and since $e^{\alpha} - e^{-\alpha} > 0$ for any $\alpha > 0$, minimising this expression with respect to $\widehat{y}$ is equivalent to minimising $W_e$ with respect to $\widehat{y}$. That is,

$$\widehat{y}^{(b)} = \arg\min_{\widehat{y}} \sum_{i=1}^{n} w_i^{(b)} \mathbb{I}\{y_i \neq \widehat{y}(\mathbf{x}_i)\}. \tag{8.11}$$

In words, the $b$th ensemble member should be trained by minimising the *weighted misclassification loss*, where each data point $(\mathbf{x}_i, y_i)$ is assigned a weight $w_i^{(b)}$. The intuition for these weights is that, at iteration $b$, we should focus our attention on the data points previously misclassified in order to 'correct the mistakes' made by the ensemble of the first $b-1$ classifiers.

> ***Time to reflect 8.2*** *In AdaBoost, we use the exponential loss for training the boosting ensemble. How come we end up training the individual ensemble members using a weighted misclassification loss (and not the unweighted exponential loss)?*

How the problem (8.11) is solved in practice depends on the choice of base classifier that we use, that is, on the specific restrictions that we put on the function $\widehat{y}$ (for


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

example a shallow classification tree). However, solving (8.11) is almost our standard classification problem, except for the weights $w_i^{(b)}$. Training the ensemble member $b$ on a *weighted* classification problem is, for most base classifiers, straightforward. Since most classifiers are trained by minimising some cost function, this simply boils down to weighting the individual terms of the cost function and solving that slightly modified problem instead.

Once the $b$th ensemble member, $\widehat{y}^{(b)}(\mathbf{x})$, has been trained for solving the weighted classification problem (8.11), it remains to learn its coefficient $\alpha^{(b)}$. This is done by solving (8.7), which amounts to minimising (8.10) once $\widehat{y}$ has been trained. By differentiating (8.10) with respect to $\alpha$ and setting the derivative to zero, we get the equation

$$-\alpha e^{-\alpha} W + \alpha(e^{\alpha} + e^{-\alpha}) W_e = 0 \Leftrightarrow W = \left(e^{2\alpha} + 1\right) W_e \Leftrightarrow \alpha = \frac{1}{2} \ln\left(\frac{W}{W_e} - 1\right).$$

Thus, by defining

$$E_{\text{train}}^{(b)} \stackrel{\text{def}}{=} \frac{W_e}{W} = \sum_{i=1}^{n} \frac{w_i^{(b)}}{\sum_{j=1}^{n} w_j^{(b)}} \mathbb{I}\{y_i \neq \widehat{y}^{(b)}(\mathbf{x}_i)\} \tag{8.12}$$

to be the weighted misclassification error for the $b$th classifier, we can express the optimal value for its coefficient as

$$\alpha^{(b)} = \frac{1}{2} \ln\left(\frac{1 - E_{\text{train}}^{(b)}}{E_{\text{train}}^{(b)}}\right). \tag{8.13}$$

The fact that $\alpha^{(b)}$ depends on the training error of the $b$th ensemble member is natural since, as mentioned above, we can interpret $\alpha^{(b)}$ as the confidence in this member's predictions. This completes the derivation of the AdaBoost algorithm, which is summarised in Method 8.2. In the algorithm we exploit the fact that the weights (8.8) can be computed recursively by using the expression (8.6) in line 6 in the learning section. Furthermore, we have added an explicit weight normalisation (line 7), which is convenient in practice and which does not affect the derivation of the method above.

The derivation of AdaBoost assumes that all coefficients $\{\alpha^{(b)}\}_{b=1}^{(B)}$ are positive. To see that this is indeed the case when the coefficients are computed according to (8.13),

note that the function $\ln((1-x)/x)$ is positive for any $0 < x < 0.5$. Thus, $\alpha^{(b)}$ will be positive as long as the weighted training error for the $b$th classifier, $E_{\text{train}}^{(b)}$, is less than 0.5. That is, the classifier just has to be slightly better than a coin flip, which is always the case in practice (note that $E_{\text{train}}^{(b)}$ is the *training* error). (Indeed, if $E_{\text{train}}^{(b)} > 0.5$, then we could simply flip the sign of all predictions made by $\widehat{y}^{(b)}(\mathbf{x})$ to reduce the error below 0.5.)

**Learn** an AdaBoost classifier

**Data:** Training data $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^n$
**Result:** $B$ weak classifiers

Assign weights $w_i^{(1)} = 1/n$ to all data points.
**for** $b = 1, \ldots, B$ **do**
  Train a weak classifier $\widehat{y}^{(b)}(\mathbf{x})$ on the weighted training data $\{(\mathbf{x}_i, y_i, w_i^{(b)})\}_{i=1}^n$.
  Compute $E_{\text{train}}^{(b)} = \sum_{i=1}^n w_i^{(b)} \mathbb{I}\{y_i \neq \widehat{y}^{(b)}(\mathbf{x}_i)\}$.
  Compute $\alpha^{(b)} = 0.5 \ln((1 - E_{\text{train}}^{(b)})/E_{\text{train}}^{(b)})$.
  Compute $w_i^{(b+1)} = w_i^{(b)} \exp(-\alpha^{(b)} y_i \widehat{y}^{(b)}(\mathbf{x}_i))$, $i = 1, \ldots, n$.
  Set $w_i^{(b+1)} \leftarrow w_i^{(b+1)} / \sum_{j=1}^n w_j^{(b+1)}$, for $i = 1, \ldots, n$.
**end**

**Predict** with the AdaBoost classifier

**Data:** $B$ weak classifiers with confidence values $\{\widehat{y}^{(b)}(\mathbf{x}), \alpha^{(b)}\}_{b=1}^B$ and test input $\mathbf{x}_\star$
**Result:** Prediction $\widehat{y}_{\text{boost}}^{(B)}(\mathbf{x}_\star)$

Output $\widehat{y}_{\text{boost}}^{(B)}(\mathbf{x}_\star) = \text{sign}\left\{\sum_{b=1}^B \alpha^{(b)} \widehat{y}^{(b)}(\mathbf{x}_\star)\right\}$.

**Method 8.2:** AdaBoost

### Example 8.6 AdaBoost and bagging for a binary classification example

Consider the same binary classification problem as in Example 8.4. We now compare how AdaBoost and bagging perform on this problem, when using trees of depth one (decision stumps) and three. It should be noted that this comparison is made to illustrate the difference between the methods. In practice, we would typically not use bagging with such shallow trees.

The decision boundaries for each method with $B = 1, 5, 20$, and 100 ensemble members are shown in Figure 8.8. Despite using quite weak ensemble members (a shallow tree has high bias), AdaBoost adapts quite well to the data. This is in contrast to bagging, where the decision boundary does not become much more flexible despite using many ensemble members. In other words, AdaBoost reduces the bias of the base model, whereas bagging only has minor effect on the bias.


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

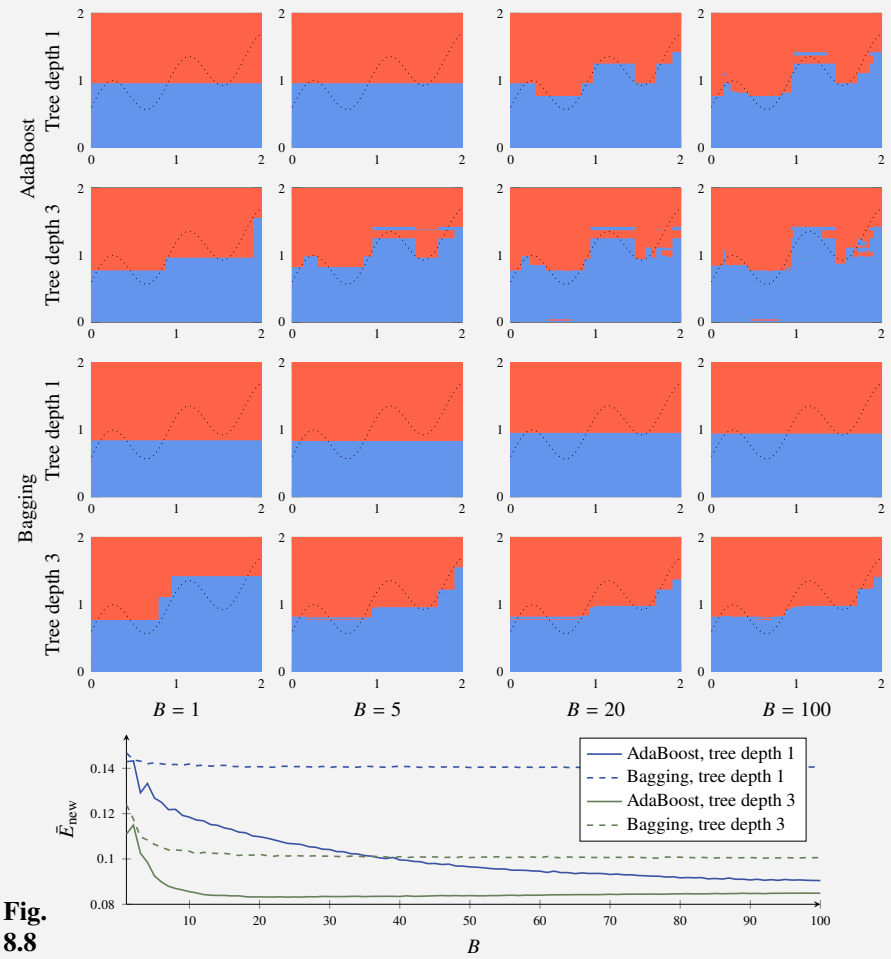



**Fig. 8.8**

We also numerically compute $\bar{E}_{\text{new}}$ for this problem, as a function of $B$, which is shown at the bottom of Figure 8.8. Remember that $\bar{E}_{\text{new}}$ depends on both the bias and the variance. As discussed, the main effect of bagging is variance reduction, but that does not help much since the base model is already quite low-variance (but high-bias). Boosting, on the other hand, reduces bias, which has a much bigger effect in this case. Furthermore, bagging does not overfit as $B \to \infty$, but that is *not* the case for boosting! We can indeed see that for trees of depth 3, the smallest $\bar{E}_{\text{new}}$ is obtained for $B \approx 25$, and there is actually a slight increase in $\bar{E}_{\text{new}}$ for larger values of $B$. Hence, AdaBoost with depth-3 trees suffers from a (minor) overfit as $B \gtrsim 25$ in this problem.

## Design Choices for AdaBoost

AdaBoost, and in fact any boosting algorithm, has two important design choices: *(i)* which base classifier to use and *(ii)* how many iterations $B$ to run the boosting algorithm for. As previously pointed out, we can use essentially any classification method as the base classifier. However, the most common choice in practice is to use a shallow

classification tree, or even a decision stump (a tree of depth one; see Example 8.5). This choice is guided by the fact that boosting reduces bias efficiently and can thereby learn good models despite using a very weak (high-bias) base model. Since shallow trees can be trained quickly, they are a good default choice. Practical experience suggests that trees with a handful of terminal nodes may work well as base models, but trees of depth one (only $M = 2$ terminal nodes in binary classification) are perhaps even more commonly used. In fact, using deep classification trees (high-variance models) as base classifiers typically deteriorates performance.

The base models are trained sequentially in boosting: each iteration introduces a new base model aiming at reducing the errors made by the current model. As a consequence, the boosting model becomes more and more flexible as the number of iterations $B$ increases, and using too many base models can result in overfitting (in contrast to bagging, where increased $B$ cannot lead to overfit). It has been observed in practice, however, that this overfitting often occurs slowly, and the performance tends to be rather insensitive to the choice of $B$. Nevertheless, it is a good practice to select $B$ in some systematic way, for instance using early stopping during training. Another unfortunate aspect of the sequential nature of boosting is that it is not possible to parallelise the training.

In the method discussed above, we have assumed that each base classifier outputs a class prediction, $\widehat{y}^{(b)}(\mathbf{x}) \in \{-1, 1\}$. However, many classification models output $g(\mathbf{x})$, which is an estimate of the class probability $p(y = 1 \,|\, \mathbf{x})$. In AdaBoost it is possible to use the predicted probabilities $g(\mathbf{x})$ (instead of the binary prediction $\widehat{y}(\mathbf{x})$) when constructing the prediction; however, this is at the cost of a more complicated expression than (8.4). This extension of Method 8.2 is referred to as Real AdaBoost.

## 8.4 Gradient Boosting

It has been seen in practice that AdaBoost often performs well if there is little noise in the data. However, as the data becomes more noisy, either due to outliers (mislabelled data) or high uncertainty in the true input–output relationship, the performance of the method can deteriorate. This is not an artefact of the boosting idea but rather of the exponential loss function used in the construction of AdaBoost. As we discussed in Section 5.2, the exponential loss will heavily penalise large negative margins, making it sensitive to noise; see Figure 5.2. To mitigate this issue and construct more robust boosting algorithms, we can consider choosing some other (more robust) loss function. However, this will be at the expense of a more computationally involved training procedure.

To lay the foundation for more general boosting algorithms, we will start by presenting a slightly different view on boosting. In the discussion above, we have described boosting as *learning a sequence of weak classifiers, where each classifier tries to correct the mistakes made by the previous ones*. This is an intuitive interpretation, but from a mathematical perspective, a perhaps more useful interpretation is that boosting is as a way to train an additive model. The fundamental task of supervised

learning is to approximate some unknown function, mapping inputs to outputs, based on observed data. A very useful – and indeed common – way of constructing a flexible function approximator is by using an additive model of the form

$$f^{(B)}(\mathbf{x}) = \sum_{b=1}^{B} \alpha^{(b)} f^{(b)}(\mathbf{x}), \tag{8.14}$$

where $\alpha^{(b)}$ are real-valued coefficients and $f^{(b)}(\mathbf{x})$ are some 'basis functions'. For a regression problem, the function $f^{(B)}(\mathbf{x})$ can be used directly as the model's prediction. For a classification problem, it can be thresholded by a sign function to obtain a hard class prediction or transformed into a class probability by passing it through a logistic function.[4]

Comparing (8.14) with (8.4), it is clear that AdaBoost follows this additive form, where the weak learners (ensemble members) are the basis functions, and their confidence scores are the coefficients. However, we have in fact seen other examples of additive models before. To put boosting algorithms in a broader context, we provide a couple of examples:

If the basis functions $f^{(b)}(\mathbf{x})$ are fixed *a priori*, then the only learnable parameters are the coefficients $\alpha^{(b)}$. The model (8.14) is then nothing but a linear regression, or generalised linear model. For instance, in Chapter 3, we discussed polynomial regression, where the basis functions are defined as polynomial transformations of the input. In Chapter 9, we will discuss more systematic ways of constructing (fixed) basis functions for additive models.

A more flexible model can be obtained if we also allow the basis functions themselves to be learnable. This is also something that we have come across before. In Chapter 6, we introduced the neural network model, and writing out the expression for a two-layer regression network, it can be seen that it corresponds to an additive model.

> ***Time to reflect 8.3*** *If we write a two-layer regression neural network in the form of an additive model, then what do B, $\alpha^{(b)}$, and $f^{(b)}(\mathbf{x})$ correspond to?*

An important consequence of this interpretation of boosting as an additive model is that the individual ensemble members do not necessarily have to correspond to 'weak learners' for the specific problem under study. Put differently, for a classification problem, each ensemble member does not have to correspond to a classifier trained to solve (some modified version of) the original problem. What is important is just that the sum over all ensemble members in (8.14) results in a useful model! We will see an example of this below, when we discuss how regression trees can be used to solve classification problems in the context of gradient boosting. This is also the reason why we use $f$ instead of $\widehat{y}$ in the notation above. Even for a classification problem, the

[4] Similarly, we can use other link functions to turn the additive model $f^{(B)}(\mathbf{x})$ into a likelihood that is suitable for the properties of the data under study, akin to generalised linear models (see Section 3.4).

outputs from the ensemble members do not have to correspond to class predictions in general.

Instead, there are two properties that distinguish boosting from other additive models.

(i) The basis functions are learned from data and, specifically, each function (that is ensemble member) corresponds to a machine learning model itself – the base model of the boosting procedure.

(ii) The basis functions and their coefficients are learned sequentially. That is, we add one component to the sum in (8.14) at each iteration, and after $B$ iterations the learning algorithm terminates.

The goal when training an additive model is to select $\{\alpha^{(b)}, f^{(b)}(\mathbf{x})\}_{b=1}^{B}$ such that the final $f^{(B)}(\mathbf{x})$ minimises

$$J(f(\mathbf{X})) = \frac{1}{n}\sum_{i=1}^{n} L(y_i, f(\mathbf{x}_i)) \tag{8.15}$$

for some arbitrary loss function $L$; see Section 5.2. For instance, in a binary classification setting, choosing the logistic loss (or some other robust loss function) instead of the exponential loss will result in a model which is less sensitive to outliers. Here we define $f(\mathbf{X}) = [f(\mathbf{x}_1) \ \cdots \ f(\mathbf{x}_n)]^\mathsf{T}$ as the vector of function values obtained by evaluating the model $f(\mathbf{x})$ at the $n$ training data points. Since we do not have an explicit parametric form for $f(\mathbf{x})$, we consider $J$ to be a function of the model $f(\mathbf{x})$ itself.

A consequence of the first point in the list above – that the basis function themselves are generic machine learning models – is that the objective (8.15) will lack a closed form minimiser, and we thus need to resort to some approximate numerical solution. The sequential learning (second point above) can be viewed as one way of handling this, by using 'greedy' step-wise training. Connecting this back to the AdaBoost algorithm, using the exponential loss function is convenient since it results in tractable expressions for each step of this iterative training procedure. However, this is not strictly necessary. Indeed, by similar arguments as in numerical optimisation, we can improve the model at each iteration as long as we 'move in the right direction'. That is, at iteration $b$ we introduce a new ensemble member with the objective of *reducing the value* of the cost function (8.15) but without requiring that it is (greedily) minimised. This leads us in to the idea of gradient boosting.

Consider the $b$th iteration of the training procedure. As before, we can use the sequential nature of the method to write

$$f^{(b)}(\mathbf{x}) = f^{(b-1)}(\mathbf{x}) + \alpha^{(b)} f^{(b)}(\mathbf{x}), \tag{8.16}$$

and the goal is to select $\{\alpha^{(b)}, f^{(b)}(\mathbf{x})\}$ to reduce the value of the cost function (8.15). That is, we want to choose the $b$th ensemble member such that

$$J\left(f^{(b-1)}(\mathbf{X}) + \alpha^{(b)} f^{(b)}(\mathbf{X})\right) < J\left(f^{(b-1)}(\mathbf{X})\right). \tag{8.17}$$

Akin to the gradient descent algorithm (see Section 5.4), we do this by taking a step in the negative direction of the gradient of the cost function.

However, in the context of boosting, we do not assume a specific parametric form for the basis function but rather construct each ensemble member using a learnable base model, such as a tree. What, then, should we compute the gradient of the cost function with respect to? The idea behind gradient boosting, which allows us to address this question, is to take a non-parametric approach and represent the model $c(\mathbf{x})$ by the values it assigns to the $n$ training data points. That is, we compute the gradient of the cost function directly with respect to the (vector of) function values $f(\mathbf{X})$. This gives us an $n$-dimensional gradient vector

$$\nabla_c J(c^{(b-1)}(\mathbf{X})) \stackrel{\text{def}}{=} \begin{bmatrix} \frac{\partial J(f(\mathbf{X}))}{\partial f(\mathbf{x}_1)} \\ \vdots \\ \frac{\partial J(f(\mathbf{X}))}{\partial f(\mathbf{x}_n)} \end{bmatrix}_{|f(\mathbf{X})=f^{(b-1)}(\mathbf{X})} = \frac{1}{n} \begin{bmatrix} \frac{\partial L(y_1,f)}{\partial f}_{|f=f^{(b-1)}(\mathbf{x}_1)} \\ \vdots \\ \frac{\partial L(y_n,f)}{\partial f}_{|f=f^{(b-1)}(\mathbf{x}_n)} \end{bmatrix}, \quad (8.18)$$

where we have assumed that the loss function $L$ is differentiable. Hence, to satisfy (8.17), we should select $f^{(b)}(\mathbf{X}) = -\nabla_c J(c^{(b-1)}(\mathbf{X}))$ and then pick the coefficient $\alpha^{(b)}$ – which takes the role of the step length in the gradient descent analogy – in some suitable way, for instance by line search.

However, selecting the $b$th ensemble member $f^{(b)}(\mathbf{X})$ so that it *exactly* matches the negative gradient is typically not possible. The reason is that the ensemble members $f(\mathbf{x})$ are restricted to some specific functional form, for instance the set of functions that can be represented by a tree-based model of a certain depth. Neither would it be desirable in general, since exactly matching the gradient at all training data points could easily lead to overfitting. Indeed, as usual we are not primarily interested in finding a model that fits the training data as well as possible but rather one that generalises to new data. Restricting our attention to a class of functions that generalise beyond the observed training data is therefore a key requirement.

To proceed, we will therefore train the $b$th ensemble member $f^{(b)}(\mathbf{x})$ as a machine learning model, with the training objective that its predictions on the training data points (that is, the vector $f^{(b)}(\mathbf{X})$) are close to the negative gradient (8.18). Closeness can be evaluated by any suitable distance function, such as the squared distance. This corresponds to solving a *regression problem* where the target values are the elements of the gradient, and the loss function (for example squared loss) determines how we measure closeness. Note that, even when the actual problem under study is classification, the gradient values in (8.18) will be real-valued in general.

Having found the $b$th ensemble member, it remains to compute the coefficient $\alpha^{(b)}$. As pointed out above, this corresponds to the step size (or learning rate) in gradient

descent. In the simplest version of gradient descent, it is considered a tuning parameter left to the user. However, it can also be found by solving a line-search optimisation problem at each iteration. For gradient boosting, it is most often handled in the latter way. If multiplying the optimal $\alpha^{(b)}$ with a constant <1, a regularising effect is obtained which has proven useful in practice. We summarise gradient boosting in Method 8.3.

**Learn** a simple gradient boosting classifier

**Data:** Training data $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^n$, step size multiplier $\gamma < 1$
**Result:** A boosted classifier $f^{(B)}(\mathbf{x})$

Initialise (as a constant) $f^0(\mathbf{x}) \equiv \arg\min_c \sum_{i=1}^n L(y_i, c)$.
**for** $b = 1, \ldots, B$ **do**
  Compute the negative gradient of the loss function $d_i^{(b)} = -\frac{1}{n}\left[\frac{\partial L(y_i, c)}{\partial c}\right]_{c=f^{(b-1)}(\mathbf{x}_i)}$.
  Learn a *regression* model $f^{(b)}(\mathbf{x})$ from the input-output training data $\{\mathbf{x}_i, d_i^{(b)}\}_{i=1}^n$.
  Compute $\alpha^{(b)} = \arg\min_\alpha \sum_{i=1}^n L(y_i, f^{(b-1)}(\mathbf{x}_i) + \alpha f^{(b)}(\mathbf{x}_i))$.
  Update the boosted model $f^{(b)}(\mathbf{x}) = f^{(b-1)}(\mathbf{x}) + \gamma\alpha^{(b)} f^{(b)}(\mathbf{x})$.
**end**

**Predict** with the gradient boosting classifier

**Data:** $B$ weak classifiers and test input $\mathbf{x}_\star$
**Result:** Prediction $\widehat{y}_{\text{boost}}^{(B)}(\mathbf{x}_\star)$

Output $\widehat{y}_{\text{boost}}^{(B)}(\mathbf{x}) = \text{sign}\{f^{(B)}(\mathbf{x})\}$.

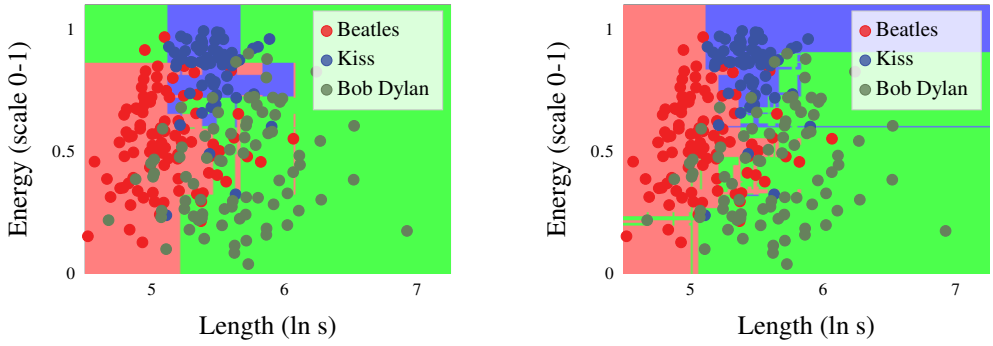
**Method 8.3:** A simple gradient boosting algorithm

When using trees as base models, optimising $\alpha^{(b)}$ can be done jointly with learning $f^{(b)}(\mathbf{x})$. Specifically, instead of first computing constant predictions for all terminal nodes of the tree (see Section 2.3) and then multiplying these by a constant $\alpha^{(b)}$, we can solve one separate line search problem for each terminal node in the tree directly.

***Time to reflect 8.4*** *We have described boosting algorithms as greedy step-wise training of an additive model. Based on this interpretation, another approach for training these models is by coordinate ascent (see Section 5.4). That is, instead of adding a new component at each iteration of the training algorithm and stopping after B iterations, we can fix the number of components and cycle through them (updating one component at a time) until convergence. What are the possible drawbacks and benefits of this alternative training approach?*

While presented for classification in Method 8.3, gradient boosting can also be


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September* 1, 2026, `http://smlbook.org`




**(a)** AdaBoost

**(b)** A gradient boosting classifier

**Figure 8.9:** Two boosting algorithms applied to the music classification problem from Example 2.1.

used for regression, with minor modifications. As mentioned above, gradient boosting requires a certain amount of smoothness in the loss function. A minimal requirement is that it is differentiable almost everywhere, so that it is possible to compute the gradient of the loss function. However, some implementations of gradient boosting require stronger conditions, such as second order differentiability. The logistic loss (see Section 5.2) is in this respect a 'safe choice' as it is infinitely differentiable and convex while still enjoying good statistical properties. As a consequence, the logistic loss is one of the most commonly used loss functions in practice.

We conclude this chapter by applying AdaBoost and gradient boosting to the music classification problem from Example 2.1; see Figure 8.9.

## 8.5 Further Reading

The general bagging idea was initially proposed by Breiman (1996), whereas the more specific random forest algorithm dates back to T. K. Ho (1995), who essentially proposed to limit the set of possible splitting variables for each tree. The idea to also use a bootstraped data set (that is, bagging) is due to Breiman (2001).

Boosting was popularised by the introduction of AdaBoost by Freund and Schapire (1996), who were also awarded the prestigious Gödel Prize in 2003 for their algorithm. Real AdaBoost was proposed by Friedman et al. (2000), and gradient boosting by Friedman (2001) and Mason et al. (1999). Efficient and widely used implementations of gradient boosting include the XGBoost package by Tianqi Chen and Guestrin (2016) and LightGBM by Ke et al. (2017).

# 9 Non-linear Input Transformations and Kernels

In this chapter, we will continue to develop the idea from Chapter 3 of creating new input features by using non-linear transformations $\boldsymbol{\phi}(\mathbf{x})$. It turns out that by the so-called *kernel trick*, we can have *infinitely* many such non-linear transformations, and we can extend our basic methods, such as linear regression and $k$-NN, into more versatile and flexible ones. When we also change the loss function of linear regression, we obtain support vector regression and its classification counterpart support vector classification, two powerful off-the-shelf machine learning methods. The concept of kernels is important also to the next chapter (10), where a Bayesian perspective of linear regression and kernels leads us to the Gaussian process model.

## 9.1 Creating Features by Non-linear Input Transformations

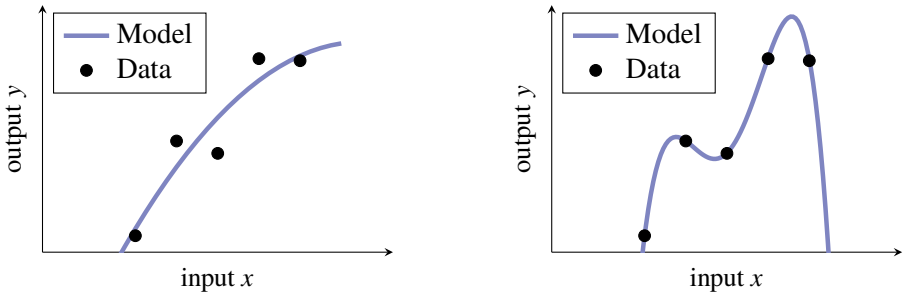
The reason for the word 'linear' in the name 'linear regression' is that the output is modelled as a *linear* combination of the inputs. However, we have not provided a clear definition of what an input is. Recall the car stopping distance problem in Example 2.2. If the speed is an input in that example, then could the kinetic energy – the square of the speed – not also be considered as another input? The answer is: yes, it can. We can in fact make use of arbitrary non-linear transformations of the 'original' input variables in any model, including linear regression. For example, if we only have a one-dimensional input $x$, the vanilla linear regression model (3.2) is

$$y = \theta_0 + \theta_1 x + \varepsilon. \tag{9.1}$$

Starting from this, we can extend the model with $x^2, x^3, \ldots, x^{d-1}$ as inputs ($d$ is a user-choice) and thus obtain a linear regression model which is a polynomial in $x$:

$$y = \theta_0 + \theta_1 x + \theta_2 x^2 + \cdots + \theta_{d-1} x^{d-1} + \varepsilon = \boldsymbol{\theta}^\mathsf{T} \boldsymbol{\phi}(x) + \varepsilon. \tag{9.2}$$

Since $x$ is known, we can directly compute $x^2, \ldots, x^{d-1}$. Note that this is still a linear regression model since *the parameters $\boldsymbol{\theta}$ appear in a linear fashion* with $\boldsymbol{\phi}(x) = [1 \;\; x \;\; x^2 \;\; \ldots \;\; x^{d-1}]^\mathsf{T}$ as a new input vector. We refer to a transformation of $\mathbf{x}$




**(a)** A linear regression model with a 2nd order polynomial, trained with squared error loss. The line is no longer straight (as in Figure 3.1), but this is merely an artifact of the plot: in a three-dimensional plot with each feature (here, $x$ and $x^2$) on a separate axis, it would still be an affine model.

**(b)** A linear regression model with a 4th order polynomial, trained with squared error loss. Note that a 4th order polynomial implies five unknown parameters, which roughly means that we can expect the learned model to fit five data points exactly, a typical case of overfitting.

**Figure 9.1:** A linear regression model with 2nd and 4th order polynomials in the input $x$, as in (9.2).

as a *feature*[1] and the vector of transformed inputs $\boldsymbol{\phi}(\mathbf{x})$, a vector of dimension $d \times 1$, as a *feature vector*. The parameters $\widehat{\boldsymbol{\theta}}$ are still learned in the same way, but we

$$\text{replace the original } \mathbf{X} = \underbrace{\begin{bmatrix} \mathbf{x}_1^\mathsf{T} \\ \mathbf{x}_2^\mathsf{T} \\ \vdots \\ \mathbf{x}_n^\mathsf{T} \end{bmatrix}}_{n \times p+1} \text{ with the transformed } \boldsymbol{\Phi}(\mathbf{X}) = \underbrace{\begin{bmatrix} \boldsymbol{\phi}(\mathbf{x}_1)^\mathsf{T} \\ \boldsymbol{\phi}(\mathbf{x}_2)^\mathsf{T} \\ \vdots \\ \boldsymbol{\phi}(\mathbf{x}_n)^\mathsf{T} \end{bmatrix}}_{n \times d}. \tag{9.3}$$

For linear regression, this means that we can learn the parameters by making the substitution (9.3) directly in the normal equations (3.13).

The idea of non-linear input transformations is not unique to linear regression, and any choice of non-linear transformation $\boldsymbol{\phi}(\cdot)$ can be used with any supervised machine learning method. The non-linear transformation is first applied to the input, like a pre-processing step, and the transformed input is thereafter used when training, evaluating, and using the model. We illustrated this for regression already in Example 3.5 in Chapter 3 and for classification in Example 9.1.

[1] The original input $\mathbf{x}$ is sometimes also referred to as a feature.


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

***Time to reflect 9.1*** *Figure 9.1 shows an example of two linear regression models with transformed (polynomial) inputs. When studying the figure, one may ask how a* linear *regression model can result in a* curved *line? Are linear regression models not restricted to linear (or affine) straight lines?*

**Example 9.1 Non-linear feature transformations for classification**

Consider the data for a binary classification problem in the left panel of Figure 9.2, with $\mathbf{x} = [x_1 \;\; x_2]^\mathsf{T}$ and with a blue and a red class. By just looking at the data, we can conclude that a linear classifier would not be able to perform well on this problem.

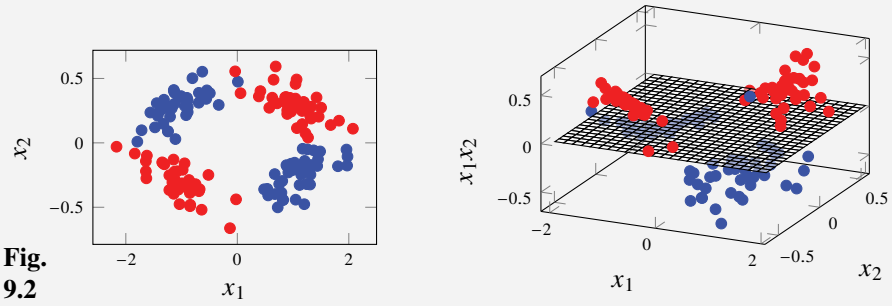



**Fig. 9.2**

However, by adding the non-linear transformation $x_1x_2$ as a feature, such that $\boldsymbol{\phi}(\mathbf{x}) = [x_1 \;\; x_2 \;\; x_1x_2]^\mathsf{T}$, we get the situation in the right panel in Figure 9.2. With this relatively simple introduction of an extra feature, the problem now appears to be much better suited to a linear classifier, since the data can be separated relatively well by the sketched plane. The conclusion here is that one strategy for increasing the capability of otherwise relatively simple methods is to introduce non-linear feature transformations.

Polynomials are only one out of (infinitely) many possible choices of features $\boldsymbol{\phi}(\mathbf{x})$. One should take care when using polynomials higher than second order in practice because of their behavior outside the range where the data is observed (recall Figure 9.1b). Instead, there are several alternatives that are often more useful in practice, such as Fourier series, essentially corresponding to (for scalar $x$) $\boldsymbol{\phi}(x) = [1 \;\; \sin(x) \;\; \cos(x) \;\; \sin(2x) \;\; \cos(2x) \;\; \cdots]^\mathsf{T}$, step functions, regression splines, etc. The use of non-linear input transformations $\boldsymbol{\phi}(\mathbf{x})$ arguably makes simple models more flexible and applicable to real-world problems with non-linear characteristics. In order to obtain good performance, it is important to chose $\boldsymbol{\phi}(\mathbf{x})$ such that enough flexibility is obtained but overfitting avoided. With a very careful choice of $\boldsymbol{\phi}(\mathbf{x})$, good performance can be obtained for many problems, but that choice is problem-specific and requires some craftmanship. Let us instead explore the conceptual idea of letting the number of features $d \to \infty$ and combine this with regularisation. In a sense this will automate the choice of features, and it leads us to a family of powerful off-the-shelf machine learning tools called kernel methods.

## 9.2 Kernel Ridge Regression

A carefully engineered transformation $\boldsymbol{\phi}(x)$ in linear regression, or any other method for that matter, may indeed perform well for a specific machine learning problem. However, we would like $\boldsymbol{\phi}(x)$ to contain a lot of transformations that could possibly be of interest for most problems, in order to obtain a general off-the-shelf method. We will, therefore, explore the idea of choosing $d$ really large, much larger than the number of data points $n$, and eventually even let $d \to \infty$. The derivation and reasoning will be done using $L^2$-regularised linear regression, but we will later see that the idea is also applicable to other model types.

### Re-formulating Linear Regression

First of all, we have to use some kind of regularisation if we are going to increase $d$ in linear regression, in order to avoid overfitting when $d > n$. For reasons that we will discuss later, we chose to use $L^2$-regularisation. Recall the equation for $L^2$-regularised linear regression:

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \frac{1}{n}\sum_{i=1}^{n} \big(\underbrace{\boldsymbol{\theta}^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}_i)}_{\widehat{y}(\mathbf{x}_i)} - y_i\big)^2 + \lambda\|\boldsymbol{\theta}\|_2^2 = (\boldsymbol{\Phi}(\mathbf{X})^\mathsf{T}\boldsymbol{\Phi}(\mathbf{X}) + n\lambda\mathbf{I})^{-1}\boldsymbol{\Phi}(\mathbf{X})^\mathsf{T}\mathbf{y}, \tag{9.4a}$$

We have not fixed the non-linear transformations $\boldsymbol{\phi}(\mathbf{x})$ to anything specific yet, but we are preparing for choosing $d \gg n$ in these transformations. The downside of choosing $d$, the dimension of $\boldsymbol{\phi}(\mathbf{x})$, large is that we also have to learn $d$ parameters when training. In linear regression, we usually first learn and store the $d$-dimensional vector $\widehat{\boldsymbol{\theta}}$, and thereafter we use it for computing a prediction

$$\widehat{y}(\mathbf{x}_\star) = \widehat{\boldsymbol{\theta}}^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}_\star). \tag{9.5}$$

To be able to choose $d$ really large, conceptually even $d \to \infty$, we have to re-formulate the model such that there are no computations or storage demands that scale with $d$. The first step is to realise that the prediction $\widehat{y}(\mathbf{x}_\star)$ can be rewritten as

$$\begin{aligned}\widehat{y}(\mathbf{x}_\star) = \underbrace{\widehat{\boldsymbol{\theta}}^\mathsf{T}}_{1\times d}\underbrace{\boldsymbol{\phi}(\mathbf{x}_\star)}_{d\times 1} &= \big(\boldsymbol{\Phi}(\mathbf{X})^\mathsf{T}\boldsymbol{\Phi}(\mathbf{X}) + n\lambda\mathbf{I}\big)^{-1}\boldsymbol{\Phi}(\mathbf{X})^\mathsf{T}\mathbf{y}\big)^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}_\star) \\ &= \underbrace{\mathbf{y}^\mathsf{T}}_{1\times n}\underbrace{\boldsymbol{\Phi}(\mathbf{X})}_{n\times d}\underbrace{(\underbrace{\boldsymbol{\Phi}(\mathbf{X})^\mathsf{T}\boldsymbol{\Phi}(\mathbf{X}) + n\lambda\mathbf{I}}_{d\times d})^{-1}\underbrace{\boldsymbol{\phi}(\mathbf{x}_\star)}_{d\times 1}}_{n\times 1},\end{aligned} \tag{9.6}$$

where the underbraces give the sizes of the corresponding vectors and matrices. This expression for $\widehat{y}(\mathbf{x}_\star)$ suggests that, instead of computing and storing the $d$-dimensional $\widehat{\boldsymbol{\theta}}$ once (independently of $\mathbf{x}_\star$), we could, for each test input $\mathbf{x}_\star$,

compute the $n$-dimensional vector $\boldsymbol{\Phi}(\mathbf{X})(\boldsymbol{\Phi}(\mathbf{X})^\mathsf{T}\boldsymbol{\Phi}(\mathbf{X}) + n\lambda\mathbf{I})^{-1}\boldsymbol{\phi}(\mathbf{x}_\star)$. By doing so, we avoid storing a $d$-dimensional vector. But this would still require the inversion of a $d \times d$ matrix. We therefore have some more work to do before we have a practically useful method where we can select $d$ arbitrarily large.

The push-through matrix identity says that $\mathbf{A}(\mathbf{A}^\mathsf{T}\mathbf{A} + \mathbf{I})^{-1} = (\mathbf{A}\mathbf{A}^\mathsf{T} + \mathbf{I})^{-1}\mathbf{A}$ holds for any matrix $\mathbf{A}$. By using it in (9.6), we can further rewrite $\widehat{y}(\mathbf{x}_\star)$ as

$$\widehat{y}(\mathbf{x}_\star) = \underbrace{\mathbf{y}^\mathsf{T}}_{1\times n}\ \underbrace{(\boldsymbol{\Phi}(\mathbf{X})\boldsymbol{\Phi}(\mathbf{X})^\mathsf{T} + n\lambda\mathbf{I})^{-1}}_{n\times n}\ \underbrace{\boldsymbol{\Phi}(\mathbf{X})\boldsymbol{\phi}(\mathbf{x}_\star)}_{n\times 1}. \tag{9.7}$$

It appears in (9.7) as if we can compute $\widehat{y}(\mathbf{x}_\star)$ *without* having to deal with any $d$-dimensional vectors or matrices, provided that the matrix multiplications $\boldsymbol{\Phi}(\mathbf{X})\boldsymbol{\Phi}(\mathbf{X})^\mathsf{T}$ and $\boldsymbol{\Phi}(\mathbf{X})\boldsymbol{\phi}(\mathbf{x}_\star)$ in (9.7) can somehow be computed. Let us therefore have a closer look at these:

$$\boldsymbol{\Phi}(\mathbf{X})\boldsymbol{\Phi}(\mathbf{X})^\mathsf{T} = \begin{bmatrix} \boldsymbol{\phi}(\mathbf{x}_1)^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}_1) & \boldsymbol{\phi}(\mathbf{x}_1)^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}_2) & \ldots & \boldsymbol{\phi}(\mathbf{x}_1)^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}_n) \\ \boldsymbol{\phi}(\mathbf{x}_2)^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}_1) & \boldsymbol{\phi}(\mathbf{x}_2)^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}_2) & \ldots & \boldsymbol{\phi}(\mathbf{x}_2)^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}_n) \\ \vdots & & \ddots & \vdots \\ \boldsymbol{\phi}(\mathbf{x}_n)^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}_1) & \boldsymbol{\phi}(\mathbf{x}_n)^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}_2) & \ldots & \boldsymbol{\phi}(\mathbf{x}_n)^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}_n) \end{bmatrix} \quad \text{and} \tag{9.8}$$

$$\boldsymbol{\Phi}(\mathbf{X})\boldsymbol{\phi}(\mathbf{x}_\star) = \begin{bmatrix} \boldsymbol{\phi}(\mathbf{x}_1)^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}_\star) \\ \boldsymbol{\phi}(\mathbf{x}_2)^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}_\star) \\ \vdots \\ \boldsymbol{\phi}(\mathbf{x}_n)^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}_\star) \end{bmatrix}. \tag{9.9}$$

Remember that $\boldsymbol{\phi}(\mathbf{x})^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}')$ is an inner product between the two $d$-dimensional vectors $\boldsymbol{\phi}(\mathbf{x})$ and $\boldsymbol{\phi}(\mathbf{x}')$. The key insight here is to note that the transformed inputs $\boldsymbol{\phi}(\mathbf{x})$ enter into (9.7) only as inner products $\boldsymbol{\phi}(\mathbf{x})^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}')$, where each inner product is a scalar. That is, if we are able to compute the inner product $\boldsymbol{\phi}(\mathbf{x})^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}')$ directly, without first explicitly computing the $d$-dimensional $\boldsymbol{\phi}(\mathbf{x})$, we have reached our goal.

As a concrete illustration, let us for simplicity consider polynomials. With $p = 1$, meaning $\mathbf{x}$ is a scalar $x$, and $\boldsymbol{\phi}(x)$ is a third-order polynomial ($d = 4$) with the second and third-term scaled by $\sqrt{3}$,[2] we have

$$\begin{aligned} \boldsymbol{\phi}(x)^\mathsf{T}\boldsymbol{\phi}(x') &= \begin{bmatrix} 1 & \sqrt{3}x & \sqrt{3}x^2 & x^3 \end{bmatrix} \begin{bmatrix} 1 \\ \sqrt{3}x' \\ \sqrt{3}x'^2 \\ x'^3 \end{bmatrix} \\ &= 1 + 3xx' + 3x^2x'^2 + x^3x'^3 = (1 + xx')^3. \end{aligned} \tag{9.10}$$

It can generally be shown that if $\boldsymbol{\phi}(x)$ is a (suitably re-scaled) polynomial of order $d-1$, then $\boldsymbol{\phi}(x)^\mathsf{T}\boldsymbol{\phi}(x') = (1 + xx')^{d-1}$. The point we want to make is that instead of first computing the two $d$-dimensional vectors $\boldsymbol{\phi}(x)$ and $\boldsymbol{\phi}(x')$ and thereafter computing

[2] The scaling $\sqrt{3}$ can be compensated for by an inverse scaling of the second and third element in $\boldsymbol{\theta}$.

their inner product, we could just evaluate the expression $(1 + xx')^{d-1}$ directly instead. With a second- or third-order polynomial, this might not make much of a difference, but consider the computational scaling in a situation where it is of interest to use $d$ in the hundreds or thousands.

The main point we are getting at is that *if we just make the choice of* $\boldsymbol{\phi}(\mathbf{x})$ *such that the inner product* $\boldsymbol{\phi}(\mathbf{x})^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}')$ *can be computed without first computing* $\boldsymbol{\phi}(\mathbf{x})$*, we can let* $d$ *be arbitrary big.* Since it is possible to define inner products between infinite-dimensional vectors, there is nothing preventing us from letting $d \to \infty$.

We have now derived a version of $L^2$-regularised linear regression that we can use in practice with an unbounded number of features $d$ in $\boldsymbol{\phi}(\mathbf{x})$, if we restrict ourselves to $\boldsymbol{\phi}(\mathbf{x})$ such that its inner product $\boldsymbol{\phi}(\mathbf{x})^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}')$ has a closed-form expression (or can, at least, be computed in such a way that it does not scale with $d$). This might appear to be of rather limited interest for a machine learning engineer, since one still has to come up with a non-linear transformation $\boldsymbol{\phi}(\mathbf{x})$, choose $d$ (possibly $\infty$), and thereafter make a pen-and-paper derivation (like (9.10)) of $\boldsymbol{\phi}(\mathbf{x})^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}')$. Fortunately it is possible to bypass this by introducing the concept of a *kernel*.

## Introducing the Kernel Idea

A kernel $\kappa(\mathbf{x}, \mathbf{x}')$ is (in this book) any function that takes two arguments $\mathbf{x}$ and $\mathbf{x}'$ from the same space and returns a scalar. Throughout this book, we will limit ourselves to kernels that are real-valued and symmetric, that is, $\kappa(\mathbf{x}, \mathbf{x}') = \kappa(\mathbf{x}', \mathbf{x}) \in \mathbb{R}$ for all $\mathbf{x}$ and $\mathbf{x}'$. Equation (9.10), for example, is such a kernel. And more generally, the inner product of two non-linear input transformations is an example of a kernel:

$$\kappa(\mathbf{x}, \mathbf{x}') = \boldsymbol{\phi}(\mathbf{x})^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}'). \tag{9.11}$$

The important point at this stage is that since $\boldsymbol{\phi}(\mathbf{x})$ only appears in the linear regression model (9.7) via inner products, we do not have to design a $d$-dimensional vector $\boldsymbol{\phi}(\mathbf{x})$ and derive its inner product. Instead, we can just choose a kernel $\kappa(\mathbf{x}, \mathbf{x}')$ directly. This is known as the *kernel trick*:

> If $\mathbf{x}$ enters the model as $\boldsymbol{\phi}(\mathbf{x})^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}')$ only, we can choose a kernel $\kappa(\mathbf{x}, \mathbf{x}')$ instead of chosing $\boldsymbol{\phi}(\mathbf{x})$.

To be clear on what this means in practice, we rewrite (9.7) using the kernel (9.11):

$$\widehat{y}(\mathbf{x}_\star) = \underbrace{\mathbf{y}^\mathsf{T}}_{1\times n}\underbrace{(\boldsymbol{K}(\mathbf{X}, \mathbf{X}) + n\lambda\mathbf{I})}_{n\times n}^{-1}\underbrace{\boldsymbol{K}(\mathbf{X}, \mathbf{x}_\star)}_{n\times 1}, \tag{9.12a}$$

$$\text{where } \boldsymbol{K}(\mathbf{X}, \mathbf{X}) = \begin{bmatrix} \kappa(\mathbf{x}_1, \mathbf{x}_1) & \kappa(\mathbf{x}_1, \mathbf{x}_2) & \dots & \kappa(\mathbf{x}_1, \mathbf{x}_n) \\ \kappa(\mathbf{x}_2, \mathbf{x}_1) & \kappa(\mathbf{x}_2, \mathbf{x}_2) & \dots & \kappa(\mathbf{x}_2, \mathbf{x}_n) \\ \vdots & & \ddots & \vdots \\ \kappa(\mathbf{x}_n, \mathbf{x}_1) & \kappa(\mathbf{x}_n, \mathbf{x}_2) & \dots & \kappa(\mathbf{x}_n, \mathbf{x}_n) \end{bmatrix} \text{ and} \tag{9.12b}$$


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

$$\boldsymbol{K}(\mathbf{X}, \mathbf{x}_\star) = \begin{bmatrix} \kappa(\mathbf{x}_1, \mathbf{x}_\star) \\ \kappa(\mathbf{x}_2, \mathbf{x}_\star) \\ \vdots \\ \kappa(\mathbf{x}_n, \mathbf{x}_\star) \end{bmatrix}. \tag{9.12c}$$

These equations describe linear regression with $L^2$-regularisation using a kernel $\kappa(\mathbf{x}, \mathbf{x}')$. Since $L^2$-regularisation is also called ridge regression, we refer to (9.12) as kernel ridge regression. The $n \times n$ matrix $\boldsymbol{K}(\mathbf{X}, \mathbf{X})$ is obtained by evaluating the kernel at all pairs of training inputs and is called the *Gram matrix*. We initially argued that linear regression with a possibly infinite-dimensional non-linear transformation vector $\boldsymbol{\phi}(\mathbf{x})$ could be an interesting model, and (9.12) is (for certain choices of $\boldsymbol{\phi}(\mathbf{x})$ and $\kappa(\mathbf{x}, \mathbf{x}')$) equivalent to this. The design choice for the user is now to select a kernel $\kappa(\mathbf{x}, \mathbf{x}')$ instead of $\boldsymbol{\phi}(\mathbf{x})$. In practice, choosing $\kappa(\mathbf{x}, \mathbf{x}')$ is a much less tedious problem than choosing $\boldsymbol{\phi}(\mathbf{x})$.

As users, we may in principle choose the kernel $\kappa(\mathbf{x}, \mathbf{x}')$ arbitrarily, as long as we can compute (9.12a). This requires that the inverse of $\boldsymbol{K}(\mathbf{X}, \mathbf{X}) + n\lambda\mathbf{I}$ exists. We are, therefore, on the safe side if we restrict ourselves to kernels for which the Gram matrix $\boldsymbol{K}(\mathbf{X}, \mathbf{X})$ is always positive semidefinite. Such kernels are called positive semidefinite kernels.[3] Hence, the user of kernel ridge regression chooses a positive semidefinite kernel $\kappa(\mathbf{x}, \mathbf{x}')$ and neither has to select nor compute $\boldsymbol{\phi}(\mathbf{x})$. However, a corresponding $\boldsymbol{\phi}(\mathbf{x})$ always exists for a positive semidefinite kernel, as we will discuss in Section 9.4.

There is a number of positive semidefinite kernels commonly used in practice. One positive semidefinite kernel is the squared exponential kernel (also known as the RBF, exponentiated quadratic or Gaussian kernel),

$$\kappa(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|_2^2}{2\ell^2}\right), \tag{9.13}$$

where the hyperparameter $\ell > 0$ is a design choice left to the user, for example to be chosen using cross validation. Another example of a positive semidefinite kernel mentioned earlier is the polynomial kernel $\kappa(\mathbf{x}, \mathbf{x}') = (c + \mathbf{x}^\mathsf{T}\mathbf{x}')^{d-1}$. A special case thereof is the linear kernel $\kappa(\mathbf{x}, \mathbf{x}') = \mathbf{x}^\mathsf{T}\mathbf{x}'$. We will give more examples later.

From the formulation (9.12), it may seem as if we have to compute the inverse of $\boldsymbol{K}(\mathbf{X}, \mathbf{X}) + n\lambda\mathbf{I}$ every time we want to make a prediction. That is, however, not necessary since it does not depend on the test input $\mathbf{x}_\star$. It is, therefore, wise to introduce the $n$-dimensional vector

$$\widehat{\boldsymbol{\alpha}} = \begin{bmatrix} \widehat{\alpha}_1 \\ \widehat{\alpha}_2 \\ \vdots \\ \widehat{\alpha}_n \end{bmatrix} = \mathbf{y}^\mathsf{T}(\boldsymbol{K}(\mathbf{X}, \mathbf{X}) + n\lambda\mathbf{I})^{-1}, \tag{9.14a}$$

[3] Confusingly enough, such kernels are called positive definite in some texts.

which allows us to rewrite kernel ridge regression (9.12) as

$$\widehat{y}(\mathbf{x}_\star) = \widehat{\boldsymbol{\alpha}}^\mathsf{T} \boldsymbol{K}(\mathbf{X}, \mathbf{x}_\star). \tag{9.14b}$$

That is, instead of computing and storing a $d$-dimensional vector $\widehat{\boldsymbol{\theta}}$ as in standard linear regression, we now compute and store an $n$-dimensional vector $\widehat{\boldsymbol{\alpha}}$. However, we also need to store $\mathbf{X}$, since we have to compute $\boldsymbol{K}(\mathbf{X}, \mathbf{x}_\star)$ for every prediction.

We summarise kernel ridge regression in Method 9.1 and illustrate it by Example 9.2. Kernel ridge regression is in itself a practically useful method. That being said, we will next take a step back and discuss what we have derived, in order to prepare for more kernel methods. We will also come back to kernel ridge regression in Chapter 10, where it is used as a stepping stone in deriving the Gaussian process regression model.

**Example 9.2 Linear regression with kernels**

We consider again the car stopping distance problem from Example 2.2 and apply kernel ridge regression to it. We use $\lambda = 0.01$ here and explore what happens when using different kernels.

We start, in the left panel in Figure 9.3, using the squared exponential kernel with $\ell = 1$ (blue line). We see that it does not really interpolate well between the data points, whereas $\ell = 3$ (green line) gives a more sensible behavior. (We could select $\ell$ using cross validation, but we do not pursue that any further here.)

It is interesting to note that prediction reverts to zero when extrapolating beyond the range of the training data. This is, in fact, a general property of the squared exponential kernel as well as many other commonly used kernels. The reason for this behavior is that, by construction, the kernel $\kappa(\mathbf{x}, \mathbf{x}')$ drops to zero as the distance between $\mathbf{x}$ and $\mathbf{x}'$ increases. Intuitively, this means that the resulting predictions are based on local interpolation, and as we extrapolate far beyond the range of the training data, the method will revert to a 'default prediction' of zero. This can be seen from (9.14b) – if $\mathbf{x}_\star$ is far from the training data points, then all elements of the vector $\boldsymbol{K}(\mathbf{X}, \mathbf{x}_\star)$ will be close to zero (for a kernel with the aforementioned property) and so will the resulting prediction.

We have previously seen that this data, to some extent, follows a quadratic function. As we will discuss in Section 9.4, the sum of two kernels is another kernel. In the right panel in Figure 9.3, we therefore try using the sum of the squared exponential kernel (with $\ell = 3$) *and* the polynomial kernel of degree 2 ($d = 3$) (red line). As a reference, we also include kernel ridge regression with only the polynomial kernel of degree 2 ($d = 3$) (dashed blue line; equivalent to $L^2$-regularised polynomial regression). The combined kernel gives a more flexible model than only a quadratic function, but it also (for this example) seems to extrapolate better than only using the squared exponential kernel.

This can be understood by noting that the polynomial kernel is *not local* (in the sense discussed above). That is, it does not drop to zero for test data points that are far from the training data. Instead it corresponds to a polynomial trend, and the predictions will follow this trend when extrapolating. Note that the two kernels considered here result in very similar extrapolations. The reason for this is that the


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

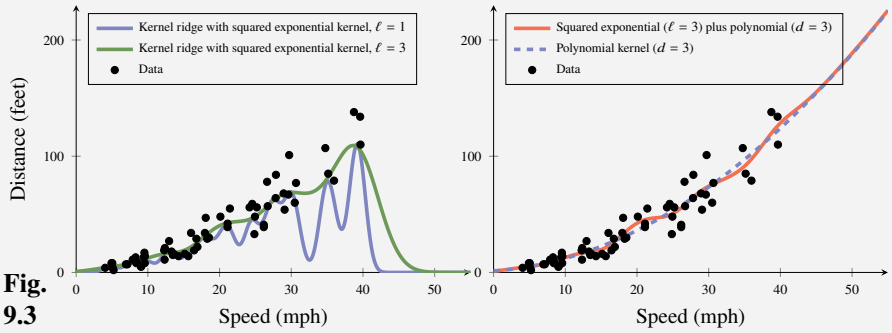



**Fig. 9.3**

squared exponential component of the combined kernel will only 'be active' when interpolating. For extrapolation, the combined kernel will thus revert to using only the polynomial component.

By studying Figure 9.3, we can see that kernel ridge regression is a very flexible model, and the result is highly dependent on the choice of kernel. As we will stress throughout this (and the next) chapter, the kernel is indeed a crucial choice for the machine learning engineer when using kernel methods.

***Time to reflect 9.2*** *Verify that you retrieve $L^2$-regularised linear regression, without any non-linear transformations, by using the linear kernel $\kappa(\mathbf{x}, \mathbf{x}') = \mathbf{x}^\mathsf{T}\mathbf{x}$ in* (9.12).

**Learn** Kernel ridge regression

**Data:** Training data $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^n$ and a kernel $\kappa$
**Result:** Learned dual parameters $\widehat{\boldsymbol{\alpha}}$

1 Compute $\widehat{\boldsymbol{\alpha}}$ as per (9.14a)

**Predict** with Kernel ridge regression

**Data:** Learned dual parameters $\widehat{\boldsymbol{\alpha}}$ and test input $\mathbf{x}_\star$
**Result:** Prediction $\widehat{y}(\mathbf{x}_\star)$

1 Compute $\widehat{y}(\mathbf{x}_\star)$ as (9.14b).

**Method 9.1:** Kernel ridge regression.

## 9.3 Support Vector Regression

Kernel ridge regression, as we just derived, is our fist kernel method for regression, which is a useful method on its own. We will now extend kernel ridge regression into

support vector regression by replacing the loss function. First, however, we take a step back and make an interesting observation that suggests the so-called representer theorem, which will be useful later in this chapter.

### Preparing for More Kernel Methods: The Representer Theorem

The formulation (9.14) is not only practical for implementation; it is also important for theoretical understanding. We can interpret (9.14) as a *dual formulation* of linear regression, where we have the *dual parameters* $\boldsymbol{\alpha}$ instead of the primal formulation (9.4) with primal parameters $\boldsymbol{\theta}$. Remember that $d$, the (possibly infinite) number of primal parameters in $\boldsymbol{\theta}$, is a user design choice, whereas $n$, the number of dual parameters in $\boldsymbol{\alpha}$, is the number of data points.

By comparing (9.14b) and (9.5), we have that

$$\widehat{y}(\mathbf{x}_\star) = \widehat{\boldsymbol{\theta}}^\mathsf{T} \boldsymbol{\phi}(\mathbf{x}_\star) = \widehat{\boldsymbol{\alpha}}^\mathsf{T} \underbrace{\boldsymbol{\Phi}(\mathbf{X})\boldsymbol{\phi}(\mathbf{x}_\star)}_{\boldsymbol{K}(\mathbf{X},\mathbf{x}_\star)} \tag{9.15}$$

for all $\mathbf{x}_\star$, which suggests that

$$\widehat{\boldsymbol{\theta}} = \boldsymbol{\Phi}(\mathbf{X})^\mathsf{T} \widehat{\boldsymbol{\alpha}}. \tag{9.16}$$

This relationship between the primal parameters $\boldsymbol{\theta}$ and the dual parameters $\boldsymbol{\alpha}$ is not specific for kernel ridge regression, but (9.16) is the consequence of a general result called *the representer theorem*.

In essence, the representer theorem states that if $\widehat{y}(\mathbf{x}) = \boldsymbol{\theta}^\mathsf{T}\boldsymbol{\phi}(\mathbf{x})$, the equation (9.16) holds when $\boldsymbol{\theta}$ is learned using (almost) any loss function and $L^2$-regularisation. A full treatment is beyond the scope of this chapter, but we give a complete statement of the theorem in Section 9.A. An implication of the representer theorem is that $L^2$-regularisation is crucial in order to obtain kernel ridge regression (9.14), and we could not have achieved it using, say, $L^1$ regularisation instead. The representer theorem is a cornerstone of most kernel methods, since it tells us that *we can express some models in terms of dual parameters $\boldsymbol{\alpha}$ (of finite length $n$) and a kernel $\kappa(\mathbf{x}, \mathbf{x}')$, instead of the primal parameters $\boldsymbol{\theta}$ (possibly of infinite length $d$) and a non-linear feature transformation $\boldsymbol{\phi}(\mathbf{x})$*, just like we did with linear regression in (9.14).

### Support Vector Regression

We will now look at support vector regression, another off-the-shelf kernel method for regression. From a model perspective, the only difference to kernel ridge regression is a change of loss function. This new loss function has an interesting effect in that the dual parameter vector $\widehat{\boldsymbol{\alpha}}$ in support vector regression becomes *sparse*, meaning that several elements of $\widehat{\boldsymbol{\alpha}}$ are exactly zero. Recall that we can associate each element in $\widehat{\boldsymbol{\alpha}}$ with one training data point. The training data points corresponding to the


Machine Learning – A First Course for Engineers and Scientists
Second Edition, online draft version September 1, 2026, http://smlbook.org

non-zero elements of $\widehat{\boldsymbol{\alpha}}$ are referred to as *support vectors*, and the prediction $\widehat{y}(\mathbf{x}_\star)$ will depend only on these (in contrast to kernel ridge regression (9.14b), where all training data points are needed to compute $\widehat{y}(\mathbf{x}_\star)$). This makes support vector regression an example of a so-called *support vector machine* (SVM), a family of methods with sparse dual parameter vectors.

The loss function we will use for support vector regression is the $\epsilon$-insensitive loss,

$$L(y, \widehat{y}) = \begin{cases} 0 & \text{if } |y - \widehat{y}| < \epsilon, \\ |y - \widehat{y}| - \epsilon & \text{otherwise,} \end{cases} \tag{9.17}$$

or equivalently $L(y, \widehat{y}) = \max\{0, |y - \widehat{y}| - \epsilon\}$, which was introduced in (5.9) in Chapter 5. The parameter $\epsilon$ is a user design choice. In its primal formulation, support vector regression also makes use of the linear regression model

$$\widehat{y}(\mathbf{x}_\star) = \boldsymbol{\theta}^\mathsf{T} \boldsymbol{\phi}(\mathbf{x}_\star), \tag{9.18a}$$

but instead of the least square cost function in (9.4), we now have

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \frac{1}{n} \sum_{i=1}^{n} \max\{0, |y_i - \underbrace{\boldsymbol{\theta}^\mathsf{T} \boldsymbol{\phi}(\mathbf{x}_i)}_{\widehat{y}(\mathbf{x}_i)}| - \epsilon\} + \lambda \|\boldsymbol{\theta}\|_2^2. \tag{9.18b}$$

As with kernel ridge regression, we reformulate the primal formulation (9.18) into a dual formulation with $\boldsymbol{\alpha}$ instead of $\boldsymbol{\theta}$ and use the kernel trick. For the dual formulation, we cannot repeat the convenient closed-form derivation along the lines of (9.4–9.14) since there is no closed-form solution for $\widehat{\boldsymbol{\theta}}$. Instead we have to use optimisation theory, introduce slack variables, and construct the Lagrangian of (9.18b). We do not give the full derivation here (it is similar to the derivation of support vector classification in Appendix 9.B), but as it turns out, the dual formulation becomes

$$\widehat{y}(\mathbf{x}_\star) = \widehat{\boldsymbol{\alpha}}^\mathsf{T} \boldsymbol{K}(\mathbf{X}, \mathbf{x}_\star), \tag{9.19a}$$

where $\widehat{\boldsymbol{\alpha}}$ is the solution to the optimisation problem

$$\widehat{\boldsymbol{\alpha}} = \arg\min_{\boldsymbol{\alpha}} \frac{1}{2} \boldsymbol{\alpha}^\mathsf{T} \boldsymbol{K}(\mathbf{X}, \mathbf{X}) \boldsymbol{\alpha} - \boldsymbol{\alpha}^\mathsf{T} \mathbf{y} + \epsilon \|\boldsymbol{\alpha}\|_1, \tag{9.19b}$$

$$\text{subject to } |\alpha_i| \leq \frac{1}{2n\lambda}. \tag{9.19c}$$

Note that (9.19a) is identical to the corresponding expression for kernel ridge regression (9.14b). This is a consequence of the representer theorem. The only difference to kernel ridge regression is how the dual parameters $\boldsymbol{\alpha}$ are learned: by numerically solving the optimisation problem (9.19b) instead of using the closed-form solution in (9.14a).

The $\epsilon$-insensitive loss function could be used for any regression model, but it is particularly interesting in this kernel context since the dual parameter vector $\boldsymbol{\alpha}$ becomes sparse (meaning that only some elements are non-zero). Remember that $\boldsymbol{\alpha}$ has one entry per training data point. This implies that the prediction (9.19a) depends only on *some* of the training data points, namely those whose corresponding $\alpha_i$ is non-zero, the so-called *support vectors*. In fact it can be shown that the support vectors are the data points for which the loss function is non-zero, that is, the data points for which $|\widehat{y}(\mathbf{x}_i) - y_i| \geq \epsilon$. That is, a larger $\epsilon$ results in fewer support vectors, and vice versa. This effect can also be understood by interpreting $\epsilon$ as a regularisation parameter in an $L^1$ penalty in the dual formulation (9.19b). (The number of support vectors is, however, also affected by $\lambda$, since $\lambda$ influences the shape of $\widehat{y}(\mathbf{x})$.) We illustrate this in Example 9.3.

All training data is indeed used at training time (that is, solving (9.19b)), but when making predictions (using (9.19a)), only the support vectors contribute. This can significantly reduce the computational burden. The larger $\epsilon$ chosen by the user, the fewer support vectors and the fewer computations needed when making predictions. It can, therefore, be said that $\epsilon$ has a regularising effect, in the sense that the more/fewer support vectors that are used, the more/less complicated model. We summarise support vector regression in Method 9.2.

**Learn** support vector regression

**Data:** Training data $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^n$
**Result:** Learned parameters $\widehat{\boldsymbol{\alpha}}$

1 Compute $\widehat{\boldsymbol{\alpha}}$ by numerically solving (9.19b–9.19c)

**Predict** with kernel ridge regression

**Data:** Learned parameters $\widehat{\boldsymbol{\alpha}}$ and test input $\mathbf{x}_\star$
**Result:** Prediction $\widehat{y}(\mathbf{x}_\star)$

1 Compute $\widehat{y}(\mathbf{x}_\star)$ as per (9.19a).

**Method 9.2:** Support vector regression.

### Example 9.3 Support vector regression and kernel ridge regression

We consider yet again the car stopping distance problem from Example 2.2 in Figure 9.4. With the combined squared exponential and polynomial kernel from Example 9.2, $\lambda = 0.01$ and $\epsilon = 15$, we apply support vector regression to the data (red line). As a reference, we also show the corresponding kernel ridge regression (blue line).


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September* 1, 2026, `http://smlbook.org`

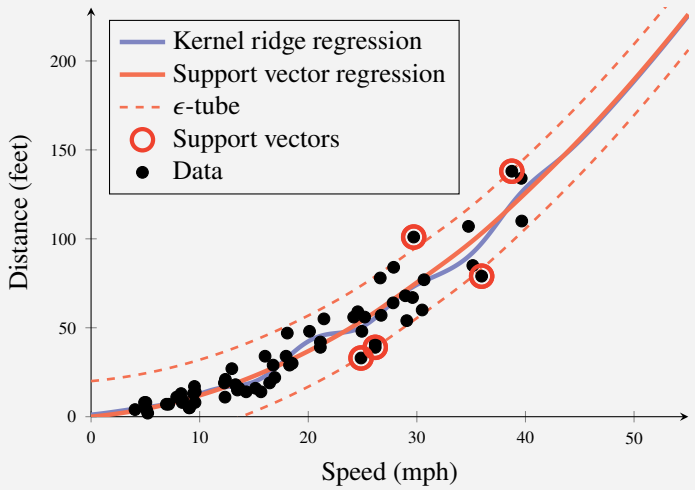



**Fig. 9.4**

In Figure 9.4, we have circled (in red) all data points for which $\alpha_i \neq 0$, the so-called support vectors. We have also included the '$\epsilon$-tube' ($\widehat{y}(\mathbf{x}) \pm \epsilon$; dotted lines), and we can confirm that all support vectors are located outside the '$\epsilon$-tube'. This is a direct effect of using the $\epsilon$-insensitive loss, which explicitly encodes that the loss function for data points within $\epsilon$ from $\widehat{y}(\mathbf{x})$ is exactly zero. If choosing a smaller $\epsilon$, we would have more support vectors, and vice versa. Another consequence of the sparsity of $\boldsymbol{\alpha}$ is that when computing a prediction (9.19a) with support vector regression, it is sufficient to do the computation using (in this case) only five data points. For kernel ridge regression, which does not have a sparse $\boldsymbol{\alpha}$, the prediction (9.14b) depends on all 62 data points.

The $\epsilon$-insensitive loss makes the dual parameter vector $\boldsymbol{\alpha}$ sparse. Note, however, that this does *not* mean that the corresponding primal parameter vector $\boldsymbol{\theta}$ is sparse (their relationship is given by (9.16)). Also note that (9.19b) is a *constrained* optimisation problem (there is a constraint given by (9.19c)), and more theory than we presented in Section 5.4 is needed to derive a good solver.

The feature vector $\boldsymbol{\Phi}(\mathbf{x})$ corresponding to some kernels, such as the squared exponential kernel (9.13), does not have a constant offset term. Therefore, an additional $\theta_0$ is sometimes included in (9.19a) for support vector regression, which adds the constraint $\sum_i \alpha_i = 0$ to the optimisation problem in (9.19b). The same addition could also be made to kernel ridge regression (9.14b), but that would break the closed-form calculation of $\boldsymbol{\alpha}$ (9.14b).

### Conclusions on Using Kernels for Regression

With kernel ridge regression and support vector regression, we have been dealing with the interplay between three different concepts, each of them interesting on its own. To

clarify this, we repeat them in an ordered list:

(i) We have considered the **primal and dual formulations** of a model. The primal formulation expresses the model in terms of $\boldsymbol{\theta}$ (fixed size $d$), whereas the dual formulation uses $\boldsymbol{\alpha}$ (one $\alpha_i$ per data point $i$, hence $\boldsymbol{\alpha}$ has size is $n$ no matter what $d$ happens to be). Both formulations are mathematically equivalent but more or less useful in practice depending on whether $d > n$ or $n > d$.

(ii) We have introduced **kernels** $\kappa(\mathbf{x}, \mathbf{x}')$, which allows us to let $d \to \infty$ without explicitly formulating an infinite vector of non-linear transformations $\boldsymbol{\phi}(\mathbf{x})$. This idea is practically useful only when using the dual formulation with $\boldsymbol{\alpha}$, since $d$ is the dimension of $\boldsymbol{\theta}$.

(iii) We have used **different loss functions**. Kernel ridge regression makes use of squared error loss, whereas support vector regression uses the $\epsilon$-insensitive loss. The $\epsilon$-insensitive loss is particularly interesting in the dual formulation, since it gives *sparse* $\boldsymbol{\alpha}$. (We will later also use the hinge loss for support vector classification in Section 9.5, which has a similar effect.)

We will now spend some additional effort on understanding the kernel concept in Section 9.4 and thereafter introduce support vector classification in Section 9.5.

## 9.4 Kernel Theory

We have defined a kernel as being any function taking two arguments from the same space and returning a scalar. We have also suggested that we can often restrict ourselves to positive semidefinite kernels, and presented two practically useful algorithms – kernel ridge regression and support vector regression. Before we continue and introduce support vector classification, we will discuss the kernel concept further and also give a flavour of the available theory behind it. To make the discussion more concrete, let us start by introducing another kernel method, namely a kernel version of $k$-NN.

### Introducing Kernel *k*-NN

As you know from Chapter 2, $k$-NN constructs the prediction for $\mathbf{x}_\star$ by taking the average or a majority vote among the $k$ nearest neighbours to $\mathbf{x}_\star$. In its standard


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

formulation, 'nearest' is defined by the Euclidean distance. Since the Euclidean distance is always positive, we can equivalently consider the squared Euclidean distance instead, which can be written in terms of the linear kernel $\kappa(\mathbf{x}, \mathbf{x}') = \mathbf{x}^\mathsf{T}\mathbf{x}'$ as

$$\begin{aligned}\|\mathbf{x} - \mathbf{x}'\|_2^2 &= (\mathbf{x} - \mathbf{x}')^\mathsf{T}(\mathbf{x} - \mathbf{x}') = \mathbf{x}^\mathsf{T}\mathbf{x} + \mathbf{x}'^\mathsf{T}\mathbf{x}' - 2\mathbf{x}^\mathsf{T}\mathbf{x}' \\ &= \kappa(\mathbf{x}, \mathbf{x}) + \kappa(\mathbf{x}', \mathbf{x}') - 2\kappa(\mathbf{x}, \mathbf{x}'). \end{aligned} \tag{9.20}$$

To generalise the $k$-NN algorithm to use kernels, we allow the linear kernel to be replaced by any, say, positive semidefinite kernel $\kappa(\mathbf{x}, \mathbf{x}')$ in (9.20). *Kernel $k$-NN* thus works the same as standard $k$-NN but determines proximity between the data points using the right hand side of (9.20) with a user-chosen kernel $\kappa(\mathbf{x}, \mathbf{x}')$ instead of the left hand side of (9.20).

For many (but not all) kernels, it holds that $\kappa(\mathbf{x}, \mathbf{x}) = \kappa(\mathbf{x}', \mathbf{x}') =$ constant for all $\mathbf{x}$ and $\mathbf{x}'$, suggesting that the most interesting part of the right hand side of (9.20) is the term $-2\kappa(\mathbf{x}, \mathbf{x}')$. Thus, if $\kappa(\mathbf{x}, \mathbf{x}')$ takes a large value, the two data points $\mathbf{x}$ and $\mathbf{x}'$ are considered to be close, and vice versa. That is, *the kernel determines how close any two data points are*.

Furthermore, kernel $k$-NN allows us to also use $k$-NN for data where the Euclidean distance has no natural meaning. As long as we have a kernel which acts on the input space, we can apply kernel $k$-NN even if the Euclidean distance is not defined for that input type. We can thereby apply kernel $k$-NN to input data that is neither numerical nor categorical, such as text snippets, as illustrated by Example 9.4.

**Example 9.4 Kernel $k$-NN for interpreting words**

This example illustrates how kernel $k$-NN can be applied to text data, where the Euclidean distance has no meaning, and standard $k$-NN therefore cannot be applied. In this example, the input is single words (or more technically, character strings), and we use the so-called Levenshtein distance to construct a kernel. The Levenshtein distance is the number of single-character edits needed to transform one word (string) into another. It takes two strings and returns a non-negative integer, which is zero only if the two strings are equivalent. It fulfills the properties of being a metric on the space of character strings, and we can thereby use it, for example, in the squared exponential to construct a kernel as $\kappa(x, x') = \exp\left(-\frac{(\mathrm{LD}(x,x'))^2}{2\ell^2}\right)$ (where LD is the Levenshtein distance) with, say, $\ell = 5$.

In this very small example, we consider a training dataset of 10 adjectives shown below ($x_i$), each labelled ($y_i$) `Positive` or `Negative`, according to their meaning. We will now use kernel $k$-NN (with the kernel defined above) to predict whether the word 'horrendous' ($x_\star$) is a positive or negative word. In the third column below, we have therefore computed the Levenshtein distance (LD) between each labelled word ($x_i$) and 'horrendous' ($x_\star$). The rightmost column shows the value of the right hand side of (9.20), which is the value that kernel $k$-NN uses to determine how close two data points are.

| Word, $x_i$ | Meaning, $y_i$ | Levenshtein dist. from $x_i$ to $x_\star$ | $\kappa(x_i, x_i) + \kappa(x_\star, x_\star) - 2\kappa(x_i, x_\star)$ |
|---|---|---|---|
| Awesome | `Positive` | 8 | 1.44 |
| Excellent | `Positive` | 10 | 1.73 |
| Spotless | `Positive` | 9 | 1.60 |
| Terrific | `Positive` | 8 | 1.44 |
| Tremendous | `Positive` | 4 | 0.55 |
| Awful | `Negative` | 9 | 1.60 |
| Dreadful | `Negative` | 6 | 1.03 |
| Horrific | `Negative` | 6 | 1.03 |
| Outrageous | `Negative` | 6 | 1.03 |
| Terrible | `Negative` | 8 | 1.44 |

Inspecting the rightmost column, the closest word to horrendous is the positive word tremendous. Thus, if we use $k = 1$, the conclusion would be that horrendous is a positive word. However, the second, third, and fourth closest words are all negative (dreadful, horrific, outrageous), and with $k = 3$ or $k = 4$, the conclusion thereby becomes that horrendous is a negative word (which also happens to be correct in this case).

The purpose of this example is to illustrate how a kernel allows a basic method such as $k$-NN to be used for a problem where the input has a more intricate structure than just being numerical. For the particular application of predicting word semantics, the character-by-character similarity is clearly an oversimplified approach, and more elaborate approaches exist such as embeddings (Section 7.3).

### The Meaning of a Kernel

From kernel $k$-NN we got (at least) two lessons about kernels that are generally applicable to all supervised machine learning methods that use kernels:

- The kernel defines how close/similar any two data points are. If, say, $\kappa(\mathbf{x}_i, \mathbf{x}_\star) > \kappa(\mathbf{x}_j, \mathbf{x}_\star)$, then $\mathbf{x}_\star$ is considered to be more similar to $\mathbf{x}_i$ than $\mathbf{x}_j$. Intuitively speaking, for most methods, the prediction $\widehat{y}(\mathbf{x}_\star)$ is most influenced by the training data points that are closest/most similar to $\mathbf{x}_\star$. The kernel thereby plays an important role in determining the individual influence of each training data point when making a prediction.

- Even though we started by introducing kernels via the inner product $\boldsymbol{\phi}(\mathbf{x})^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}')$, we do not have to bother about inner product for the space in which $\mathbf{x}$ itself lives. As we saw in Example 9.4, we can also apply a positive semidefinite kernel method to text strings without worrying about inner products of strings, as long as we have a kernel for that type of data.

In addition to this, the kernel also plays a somewhat more subtle role in methods that build on the representer theorem (such as kernel ridge regression, support vector regression, and support vector classification, but not kernel $k$-NN). Remember that the


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

primal formulation of those methods, by virtue of the representer theorem, contains the $L^2$-regularisation term $\lambda\|\boldsymbol{\theta}\|_2^2$. Even though we do not solve the primal formulation explicitly when using kernels (we solve the dual instead), it is nevertheless an equivalent representation, and we may ask what impact the regularisation $\lambda\|\boldsymbol{\theta}\|_2^2$ has on the solution?

The $L^2$-regularisation means that primal parameter values $\boldsymbol{\theta}$ close to zero are favoured. Besides the regularisation term, $\boldsymbol{\theta}$ only appears in the expression $\boldsymbol{\theta}^\mathsf{T}\boldsymbol{\phi}(\mathbf{x})$. The solution $\widehat{\boldsymbol{\theta}}$ to the primal problem is therefore an interplay between the feature vector $\boldsymbol{\phi}(\mathbf{x})$ and the $L^2$-regularisation term. Consider two different choices of feature vectors, $\boldsymbol{\phi}_1(\mathbf{x})$ and $\boldsymbol{\phi}_2(\mathbf{x})$. If they both span the same space of functions, there exist $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$ such that $\boldsymbol{\theta}_1^\mathsf{T}\boldsymbol{\phi}_1(\mathbf{x}) = \boldsymbol{\theta}_2^\mathsf{T}\boldsymbol{\phi}_2(\mathbf{x})$ for all $\mathbf{x}$, and it might appear irrelevant which feature vector that is used. However, the $L^2$-regularisation complicates the situation because it acts directly on $\boldsymbol{\theta}$, and it therefore matters whether we use $\boldsymbol{\phi}_1(\mathbf{x})$ or $\boldsymbol{\phi}_2(\mathbf{x})$. In the dual formulation, we choose the kernel $\kappa(\mathbf{x}, \mathbf{x}')$ instead of feature vector $\boldsymbol{\phi}(\mathbf{x})$, but since that choice implicitly corresponds to a feature vector, the effect is still present, and we may add one more bullet point about the meaning of a kernel:

- The choice of kernel corresponds to a choice of a regularisation functional. That is, the kernel implies a preference for certain functions in the space of all functions that are spanned by the feature vector. For example, the squared exponential kernel implies a preference for smooth functions.

Using a kernel makes a method quite flexible, and one could perhaps expect it to suffer heavily from overfitting. However, the regularising role of the kernel explains why that rarely is the case in practice.

All three bullet points above are central to understanding the usefulness and versatility of kernel methods. They also highlight the importance for the machine learning engineer of choosing the kernel wisely and not simply resorting to 'default' choices.

### Valid Choices of Kernels

We introduced kernels as a way to compactly work with non-linear feature transformations like (9.11). A direct consequence of this is that it is now sufficient to consider $\kappa(\mathbf{x}, \mathbf{x}')$, and not $\boldsymbol{\phi}(\mathbf{x})$. A natural question to ask is whether an arbitrary kernel $\kappa(\mathbf{x}, \mathbf{x}')$ always corresponds to a feature transformation $\boldsymbol{\phi}(\mathbf{x})$, such that it can be written as the inner product

$$\kappa(\mathbf{x}, \mathbf{x}') = \boldsymbol{\phi}(\mathbf{x})^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}')? \tag{9.21}$$

Before answering the question, we have to be aware that this question is primarily of theoretical nature. As long as we can use $\kappa(\mathbf{x}, \mathbf{x}')$ when computing predictions, it serves its purpose, no matter whether it admits the factorisation (9.21) or not. The specific requirements on $\kappa(\mathbf{x}, \mathbf{x}')$ are different for different methods – for example, the inverse $(\boldsymbol{K}(\mathbf{X}, \mathbf{X}) + n\lambda\mathbf{I})^{-1}$ is needed for kernel ridge regression but not for support

vector regression. Furthermore, whether a kernel admits the factorisation (9.21) or not has no direct correspondence to how well it performs in terms of $E_{\text{new}}$. For any practical machine learning problem, the performance still has to be evaluated using cross-validation or similarly.

That being said, we will now have a closer look at the important family of positive semidefinite kernels. A kernel is said to be positive semidefinite if the Gram matrix $\boldsymbol{K}(\mathbf{X}, \mathbf{X})$ as defined in (9.12b) is positive semidefinite (has no negative eigenvalues) for any choice of $\mathbf{X}$.

First, it holds that any kernel $\kappa(\mathbf{x}, \mathbf{x}')$ that is defined as an inner product between feature vectors $\boldsymbol{\phi}(\mathbf{x})$, as in (9.21), is always positive semidefinite. It can, for example, be shown from the equivalent definition of positive semidefinite that $\mathbf{v}^\mathsf{T}\boldsymbol{K}(\mathbf{X}, \mathbf{X})\mathbf{v} \geq 0$ holds for any vector $\mathbf{v} = \begin{bmatrix} v_1 & \dots & v_n \end{bmatrix}^\mathsf{T}$. By using (9.12b), the definition of matrix multiplication (first equality below), and properties of the inner product (second equality below), we can indeed conclude that

$$\mathbf{v}^\mathsf{T}\boldsymbol{K}(\mathbf{X}, \mathbf{X})\mathbf{v} = \sum_{i=1}^{n}\sum_{j=1}^{n} v_i(\boldsymbol{\phi}(\mathbf{x}_i))^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}_j)v_j = \left(\sum_{i=1}^{n} v_i\boldsymbol{\phi}(\mathbf{x}_i)\right)^\mathsf{T}\left(\sum_{j=1}^{n} v_j\boldsymbol{\phi}(\mathbf{x}_j)\right) \geq 0. \tag{9.22}$$

Less trivially, the other direction also holds – that is, for any positive semidefinite kernel $\kappa(\mathbf{x}, \mathbf{x}')$, there always exists a feature vector $\boldsymbol{\phi}(\mathbf{x})$ such that $\kappa(\mathbf{x}, \mathbf{x}')$ can be written as an inner product (9.21). Technically it can be shown that for any positive semidefinite kernel $\kappa(\mathbf{x}, \mathbf{x}')$, it is possible to construct a function space, more specifically a Hilbert space, that is spanned by a feature vector $\boldsymbol{\phi}(\mathbf{x})$ for which (9.21) holds. The dimensionality of the Hilbert space, and thereby also the dimension of $\boldsymbol{\phi}(\mathbf{x})$, can, however, be infinite.

Give a kernel $\kappa(\mathbf{x}, \mathbf{x}')$, there are multiple ways to construct a Hilbert space spanned by $\boldsymbol{\phi}(\mathbf{x})$, and we will only mention some directions here. One alternative is to consider the so-called reproducing kernel map. The reproducing kernel map is obtained by consider one argument, say the latter, to $\kappa(\mathbf{x}, \mathbf{x}')$ fixed and let $\kappa(\cdot, \mathbf{x}')$ span the Hilbert space with an inner product $\langle \cdot, \cdot \rangle$ such that $\langle \kappa(\cdot, \mathbf{x}), \kappa(\cdot, \mathbf{x}') \rangle = \kappa(\mathbf{x}, \mathbf{x}')$. This inner product has a so-called reproducing property, and it is the main building block for the so-called reproducing kernel Hilbert space. Another alternative is to use the so-called Mercer kernel map, which constructs the Hilbert space using eigenfunctions to an integral operator which is related to the kernel.

A given Hilbert space uniquely defines a kernel, but for a given kernel, there exist multiple Hilbert spaces which correspond to it. In practice this means that given a kernel $\kappa(\mathbf{x}, \mathbf{x}')$, the corresponding feature vector $\boldsymbol{\phi}(\mathbf{x})$ is not unique; in fact not even its dimensionality is unique. As a simple example, consider the linear kernel $\kappa(\mathbf{x}, \mathbf{x}') = \mathbf{x}^\mathsf{T}\mathbf{x}'$, which can either be expressed as an inner product between $\boldsymbol{\phi}(\mathbf{x}) = \mathbf{x}$ (one-dimensional $\boldsymbol{\phi}(\mathbf{x})$) or as an inner product between $\boldsymbol{\phi}(\mathbf{x}) = \begin{bmatrix} \frac{1}{\sqrt{2}}\mathbf{x} & \frac{1}{\sqrt{2}}\mathbf{x} \end{bmatrix}^\mathsf{T}$ (two-dimensional $\boldsymbol{\phi}(\mathbf{x})$).


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

### Examples of Kernels

We will now give a list of some commonly used kernels, of which we have already introduced some. These examples are only for the case where $\mathbf{x}$ is a numeric variable. For other types of input variables (such as in Example 9.4), we have to resort to the more application-specific literature. We start with some positive semidefinite kernels, where the *linear kernel* might be the simplest one:

$$\kappa(\mathbf{x}, \mathbf{x}') = \mathbf{x}^\mathsf{T}\mathbf{x}'. \tag{9.23}$$

A generalisation thereof, still positive semidefinite, is the *polynomial kernel*,

$$\kappa(\mathbf{x}, \mathbf{x}') = (c + \mathbf{x}^\mathsf{T}\mathbf{x}')^{d-1}, \tag{9.24}$$

with hyperparameter $c \geq 0$ and polynomial order $d - 1$ (integer). The polynomial kernel corresponds to a finite-dimensional feature vector $\boldsymbol{\phi}(\mathbf{x})$ of monomials up to order $d - 1$. The polynomial kernel does not therefore conceptually enable anything that could not be achieved by instead implementing the primal formulation and the finite-dimensional $\boldsymbol{\phi}(\mathbf{x})$ explicitly. The other positive semidefinite kernels below, on the other hand, all correspond to infinite-dimensional feature vectors $\boldsymbol{\phi}(\mathbf{x})$.

We have previously mentioned the *squared exponential kernel*,

$$\kappa(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|_2^2}{2\ell^2}\right), \tag{9.25}$$

with hyperparameter $\ell > 0$ (usually called lengthscale). As we saw in Example 9.2, this kernel has more of a 'local' nature compared to the polynomial since $\kappa(\mathbf{x}, \mathbf{x}') \to 0$ as $\|\mathbf{x} - \mathbf{x}'\| \to \infty$. This property makes sense in many problems and is perhaps the reason why this might be the most commonly used kernel.

Somewhat related to the squared exponential, we have the family of *Matérn kernels*,

$$\kappa(\mathbf{x}, \mathbf{x}') = \frac{2^{1-\nu}}{\Gamma(\nu)}\left(\frac{\sqrt{2\nu}\|\mathbf{x} - \mathbf{x}'\|_2}{\ell}\right)^\nu k_\nu\left(\frac{\sqrt{2\nu}\|\mathbf{x} - \mathbf{x}'\|}{\ell}\right), \tag{9.26}$$

with hyperparameters $\ell > 0$ and $\nu > 0$, where the latter is a type of smoothness parameter. Here, $\Gamma$ is the Gamma function and $k_\nu$ is a modified Bessel function of the second kind. All Matérn kernels are positive semidefinite. Of particular interest are the cases $\nu = \frac{1}{2}, \frac{3}{2}$, and $\frac{5}{2}$, when (9.26) simplifies to

$$\nu = \frac{1}{2} \Rightarrow \qquad \kappa(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|_2}{\ell}\right), \tag{9.27}$$

$$\nu = \frac{3}{2} \Rightarrow \qquad \kappa(\mathbf{x}, \mathbf{x}') = \left(1 + \frac{\sqrt{3}\|\mathbf{x} - \mathbf{x}'\|_2}{\ell}\right)\exp\left(-\frac{\sqrt{3}\|\mathbf{x} - \mathbf{x}'\|_2}{\ell}\right), \tag{9.28}$$

$$\nu = \frac{5}{2} \Rightarrow \qquad \kappa(\mathbf{x}, \mathbf{x}') = \left(1 + \frac{\sqrt{5}\|\mathbf{x} - \mathbf{x}'\|_2}{\ell} + \frac{5\|\mathbf{x} - \mathbf{x}'\|_2^2}{3\ell^2}\right)\exp\left(-\frac{\sqrt{5}\|\mathbf{x} - \mathbf{x}'\|_2}{\ell}\right). \tag{9.29}$$

The Matérn kernel with $\nu = \frac{1}{2}$ is also called the *exponential kernel*. It can furthermore be shown that the Matérn kernel (9.26) equals the squared exponential (9.25) when $\nu \to \infty$.

As a final example of a positive semidefinite kernel, we mention the *rational quadratic kernel*,

$$\kappa(\mathbf{x}, \mathbf{x}') = \left(1 + \frac{\|\mathbf{x} - \mathbf{x}'\|_2^2}{2a\ell^2}\right)^{-a}, \tag{9.30}$$

with hyperparameters $\ell > 0$ and $a > 0$. The squared exponential, Matérn, and rational quadratic kernel are all examples of *stationary* kernels, since they are functions of only $\mathbf{x} - \mathbf{x}'$. In fact they are also *isotropic* kernels, since they are only functions of $\|\mathbf{x} - \mathbf{x}'\|_2$. The linear kernel is neither isotropic nor stationary.

Going back to the discussion in connection to (9.21), positive semidefinite kernels are a subset of all kernels, for which we know that certain theoretical properties hold. In practice, however, a kernel is potentially useful as long as we can compute a prediction using it, regardless of its theoretical properties. One (at least historically) popular kernel for SVMs which is not positive semidefinite is the *sigmoid kernel*

$$\kappa(\mathbf{x}, \mathbf{x}') = \tanh\left(a\mathbf{x}^\mathsf{T}\mathbf{x}' + b\right), \tag{9.31}$$

where $a > 0$ and $b < 0$ are hyperparameters. The fact that it is not positive semidefinite can, for example, be seen by computing the eigenvalues of $\boldsymbol{K}(\mathbf{X}, \mathbf{X})$ with $a = 1$, $b = -1$, and $\mathbf{X} = [1\ 2]^\mathsf{T}$. Since this kernel is not positive semidefinite, the inverse $(\boldsymbol{K}(\mathbf{X}, \mathbf{X}) + n\lambda\mathbf{I})^{-1}$ does not always exists, and it is therefore not suitable for kernel ridge regression. It can, however, be used in support vector regression and classification, where that inverse is not needed. For certain values of $b$, it can be shown to be a so-called conditional positive semidefinite kernel (a weaker property than positive semidefinite).

It is possible to construct 'new' kernels by modifying or combining existing ones. In particular there is a set of operations that preserve the positive semidefinite property: If $\kappa(\mathbf{x}, \mathbf{x}')$ is a positive semidefinite kernel, then so is $a\kappa(\mathbf{x}, \mathbf{x}')$ if $a > 0$ (scaling). Furthermore, if both $\kappa_1(\mathbf{x}, \mathbf{x}')$ and $\kappa_2(\mathbf{x}, \mathbf{x}')$ are positive semidefinite kernels, then so are $\kappa_1(\mathbf{x}, \mathbf{x}') + \kappa_2(\mathbf{x}, \mathbf{x}')$ (addition) and $\kappa_1(\mathbf{x}, \mathbf{x}')\kappa_2(\mathbf{x}, \mathbf{x}')$ (multiplication).

Most kernels contain a few hyperparameters that are left for the user to choose. Much as cross-validation can provide valuable help in choosing between different kernels, it can also help in choosing hyperparameters with grid search, as discussed in Chapter 5.

## 9.5 Support Vector Classification

We have spent most of our time so far deriving two kernel versions of linear regression: kernel ridge regression and support vector regression. We will now focus on classification. Unfortunately the derivations become more technically involved than


Machine Learning – A First Course for Engineers and Scientists
Second Edition, online draft version September 1, 2026, http://smlbook.org

for kernel ridge regression, and we have placed the details in the chapter appendix. However, the intuition carries over from regression, as well as the main ideas of the dual formulation, the kernel trick, and the change of loss function.

It is possible to derive a kernel version of logistic regression with $L^2$-regularisation. The derivation can be made by first replacing $\mathbf{x}$ with $\boldsymbol{\phi}(\mathbf{x})$, and then using the representer theorem to derive its dual formulation and applying the kernel trick. However, since kernel logistic regression is rarely used in practice, we will instead go straight to support vector classification. As the name suggests, support vector classification is the classification counterpart of support vector regression. Both support vector regression and classification are SVMs since they both have sparse dual parameter vectors.

We consider the binary classification problem $y \in \{-1, 1\}$ and start with the margin formulation (see (5.12) in Chapter 5) of the logistic regression classifier

$$\widehat{y}(\mathbf{x}_\star) = \operatorname{sign}\left\{\boldsymbol{\theta}^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}_\star)\right\}. \tag{9.32}$$

If we now were to learn $\boldsymbol{\theta}$ using the logistic loss (5.13), we would obtain logistic regression with a non-linear feature transformation $\boldsymbol{\phi}(\mathbf{x})$, from which kernel logistic regression eventually would follow. However, inspired by support vector regression, we will instead make use of the hinge loss function (5.16),

$$L(\mathbf{x}, y, \boldsymbol{\theta}) = \max\left\{0, 1 - y_i\boldsymbol{\theta}^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}_i)\right\} = \begin{cases} 1 - y\boldsymbol{\theta}^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}) & \text{if } y\boldsymbol{\theta}^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}) < 1 \\ 0, & \text{otherwise} \end{cases}. \tag{9.33}$$

From Figure 5.2, it is not immediately clear what advantages the hinge loss has over the logistic loss. Analogously to the $\epsilon$-insensitive loss, the main advantage of the hinge loss comes when we look at the dual formulation using $\boldsymbol{\alpha}$ instead of the primal formulation with $\boldsymbol{\theta}$. But before introducing a dual formulation, we first have to consider the primal one. Since the representer theorem is behind all this, we have to use $L^2$-regularisation, which together with (9.33) gives the primal formulation

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \frac{1}{n}\sum_{i=1}^{n} \max\left\{0, 1 - y_i\boldsymbol{\theta}^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}_i)\right\} + \lambda\|\boldsymbol{\theta}\|_2^2. \tag{9.34}$$

The primal formulation does not allow for the kernel trick, since the feature vector does not appear as $\boldsymbol{\phi}(\mathbf{x})^\mathsf{T}\boldsymbol{\phi}(\mathbf{x})$. By using optimisation theory and constructing the Lagrangian (the details can be found in Appendix 9.B), we can arrive at the dual formulation of (9.34),[4]

$$\widehat{\boldsymbol{\alpha}} = \arg\min_{\boldsymbol{\alpha}} \ \frac{1}{2}\boldsymbol{\alpha}^\mathsf{T}\boldsymbol{K}(\mathbf{X}, \mathbf{X})\boldsymbol{\alpha} - \boldsymbol{\alpha}^\mathsf{T}\mathbf{y} \tag{9.35a}$$

$$\text{subject to } \ |\alpha_i| \leq \frac{1}{2n\lambda} \text{ and } 0 \leq \alpha_i y_i \tag{9.35b}$$

[4] In other texts, it is common to let the Lagrange multipliers (see Appendix 9.B) also be the dual variables. It is mathematically equivalent, but we have instead chosen this formulation to highlight the similarities to the other kernel methods and the importance of the representer theorem.

with

$$\widehat{y}(\mathbf{x}_\star) = \operatorname{sign}\left(\widehat{\boldsymbol{\alpha}}^\mathsf{T} \boldsymbol{K}(\mathbf{X}, \mathbf{x}_\star)\right) \tag{9.35c}$$

instead of (9.32). Note that we have also made use of the kernel trick here by replacing $\boldsymbol{\phi}(\mathbf{x})^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}')$ with $\kappa(\mathbf{x}, \mathbf{x}')$, which gives $\boldsymbol{K}(\mathbf{X}, \mathbf{X})$ in (9.35a) and $\boldsymbol{K}(\mathbf{X}, \mathbf{x}_\star)$ in (9.35c). As for support vector regression, adding an offset term $\theta_0$ to (9.32) corresponds to enforcing an additional constraint $\sum_{i=1}^n \alpha_i = 0$ in (9.35).

Because of the representer theorem, the formulation (9.35c) should come as no surprise to us, since it simply corresponds to inserting (9.16) into (9.32). However, the representer theorem only tells us that this dual formulation exists; the solution (9.35a)–(9.35b) does not follow automatically from the representer theorem but requires its own derivation.

Perhaps the most interesting property of the constrained optimisation problem (9.35) is that its solution $\widehat{\boldsymbol{\alpha}}$ turns out to be sparse. This is exactly the same phenomenon as with support vector regression, and it explains why (9.35) is also an SVM. More specifically, (9.35) is called support vector classification. The strength of this method, like support vector regression, is that the model has the full flexibility of being a kernel method, and yet the prediction (9.35c) only explicitly depends on a subset of the training data points (the support vectors). It is, however, important to realise that all training data points are needed when solving (9.35a). We summarise as Method 9.3.

The support vector property is due to the fact that the loss function is exactly equal to zero when the margin (see Chapter 5) is $\geq 1$. In the dual formulation, the parameter $\alpha_i$ becomes nonzero only if the margin for data point $i$ is $\leq 1$, which makes $\widehat{\boldsymbol{\alpha}}$ sparse. It can be shown that this corresponds to data points being either on the 'wrong side' of the decision boundary or within $\frac{1}{\|\boldsymbol{\theta}\|_2} = \frac{1}{\|\boldsymbol{\Phi}(\mathbf{X})^\mathsf{T}\boldsymbol{\alpha}\|_2}$ of the decision boundary in the feature space $\boldsymbol{\phi}(\mathbf{x})$. We illustrate this by Example 9.5.

### Example 9.5 Support vector classification

We consider the binary classification problem with the data given in Figure 9.5 and apply support vector classification with linear and squared exponential kernels, respectively, in Figure 9.6. We mark the support vectors with yellow circles. For the linear kernel, the locations of the support vectors are either on the 'wrong side' of the decision boundary or within $\frac{1}{\|\boldsymbol{\theta}\|_2}$ of it, marked with dashed white lines. As we decrease, $\lambda$ we allow for larger $\boldsymbol{\theta}$ and thereby a smaller band $\frac{1}{\|\boldsymbol{\theta}\|_2}$ and consequently fewer support vectors.

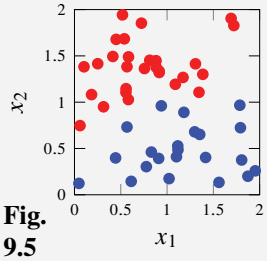



**Fig. 9.5**


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

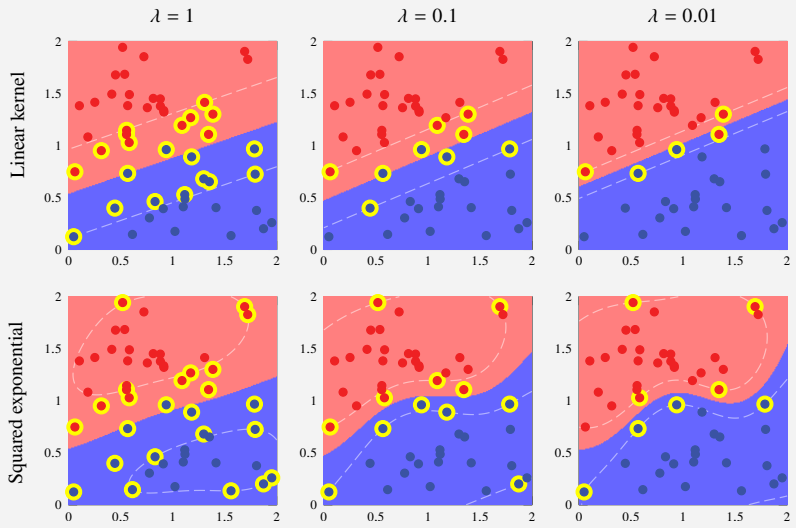



**Fig. 9.6**

When using the (indeed non-linear) squared exponential kernel, the situation becomes somewhat harder to interpret. It still holds that the support vectors are either on the 'wrong side' of the decision boundary or within $\frac{1}{\|\theta\|_2}$ of it, but the distance is measured in infinite dimensional $\boldsymbol{\phi}(\mathbf{x})$-space. Mapping this back to the original input space, we observe this heavily non-linear behavior. The meaning of the dashed white lines is the same as above. This also serves as a good illustration of the power of using kernels.

---

**Learn** Support vector classification

---

**Data:** Training data $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^n$
**Result:** Learned parameters $\widehat{\alpha}$

1 Compute $\widehat{\alpha}$ by numerically solving (9.35a)–(9.35b)

---

**Predict** with Support vector classification

---

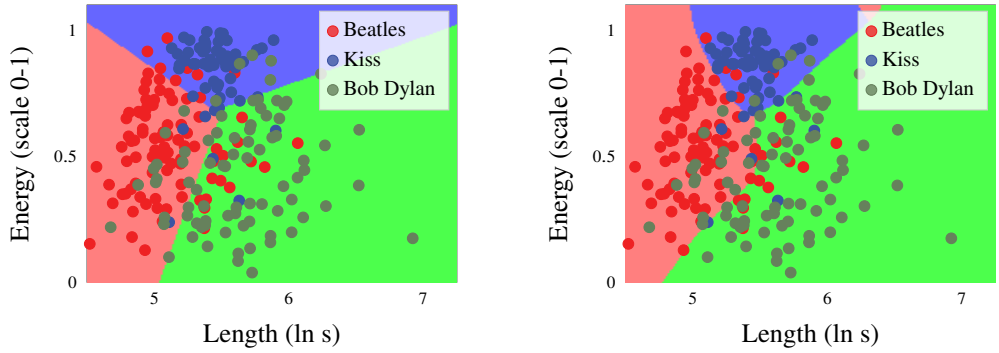
**Data:** Learned parameters $\widehat{\alpha}$ and test input $\mathbf{x}_\star$
**Result:** Prediction $\widehat{y}(\mathbf{x}_\star)$

1 Compute $\widehat{y}(\mathbf{x}_\star)$ as per (9.35c).

---

**Method 9.3:** Support vector classification.

In the SVM literature, it is common to use an equivalent formulation with $C = \frac{1}{2\lambda}$ or $C = \frac{1}{2n\lambda}$ as the regularisation hyperparameter. There also exists a slightly different formulation using another hyperparameter called $\nu$ as, effectively, a regularisation hyperparameter. Those primal and dual problems become slightly more involved, and we do not include them here, but $\nu$ has a somewhat more natural interpretation as it bounds the number of support vectors. To distinguish between the different versions, (9.35) is commonly referred to as $C$-support vector classification and the other version




**(a)** Linear kernel.

**(b)** Squared exponential kernel.

**Figure 9.7:** The decision boundaries for support vector classification with linear and squared exponential kernels, respectively. In this multiclass problem, the one-versus-one strategy has been used. It is clear from this figure that the support vector classification is a linear classifier when using a linear kernel and otherwise is a non-linear classifier.

as $\nu$-support vector classification. A corresponding '$\nu$-version' also exists for support vector regression.

As a consequence of using the hinge loss, as we discussed in Chapter 5, support vector classification does not provide probability estimates $g(\mathbf{x})$ but only a 'hard' classification $\widehat{y}(\mathbf{x}_\star)$. The predicted margin, which is $\widehat{\boldsymbol{\alpha}}\boldsymbol{K}(\mathbf{X}, \mathbf{x}_\star)$ for support vector classification, is not possible to interpret as a class probability estimate, because of the asymptotic minimiser of the hinge loss. As an alternative, it is instead possible to use the squared hinge loss or the Huberised squared hinge loss, which allows for a probability interpretation of the margin. Since all these loss functions are exactly zero for margins $\geq 1$, they retain the support vector property with a sparse $\widehat{\boldsymbol{\alpha}}$. However, by using a different loss function than the hinge loss, we will obtain a different optimisation problem and a different solution to the one discussed above.

The support vector classifier is most often formulated as a solution to the binary classification problem. The generalisation to the multiclass problem is, unfortunately, not straightforward, since it requires a multiclass generalisation of the loss function, as discussed in Chapter 5. In practice, it is common to construct a multiclass classifier from multiple binary ones using either the one-versus-rest or the one-versus-one strategy (see page 106). The latter is used when we apply it to the music classification problem in Figure 9.7.

It is not *necessary* to make use of kernels in support vector classification. It is perfectly possible to use the linear kernel $\kappa(\mathbf{x}, \mathbf{x}') = \mathbf{x}^\mathsf{T}\mathbf{x}'$ or any other kernel corresponding to a finite dimensional $\boldsymbol{\phi}(\mathbf{x})$ in (9.35). That would indeed limit the flexibility of the classifier; the linear kernel limits the classifier to linear decision boundaries, as was illustrated in Example 9.5. The possible benefit of not making full use of kernels, however, is that it suffices to implement and solve the primal (and not the dual) formulation (9.34), since $\boldsymbol{\theta}$ is of finite dimension. The support vector


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

property would still be present but much less visible since $\boldsymbol{\alpha}$ is not explicitly computed. If only using the primal formulation, the representer theorem is not needed, and one could therefore also replace the $L^2$-regularisation with any other regularisation method.

## 9.6 Further Reading

A comprehensive textbook on kernel methods is Schölkopf and Smola (2002), which includes a thorough discussion on SVM and kernel theory as well as several references to original work which we do not repeat here. A commonly used software package for solving SVM problems is Chang and Lin (2011). Kernel $k$-NN is described by Yu et al. (2002) (and the specific kernel based on the Levenshtein distance in example Example 9.4 is found in Xu and X. Zhang (2004)) and kernel logistic regression by Zhu and Hastie (2005). Some more discussion related to the comment about the presence or absence of the offset term in the SVM formulation on page 229 is found in Poggio et al. (2001) and Steinwart et al. (2011).

## 9.A The Representer Theorem

We give a slightly simplified version of the representer theorem by Schölkopf, Herbrich, et al. (2001) adapted to our notation and terminology, without using the reproducing kernel Hilbert space formalism. It is stated in a regression-like setting, since $\widehat{y}(\mathbf{x}) = \boldsymbol{\theta}^\mathsf{T}\boldsymbol{\phi}(\mathbf{x})$, and we discuss how it also applies to classification below.

**Theorem 9.1 (The representer theorem)** *Let $\widehat{y}(\mathbf{x}) = \boldsymbol{\theta}^\mathsf{T}\boldsymbol{\phi}(\mathbf{x})$ with fixed non-linear feature transformations $\boldsymbol{\phi}(\mathbf{x})$ and $\boldsymbol{\theta}$ to be learned from training data $\{\mathbf{x}_i, y_i\}_{i=1}^n$. (The dimensionality of $\boldsymbol{\theta}$ and $\boldsymbol{\phi}(\mathbf{x})$ does not have to be finite.) Furthermore, let $L(y, \widehat{y})$ be any arbitrary loss function and $h : [0, \infty] \mapsto \mathbb{R}$ a strictly monotonically increasing function. Then each minimiser $\boldsymbol{\theta}$ to the regularised cost function*

$$\frac{1}{n}\sum_{i=1}^{n} L(y_i, \underbrace{\boldsymbol{\theta}^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}_i)}_{\widehat{y}(\mathbf{x}_i)}) + h(\|\boldsymbol{\theta}\|_2^2) \tag{9.36}$$

*can be written as $\boldsymbol{\theta} = \boldsymbol{\Phi}(\mathbf{X})^\mathsf{T}\boldsymbol{\alpha}$ (or, equivalently, $\widehat{y}(\mathbf{x}) = \boldsymbol{\alpha}\mathbf{K}(\mathbf{X}, \mathbf{x}_\star)$) with some $n$-dimensional vector $\boldsymbol{\alpha}$.*

**Proof:** For a given $\mathbf{X}$, any $\boldsymbol{\theta}$ can be decomposed into one part $\boldsymbol{\Phi}(\mathbf{X})^\mathsf{T}\boldsymbol{\alpha}$ (with some $\boldsymbol{\alpha}$) that lives in the row span of $\boldsymbol{\Phi}(\mathbf{X})$ and one part $\mathbf{v}$ orthogonal to it, that is, $\boldsymbol{\theta} = \boldsymbol{\Phi}(\mathbf{X})^\mathsf{T}\boldsymbol{\alpha} + \mathbf{v}$ with $\mathbf{v}$ being orthogonal to all rows $\boldsymbol{\phi}(\mathbf{x}_i)$ of $\boldsymbol{\Phi}(\mathbf{X})$.

For any $\mathbf{x}_i$ in $\{\mathbf{x}_i, y_i\}_{i=1}^n$, it therefore holds that

$$\begin{aligned}\widehat{y}(\mathbf{x}_i) &= \boldsymbol{\theta}^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}_i) = (\boldsymbol{\Phi}(\mathbf{X})^\mathsf{T}\boldsymbol{\alpha} + \mathbf{v})^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}_i) \\ &= \boldsymbol{\alpha}^\mathsf{T}\boldsymbol{\Phi}(\mathbf{X})\boldsymbol{\phi}(\mathbf{x}_i) + \underbrace{\mathbf{v}^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}_i)}_{=0} = \boldsymbol{\alpha}^\mathsf{T}\boldsymbol{\Phi}(\mathbf{X})\boldsymbol{\phi}(\mathbf{x}_i).\end{aligned} \tag{9.37}$$

The first term in (9.36) is therefore independent of $\mathbf{v}$. Concerning the second term in (9.36), we have

$$h\left(\|\boldsymbol{\theta}\|_2^2\right) = h\left(\|\boldsymbol{\Phi}(\mathbf{X})^\mathsf{T}\boldsymbol{\alpha} + \mathbf{v}\|_2^2\right) = h\left(\|\boldsymbol{\Phi}(\mathbf{X})^\mathsf{T}\boldsymbol{\alpha}\|_2^2 + \|\mathbf{v}\|_2^2\right) \geq h\left(\|\boldsymbol{\Phi}(\mathbf{X})^\mathsf{T}\boldsymbol{\alpha}\|_2^2\right), \tag{9.38}$$

where the second inequality follows from the fact that $\mathbf{v}$ is orthogonal to $\boldsymbol{\Phi}(\mathbf{X})^\mathsf{T}\boldsymbol{\alpha}$, and equality in the last step only holds if $\mathbf{v} = 0$. The equation (9.38) therefore implies that the minimum of (9.36) is found for $\mathbf{v} = 0$, from which the theorem follows. ■

The assumption that the model is linear in both parameters and features, $\boldsymbol{\theta}^\mathsf{T}\boldsymbol{\phi}(\mathbf{x})$, is indeed crucial for Theorem 9.1. That is not an issue when we consider models of linear regression type, but in order to apply it to, for example, logistic regression, we have to find a linear formulation of that model. Not all models are possible to formulate as linear, but logistic regression can (instead of (3.29a)) be understood as a linear model predicting the so-called log-odds, $\boldsymbol{\theta}^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}) = \ln\left(\frac{p(y=1\,|\,\mathbf{x})}{p(y=-1\,|\,\mathbf{x})}\right)$, and the representer theorem is therefore applicable to it. Furthermore, support vector classification is also a linear model if we consider the function $c(\mathbf{x}) = \boldsymbol{\theta}^\mathsf{T}\boldsymbol{\phi}(\mathbf{x})$ rather than the predicted class $\widehat{y}(\mathbf{x}_\star) = \operatorname{sign}\left\{\boldsymbol{\theta}^\mathsf{T}\boldsymbol{\phi}(\mathbf{x})\right\}$.

## 9.B Derivation of Support Vector Classification

We will derive (9.35) from (9.34),

$$\underset{\boldsymbol{\theta}}{\text{minimise}}\ \frac{1}{n}\sum_{i=1}^{n}\max\left\{0, 1 - y_i\boldsymbol{\theta}^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}_i)\right\} + \lambda\|\boldsymbol{\theta}\|_2^2,$$

by first re-formulating it into an equivalent formulation using slack variables $\boldsymbol{\xi} = [\xi_1 \ldots \xi_n]^\mathsf{T}$:

$$\begin{aligned}
\underset{\boldsymbol{\theta},\boldsymbol{\xi}}{\text{minimise}} \quad & \frac{1}{n}\sum_{i=1}^{n}\xi_i + \lambda\|\boldsymbol{\theta}\|_2^2, && \text{(9.39a)}\\
\text{subject to} \quad & \xi_i \geq 1 - y_i\boldsymbol{\theta}^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}_i), && \text{(9.39b)}\\
(i = 1, \ldots, n) \quad & \xi_i \geq 0. && \text{(9.39c)}
\end{aligned}$$

The Lagrangian for (9.39) is then

$$L(\boldsymbol{\theta}, \boldsymbol{\xi}, \boldsymbol{\beta}, \boldsymbol{\gamma}) = \frac{1}{n}\sum_{i=1}^{n}\xi_i + \lambda\|\boldsymbol{\theta}\|_2^2 - \sum_{i=1}^{n}\beta_i(\xi_i + y_i\boldsymbol{\theta}^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}_i) - 1) - \sum_{i=1}^{n}\gamma_i\xi_i, \tag{9.40}$$

with Lagrange multipliers $\beta_i \geq 0$ and $\gamma_i \geq 0$. According to Lagrange duality theory, instead of solving (9.34), we can minimise (9.40) with respect to $\boldsymbol{\theta}$ and $\boldsymbol{\xi}$ and maximise it with respect to $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$. Two necessary conditions for optimality of (9.40) are

$$\frac{\partial}{\partial\boldsymbol{\theta}}L(\boldsymbol{\theta}, \boldsymbol{\xi}, \boldsymbol{\beta}, \boldsymbol{\gamma}) = 0, \tag{9.41a}$$


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

$$\frac{\partial}{\partial \boldsymbol{\xi}} L(\boldsymbol{\theta}, \boldsymbol{\xi}, \boldsymbol{\beta}, \boldsymbol{\gamma}) = 0. \tag{9.41b}$$

In more detail, and using the fact that $\|\boldsymbol{\theta}\|_2^2 = \boldsymbol{\theta}^\mathsf{T}\boldsymbol{\theta}$, and hence $\frac{\partial}{\partial \boldsymbol{\theta}}\|\boldsymbol{\theta}\|_2^2 = 2\boldsymbol{\theta}$, (9.41a) gives

$$\boldsymbol{\theta} = \frac{1}{2\lambda}\sum_{i=1}^{n} y_i \beta_i \boldsymbol{\phi}(\mathbf{x}_i), \tag{9.41c}$$

and (9.41b) gives

$$\gamma_i = \frac{1}{n} - \beta_i. \tag{9.41d}$$

Inserting (9.41c) and (9.41d) into (9.40) and scaling it with $\frac{1}{2\lambda}$ (assuming $\lambda > 0$), we now seek to maximise

$$\tilde{L}(\boldsymbol{\beta}) = \sum_{i=1}^{n} \frac{\beta_i}{2\lambda} - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} y_i y_j \frac{\beta_i \beta_j}{4\lambda^2} \boldsymbol{\phi}^\mathsf{T}(\mathbf{x}_i)\boldsymbol{\phi}(\mathbf{x}_j). \tag{9.42}$$

For (9.42), we have the constraint $0 \leq \beta_i \leq \frac{1}{n}$, where the upper bound on $b_i$ comes from (9.41d) and the fact that $\gamma_i \geq 0$. With $\alpha_i = \frac{y_i \beta_i}{2\lambda}$, we see (noting that $y_i = 1/y_i$ since $y_i \in -1, 1$) that maximising (9.42) is equivalent to solving

$$\underset{\boldsymbol{\alpha}}{\text{minimise}}\ \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i \alpha_j \boldsymbol{\phi}^\mathsf{T}(\mathbf{x}_i)\boldsymbol{\phi}(\mathbf{x}_j) - \sum_{i=1}^{n} y_i \alpha_i \tag{9.43}$$

or, using matrix notation,

$$\underset{\boldsymbol{\alpha}}{\text{minimise}}\ \frac{1}{2}\boldsymbol{\alpha}^\mathsf{T}\boldsymbol{K}(\mathbf{X}, \mathbf{X})\boldsymbol{\alpha} - \boldsymbol{\alpha}^\mathsf{T}\mathbf{y}. \tag{9.44}$$

Finally, we note that by (9.41c) we have

$$\operatorname{sign}(\boldsymbol{\theta}^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}_\star)) = \operatorname{sign}\left(\frac{1}{2\lambda}\sum_{i=1}^{n} y_i \beta_i \boldsymbol{\phi}^\mathsf{T}(\mathbf{x}_i)\boldsymbol{\phi}(\mathbf{x}_\star)\right) = \operatorname{sign}\left(\boldsymbol{\alpha}^\mathsf{T}\boldsymbol{K}(\mathbf{X}, \mathbf{x}_\star)\right), \tag{9.45}$$

and we have arrived at (9.35).

# 10 The Bayesian Approach and Gaussian Processes

So far, learning a parametric model has amounted to somehow finding a parameter value $\widehat{\boldsymbol{\theta}}$ that best fits the training data. With the *Bayesian approach* (also called the *probabilistic approach*), learning amounts to instead finding the *distribution* of parameter values $\boldsymbol{\theta}$ conditioned on the observed training data $\mathcal{T}$, that is, $p(\boldsymbol{\theta} \mid \mathcal{T})$. Furthermore, with the Bayesian approach, the prediction is a distribution $p(y_\star \mid \mathbf{x}_\star, \mathcal{T})$ instead of a single value. Before we get into the details, let us just say that on a theoretical (or even philosophical) level, the Bayesian approach is rather different to what we have previously seen in this book. However, it opens up for a family of new, versatile, and practically useful methods, and the extra effort required to understand this somewhat different approach pays off well and provides another interesting perspective on supervised machine learning. As the Bayesian approach makes repeated use of probability distributions, it is also natural that this chapter will be heavier on the probability theory side compared to the rest of the book.

We will start this chapter by first giving a general introduction to the Bayesian idea. We thereafter go back to basics and apply the Bayesian approach to linear regression, which we thereafter extend to the non-parametric Gaussian process model.

## 10.1 The Bayesian Idea

In the Bayesian approach, the parameters $\boldsymbol{\theta}$ of any model are consistently treated as being random variables. As a consequence, in this chapter we will use the term *model* with a very specific meaning. A model, in this chapter, refers to the *joint* distribution over all outputs $\mathbf{y}$ and the parameters $\boldsymbol{\theta}$ given all inputs $\mathbf{X}$, that is, $p(\mathbf{y}, \boldsymbol{\theta} \mid \mathbf{X})$. To ease the notation, we will, however, consistently omit $\mathbf{X}$ in the conditioning (mathematically we motivate this by the fact that we only consider $\mathbf{y}$, and not $\mathbf{X}$, to be a random variable) and simply write $p(\mathbf{y}, \boldsymbol{\theta})$.

Learning in the Bayesian setting amounts to computing the distribution of $\boldsymbol{\theta}$ *conditional on* training data $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^n = \{\mathbf{X}, \mathbf{y}\}$. Since we omit $\mathbf{X}$, we denote this distribution as $p(\boldsymbol{\theta} \mid \mathbf{y})$. The computation of $p(\boldsymbol{\theta} \mid \mathbf{y})$ is done using the laws of probabilities. First, we use the rule of conditioning to factorise the joint distribution into two factors: $p(\mathbf{y}, \boldsymbol{\theta}) = p(\mathbf{y} \mid \boldsymbol{\theta}) p(\boldsymbol{\theta})$. By using the rule of conditioning once more, now conditioning on $\mathbf{y}$, we arrive at Bayes' theorem,

$$p(\boldsymbol{\theta} \mid \mathbf{y}) = \frac{p(\mathbf{y} \mid \boldsymbol{\theta}) p(\boldsymbol{\theta})}{p(\mathbf{y})}, \tag{10.1}$$

which is the reason why it is called the Bayesian approach. The left hand side of (10.1) is the sought-after distribution $p(\boldsymbol{\theta} \mid \mathbf{y})$. The right hand side of (10.1) contains some important elements: $p(\mathbf{y} \mid \boldsymbol{\theta})$ is the distribution of the observations in view of the parameters, and $p(\boldsymbol{\theta})$ is the distribution of $\boldsymbol{\theta}$ before any observations are made (that is, not conditional on training data). By definition $p(\boldsymbol{\theta})$ cannot be computed but has to be postulated by the user. Finally $p(\mathbf{y})$ can, by the laws of probabilities, be rewritten as

$$p(\mathbf{y}) = \int p(\mathbf{y}, \boldsymbol{\theta}) d\boldsymbol{\theta} = \int p(\mathbf{y} \mid \boldsymbol{\theta}) p(\boldsymbol{\theta}) d\boldsymbol{\theta}, \tag{10.2}$$

which is an integral that, at least in theory, can be computed. In other words, training a parametric model (in the Bayesian fashion) amounts to conditioning $\boldsymbol{\theta}$ on $\mathbf{y}$, that is, computing $p(\boldsymbol{\theta} \mid \mathbf{y})$. After being trained, a model can be used to compute predictions. Again this is a matter of computing a distribution $p(y_\star \mid \mathbf{y})$ (rather than a point prediction $\widehat{y}$) for a test input $\mathbf{x}_\star$, which, since $\boldsymbol{\theta}$ is a random variable, connects to $p(\boldsymbol{\theta} \mid \mathbf{y})$ via the marginalisation

$$p(y_\star \mid \mathbf{y}) = \int p(y_\star \mid \boldsymbol{\theta}) p(\boldsymbol{\theta} \mid \mathbf{y}) d\boldsymbol{\theta}. \tag{10.3}$$

Here $p(y_\star \mid \boldsymbol{\theta})$ encodes the distribution of the test data output $y_\star$ (again the corresponding input $\mathbf{x}_\star$ is omitted in the notation).

Often $p(\mathbf{y} \mid \boldsymbol{\theta})$ is referred to as the likelihood.[1] The other elements involved in the Bayesian approach are traditionally given the names

- $p(\boldsymbol{\theta})$ *prior*,
- $p(\boldsymbol{\theta} \mid \mathbf{y})$ *posterior*,
- $p(y_\star \mid \mathbf{y})$ *posterior predictive*.

These names are useful when talking about the various component of the Bayesian approach, but it is important to remember that they are nothing but different probability distributions connected to each other via the likelihood $p(\mathbf{y} \mid \boldsymbol{\theta})$. In addition, $p(\mathbf{y})$ is often called the *marginal likelihood* or *evidence*.

## A Representation of Beliefs

The main feature of the Bayesian approach is its use of probability distributions. It is possible to interpret those distributions as representing *beliefs* in the following

[1] Remember that $p(\mathbf{y} \mid \boldsymbol{\theta})$ was also used for the maximum likelihood perspective; one example of $p(\mathbf{y} \mid \boldsymbol{\theta})$ is linear regression (3.17)–(3.18).


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

sense: The prior $p(\boldsymbol{\theta})$ represents our beliefs about $\boldsymbol{\theta}$ *a priori*, that is, before any data has been observed. The likelihood, encoded as $p(\mathbf{y} \mid \boldsymbol{\theta})$, defines how data $\mathbf{y}$ relates to the parameter $\boldsymbol{\theta}$. Using Bayes' theorem (10.1), we *update* the belief about $\boldsymbol{\theta}$ to the posterior $p(\boldsymbol{\theta} \mid \mathbf{y})$ which also takes the observed data $\mathbf{y}$ into account. In everyday language, these distributions could be said to represent the uncertainty about $\boldsymbol{\theta}$ before and after observing the data $\mathbf{y}$, respectively.

An interesting and practically relevant consequence is that the Bayesian approach is less prone to overfitting, compared to using a maximum-likelihood based method. In maximum-likelihood, we obtain a single value $\widehat{\boldsymbol{\theta}}$ and use that value to make our prediction according to $p(y_\star \mid \widehat{\boldsymbol{\theta}})$. In the Bayesian approach, we instead obtained an entire distribution $p(\boldsymbol{\theta} \mid \mathbf{y})$ representing different hypotheses of the value for our model parameters. We account for all of these hypotheses we do the marginalisation in (10.3) to compute the posterior predictive. In the regime of small datasets in particular, the 'uncertainty' seen in the posterior $p(\boldsymbol{\theta} \mid \mathbf{y})$ reflects how much (or little) can be said about $\boldsymbol{\theta}$ from the (presumably) limited information in $\mathbf{y}$, under the assumed conditions.

The posterior $p(\boldsymbol{\theta} \mid \mathbf{y})$ is a combination of the prior belief $p(\boldsymbol{\theta})$ and the information about $\boldsymbol{\theta}$ carried by $\mathbf{y}$ through the likelihood. Without a meaningful prior $p(\boldsymbol{\theta})$, the posterior $p(\boldsymbol{\theta} \mid \mathbf{y})$ is not meaningful either. In some applications it can be hard to make a choice of $p(\boldsymbol{\theta})$ that is not influenced by the personal experience of the machine learning engineer, which is sometimes emphasised by saying that $p(\boldsymbol{\theta})$, and thereby also $p(\boldsymbol{\theta} \mid \mathbf{y})$, represents a *subjective* belief. This notion is meant to reflect that the result is not independent of the human that designed the solution. However, no matter whether or not the Bayesian approach is used, the likelihood is often chosen based on the personal experience of the machine learning engineer, meaning that most machine learning results are in that sense subjective anyway.

An interesting situation for the Bayesian approach is when data arrives in a sequential fashion, that is, one data point after the other. Say that we have two sets of data, $\mathbf{y}_1$ and $\mathbf{y}_2$. Starting with a prior $p(\boldsymbol{\theta})$, we can condition on $\mathbf{y}_1$ by computing $p(\boldsymbol{\theta} \mid \mathbf{y}_1)$ using Bayes' theorem (10.1). However, if we thereafter want to condition on all data, $\mathbf{y}_1$ and $\mathbf{y}_2$ as $p(\boldsymbol{\theta} \mid \mathbf{y}_1, \mathbf{y}_2)$, we do not have to start over again. We can instead replace the prior $p(\boldsymbol{\theta})$ with $p(\boldsymbol{\theta} \mid \mathbf{y}_1)$ in Bayes' theorem to compute $p(\boldsymbol{\theta} \mid \mathbf{y}_1, \mathbf{y}_2)$. In a sense, the 'old posterior' becomes the 'new prior' when data arrives sequentially.

### The Marginal Likelihood as a Model Selection Tool

When using the Bayesian approach, there are often some hyperparameters in the likelihood or the prior, say $\eta$, that need to be chosen. It is an option to assume a 'hyper'-prior $p(\eta)$ and compute the posterior also for the hyperparameters, $p(\eta \mid \mathbf{y})$. That would be the fully Bayesian solution, but sometimes this is too computationally challenging.

A more pragmatic solution is to select a value $\widehat{\eta}$ using cross-validation instead. It

is perfectly possible to use cross-validation, but the Bayesian approach also comes with an alternative for selecting hyperparameters $\eta$ by maximising the marginal likelihood (10.2):

$$\widehat{\eta} = \arg\max_{\eta} p_{\eta}(\mathbf{y}), \tag{10.4}$$

where we have added an index $\eta$ to emphasise the fact that $p(\mathbf{y})$ depends on $\eta$. This approach is sometimes referred to as *empirical Bayes*. Choosing hyperparameter $\eta$ is, in a sense, a selection of a likelihood (and/or prior), and we can therefore understand the marginal likelihood as a tool for selecting a likelihood. Maximising the marginal likelihood is, however, not equivalent to using cross-validation (the obtained hyperparameter value might differ), and unlike cross-validation, the marginal likelihood does not give an estimate of $E_{\text{new}}$. In many situations, however, it is relatively easy to compute (and maximise) the marginal likelihood, compared to employing a full cross-validation procedure.

In the previous section we argued that the Bayesian approach was less prone to overfitting compared to the maximum likelihood approach. However, maximising the marginal likelihood is, in a sense, a kind of a maximum likelihood approach, and one may ask if there is a risk of overfitting when maximising the marginal likelihood. To some extent that might be the case, but the key point is that handling one (or, at most, a few) hyperparameters $\eta$ with maximum (marginal) likelihood typically does not cause any severe overfitting, much like there rarely are overfitting issues when learning a straight line with plain linear regression. In other words, we can usually 'afford' to learn one or a few (hyper)parameters by maximising the (marginal) likelihood; overfitting typically only becomes a potential issue when learning a larger number of (hyper)parameters.

## 10.2 Bayesian Linear Regression

As a first example of the Bayesian approach, we will apply it to linear regression. In itself Bayesian linear regression is perhaps not the most versatile method, but just like ordinary linear regression, it is a good starting point and illustrates the main concepts well. Just like ordinary linear regression, it is possible to extend it in various directions. It also opens the way for the perhaps more interesting Gaussian process model. Before we work out the details of the Bayesian approach applied to linear regression, we will repeat some facts about the multivariate Gaussian distribution that will be useful.

### The Multivariate Gaussian Distribution

A central mathematical object for Bayesian linear regression (and later also the Gaussian process) is the multivariate Gaussian distribution. We assume that the reader already has some familiarity with multivariate random variables, or equivalently random vectors, and repeat only the most important properties of the multivariate Gaussian distribution here.


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

Let $\mathbf{z}$ denote a $q$-dimensional multivariate Gaussian random vector $\mathbf{z} = [z_1 \;\; z_2 \;\; \ldots \;\; z_q]^\mathsf{T}$. The multivariate Gaussian distribution is parametrised by a $q$-dimensional mean vector $\boldsymbol{\mu}$ and a $q \times q$ covariance matrix $\boldsymbol{\Sigma}$,

$$\boldsymbol{\mu} = \begin{bmatrix} \mu_1 \\ \mu_2 \\ \vdots \\ \mu_q \end{bmatrix}, \quad \boldsymbol{\Sigma} = \begin{bmatrix} \sigma_1^2 & \sigma_{12} & \ldots & \sigma_{1q} \\ \sigma_{21} & \sigma_2^2 & & \sigma_{2q} \\ \vdots & & & \vdots \\ \sigma_{q1} & \sigma_{q2} & \ldots & \sigma_q^2 \end{bmatrix}.$$

The covariance matrix is a real-valued positive semidefinite matrix, that is, a symmetric matrix with nonnegative eigenvalues. As a shorthand notation, we write $\mathbf{z} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ or $p(\mathbf{z}) = \mathcal{N}(\mathbf{z}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$. Note that we use the same symbol $\mathcal{N}$ to denote the univariate as well as the multivariate Gaussian distribution. The reason is that the former is just a special case of the latter.

The expected value of $\mathbf{z}$ is $\mathbb{E}[\mathbf{z}] = \boldsymbol{\mu}$, and the variance of $z_1$ is $\operatorname{var}(z_1) = \mathbb{E}\big[(z_1 - \mathbb{E}[z_1])^2\big] = \sigma_1^2$, and similarly for $z_2, \ldots, z_q$. Moreover, the covariance between $z_1$ and $z_2$ is $\operatorname{cov}(z_1, z_2) = \mathbb{E}[(z_1 - \mathbb{E}[z_1])(z_2 - \mathbb{E}[z_2])] = \sigma_{12} = \sigma_{21}$, and similarly for any other pair of $z_i, z_j$. All these properties can be derived from the probability density function of the multivariate Gaussian distribution, which is

$$\mathcal{N}(\mathbf{z}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) = (2\pi)^{-\frac{q}{2}} \det(\boldsymbol{\Sigma})^{-\frac{1}{2}} \exp\left(-\frac{1}{2}(\mathbf{z} - \boldsymbol{\mu})^\mathsf{T} \boldsymbol{\Sigma}^{-1} (\mathbf{z} - \boldsymbol{\mu})\right). \tag{10.5}$$

If all off-diagonal elements of $\boldsymbol{\Sigma}$ are 0, the elements of $\mathbf{z}$ are just independent univariate Gaussian random variables. However, if some off-diagonal element, say $\sigma_{ij}$ $(i \neq j)$, is nonzero, then there is a correlation between $z_i$ and $z_j$. Intuitively, the correlation means that $z_i$ carries information also about $z_j$, and vice versa. Some important results on how the multivariate Gaussian distribution can be manipulated are summarised in Appendix 10.A.

### Linear Regression with the Bayesian Approach

We will now apply the Bayesian approach to the linear regression model. We will first spend some effort on mapping the elements of linear regression from Chapter 3 to the Bayesian terminology, and thereafter we will derive the solution.

From Chapter 3, we have that the linear regression model is

$$y = f(\mathbf{x}) + \varepsilon, \quad f(\mathbf{x}) = \boldsymbol{\theta}^\mathsf{T} \mathbf{x}, \quad \varepsilon \sim \mathcal{N}\big(0, \sigma^2\big), \tag{10.6}$$

which can be written equivalently as

$$p(y \mid \boldsymbol{\theta}) = \mathcal{N}\big(y; \boldsymbol{\theta}^\mathsf{T} \mathbf{x}, \sigma^2\big). \tag{10.7}$$

This is an expression for one output data point $y$, and for the entire vector of all training data outputs $\mathbf{y}$, we can write

$$p(\mathbf{y} \mid \boldsymbol{\theta}) = \prod_{i=1}^{n} p(y_i \mid \boldsymbol{\theta}) = \prod_{i=1}^{n} \mathcal{N}\big(y_i; \boldsymbol{\theta}^\mathsf{T} \mathbf{x}_i, \sigma^2\big) = \mathcal{N}\big(\mathbf{y}; \mathbf{X}\boldsymbol{\theta}, \sigma^2 \mathbf{I}\big), \tag{10.8}$$

where in the last step, we used the notation $\mathbf{X}$ from (3.5) and the fact that an $n$-dimensional Gaussian random vector with a diagonal covariance matrix is equivalent to $n$ scalar Gaussian random variables.

In the Bayesian approach, there is also a need for a prior $p(\boldsymbol{\theta})$ for the unknown parameters $\boldsymbol{\theta}$. In Bayesian linear regression, the prior distribution is most often chosen as a Gaussian with mean $\boldsymbol{\mu}_0$ and covariance $\boldsymbol{\Sigma}_0$,

$$p(\boldsymbol{\theta}) = \mathcal{N}\big(\boldsymbol{\theta}; \boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0\big)\,, \tag{10.9}$$

with, for example, $\boldsymbol{\Sigma}_0 = \mathbf{I}\sigma_0^2$. The reason for this choice is frankly that it simplifies the calculations, much like the squared error loss simplifies the computations for plain linear regression.

The next step is now to compute the posterior. It is possible to derive it using Bayes' theorem, but since $p(\mathbf{y} \mid \boldsymbol{\theta})$ as well as $p(\boldsymbol{\theta})$ are multivariate Gaussian distributions, Corollary 10.1 in Appendix 10.A directly gives us that

$$p(\boldsymbol{\theta} \mid \mathbf{y}) = \mathcal{N}\big(\boldsymbol{\theta}; \boldsymbol{\mu}_n, \boldsymbol{\Sigma}_n\big)\,, \tag{10.10a}$$

$$\boldsymbol{\mu}_n = \boldsymbol{\Sigma}_n \left(\tfrac{1}{\sigma_0^2}\boldsymbol{\mu}_0 + \tfrac{1}{\sigma^2}\mathbf{X}^\mathsf{T}\mathbf{y}\right), \tag{10.10b}$$

$$\boldsymbol{\Sigma}_n = \left(\tfrac{1}{\sigma_0^2}\mathbf{I} + \tfrac{1}{\sigma^2}\mathbf{X}^\mathsf{T}\mathbf{X}\right)^{-1}. \tag{10.10c}$$

From (10.10), we can also derive the posterior predictive for $f(\mathbf{x}_\star)$ by Corollary 10.2 in Appendix 10.A:

$$p(f(\mathbf{x}_\star) \mid \mathbf{y}) = \mathcal{N}(f(\mathbf{x}_\star); m_\star, s_\star)\,, \tag{10.11a}$$

$$m_\star = \mathbf{x}_\star^\mathsf{T}\boldsymbol{\mu}_n, \tag{10.11b}$$

$$s_\star = \mathbf{x}_\star^\mathsf{T}\boldsymbol{\Sigma}_n\mathbf{x}_\star. \tag{10.11c}$$

We have so far only derived the posterior predictive for $f(\mathbf{x}_\star)$. Since we have $y_\star = f(\mathbf{x}_\star) + \varepsilon$ according to the linear regression model (10.10), where $\varepsilon$ is assumed to be independent of $f$ with variance $\sigma^2$, we can also compute the posterior predictive for $\mathbf{y}_\star$

$$p(y_\star \mid \mathbf{y}) = \mathcal{N}\big(y_\star; m_\star, s_\star + \sigma^2\big). \tag{10.11d}$$

Note, that only difference from $p(f(\mathbf{x}_\star) \mid \mathbf{y})$ is that we add the variance of the measurement noise $\sigma^2$, which reflects the the additional uncertainty we expect to get from the test data output. In both $p(f(\mathbf{x}_\star) \mid \mathbf{y})$ and $p(y_\star \mid \mathbf{y})$, the measurement noise of the training data output is reflected via the posterior.

We now have all the pieces of Bayesian linear regression in place. The main difference to plain linear regression is that we compute a posterior distribution $p(\boldsymbol{\theta} \mid \mathbf{y})$ (instead of a single value $\widehat{\boldsymbol{\theta}}$) and a posterior predictive distribution $p(y_\star \mid \mathbf{y})$ instead of a prediction $\widehat{y}$. We summarise it as Method 10.1 and illustrate it with Example 10.1.

**Learn** Bayesian linear regression

**Data:** Training data $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^n$
**Result:** Posterior $p(\boldsymbol{\theta} \mid \mathbf{y}) = \mathcal{N}\left(\boldsymbol{\theta}; \boldsymbol{\mu}_n, \boldsymbol{\Sigma}_n\right)$

1 Compute $\boldsymbol{\mu}_n$ and $\boldsymbol{\Sigma}_n$ as (10.10).

**Predict** with Bayesian linear regression

**Data:** Posterior $p(\boldsymbol{\theta} \mid \mathbf{y}) = \mathcal{N}\left(\boldsymbol{\theta}; \boldsymbol{\mu}_n, \boldsymbol{\Sigma}_n\right)$ and test input $\mathbf{x}_\star$
**Result:** Posterior predictive $p(f(\mathbf{x}_\star) \mid \mathbf{y}) = \mathcal{N}(f(\mathbf{x}_\star); m_\star, s_\star)$

1 Compute $m_\star$ and $s_\star$ as (10.11).

**Method 10.1:** Bayesian linear regression.

We have so far assumed that the noise variance $\sigma^2$ is fixed. Most often $\sigma^2$ is also a parameter the user has to decide, which can be done by maximising the marginal likelihood. Corollary 10.2 gives us the marginal likelihood:

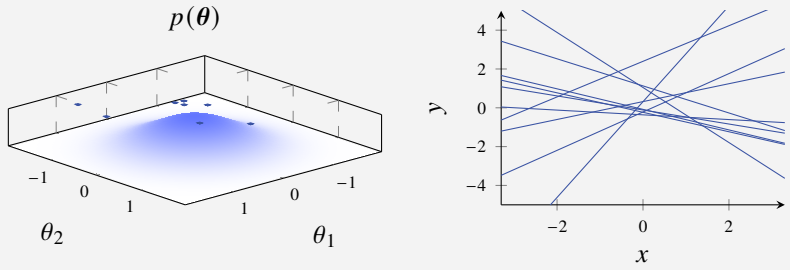
$$p(\mathbf{y}) = \mathcal{N}\left(\mathbf{y}; \mathbf{X}\boldsymbol{\mu}_0, \sigma^2\mathbf{I} + \mathbf{X}\boldsymbol{\Sigma}_0\mathbf{X}^\mathsf{T}\right). \tag{10.12}$$

It is also possible to chose the prior variance $\sigma_0^2$ by maximising (10.12).

Just as for plain linear regression, it is possible to use non-linear input transformations, such as polynomials, in Bayesian linear regression. We give an example of that in Example 10.2, where we return to the running regression example of car stopping distances. We can, however, go one step further and also use the kernel trick from Chapter 9. That will lead us to the Gaussian process, which is the topic of Section 10.3.

**Example 10.1 Bayesian linear regression**

To illustrate the inner workings of Bayesian linear regression, we consider a one-dimensional example with $y = \theta_1 + \theta_2 x + \varepsilon$. In the first row of Figure 10.1, the left panel shows the prior $p(\boldsymbol{\theta})$ (blue surface; a two-dimensional Gaussian distribution over $\theta_1$ and $\theta_2$) from which 10 samples are drawn (blue dots). Each of these samples correspond to a straight line in the plot to the right (blue lines).



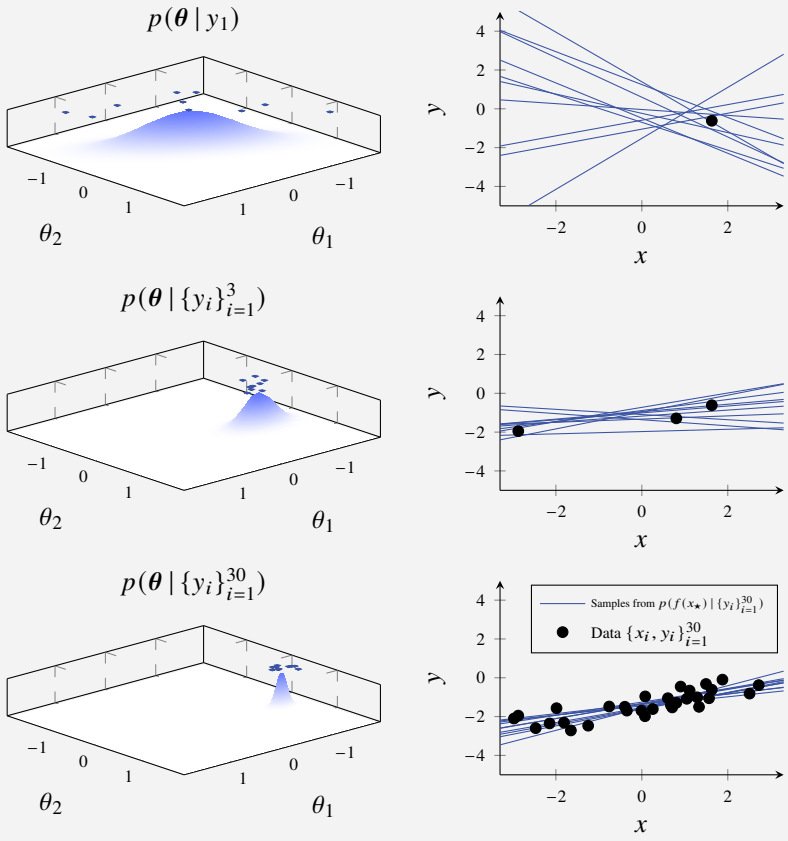



**Fig. 10.1**

In the second row of Figure 10.1 one ($n = 1$) data point $\{x_1, y_1\}$ is introduced. Its value is shown in the right panel (black dot), and the posterior for $\boldsymbol{\theta}$ in the left panel (a Gaussian; blue surface). In addition, 10 samples are drawn from the posterior (blue dots), each corresponding to a straight line in the right panel (blue lines). In the Bayesian formulation, the sampled lines can be thought of as equally likely posterior hypotheses about $f(x_\star)$. This is repeated also with $n = 3$ and $n = 30$ data points in Figure 10.1. We can in particular see how the posterior contracts ('less uncertainty') as more data arrives, in terms of the blue surface being more peaked as well as the blue lines being more concentrated.


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

**Example 10.2 Car stopping distances**

We consider the car stopping distance problem from Example 2.2 and apply probabilistic linear regression with $y = 1 + \theta_1 x + \varepsilon$ and $y = 1 + \theta_1 x + \theta_2 x^2 + \varepsilon$, respectively. We set $\sigma^2$ and $\sigma_0^2$ by maximising the marginal likelihood, which gives us $\sigma^2 = 12.0^2$ and $\sigma_0^2 = 14.1^2$ for the linear model, and $\sigma^2 = 10.1^2$ and $\sigma_0^2 = 0.3^2$ for the quadratic model.

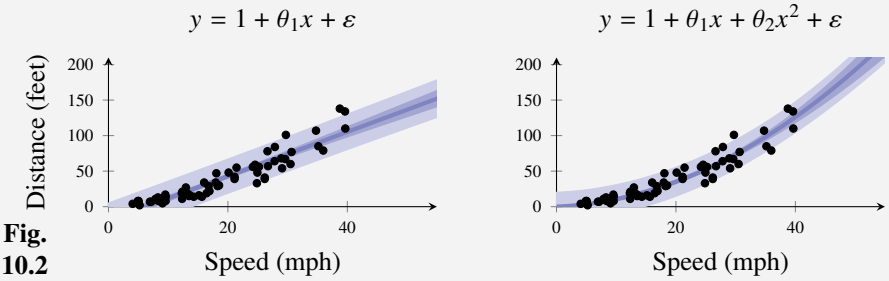



**Fig. 10.2**

In Figure 10.2 we illustrate $p(f(\mathbf{x}_\star) \mid \mathbf{y})$ and $p(y_\star \mid \mathbf{y})$ (10.11) using the somewhat darker blue line for the mean (they both have the same mean) and the shaded blue areas of different intensities to visualise two standard deviations of $p(f(\mathbf{x}_\star) \mid \mathbf{y})$ and $p(y_\star \mid \mathbf{y})$, respectively.

### Connection to Regularised Linear Regression

The main feature of the Bayesian approach is that it provides a full distribution $p(\boldsymbol{\theta} \mid \mathbf{y})$ over the parameters $\boldsymbol{\theta}$, rather than a single point estimate $\widehat{\boldsymbol{\theta}}$. There is, however, also an interesting connection between the Bayesian approach and regularisation. We will make this concrete by considering the posterior $p(\boldsymbol{\theta} \mid \mathbf{y})$ from Bayesian linear regression and the point estimate $\widehat{\boldsymbol{\theta}}_{L^2}$ obtained from $L^2$-regularised linear regression with squared error loss. Let us extract the so called *maximum a posteriori (MAP)* point estimate $\widehat{\boldsymbol{\theta}}_{\text{MAP}}$ from the posterior $p(\boldsymbol{\theta} \mid \mathbf{y})$. The MAP estimate is the value of $\boldsymbol{\theta}$ for which the posterior reaches its maximum,

$$\widehat{\boldsymbol{\theta}}_{\text{MAP}} = \arg\max_{\boldsymbol{\theta}} p(\boldsymbol{\theta} \mid \mathbf{y}) = \arg\max_{\boldsymbol{\theta}} p(\mathbf{y} \mid \boldsymbol{\theta}) p(\boldsymbol{\theta}) = \arg\max_{\boldsymbol{\theta}} \left[\ln p(\mathbf{y} \mid \boldsymbol{\theta}) + \ln p(\boldsymbol{\theta})\right], \tag{10.13}$$

where the second equality follows from the fact that $p(\boldsymbol{\theta} \mid \mathbf{y}) = \frac{p(\mathbf{y} \mid \boldsymbol{\theta}) p(\boldsymbol{\theta})}{p(\mathbf{y})}$ and that $p(\mathbf{y})$ does not depend on $\boldsymbol{\theta}$. Remember that $L^2$-regularised linear regression can be understood as using the cost function (3.48),

$$\widehat{\boldsymbol{\theta}}_{L^2} = \arg\max_{\boldsymbol{\theta}} \left[\ln p(\mathbf{y} \mid \boldsymbol{\theta}) + \lambda \|\boldsymbol{\theta}\|_2^2\right], \tag{10.14}$$

with some regularisation parameter $\lambda$. When comparing (10.14) to (10.13), we realise that if $\ln p(\boldsymbol{\theta}) \propto \|\boldsymbol{\theta}\|_2^2$, the MAP estimate and the $L^2$ regularised estimate of $\boldsymbol{\theta}$ are identical for some value of $\lambda$. With the prior $p(\boldsymbol{\theta})$ in (10.9), that is indeed the case, and the MAP estimate is in that case identical to $\widehat{\boldsymbol{\theta}}_{L^2}$.

This connection between MAP estimates and regularised maximum likelihood estimates holds as long as the regularisation is proportional to the logarithm of the prior. If, for example, we instead chose a Laplace prior for $\boldsymbol{\theta}$, the MAP estimate would be identical to $L^1$ regularisation. In general there are many regularisation methods that can be interpreted as implicitly choosing a certain prior. This connection to regularisation gives another perspective as to why the Bayesian approach is less prone to overfitting.

It is, however, important to note that the connection between the Bayesian approach and the use of regularisation does *not* imply that the two approaches are equivalent. The main point with the Bayesian approach is still that a posterior *distribution* is computed for $\boldsymbol{\theta}$, instead of just a point estimate $\widehat{\boldsymbol{\theta}}$.

## 10.3 The Gaussian Process

We introduced the Bayesian approach as the idea of considering unknown parameters $\boldsymbol{\theta}$ as random variables and consequently learning a posterior distribution $p(\boldsymbol{\theta} \mid \mathbf{y})$ instead of a single value $\widehat{\boldsymbol{\theta}}$. However, the Bayesian idea does not only apply to models with parameters but also to nonparametric models. We will now introduce the Gaussian process, where instead of considering parameters $\boldsymbol{\theta}$ as being random variables, we effectively consider an entire function $f(\mathbf{x})$ to be a stochastic process and compute the posterior $p(f(\mathbf{x}) \mid \mathbf{y})$. The Gaussian process is an interesting and commonly used Bayesian nonparametric model. In a nutshell, it is the Bayesian approach applied to kernel ridge regression (Chapter 9). We will present the Gaussian process as a method for handling regression problems, but it is possible to use it for classification problems as well (similarly to how linear regression can be modified into logistic regression).

In this section we will introduce the fundamentals of the Gaussian process, and thereafter, in Section 10.4, we describe how it can be used as a supervised machine learning method. We will first discuss what a Gaussian process is and thereafter see how we can construct a Gaussian process that connects closely to kernel ridge regression from Section 9.2.

### What Is a Gaussian Process?

A Gaussian process is a specific type of stochastic process. A stochastic process, in turn, is a generalisation of a random variable. Most commonly we think about a stochastic process as some random quantity that evolves over time. Mathematically this corresponds to a collection of random variables $\{z(t) : t \in \mathbb{R}\}$ indexed by time $t$. That is, for each time point $t$, the value of the process $z(t)$ is a random variable. Furthermore, most often, we assume that values at different time points, say $z(t)$ and


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September* 1, 2026, `http://smlbook.org`

$z(s)$, are correlated and that the correlation depends on the time difference. More abstractly, however, we can view $z(t)$ as a random function, where the input to the function is the index variable (time) $t$. With this interpretation, it is possible to generalise the concept of a stochastic process to random functions with arbitrary inputs, $\{f(\mathbf{x}) : \mathbf{x} \in \mathcal{X}\}$, where $\mathcal{X}$ denotes the (possibly high-dimensional) input space. Similarly to above, this means that we view the *function value* $f(\mathbf{x})$ for any input $\mathbf{x}$ as a random variable and that the function values $f(\mathbf{x})$ and $f(\mathbf{x}')$ for inputs $\mathbf{x}$ and $\mathbf{x}'$ are dependent. As we will see below, the dependencies can be used to control certain properties of the function. For instance, if we expect the function to be smooth (varies slowly), then the function values $f(\mathbf{x})$ and $f(\mathbf{x}')$ should be highly correlated if $\mathbf{x}$ is close to $\mathbf{x}'$. This generalisation opens the way for using random functions as priors for unknown functions (such as a regression function) in a Bayesian setting.

To introduce the Gaussian process as a random function, we will start by making the simplifying assumption that the input variable $\mathbf{x}$ is discrete and can take only $q$ different values, $\mathbf{x}_1, \ldots, \mathbf{x}_q$. Hence, the function $f(\mathbf{x})$ is completely characterised by the $q$-dimensional vector $\mathbf{f} = \begin{bmatrix} f_1 & \cdots & f_q \end{bmatrix}^\mathsf{T} = \begin{bmatrix} f(\mathbf{x}_1) & \cdots & f(\mathbf{x}_q) \end{bmatrix}^\mathsf{T}$. We can then model $f(\mathbf{x})$ as a random function by assigning a joint probability distribution to this vector $\mathbf{f}$. In the Gaussian process model, this distribution is the multivariate Gaussian distribution,

$$p(\mathbf{f}) = \mathcal{N}(\mathbf{f}; \boldsymbol{\mu}, \boldsymbol{\Sigma}), \tag{10.15}$$

with mean vector $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$. Let us partition $\mathbf{f}$ into two vectors $\mathbf{f}_1$ and $\mathbf{f}_2$ such that $\mathbf{f} = \begin{bmatrix} \mathbf{f}_1^\mathsf{T} & \mathbf{f}_2^\mathsf{T} \end{bmatrix}^\mathsf{T}$, and $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ similarly, allowing us to write

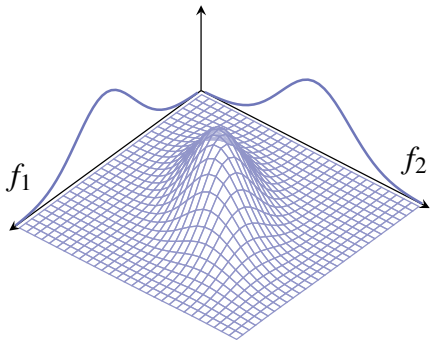
$$p\left(\begin{bmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \end{bmatrix}\right) = \mathcal{N}\left(\begin{bmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \end{bmatrix}; \begin{bmatrix} \boldsymbol{\mu}_1 \\ \boldsymbol{\mu}_2 \end{bmatrix}, \begin{bmatrix} \boldsymbol{\Sigma}_{11} & \boldsymbol{\Sigma}_{12} \\ \boldsymbol{\Sigma}_{21} & \boldsymbol{\Sigma}_{22} \end{bmatrix}\right). \tag{10.16}$$

If some elements of $\mathbf{f}$, let us say the ones in $\mathbf{f}_1$, are observed, the conditional distribution for $\mathbf{f}_2$ given the observation of $\mathbf{f}_1$ is, by Theorem 10.2 in Appendix 10.A,

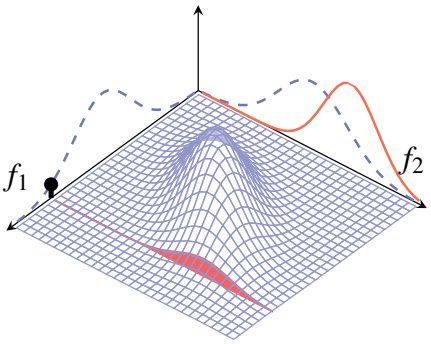
$$p(\mathbf{f}_2 \mid \mathbf{f}_1) = \mathcal{N}\Big(\mathbf{f}_2; \boldsymbol{\mu}_2 + \boldsymbol{\Sigma}_{21}\boldsymbol{\Sigma}_{11}^{-1}(\mathbf{f}_1 - \boldsymbol{\mu}_1), \boldsymbol{\Sigma}_{22} - \boldsymbol{\Sigma}_{21}\boldsymbol{\Sigma}_{11}^{-1}\boldsymbol{\Sigma}_{12}\Big). \tag{10.17}$$

The conditional distribution is nothing but another Gaussian distribution with closed-form expressions for the mean and covariance.

Figure 10.3a shows a two-dimensional example ($\mathbf{f}_1$ is a scalar $f_1$, and $\mathbf{f}_2$ is a scalar $f_2$). For this model we choose the prior where the prior mean and variances for $f_1$ and $f_2$ are the same, that is $\boldsymbol{\mu}_1 = \boldsymbol{\mu}_2$ and $\boldsymbol{\Sigma}_1 = \boldsymbol{\Sigma}_2$. We also assume a positive prior correlation between $\mathbf{f}_1$ and $\mathbf{f}_2$ before any observations have been received, reflecting the smoothness assumption we have about the underlying random function $f(x)$. That is why the multivariate Gaussian distribution in Figure 10.3a is skewed in the diagonal direction. This multivariate Gaussian distribution is now conditioned on an observation of $f_1$, which is reflected in Figure 10.3b. In Figure 10.4, we have plotted the marginal distributions from Figure 10.3. Since $\mathbf{f}_1$ and $\mathbf{f}_2$ are correlated according to the prior, the marginal distribution of $f_2$ is also affected by this observation.

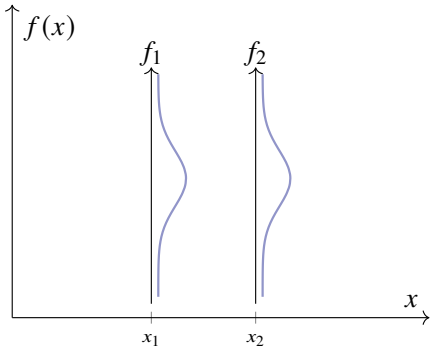



**(a)** A two-dimensional Gaussian distribution for the random variables $f_1$ and $f_2$, with a blue surface plot for the density, and the marginal distribution for each component sketched using blue lines along each axis. Note that the marginal distributions do *not* contain all information about the distribution of $f_1$ and $f_2$, since the covariance information is lacking in that representation.

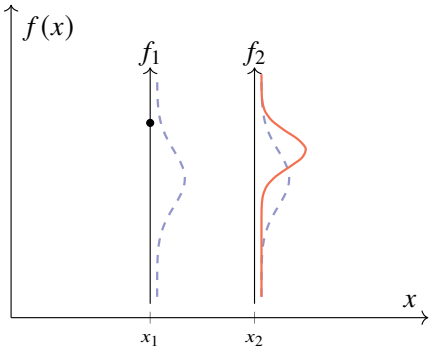



**(b)** The conditional distribution of $f_2$ (red line) when $f_1$ is observed (black dot). The conditional distribution of $f_2$ is given by (10.17), which (apart from a normalising constant) in this graphical representation is the red 'slice' of the joint distribution (blue surface). The marginals of the joint distribution from Figure 10.3a are kept for reference (blue dashed lines).

**Figure 10.3:** A two-dimensional multivariate Gaussian distribution for $f_1$ and $f_2$ in (a), and the conditional distribution for $f_2$, when a particular value of $f_1$ is observed, in (b).




**(a)** The marginal distributions for $f_1$ and $f_2$ from Figure 10.3a.

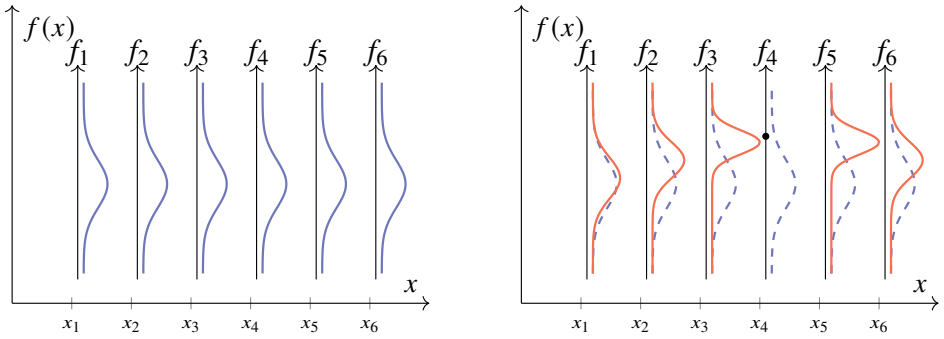



**(b)** The distribution for $f_2$ (red line) when $f_1$ is observed (black dot), as in Figure 10.3b.

**Figure 10.4:** The marginals of the distributions in Figure 10.3, here plotted slightly differently. Note that this more compact plot comes at the cost of missing the information about the covariance between $f_1$ and $f_2$.

In a similar fashion to Figure 10.4, we can plot a six-dimensional multivariate Gaussian distribution by its marginal distributions in Figure 10.5. Also in this model we assume a positive prior correlation between all elements $f_i$ and $f_j$ which decays with the distance between their corresponding inputs $x_i$ and $x_j$. Bear in mind that to fully illustrate the joint distribution for $f_1, \ldots, f_6$, a six-dimensional surface plot would be needed, whereas Figure 10.5a only contains the marginal distributions for each component. We may also condition the six-dimensional distribution underlying Figure 10.5a on an observation of, say, $f_4$. Once again, the conditional distribution is


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September* 1, 2026, `http://smlbook.org`




**(a)** A six-dimensional Gaussian distribution, plotted in the same way as Figure 10.4a, that is, only its marginals are illustrated.

**(b)** The conditional distribution $f_1$, $f_2$, $f_3$, $f_5$, and $f_6$ when $f_4$ is observed (black dot), illustrated by its marginals (red lines) similarly to Figure 10.4b.

**Figure 10.5:** A six-dimensional Gaussian distribution, illustrated in the same fashion as Figure 10.4.

another Gaussian distribution, and the marginal distributions of the five-dimensional random variable $[f_1, f_2, f_3, f_5, f_6]^\mathsf{T}$ are plotted in Figure 10.5b.

In Figures 10.4 and 10.5, we illustrated the marginal distributions for a finite-dimensional multivariate Gaussian random variable. However, we are aiming for the Gaussian process, which is a stochastic process on a continuous space.

The extension of the Gaussian distribution (defined on a finite set) to the Gaussian process (defined on a continuous space) is achieved by replacing the discrete index set $\{1, 2, 3, 4, 5, 6\}$ in Figure 10.5 by a variable $\mathbf{x}$ taking values on a continuous space, for example the real line. We then also have to replace the random variables $f_1, f_2, \ldots, f_6$ with a random function (that is, a stochastic process) $f$ which can be evaluated at any $\mathbf{x}$ as $f(\mathbf{x})$. Furthermore, in the Gaussian multivariate distribution, $\boldsymbol{\mu}$ is a vector with $q$ components, and $\Sigma$ is a $q \times q$ matrix. Instead of having a separate hyperparameter for each element in this mean vector and covariance matrix, in the Gaussian process we replace $\boldsymbol{\mu}$ by a mean *function* $\mu(\mathbf{x})$ into which we can insert any $\mathbf{x}$, and the covariance matrix $\boldsymbol{\Sigma}$ is replaced by a covariance *function* $\kappa(\mathbf{x}, \mathbf{x}')$ into which we can insert any pair $\mathbf{x}$ and $\mathbf{x}'$. This mean function and covariance function we can then parametrise with a few hyperparameters. In these functions we can also encode certain properties that we want the Gaussian process to obey, for example that two function values $f(\mathbf{x}_1)$ and $f(\mathbf{x}_2)$ should be more correlated if $\mathbf{x}_1$ and $\mathbf{x}_2$ are closer to each other than if they are further apart.

From this we can define the Gaussian process. If, for any arbitrary finite set of points $\{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$, it holds that

$$p\left(\begin{bmatrix} f(\mathbf{x}_1) \\ \vdots \\ f(\mathbf{x}_n) \end{bmatrix}\right) = \mathcal{N}\left(\begin{bmatrix} f(\mathbf{x}_1) \\ \vdots \\ f(\mathbf{x}_n) \end{bmatrix}; \begin{bmatrix} \mu(\mathbf{x}_1) \\ \vdots \\ \mu(\mathbf{x}_n) \end{bmatrix}, \begin{bmatrix} \kappa(\mathbf{x}_1, \mathbf{x}_1) & \cdots & \kappa(\mathbf{x}_1, \mathbf{x}_n) \\ \vdots & & \vdots \\ \kappa(\mathbf{x}_n, \mathbf{x}_1) & \cdots & \kappa(\mathbf{x}_n, \mathbf{x}_n) \end{bmatrix}\right), \tag{10.18}$$

then $f$ is a Gaussian process.

That is, with a Gaussian process $f$ and any choice of $\{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$, the vector of function values $[f(\mathbf{x}_1), \ldots, f(\mathbf{x}_n)]$ has a multivariate Gaussian distribution, just like the one in Figure 10.5. Since $\{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ can be chosen arbitrarily from the continuous space on which it lives, the Gaussian process defines a distribution for *all* points in that space. For this definition to make sense, $\kappa(\mathbf{x}, \mathbf{x}')$ has to be such that a positive semidefinite covariance matrix is obtained for any choice of $\{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$.

We will use the notation

$$f \sim \mathcal{GP}\left(\boldsymbol{\mu}, \kappa\right) \tag{10.19}$$

to express that the function $f(\mathbf{x})$ is distributed according to a Gaussian process with mean function $\mu(\mathbf{x})$ and covariance function $\kappa(\mathbf{x}, \mathbf{x}')$. If we want to illustrate a Gaussian process, which we do in Figure 10.6, we can choose $\{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ to correspond to the pixels on the screen or the printer dots on the paper and print the marginal distributions for each $\{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ so that it appears as a continuous line to the eye (despite the fact that we can only actually access the distribution in a finite, but arbitrary, set of points).

It is no coincidence that we use the same symbol $\kappa$ for covariance functions as we used for kernels in Chapter 9. As we will soon discuss, applying the Bayesian approach to kernel ridge regression will result in a Gaussian process where the covariance function is the kernel.

We can also condition the Gaussian process on some observations $\{f(\mathbf{x}_i), \mathbf{x}_i\}_{i=1}^n$, the Gaussian process counterpart to Figures 10.3b, 10.4b, and 10.5b. As usual we stack the observed inputs in $\mathbf{X}$ and let $f(\mathbf{X})$ denote the vector of observed outputs (we assume for now that the observations are made without any noise). We use the notations $\boldsymbol{K}(\mathbf{X}, \mathbf{X})$ and $\boldsymbol{K}(\mathbf{X}, \mathbf{x}_\star)$ as defined by (9.12b) and (9.12c) to write the joint distribution between the observed values $f(\mathbf{X})$ and the test value $f(\mathbf{x}_\star)$ as

$$p\left(\begin{bmatrix} f(\mathbf{x}_\star) \\ f(\mathbf{X}) \end{bmatrix}\right) = \mathcal{N}\left(\begin{bmatrix} f(\mathbf{x}_\star) \\ f(\mathbf{X}) \end{bmatrix}; \begin{bmatrix} \mu(\mathbf{x}_\star) \\ \mu(\mathbf{X}) \end{bmatrix}, \begin{bmatrix} \kappa(\mathbf{x}_\star, \mathbf{x}_\star) & \boldsymbol{K}(\mathbf{X}, \mathbf{x}_\star)^\mathsf{T} \\ \boldsymbol{K}(\mathbf{X}, \mathbf{x}_\star) & \boldsymbol{K}(\mathbf{X}, \mathbf{X}) \end{bmatrix}\right). \tag{10.20}$$

Now, as we have observed $f(\mathbf{X})$, we use the expressions for the Gaussian distribution to write the distribution for $f(\mathbf{x}_\star)$ conditional on the observations of $f(\mathbf{X})$ as

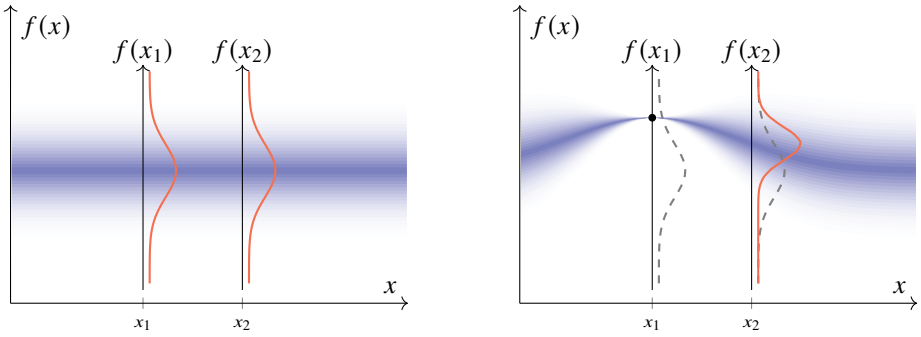
$$\begin{aligned} p\left(f(\mathbf{x}_\star) \,|\, f(\mathbf{X})\right) = \mathcal{N}\big(&f(\mathbf{x}_\star); \mu(\mathbf{x}_\star) + \boldsymbol{K}(\mathbf{X}, \mathbf{x}_\star)^\mathsf{T} \boldsymbol{K}(\mathbf{X}, \mathbf{X})^{-1} \\ &\times \left(f(\mathbf{X}) - \mu(\mathbf{X})\right), \kappa(\mathbf{x}_\star, \mathbf{x}_\star) - \boldsymbol{K}(\mathbf{X}, \mathbf{x}_\star)^\mathsf{T} \boldsymbol{K}(\mathbf{X}, \mathbf{X})^{-1} \boldsymbol{K}(\mathbf{X}, \mathbf{x}_\star)\big), \end{aligned} \tag{10.21}$$

which is another Gaussian distribution for any test input $\mathbf{x}_\star$. We illustrate this in Figure 10.6.

We have now introduced the somewhat abstract concept of a Gaussian process. In some subjects, such as signal processing, so-called white Gaussian processes are common. A white Gaussian process has a white covariance function

$$\kappa(\mathbf{x}, \mathbf{x}') = \mathbb{I}\{\mathbf{x} = \mathbf{x}'\} = \begin{cases} 1 & \text{if } \mathbf{x} = \mathbf{x}', \\ 0 & \text{otherwise,} \end{cases} \tag{10.22}$$


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`




**(a)** A Gaussian process defined on the real line, parameterised by $x$, not conditioned on any observations. The intensity of the blue colour is proportional to the (marginal) density, and the marginal distributions for two test inputs $x_1$ and $x_2$ are shown in red. Similarly to Figure 10.5, we only plot the marginal distribution for each test input, but the Gaussian process defines a full joint distribution for all points on the $x$-axis.

**(b)** The conditional Gaussian process given the observation of $f(x_1)$ in the point $x_1$. The prior distribution from Figure (a) is shown in dashed grey. Note how the conditional distribution adjusts to the observation, both in terms of mean (closer to the observation) and (marginal) variance (smaller in the proximity of the observation, but it remains more or less unchanged in areas far from it).

**Figure 10.6:** A Gaussian process. Figure (a) shows the prior distribution, whereas (b) shows the posterior distribution after conditioning on one observation (black dot).

which implies that $f(\mathbf{x})$ is uncorrelated to $f(\mathbf{x}')$ unless $\mathbf{x} = \mathbf{x}'$. White Gaussian processes are of less use in supervised machine learning, but we will instead have a look at how kernel ridge regression can be turned into a Gaussian process, where the mean function becomes zero, and the covariance function becomes the kernel from Chapter 9.

## Extending Kernel Ridge Regression into a Gaussian Process

An alternative way to obtain the Gaussian process construction is to apply the kernel trick from Section 9.2 to Bayesian linear regression from (10.11). The connection between linear regression, Bayesian linear regression, kernel ridge regression, and the Gaussian process is summarised in Figure 10.7. This will essentially lead us back to (10.21), with the kernel being the covariance function $\kappa(\mathbf{x}, \mathbf{x}')$, and the mean function being $\mu(\mathbf{x}) = 0$.

Let us now repeat the posterior predictive for Bayesian linear regression (10.11), but with two changes. The first change is that we assume the prior mean and covariance for $\boldsymbol{\theta}$ are $\boldsymbol{\mu}_0 = \mathbf{0}$ and $\boldsymbol{\Sigma}_0 = \mathbf{I}$, respectively. This assumption is not strictly needed for our purposes but simplifies the expressions. The second change is that, as in Section 9.1, we introduce non-linear feature transformations $\boldsymbol{\phi}(\mathbf{x})$ of the input variable $\mathbf{x}$ in the linear regression model. We therefore replace $\mathbf{X}$ with $\boldsymbol{\Phi}(\mathbf{X})$ in the notation. Altogether, (10.11) becomes

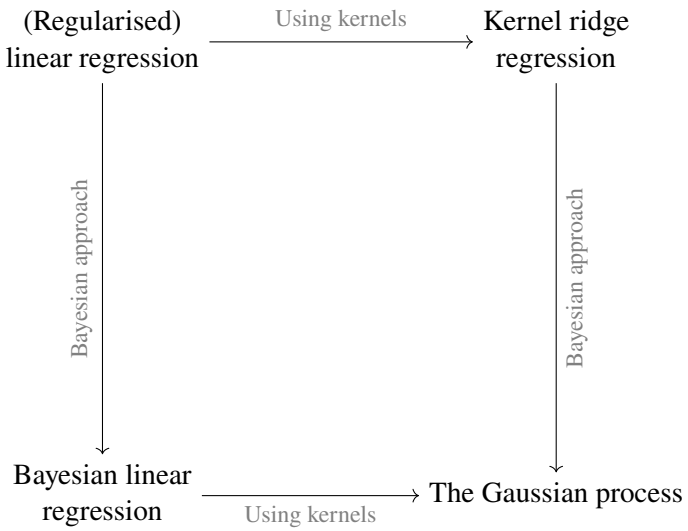



**Figure 10.7:** A graphical summary of the connections between linear regression, Bayesian linear regression, kernel ridge regression, and the Gaussian process.

$$p(f(\mathbf{x}_\star) \,|\, \mathbf{y}) = \mathcal{N}(f(\mathbf{x}_\star); m_\star, s_\star)\,, \tag{10.23a}$$

$$m_\star = \boldsymbol{\phi}(\mathbf{x}_\star)^\mathsf{T} \left(\sigma^2 \mathbf{I} + \boldsymbol{\Phi}(\mathbf{X})^\mathsf{T} \boldsymbol{\Phi}(\mathbf{X})\right)^{-1} \boldsymbol{\Phi}(\mathbf{X})^\mathsf{T} \mathbf{y}, \tag{10.23b}$$

$$s_\star = \boldsymbol{\phi}(\mathbf{x}_\star)^\mathsf{T} \left(\mathbf{I} + \tfrac{1}{\sigma^2} \boldsymbol{\Phi}(\mathbf{X})^\mathsf{T} \boldsymbol{\Phi}(\mathbf{X})\right)^{-1} \boldsymbol{\phi}(\mathbf{x}_\star). \tag{10.23c}$$

In a similar fashion to the derivation of kernel ridge regression, we use the push-through matrix identity $\mathbf{A}(\mathbf{A}^\mathsf{T}\mathbf{A} + \mathbf{I})^{-1} = (\mathbf{A}\mathbf{A}^\mathsf{T} + \mathbf{I})^{-1}\mathbf{A}$ to re-write $m_\star$ with the aim of having $\boldsymbol{\phi}(\mathbf{x})$ only appearing through inner products:

$$m_\star = \boldsymbol{\phi}(\mathbf{x}_\star)^\mathsf{T} \boldsymbol{\Phi}(\mathbf{X})^\mathsf{T} \left(\sigma^2 \mathbf{I} + \boldsymbol{\Phi}(\mathbf{X}) \boldsymbol{\Phi}(\mathbf{X})^\mathsf{T}\right)^{-1} \mathbf{y}. \tag{10.24a}$$

To re-write $s_\star$ in a similar fashion, we have to use the matrix inversion lemma $(\mathbf{I} - \mathbf{U}\mathbf{V})^{-1} = \mathbf{I} - \mathbf{U}(\mathbf{I} + \mathbf{V}\mathbf{U})^{-1}\mathbf{V}$ (which holds for any matrices $\mathbf{U}, \mathbf{V}$ of compatible dimensions):

$$s_\star = \boldsymbol{\phi}(\mathbf{x}_\star)^\mathsf{T} \boldsymbol{\phi}(\mathbf{x}_\star) - \boldsymbol{\phi}(\mathbf{x}_\star)^\mathsf{T} \boldsymbol{\Phi}(\mathbf{X})^\mathsf{T} \left(\sigma^2 \mathbf{I} + \boldsymbol{\Phi}(\mathbf{X}) \boldsymbol{\Phi}(\mathbf{X})^\mathsf{T}\right)^{-1} \boldsymbol{\Phi}(\mathbf{X}) \boldsymbol{\phi}(\mathbf{x}_\star). \tag{10.24b}$$

Analogously to the derivation of kernel ridge regression as in (9.12), we are now ready to apply the kernel trick and replace all instances of $\boldsymbol{\phi}(\mathbf{x})^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}')$ with a kernel $\kappa(\mathbf{x}, \mathbf{x}')$. With the same notation as in (9.12), we get

$$m_\star = \boldsymbol{K}(\mathbf{X}, \mathbf{x}_\star)^\mathsf{T} \left(\sigma^2 \mathbf{I} + \boldsymbol{K}(\mathbf{X}, \mathbf{X})\right)^{-1} \mathbf{y}, \tag{10.25a}$$

$$s_\star = \kappa(\mathbf{x}_\star, \mathbf{x}_\star) - \boldsymbol{K}(\mathbf{X}, \mathbf{x}_\star)^\mathsf{T} \left(\sigma^2 \mathbf{I} + \boldsymbol{K}(\mathbf{X}, \mathbf{X})\right)^{-1} \boldsymbol{K}(\mathbf{X}, \mathbf{x}_\star). \tag{10.25b}$$


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

The posterior predictive that is defined by (10.23a) and (10.25) is the Gaussian process model again, identical to (10.21) if $\mu(\mathbf{x}_\star) = 0$ and $\sigma^2 = 0$. The reason for $\mu(\mathbf{x}_\star) = 0$ is that we made this derivation starting with $\boldsymbol{\mu}_0 = 0$. When we derived (10.21), we assumed that we observed $f(\mathbf{x}_\star)$ (rather than $y_\star = f(\mathbf{x}_\star) + \varepsilon$), which is the reason why $\sigma^2 = 0$ in (10.21). The Gaussian process is thus a kernel version of Bayesian linear regression, much like kernel ridge regression is a kernel version of (regularised) linear regression, as illustrated in Figure 10.7. In order to see the connection to kernel ridge regression, note that (10.25a) is identical to (9.14) with $\sigma^2 = n\lambda$. It is, however, also important to note the difference in that there is no counterpart to (10.25b) for kernel ridge regression, simply because kernel ridge regression does not predict a probability distribution.

The fact that the kernel plays the role of a covariance function in the Gaussian process gives us another interpretation of the kernel in addition to the ones in Section 9.4, namely that the kernel $\kappa(\mathbf{x}, \mathbf{x}')$ determines how strong the correlation between $f(\mathbf{x})$ and $f(\mathbf{x}')$ is assumed to be.

> ***Time to reflect 10.1*** *Verify that you retrieve Bayesian linear regression when using the linear kernel* (9.23) *in the Gaussian process. Why is that?*

It is common not to write out the addition of $\sigma^2\mathbf{I}$ to the Gram matrix $\boldsymbol{K}(\mathbf{X}, \mathbf{X})$ but instead to add a white noise kernel (10.22) multiplied by $\sigma^2$ to the original kernel: as $\tilde{\kappa}(\mathbf{x}, \mathbf{x}') = \kappa(\mathbf{x}, \mathbf{x}') + \sigma^2\mathbb{I}\{\mathbf{x}, \mathbf{x}'\}$. We can then replace $\sigma^2\mathbf{I} + \boldsymbol{K}(\mathbf{X}, \mathbf{X})$ with $\tilde{\boldsymbol{K}}(\mathbf{X}, \mathbf{X})$, where $\tilde{\boldsymbol{K}}$ is built up using $\tilde{\kappa}(\mathbf{x}, \mathbf{x}')$ rather than $\kappa(\mathbf{x}, \mathbf{x}')$. In this notation, (10.25) simplifies to

$$m_\star = \tilde{\boldsymbol{K}}(\mathbf{X}, \mathbf{x}_\star)^\mathsf{T}\tilde{\boldsymbol{K}}(\mathbf{X}, \mathbf{X})^{-1}\mathbf{y}, \tag{10.26a}$$

$$s_\star = \tilde{\kappa}(\mathbf{x}_\star, \mathbf{x}_\star) - \tilde{\boldsymbol{K}}(\mathbf{X}, \mathbf{x}_\star)^\mathsf{T}\tilde{\boldsymbol{K}}(\mathbf{X}, \mathbf{X})^{-1}\tilde{\boldsymbol{K}}(\mathbf{X}, \mathbf{x}_\star) - \sigma^2. \tag{10.26b}$$

As in Bayesian linear regression, if we are interested in the posterior predictive $p(y_\star \mid \mathbf{y})$ instead of $p(f(\mathbf{x}_\star) \mid \mathbf{y})$, we add $\sigma^2$ to the variance of the prediction; see (10.11d). We summarise the Gaussian process as Method 10.2.

---

**Data:** Training data $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^n$, kernel $\kappa(\mathbf{x}, \mathbf{x}')$, noise variance $\sigma^2$ and test input $\mathbf{x}_\star$

**Result:** Posterior predictive $p(f(\mathbf{x}_\star) \mid \mathbf{y}) = \mathcal{N}(f(\mathbf{x}_\star); m_\star, s_\star)$

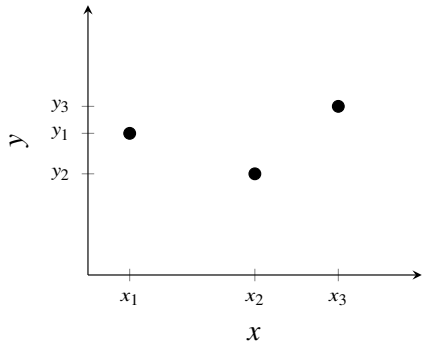
1 Compute $m_\star$ and $s_\star$ according to (10.25)

---

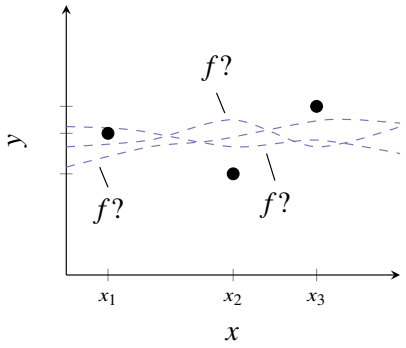
**Method 10.2:** Gaussian process regression.

## A Nonparametric Distribution over Functions

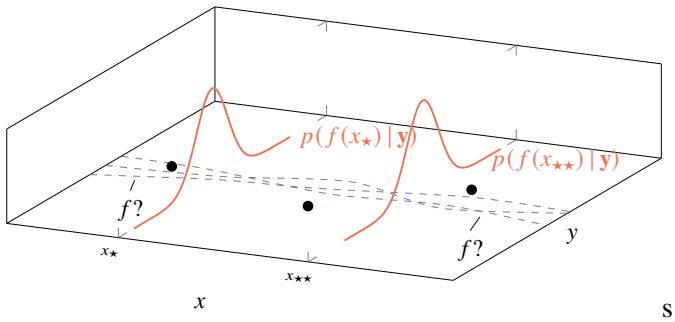
As a supervised machine learning tool, we use the Gaussian process for making predictions, that is, computing the posterior predictive $p(f(\mathbf{x}_\star) \mid \mathbf{y})$ (or $p(y_\star \mid \mathbf{y})$).




**(a)** The training data $\{x_i, y_i\}_{i=1}^3$ of a regression problem is given to us.




**(b)** The underlying assumption when we do regression is that there exists *some* function $f$, which describes the data as $y_i = f(x_i) + \varepsilon$. It is unknown to us, but the purpose of regression (no matter which method is used) is to determine $f$.




**(c)** The Gaussian process defines a distribution over $f$. We can condition that distribution on training data (that is, learning) and access it for any input, say $x_\star$ and $x_{\star\star}$. That is, we make a prediction for $x_\star$ and $x_{\star\star}$. The Gaussian process gives us a Gaussian distribution for $f(x_\star)$ and $f(x_{\star\star})$, illustrated by solid red lines. That distribution is heavily influenced by the choice of kernel, which is a design choice in the Gaussian process.

**Figure 10.8:** The Gaussian process defines a distribution over functions, which we can condition on training data and access at arbitrary points (such as $x_\star$ and $x_{\star\star}$) in order to compute predictions.

However, unlike most other methods, which only deliver a point prediction $\widehat{y}(\mathbf{x}_\star)$, the posterior predictive is a distribution. Since we can compute the posterior predictive for any $\mathbf{x}_\star$, the Gaussian process actually defines a *distribution over functions*, as we illustrate in Figure 10.8.

Much like we could derive a connection between Bayesian linear regression and $L^2$ regularised linear regression, we have also seen a similar connection between the Gaussian process and kernel ridge regression. If we only consider the mean $m_\star$ of the posterior predictive, we recover kernel ridge regression. To take full advantage of the Bayesian perspective, we also have to consider the posterior predictive variance $s_\star$. With most kernels, the predictive variance is smaller if there is a training data point nearby and larger if the closest training data point is distant. Hence, the predictive variance provides a quantification of the 'uncertainty' in the prediction. Altogether,


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

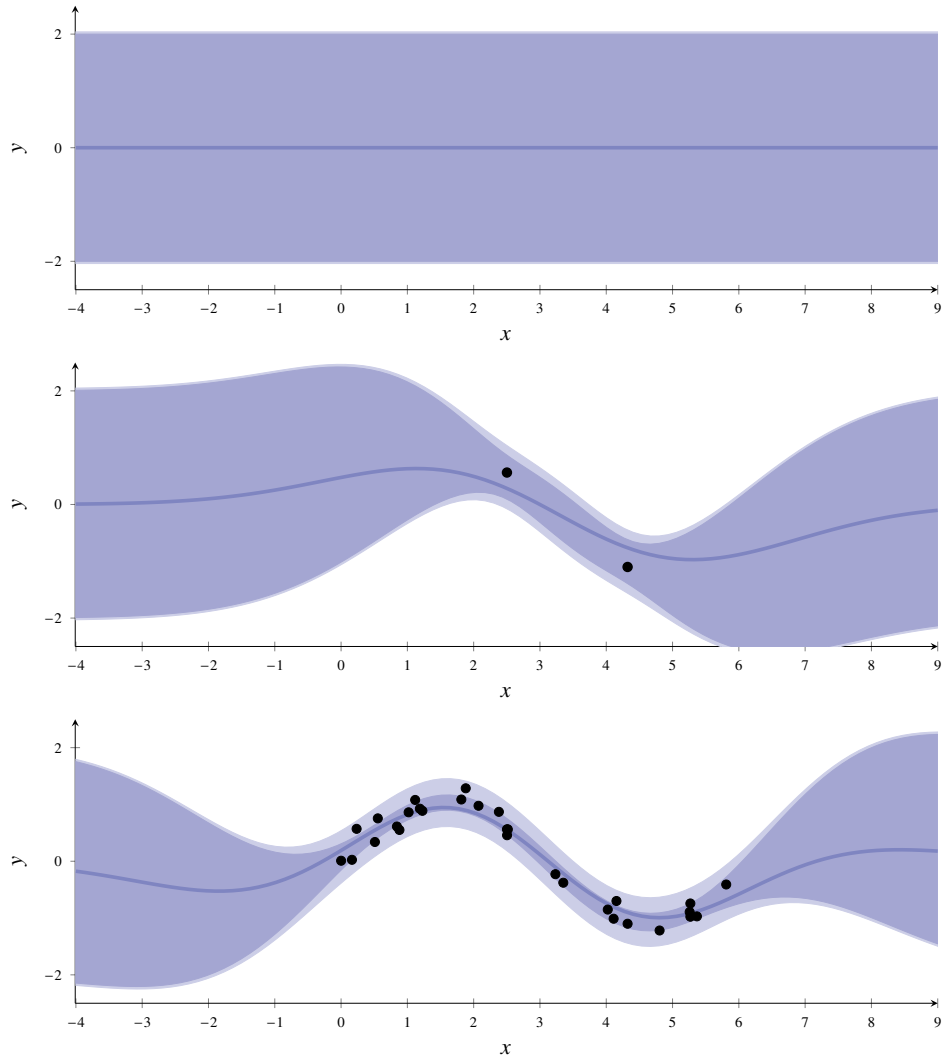



**Figure 10.9:** The Gaussian process as a supervised machine learning method: we can learn (that is, compute (10.23a) and (10.25)) the posterior predictive for $f(x_\star)$ and $y_\star$ (shaded blue; darker blue for two standard deviations for $p(y_\star \mid \mathbf{y})$, lighter blue for two standard deviations for $p(f(x_\star) \mid \mathbf{y})$, and solid blue line for the mean) learned from 0 (upper), 2 (middle), and 30 (bottom) observations (black dots). We see how the model adapts to training data, and note in particular that the variance shrinks in the regions where observations are made but remains larger in regions where no observations are made.

the Gaussian process is another useful tool for regression problems, as we illustrate in Figure 10.9.

### Drawing Samples from a Gaussian Process

When computing a prediction of $f(\mathbf{x}_\star)$ for a single input point $\mathbf{x}_\star$, the posterior predictive $p(f(\mathbf{x}_\star) \mid \mathbf{y})$ captures all information the Gaussian process has about

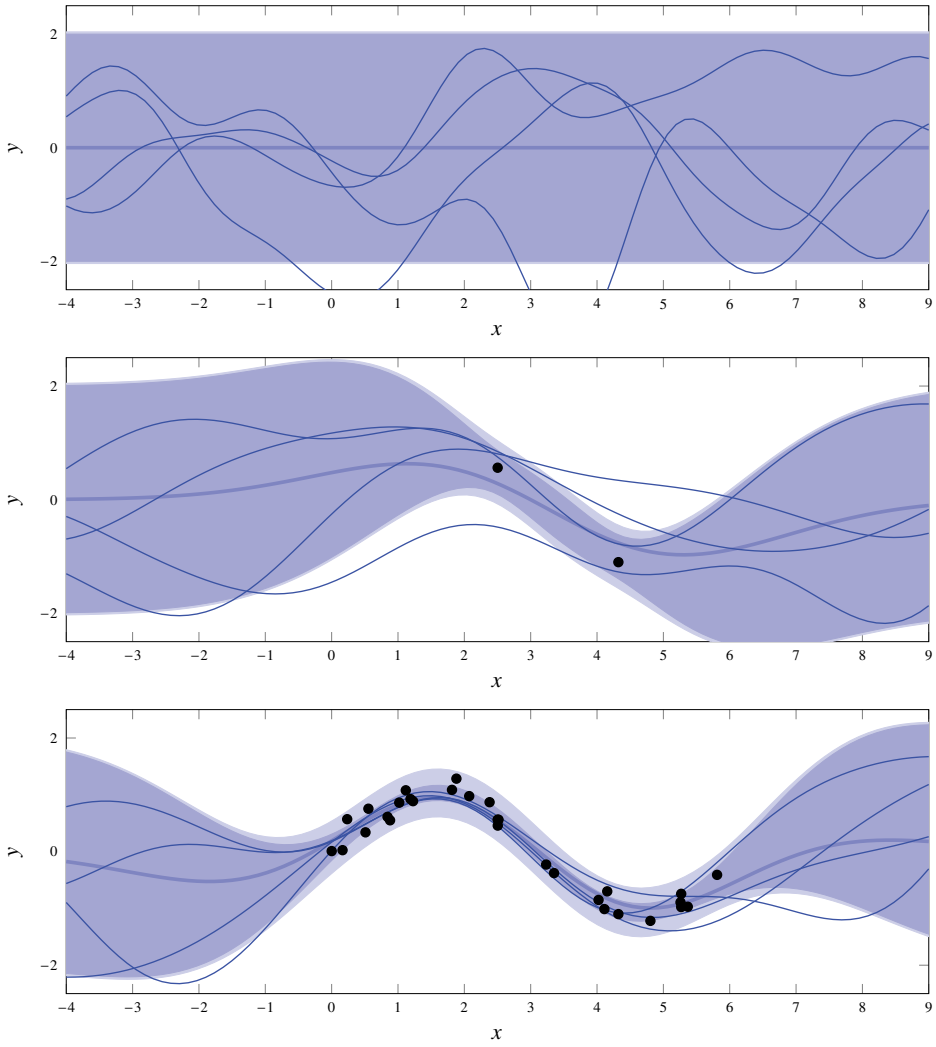



**Figure 10.10:** Figure 10.9 again, this time appended also with samples from $p(f(x_\star) \,|\, \mathbf{y})$.

$f(\mathbf{x}_\star)$. However, if we are interested not only in predicting $f(\mathbf{x}_\star)$, but also $f(\mathbf{x}_\star + \boldsymbol{\delta})$, the Gaussian process contains more information than is present in the two posterior predictive distributions $p(f(\mathbf{x}_\star) \,|\, \mathbf{y})$ and $p(f(\mathbf{x}_\star + \boldsymbol{\delta}) \,|\, \mathbf{y})$ separately. This is because the Gaussian process also contains information about the correlation between the function values $f(\mathbf{x}_\star)$ and $f(\mathbf{x}_\star + \boldsymbol{\delta})$, and the pitfall is that $p(f(\mathbf{x}_\star) \,|\, \mathbf{y})$ and $p(f(\mathbf{x}_\star + \boldsymbol{\delta}) \,|\, \mathbf{y})$ are only the marginal distributions of the joint distribution $p(f(\mathbf{x}_\star), f(\mathbf{x}_\star + \boldsymbol{\delta}) \,|\, \mathbf{y})$, just as the joint distribution in Figure 10.3 contains more information than only the marginal distributions in Figure 10.4.

If we are interested in computing predictions for a larger set of input values, it can be rather cumbersome to grasp and visualise the resulting high-dimensional posterior predictive distribution. An useful alternative can therefore be to visualise the Gaussian process posterior by samples from it. Technically this simply amounts to drawing a sample from the posterior predictive distribution, which we illustrate in Figure 10.10.


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September* 1, 2026, `http://smlbook.org`

## 10.4 Practical Aspects of the Gaussian Process

When using the Gaussian process as a method for supervised machine learning, there are a few important design choices left to the user. Like the methods presented in Chapter 9, the Gaussian process is a kernel method, and the choice of kernel is very important. Most kernels contain a few hyperparameters, which also have to be chosen. That choice can be done by maximising the marginal likelihood, which we will discuss now.

### Kernel Choice

Since the Gaussian process can be understood as the Bayesian version of kernel ridge regression, the Gaussian process also requires a positive semidefinite kernel. Any of the positive semidefinite kernels presented in Section 9.4 can also be used for Gaussian processes.

Among all kernel methods presented in this book, the Gaussian process could be the method where the exact choice of kernel has the biggest impact since the Gaussian posterior predictive $p(f(\mathbf{x}_\star \mid \mathbf{y}))$ has a mean and a variance, both of which are heavily affected by the choice of kernel. It is therefore important to make a good choice, and besides the discussion in Section 9.4, it can also be instructive to visually compare different kernel choices as in Figure 10.11, at least when working with one-dimensional problems where that visualisation is possible. For example, as can be seen from Figure 10.11, the squared exponential and the Matérn kernel with $\nu = \frac{1}{2}$ correspond to drastically different assumptions about the smoothness of $f(\mathbf{x})$.

A positive semidefinite kernel $\kappa(\mathbf{x}, \mathbf{x}')$ remains a positive semidefinite kernel when multiplied with a positive constant $\varsigma^2$, $\varsigma^2\kappa(\mathbf{x}, \mathbf{x}')$. For the kernel methods in Chapter 9, such a scaling has effectively no impact beyond what can be achieved by tuning the regularisation parameter $\lambda$. However, for the Gaussian process, it is important to choose a constant $\varsigma^2$ wisely since it becomes an important factor in the predicted variance.

In the end, the kernel, with all its hyperparameters including $\varsigma^2$ and $\sigma^2$, is a design choice left to the machine learing engineer. In the Bayesian perspective, the kernel is an important part of the prior that implements crucial assumptions about the function $f$.

### Hyperparameter Tuning

Most kernels $\kappa(\mathbf{x}, \mathbf{x}')$ contain a few hyperparameters, such as $\ell$ and $\alpha$ and some possible scaling $\varsigma^2$, in addition to the noise variance $\sigma^2$. Some of the hyperparameters might be possible to set manually, for example if they have a natural interpretation, but most often some hyperparameters are left as tuning parameters for the user. We jointly refer to all those hyperparameters that need to be chosen as $\eta$, meaning that $\eta$ could be a vector. Cross-validation can indeed be used for this purpose, but the Bayesian approach also comes with the option to maximise the marginal likelihood $p(\mathbf{y})$ as a

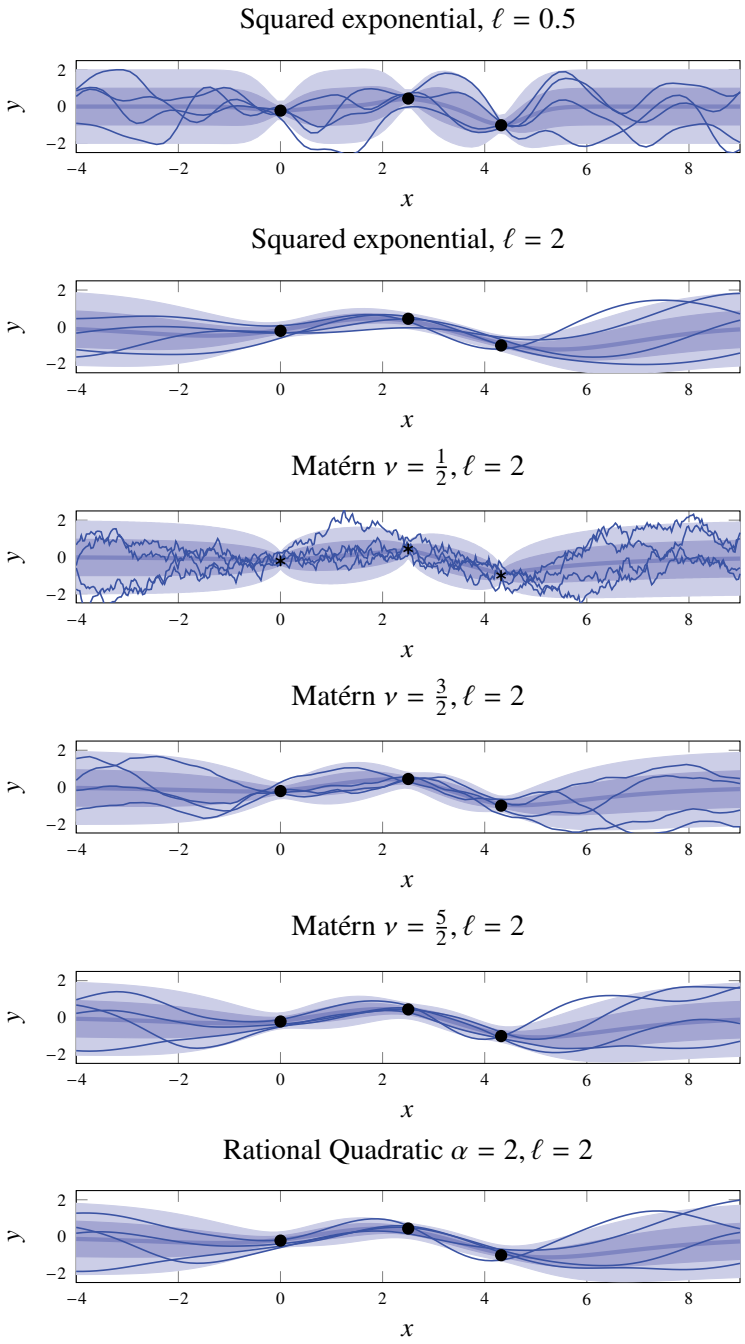



**Figure 10.11:** The Gaussian process applied to the same data with different kernels and hyperparameters, in order to illustrate what assumptions are made by the different kernels. The data is marked with black dots, and the Gaussian process is illustrated by its mean (blue thick line), variance (blue areas; darker for one standard deviation and lighter for two standard deviations), and samples (thin lines).


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September* 1, 2026, `http://smlbook.org`

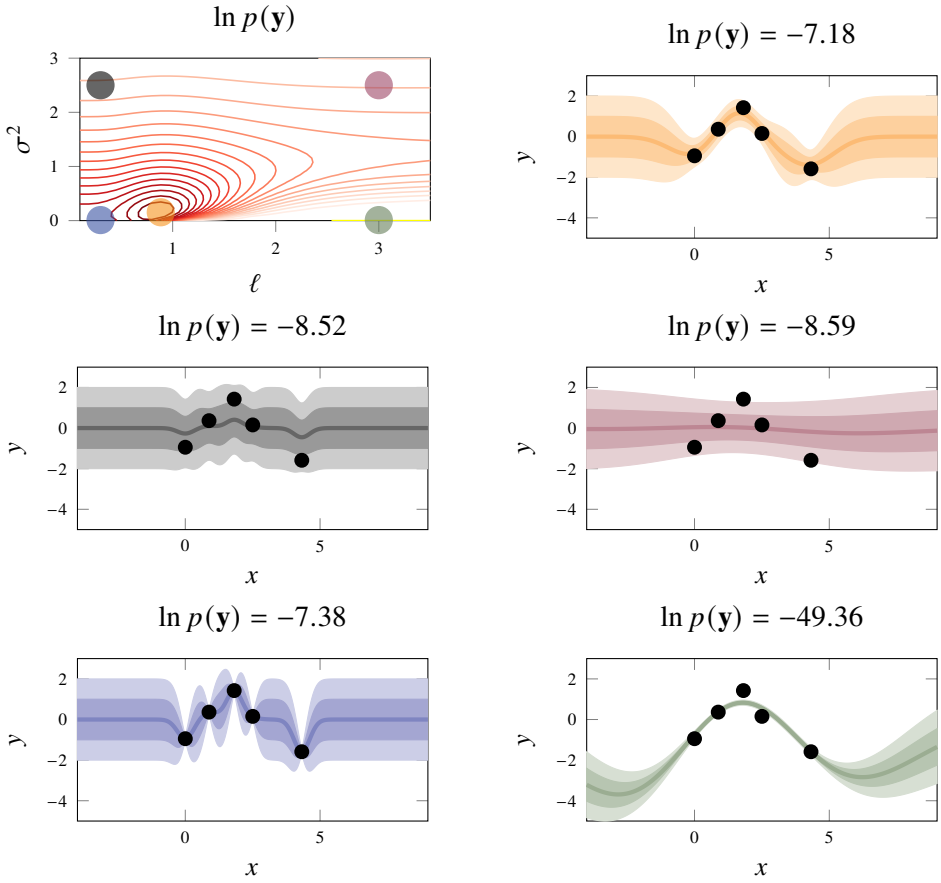



**Figure 10.12:** To choose hyperparameters $\eta = (\ell, \sigma^2)$ for the Gaussian process kernel, the marginal likelihood $p_\eta(\mathbf{y})$ can be maximised. For a given dataset of five points (black dots) and the squared exponential kernel, the landscape of the logarithm of the marginal likelihood (as a function of the hyperparameters lengthscale $\ell$ and noise variance $\sigma^2$) is shown as a contour plot in the upper left panel. Each point in that plot corresponds to a certain selection of the hyperparameters $\ell, \sigma^2$. For five such points (grey, purple, blue, green, and orange dots), the corresponding Gaussian process is shown in separate panels with the same colour. Note that the orange dot is located at the maximum of $p_\eta(\mathbf{y})$. The orange upper right plot therefore corresponds to a Gaussian process where the hyperparameters have been maximised using marginal likelihood. It is clear that optimising $p_\eta(\mathbf{y})$ does *not* mean selecting hyperparameters such that the Gaussian process follows the data as closely as possible (as the blue one does).

method for selecting hyperparameters, as in (10.4). To emphasise how $\eta$ enters into the marginal likelihood, we add the subscript $\eta$ to all terms that depends on it. From the construction of the Gaussian process, we have that $p_\eta(\mathbf{y}) = \mathcal{N}\big(\mathbf{y}; 0, \tilde{\boldsymbol{K}}_\eta(\mathbf{X}, \mathbf{X})\big)$, and consequently

$$\ln p_\eta(\mathbf{y}) = -\frac{1}{2}\mathbf{y}^\mathsf{T}\tilde{\boldsymbol{K}}_\eta(\mathbf{X}, \mathbf{X})^{-1}\mathbf{y} - \frac{1}{2}\ln\det\big(\tilde{\boldsymbol{K}}_\eta(\mathbf{X}, \mathbf{X})\big) - \frac{n}{2}\log 2\pi. \tag{10.27}$$

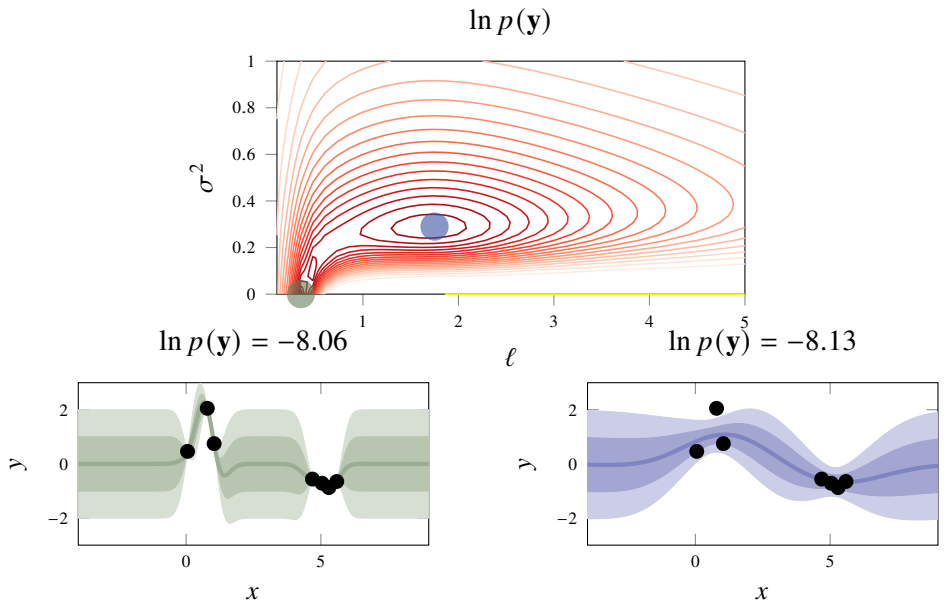



**Figure 10.13:** The landscape of $p(\mathbf{y})$ may have several local maxima. In this case there is one local maximum at the blue dot, with relatively large noise variance and length scale. There is also another local maximum, which also is the global one, at the green dot, with much less noise variance and a shorter lengthscale. There is also a third local maximum in between (not shown). It is not uncommon that the different maxima provide different 'interpretations' of the data. As a machine learning engineer, it is important to be aware that this can happen; the green one does indeed optimise the marginal likelihood, but the blue one can also be practically useful.

In other words, the hyperparameters of the Gaussian process kernel can be chosen by solving the optimisation problem of maximising (10.27) with respect to $\eta$. If using this approach, solving the optimisation problem can be seen as a part of learning the Gaussian process, which is illustrated in Figure 10.12.

When selecting hyperarameters $\eta$ of a kernel, it is important to be aware that (10.27) (as a function of the hyperparameters) may have multiple local maxima, as we illustrate in Figure 10.13. It is therefore important to carefully choose the initialisation of the optimisation procedure. The challenge with local maxima is not unique to using the marginal likelihood approach but can also arise when cross-validation is used to choose hyperparameters.

We conclude this part about the Gaussian process by applying it to the car stopping distance problem in Example 10.3.


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September* 1, 2026, `http://smlbook.org`

**Example 10.3 Car stopping distances**

We again consider the car stopping distance problem from Example 2.2. We have already discussed the application of the other kernel methods, kernel ridge regression, and support vector regression in Examples 9.2 and 9.3, respectively, both in Chapter 9. Since the results in the previous examples looked reasonable, we use the same kernel and hyperparameters again, meaning that we have

$$\tilde{\kappa}(x, x') = \exp\left(-\frac{|x - x'|^2}{2\ell^2}\right) + (1 + xx')^2 + \sigma^2 \mathbb{I}\{x = x'\},$$

with $\ell = 3$ and $\sigma^2 = \lambda n$ (with $\lambda = 0.01$ and $n = 62$ data points). However, we also have the option to introduce yet another hyperparameter $\varsigma^2$ as $\varsigma^2 \tilde{\kappa}(\mathbf{x}, \mathbf{x}')$. In the top part of Figure 10.14, we use $\varsigma^2 = 2^2$ and $\varsigma^2 = 40^2$ to illustrate the fundamental impact that $\varsigma^2$ has on the posterior predictive. (Note that only the variance, and not the mean, is affected by $\varsigma^2$. This can be confirmed by the fact that $\varsigma^2$ cancels algebraically in (10.26a) but not in (10.26b).)

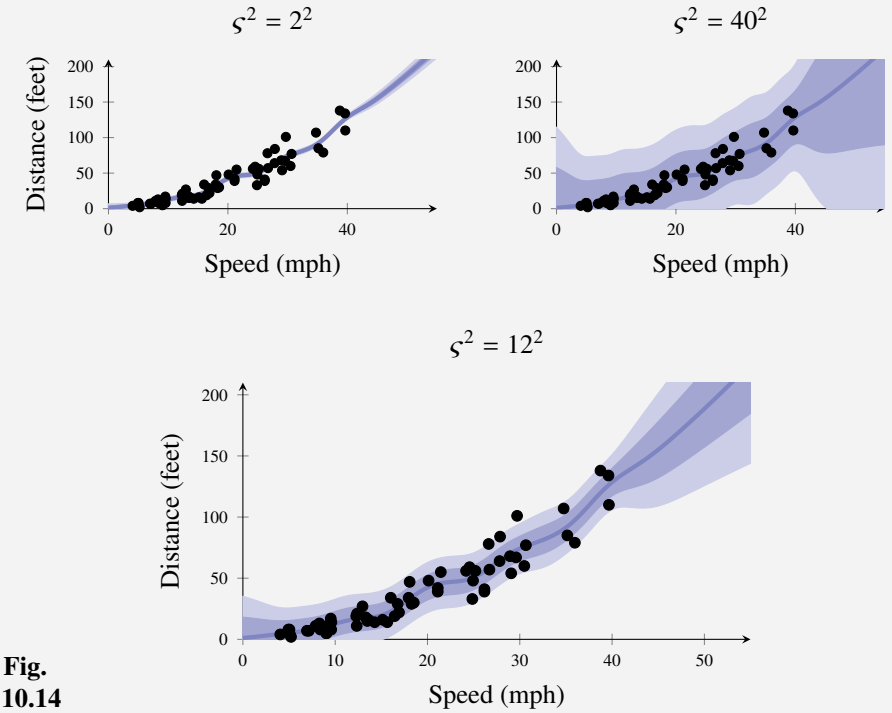



**Fig. 10.14**

To select $\varsigma^2$, we therefore maximise the marginal likelihood with respect to it, which gives us $\varsigma^2 = 12^2$, as shown in the bottom panel of Figure 10.14. Indeed, it seems to have a very reasonable variance ('uncertainty') in the posterior predictive distribution.

## 10.5 Other Bayesian Methods in Machine Learning

Besides introducing the Bayesian approach in general, this chapter contains the Bayesian treatment of linear regression (Bayesian linear regression, Section 10.2) and kernel ridge regression (Gaussian processes, Section 10.3–10.4). The Bayesian approach is, however, applicable to all methods that somehow learn a model from training data. The reason for the selection of methods in this chapter is frankly that Bayesian linear regression and Gaussian process regression are among the few Bayesian supervised machine learning methods where the posterior and/or the posterior predictive are easy to compute.

Most often, the Bayesian approach requires numerical integration routines as well as numerical methods for representing the posterior distribution (the posterior does not have to be a Gaussian or any other standard distribution) when applied to various models. There are two major families of such numerical methods, called variational inference and Monte Carlo methods. The idea of variational inference is to approximate probability distributions in such a way that the problem becomes sufficiently tractable, whereas Monte Carlo methods represent probability distributions using random samples from them.

The Gaussian process model is an example of a method belonging to the family of Bayesian nonparametric methods. Another method in that family is the Dirichlet process, which can be used for the unsupervised clustering problem (see Section 11.3), with no need to specify the number of clusters beforehand.

Another direction is the Bayesian approach applied to deep learning, often referred to as Bayesian deep learning. In short, Bayesian deep learning amounts to computing the posterior $p(\boldsymbol{\theta} \mid \mathbf{y})$, instead of only parameter values $\widehat{\boldsymbol{\theta}}$. In doing so, stochastic gradient descent has to be replaced with either some version of variational inference or some Monte Carlo method. Due to the massive number of parameters often used in deep learning, that is a computationally challenging problem.

## 10.6 Further Reading

The Bayesian approach has a long history within statistics. The name originates from Thomas Bayes and his 1763 posthumously published work 'An Essay towards solving a Problem in the Doctrine of Chances', but Pierre-Simon Laplace also made significant contributions to the idea in the late 18th and early 19th century. For an overview of its use in statistics and its historical controversies, we refer to Efron and Hastie (2016, Part I).

A relatively short review article on modern Bayesian machine learning with many suggestions for further reading is Ghahramani (2015). There are also several textbooks on the modern use of the Bayesian approach in machine learning, including Barber (2012), Gelman et al. (2014), and Rogers and Girolami (2017), and for some aspects also C. M. Bishop (2006) and Murphy (2012).

Gaussian processes are covered in depth by the textbook Rasmussen and Williams (2006). Other Bayesian nonparametric models in general and the Dirichlet process in particular are introduced in Gershman and Blei (2012), Ghahramani (2013), and Hjort et al. (2010).

As mentioned above, the Bayesian approach often requires more advanced computational methods not discussed in this chapter. Two entry points for further studies of variational inference are C. M. Bishop (2006, Chapter 10) and Blei et al. (2017). Introductions to Monte Carlo methods are found in Owen (2013), Robert and Casella (2004), and Gelman et al. (2014, Part III).

Although a very recent research topic, the idea of Bayesian learning of neural networks was laid out already in the 90s (Neal 1996). Some more recent contributions include Blundell et al. (2015), Dusenberry et al. (2020), Fort et al. (2019), Kendall and Gal (2017), and R. Zhang et al. (2020).

## 10.A The Multivariate Gaussian Distribution

This appendix contains some results on the multivariate Gaussian distribution that are essential for Bayesian linear regression and the Gaussian process. Figure 10.15 summarises how they relate to each other.

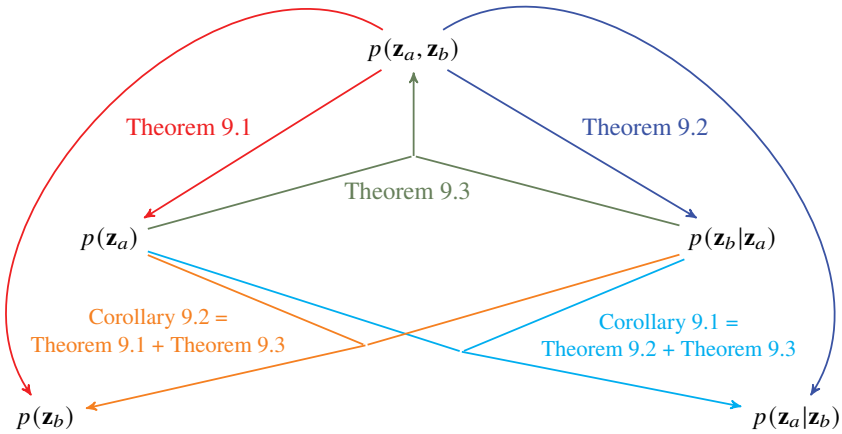



**Figure 10.15:** A graphical summary of how Theorems 10.1–10.3 and Corollaries 10.1–10.2 relate to each other. In all results, $\mathbf{z}_a$ and $\mathbf{z}_b$ are dependent multivariate Gaussian random variables.

**Theorem 10.1 (Marginalisation)** *Partition the Gaussian random vector* $\mathbf{z} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ *according to*

$$\mathbf{z} = \begin{pmatrix} \mathbf{z}_a \\ \mathbf{z}_b \end{pmatrix}, \qquad \boldsymbol{\mu} = \begin{pmatrix} \boldsymbol{\mu}_a \\ \boldsymbol{\mu}_b \end{pmatrix}, \qquad \boldsymbol{\Sigma} = \begin{pmatrix} \boldsymbol{\Sigma}_{aa} & \boldsymbol{\Sigma}_{ab} \\ \boldsymbol{\Sigma}_{ba} & \boldsymbol{\Sigma}_{bb} \end{pmatrix}. \tag{10.28}$$

*The marginal distribution* $p(\mathbf{z}_a)$ *is then given by*

$$p(\mathbf{z}_a) = \mathcal{N}\big(\mathbf{z}_a; \boldsymbol{\mu}_a, \boldsymbol{\Sigma}_{aa}\big)\,. \tag{10.29}$$

**Theorem 10.2 (Conditioning)** *Partition the Gaussian random vector* $\mathbf{z} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ *according to*

$$\mathbf{z} = \begin{pmatrix} \mathbf{z}_a \\ \mathbf{z}_b \end{pmatrix}, \qquad \boldsymbol{\mu} = \begin{pmatrix} \boldsymbol{\mu}_a \\ \boldsymbol{\mu}_b \end{pmatrix}, \qquad \boldsymbol{\Sigma} = \begin{pmatrix} \boldsymbol{\Sigma}_{aa} & \boldsymbol{\Sigma}_{ab} \\ \boldsymbol{\Sigma}_{ba} & \boldsymbol{\Sigma}_{bb} \end{pmatrix}. \tag{10.30}$$

*The conditional distribution* $p(\mathbf{z}_a \mid \mathbf{z}_b)$ *is then given by*

$$p(\mathbf{z}_a \mid \mathbf{z}_b) = \mathcal{N}\Big(\mathbf{z}_a; \boldsymbol{\mu}_{a|b}, \boldsymbol{\Sigma}_{a|b}\Big)\,, \tag{10.31a}$$

*where*

$$\boldsymbol{\mu}_{a|b} = \boldsymbol{\mu}_a + \boldsymbol{\Sigma}_{ab}\boldsymbol{\Sigma}_{bb}^{-1}(\mathbf{z}_b - \boldsymbol{\mu}_b), \tag{10.31b}$$

$$\boldsymbol{\Sigma}_{a|b} = \boldsymbol{\Sigma}_{aa} - \boldsymbol{\Sigma}_{ab}\boldsymbol{\Sigma}_{bb}^{-1}\boldsymbol{\Sigma}_{ba}. \tag{10.31c}$$

**Theorem 10.3 (Affine transformation)** *Assume that* $\mathbf{z}_a$ *as well as* $\mathbf{z}_b \mid \mathbf{z}_a$ *are both Gaussian distributed according to*

$$p(\mathbf{z}_a) = \mathcal{N}\big(\mathbf{z}_a; \boldsymbol{\mu}_a, \boldsymbol{\Sigma}_a\big)\,, \tag{10.32a}$$

$$p(\mathbf{z}_b \mid \mathbf{z}_a) = \mathcal{N}\big(\mathbf{z}_b; \mathbf{A}\mathbf{z}_a + \mathbf{b}, \boldsymbol{\Sigma}_{b|a}\big)\,. \tag{10.32b}$$

*Then the joint distribution of* $\mathbf{z}_a$ *and* $\mathbf{z}_b$ *is*

$$p(\mathbf{z}_a, \mathbf{z}_b) = \mathcal{N}\left(\begin{bmatrix} \mathbf{z}_a \\ \mathbf{z}_b \end{bmatrix}; \begin{bmatrix} \boldsymbol{\mu}_a \\ \mathbf{A}\boldsymbol{\mu}_a + \mathbf{b} \end{bmatrix}, \mathbf{R}\right) \tag{10.33a}$$

*with*

$$\mathbf{R} = \begin{bmatrix} \boldsymbol{\Sigma}_a & \boldsymbol{\Sigma}_a\mathbf{A}^\mathsf{T} \\ \mathbf{A}\boldsymbol{\Sigma}_a & \boldsymbol{\Sigma}_{b|a} + \mathbf{A}\boldsymbol{\Sigma}_a\mathbf{A}^\mathsf{T} \end{bmatrix}. \tag{10.33b}$$

**Corollary 10.1 (Affine transformation – conditional)** *Assume that* $\mathbf{z}_a$ *as well as* $\mathbf{z}_b \mid \mathbf{z}_a$ *are both Gaussian distributed according to*

$$p(\mathbf{z}_a) = \mathcal{N}\left(\mathbf{z}_a; \boldsymbol{\mu}_a, \boldsymbol{\Sigma}_a\right), \tag{10.34a}$$
$$p(\mathbf{z}_b \mid \mathbf{z}_a) = \mathcal{N}\left(\mathbf{z}_b; \mathbf{A}\mathbf{z}_a + \mathbf{b}, \boldsymbol{\Sigma}_{b|a}\right). \tag{10.34b}$$

*Then the conditional distribution of* $\mathbf{z}_a$ *given* $\mathbf{z}_b$ *is*

$$p(\mathbf{z}_a \mid \mathbf{z}_b) = \mathcal{N}\Big(\mathbf{z}_a; \boldsymbol{\mu}_{a|b}, \boldsymbol{\Sigma}_{a|b}\Big), \tag{10.35a}$$

*with*

$$\boldsymbol{\mu}_{a|b} = \boldsymbol{\Sigma}_{a|b}\left(\boldsymbol{\Sigma}_a^{-1}\boldsymbol{\mu}_a + \mathbf{A}^\mathsf{T}\boldsymbol{\Sigma}_{b|a}^{-1}(\mathbf{z}_b - \mathbf{b})\right), \tag{10.35b}$$
$$\boldsymbol{\Sigma}_{a|b} = \left(\boldsymbol{\Sigma}_a^{-1} + \mathbf{A}^\mathsf{T}\boldsymbol{\Sigma}_{b|a}^{-1}\mathbf{A}\right)^{-1}. \tag{10.35c}$$

**Corollary 10.2 (Affine transformation – Marginalisation)** *Assume that* $\mathbf{z}_a$ *as well as* $\mathbf{z}_b \mid \mathbf{z}_a$ *are both Gaussian distributed according to*

$$p(\mathbf{z}_a) = \mathcal{N}\left(\mathbf{z}_a; \boldsymbol{\mu}_a, \boldsymbol{\Sigma}_a\right), \tag{10.36a}$$
$$p(\mathbf{z}_b \mid \mathbf{z}_a) = \mathcal{N}\left(\mathbf{z}_b; \mathbf{A}\mathbf{z}_a + \mathbf{b}, \boldsymbol{\Sigma}_{b|a}\right). \tag{10.36b}$$

*Then the marginal distribution of* $\mathbf{z}_b$ *is given by*

$$p(\mathbf{z}_b) = \mathcal{N}\left(\mathbf{z}_b; \boldsymbol{\mu}_b, \boldsymbol{\Sigma}_b\right), \tag{10.37a}$$

*where*

$$\boldsymbol{\mu}_b = \mathbf{A}\boldsymbol{\mu}_a + \mathbf{b}, \tag{10.37b}$$
$$\boldsymbol{\Sigma}_b = \boldsymbol{\Sigma}_{b|a} + \mathbf{A}\boldsymbol{\Sigma}_a\mathbf{A}^\mathsf{T}. \tag{10.37c}$$

# 11 Generative Models

So far in this book, we have spent most of our energy on the supervised learning problem: given training data $\{\mathbf{x}_i, y_i\}_{i=1}^n$ of input–output pairs, learn to predict the output $y$ for a new input $\mathbf{x}$. In Chapter 2 we already noted that this seemingly narrow problem formulation is more powerful than it may appear, and we have at a couple of points extended it without leaving the overall framework. In Section 2.4 we discussed how *self-supervised learning* lets us manufacture a supervised problem out of unlabelled sequential data, simply by treating a window of past values as the input and the next value as the output. In Section 3.5 we extended the supervised perspective in a different direction by considering *conditional generative models*, where the goal is not to predict a single best output but to generate samples from the predictive distribution $p(y \mid \mathbf{x})$.

In this chapter, we take a more decisive step away from the supervised paradigm, by turning our attention to *generative models*. A model of the conditional distribution $p(y \mid \mathbf{x})$ – whether it is used to produce a point prediction $\widehat{y}$ or to generate samples, as in Section 3.5 – always takes a given input $\mathbf{x}$ as its starting point and then says something about the output $y$. In this sense it is a *discriminative* model: it describes how to discriminate between the possible outputs $y$ once the input $\mathbf{x}$ is known, but it remains silent about the inputs $\mathbf{x}$ themselves. A *generative* model takes a different and more ambitious stance. It models the *joint* distribution $p(\mathbf{x}, y)$ of inputs and outputs, or just the *marginal* distribution $p(\mathbf{x})$ of the inputs in case there are no outputs.

Modelling the full joint distribution $p(\mathbf{x}, y)$ is a richer description than modelling only the conditional $p(y \mid \mathbf{x})$, and it unlocks several capabilities that a purely discriminative model does not have. First, since the model describes how the data is distributed, we can *sample* from it, generating new synthetic data points – either complete pairs $(\mathbf{x}, y)$ or, ignoring the labels, inputs $\mathbf{x}$ on their own. Second, from a model of the joint $p(\mathbf{x}, y)$ we can always *derive* the conditional $p(y \mid \mathbf{x})$, and thereby obtain a discriminative method for classification or regression as a by-product, as will happen in Section 11.1. Third, because the model contains an explicit description of the distribution of the inputs, it can be learned also from *unlabelled* data, where parts (Section 11.2) or all (Section 11.3) of the corresponding outputs $y_i$ are missing.

Generative modelling plays an important role in modern machine learning. Many of its most spectacular recent successes rely on deep neural networks, which is the topic of Chapter 12. To introduce the generative concept, we will in this chapter use a simpler, but nevertheless useful, model as our main tool: the *Gaussian mixture model* (GMM).

**Example 11.1 Our first generative model: The GMM**

The Gaussian Mixture Model (GMM) is a generative model for a pair $(\mathbf{x}, y)$, where $\mathbf{x} \in \mathbb{R}^p$ is numerical and $y$ is a categorical variable taking values in $\{1, \ldots, M\}$. This is the same setting as for classification, but the modelling perspective is different: rather than directly modelling the conditional $p(y \mid \mathbf{x})$, the GMM models the joint distribution $p(\mathbf{x}, y)$. The GMM factorises the joint distribution as

$$p(\mathbf{x}, y) = p(\mathbf{x} \mid y)p(y). \tag{11.1a}$$

The second factor is the marginal distribution of $y$, which is categorical with $M$ parameters $\{\pi_m\}_{m=1}^M$:

$$p(y = 1) = \pi_1, \quad \cdots, \quad p(y = M) = \pi_M. \tag{11.1b}$$

The first factor $p(\mathbf{x} \mid y)$ is the class-conditional distribution of the input. In a classification-like setting, it is natural to assume that these distributions are different for different classes – indeed, if it is possible to predict the class $y$ from $\mathbf{x}$, then the distribution of $\mathbf{x}$ should depend on $y$. The basic assumption that gives the GMM its name is that each class-conditional distribution is a Gaussian,

$$p(\mathbf{x} \mid y) = \mathcal{N}\Big(\mathbf{x}; \boldsymbol{\mu}_y, \boldsymbol{\Sigma}_y\Big), \tag{11.1c}$$

with class-dependent mean vector $\boldsymbol{\mu}_y$ and covariance matrix $\boldsymbol{\Sigma}_y$. In words, the model (11.1) starts from a categorical distribution over $y$, and, for each value of $y$, assumes a (possibly different) Gaussian distribution for $\mathbf{x}$. The marginal distribution over $\mathbf{x}$ alone, obtained by summing out $y$, is

$$p(\mathbf{x}) = \sum_{m=1}^{M} p(\mathbf{x} \mid y = m)\, p(y = m) = \sum_{m=1}^{M} \pi_m \mathcal{N}\big(\mathbf{x}; \boldsymbol{\mu}_m, \boldsymbol{\Sigma}_m\big), \tag{11.2}$$

a weighted combination – a *mixture* – of $M$ Gaussian distributions (one component for each value of $y$). This is the reason for the name.[a]

[a] Unlike this book, some sources refer to the marginal distribution (11.2) rather than the full joint model (11.1), when using the term GMM.

The GMM is only an example of a generative model, but it will be our main theme for this chapter since it is flexible enough to be practically useful, and yet simple enough to allow for algebraic manipulations. It is fully specified once we fix the parameters $\boldsymbol{\theta} = \{\boldsymbol{\mu}_m, \boldsymbol{\Sigma}_m, \pi_m\}_{m=1}^M$. We will of course want to *learn* these parameters from data, but we postpone that discussion to the next section. For now we simply assume that the parameters are given, so that we have a complete generative model in hand, and explore what it means for such a model to be generative. The defining feature is that we can use it to *generate* data, that is, to draw samples. We illustrate this in Example 11.2.


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

### Example 11.2 Sampling from the Gaussian Mixture Model

Consider the two-class example ($M = 2$) illustrated in the left panel below, with equal mixing weights $\pi_1 = \pi_2 = \frac{1}{2}$ and two class-conditional Gaussians with the means and covariances indicated by the red and blue contour lines.

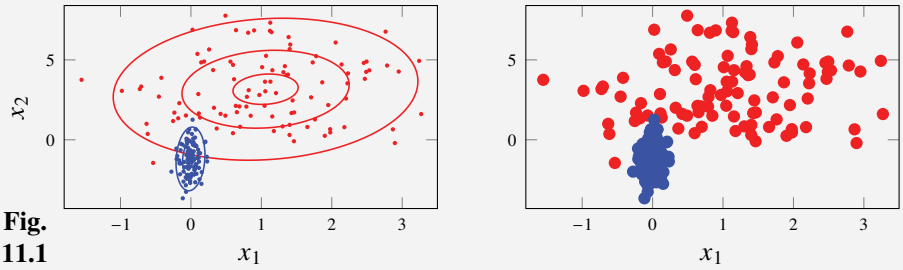



**Fig. 11.1**

Even though the GMM is a joint model for the pair $(\mathbf{x}, y)$, drawing a sample from it is straightforward and proceeds in two steps. To generate one data point we:

(i) draw a class label $y$ from the categorical distribution $p(y)$. In this case this amounts to flipping a coin, since $\pi_1 = \pi_2 = \frac{1}{2}$; and

(ii) draw an input $\mathbf{x}$ from the Gaussian $\mathcal{N}\left(\mathbf{x}; \boldsymbol{\mu}_y, \boldsymbol{\Sigma}_y\right)$ of the class that was selected in step (i).

The result is a pair $(\mathbf{x}, y)$, where the colour of the point records the sampled class $y$. Repeating this procedure many times produces the cloud of points shown in the right panel above. If we are only interested in the inputs, we simply discard the labels $y$ (the colors in the figure) and keep the $\mathbf{x}$-values; the points then follow the marginal mixture distribution (11.2).

The two-step recipe is worth dwelling on, because it captures the essence of what makes a model generative: it encodes a concrete procedure for producing data that 'looks like' the real thing. The same idea – sample a simple latent quantity first, then turn it into a data point – reappears in much more elaborate guises when we get to deep generative models in Chapter 12.

We have now defined the GMM and seen that, once its parameters are fixed, we can generate data from it. What we have *not* yet discussed is the reverse question: given a set of observed data points, how do we learn the parameters $\boldsymbol{\theta}$? This is the subject of the next section, where we learn the GMM from ordinary supervised training data and then put the learned model to work as a classifier. After that, we will see how the generative perspective lets us go further than supervised learning allows: in Section 11.2 we learn the GMM from partially labelled data, *semi-supervised learning*, and in Section 11.3 we learn it from completely unlabelled data, *unsupervised learning*, where it becomes a method for *clustering*.

## 11.1 Learning a Generative Model and Using It for Classification

In the introduction we defined the GMM and treated its parameters as given. We now address the question that any useful machine learning model must answer: how do we *learn* it from data? We thereafter consider how a learned GMM model can be used as a method for solving classification problems.

### Supervised Learning of the Gaussian Mixture Model

Like any machine learning model, the GMM (11.1) is learned from training data. The unknown parameters to be learned are $\boldsymbol{\theta} = \{\boldsymbol{\mu}_m, \boldsymbol{\Sigma}_m, \pi_m\}_{m=1}^M$. We start with the supervised case, where the training data contains both inputs $\mathbf{x}_i$ and corresponding labels $y_i$, that is, $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^n$.

We learn the GMM by maximising the log-likelihood of the training data,[1]

$$\widehat{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta}} \ln p(\{\mathbf{x}_i, y_i\}_{i=1}^n \mid \boldsymbol{\theta}). \tag{11.3a}$$

Due to the generative nature of the model, this is based on the joint likelihood of both inputs and outputs. It follows from the model definition (11.1) that the log-likelihood is

$$\begin{aligned}\ln p(\{\mathbf{x}_i, y_i\}_{i=1}^n \mid \boldsymbol{\theta}) &= \sum_{i=1}^n \left(\ln p(\mathbf{x}_i \mid y_i, \boldsymbol{\theta}) + \ln p(y_i \mid \boldsymbol{\theta})\right) \\ &= \sum_{i=1}^n \sum_{m=1}^M \mathbb{I}\{y_i = m\} \left(\ln \mathcal{N}\left(\mathbf{x}_i; \boldsymbol{\mu}_m, \boldsymbol{\Sigma}_m\right) + \ln \pi_m\right),\end{aligned} \tag{11.3b}$$

where the indicator function $\mathbb{I}\{y_i = m\}$ effectively splits the log-likelihood into $M$ independent sums, one for each class.

The optimisation problem (11.3) turns out to have a closed-form solution. Starting with the marginal class probabilities $\{\pi_m\}_{m=1}^M$, we get

$$\widehat{\pi}_m = \frac{n_m}{n}, \tag{11.4a}$$

where $n_m$ is the number of training data points in class $m$. Consequently, $\sum_m n_m = n$ and thus $\sum_m \widehat{\pi}_m = 1$. The probability of a certain class $y = m$, without having any additional information, is estimated as the proportion of this class in the training data. The mean vector $\boldsymbol{\mu}_m$ of each class is estimated as

$$\widehat{\boldsymbol{\mu}}_m = \frac{1}{n_m} \sum_{i:y_i=m} \mathbf{x}_i, \tag{11.4b}$$

[1] Alternatively, one can learn the GMM by following the Bayesian approach, but we do not pursue that any further here. See Section 11.4 for suggestions for further reading.


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

the empirical mean among all training data points of class $m$, and the covariance matrix $\boldsymbol{\Sigma}_m$ for each class $m = 1, \ldots, M$, is estimated as[2]

$$\widehat{\boldsymbol{\Sigma}}_m = \frac{1}{n_m} \sum_{i:y_i=m} (\mathbf{x}_i - \widehat{\boldsymbol{\mu}}_m)(\mathbf{x}_i - \widehat{\boldsymbol{\mu}}_m)^\mathsf{T}. \tag{11.4c}$$

The expressions (11.4b)–(11.4c) fit a Gaussian distribution for $\mathbf{x}$ to each class by matching the empirical mean and covariance – so-called moment-matching. We can compute the parameters $\widehat{\boldsymbol{\theta}}$ regardless of whether the data actually comes from a Gaussian distribution or not. If the data shown in the right panel of Figure 11.1 is taken to be our supervised training data, then learning the GMM in this way recovers the two class-conditional Gaussians shown in the left panel.

It is worth pausing to appreciate what has – and has not – happened in this step. The learning problem (11.3) made no reference whatsoever to a prediction task. We did not ask the model to classify anything; we merely estimated a description of how the data is distributed, by fitting one Gaussian per class together with the class proportions according to the GMM assumptions. The outcome is a complete, self-contained generative model. *Only now*, with the learned model in hand, do we get to ask the separate question of what to do with it. This clean separation between modelling the data and using the model is a characteristic of the generative approach, and – as we will see – it is exactly what gives generative models their interesting flexibility.

### Using the Model as a Classifier: Discriminant Analysis

We have described the generative GMM $p(\mathbf{x}, y)$ and learned its parameters from training data. We will now see how this model can be used as a classifier for supervised machine learning.

The key insight for using a generative model $p(\mathbf{x}, y)$ to make predictions is to realise that predicting the output $y$ for a known value $\mathbf{x}$ amounts to computing the conditional distribution $p(y \mid \mathbf{x})$. From probability theory, we have

$$p(y \mid \mathbf{x}) = \frac{p(\mathbf{x}, y)}{p(\mathbf{x})} = \frac{p(\mathbf{x}, y)}{\sum_{j=1}^{M} p(\mathbf{x}, y = j)}. \tag{11.5}$$

The left-hand side $p(y \mid \mathbf{x})$ is the predictive distribution, whereas all expressions on the right-hand side are defined by the generative GMM (11.1). We therefore get the classifier

$$p(y = m \mid \mathbf{x}_\star) = \frac{\widehat{\pi}_m \mathcal{N}\left(\mathbf{x}_\star; \widehat{\boldsymbol{\mu}}_m, \widehat{\boldsymbol{\Sigma}}_m\right)}{\sum_{j=1}^{M} \widehat{\pi}_j \mathcal{N}\left(\mathbf{x}_\star; \widehat{\boldsymbol{\mu}}_j, \widehat{\boldsymbol{\Sigma}}_j\right)}. \tag{11.6}$$

[2] A common alternative is to normalise the estimate by $n_m - 1$ instead of $n_m$, resulting in an unbiased estimate of the covariance matrix, but that is in fact not the maximum likelihood solution. The two options are not mathematically equivalent, but for machine learning purposes the practical difference is often minor.

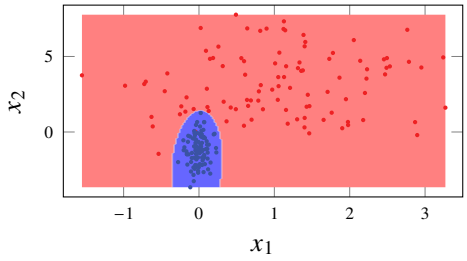



**Figure 11.2:** The decision boundary for the QDA classifier (obtained from (11.6) and (11.8)) corresponding to the GMM shown in the left panel of Figure 11.1.

As usual, we can obtain 'hard' predictions $\widehat{y}_\star$ by selecting the most probable class,

$$\widehat{y}_\star = \arg\max_m p(y = m \mid \mathbf{x}_\star). \tag{11.7}$$

Taking the logarithm and noting that only the numerator in (11.6) depends on $m$, we can equivalently write this as

$$\widehat{y}_\star = \arg\max_m \left\{ \ln \widehat{\pi}_m + \ln \mathcal{N}\left(\mathbf{x}_\star; \widehat{\boldsymbol{\mu}}_m, \widehat{\boldsymbol{\Sigma}}_m\right) \right\}. \tag{11.8}$$

Since the logarithm of the Gaussian probability density function is a quadratic function in $\mathbf{x}$, the decision boundary for this classifier is also quadratic, and the method is therefore (for historical reasons) referred to as *quadratic discriminant analysis* (QDA).We summarise the method in Method 11.1 and illustrate it in Figure 11.3; in Figure 11.2 we show the decision boundary obtained when the GMM from Figure 11.1 is turned into a QDA classifier.

The QDA method arises naturally from the GMM. However, if we make an additional simplifying assumption about the model, we instead obtain the even more well-known and commonly used classifier[3] *linear discriminant analysis* (LDA). The additional assumption is that the covariance matrix is equal for all classes, that is, $\boldsymbol{\Sigma}_1 = \boldsymbol{\Sigma}_2 = \cdots = \boldsymbol{\Sigma}_M = \boldsymbol{\Sigma}$ in (11.1c). With this restriction, we only have a single covariance matrix to learn, and (11.4c) is replaced by[4]

$$\widehat{\boldsymbol{\Sigma}} = \frac{1}{n} \sum_{m=1}^{M} \sum_{i:y_i=m} (\mathbf{x}_i - \widehat{\boldsymbol{\mu}}_m)(\mathbf{x}_i - \widehat{\boldsymbol{\mu}}_m)^\mathsf{T}. \tag{11.9}$$

Using this assumption in (11.6) results in a convenient cancellation of all quadratic terms when computing the class predictions in (11.8), and the LDA classifier will therefore have linear decision boundaries. Consequently, LDA is a linear classifier, just like logistic regression, and the two methods will often perform similarly. They

[3] Not to be confused with Latent Dirichlet Allocation, sometimes abbreviated LDA, which is a completely different method.

[4] Similarly to the comment about (11.4c), the sum can alternatively be normalised by $n - M$ instead of $n$.


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

**Learn** the Gaussian mixture model

**Data:** Training data $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^n$
**Result:** Gaussian mixture model

**for** $m = 1, \ldots, M$ **do**
  Compute $\widehat{\pi}_m$ (11.4a), $\widehat{\boldsymbol{\mu}}_m$ (11.4b) and $\widehat{\boldsymbol{\Sigma}}_m$ (11.4c)
**end**

**Predict** with Gaussian mixture model

**Data:** Gaussian mixture model and test input $\mathbf{x}_\star$
**Result:** Prediction $\widehat{y}_\star$

**for** $m = 1, \ldots, M$ **do**
  Compute $\delta_m \overset{\text{def}}{=} \ln \widehat{\pi}_m + \ln \mathcal{N}\left(\mathbf{x}_\star; \widehat{\boldsymbol{\mu}}_m, \widehat{\boldsymbol{\Sigma}}_m\right)$
**end**
Set $\widehat{y}_\star = \arg\max_m \delta_m$.

**Method 11.1:** Quadratic Discriminant Analysis, QDA.

are not equivalent, however, since the parameters are learned in different ways. This usually results in small differences in their respective decision boundaries. Note that LDA is obtained by replacing (11.4c) with (11.9) in Method 11.1. We compare LDA and QDA in Figure 11.4 by applying both of them to the music classification problem from Example 2.1.

> ***Time to reflect 11.1*** *In the GMM, it was assumed that $p(\mathbf{x} \mid y)$ is Gaussian. When applying LDA or QDA 'off the shelf' for a classification problem, is there any check that the Gaussian assumption actually holds? If yes, what? If no, is that a problem?*

Let us be explicit about what we have just done. We started from a generative model of the joint distribution $p(\mathbf{x}, y)$, and by computing the conditional distribution (11.5) we obtained the predictive distribution $p(y \mid \mathbf{x})$. In effect, we have *turned our generative model into a discriminative classifier*: the QDA (or LDA) classifier can be employed exactly like any other classifier from earlier in the book, even though it was never trained to discriminate. If we keep the full distribution $p(y \mid \mathbf{x}_\star)$ in (11.6), rather than collapsing it to the single most probable class, we can equally well view the result as a *conditional generative model* in the sense of Section 3.5: it lets us reason about, and sample from, the distribution over plausible labels for a given input. And, moreover, we still also have the possibility to sample completely new data pairs $\mathbf{x}, y$ as we did in Example 11.2.

However, if there is no need for generating samples in the application and the task at hand is purely discriminative, it is a natural question whether anything is

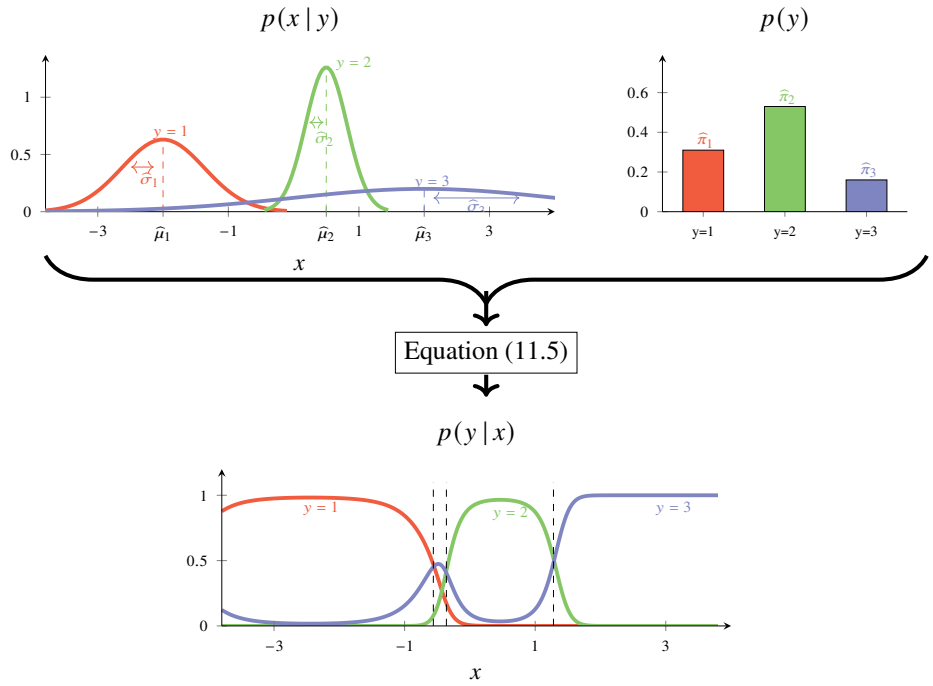
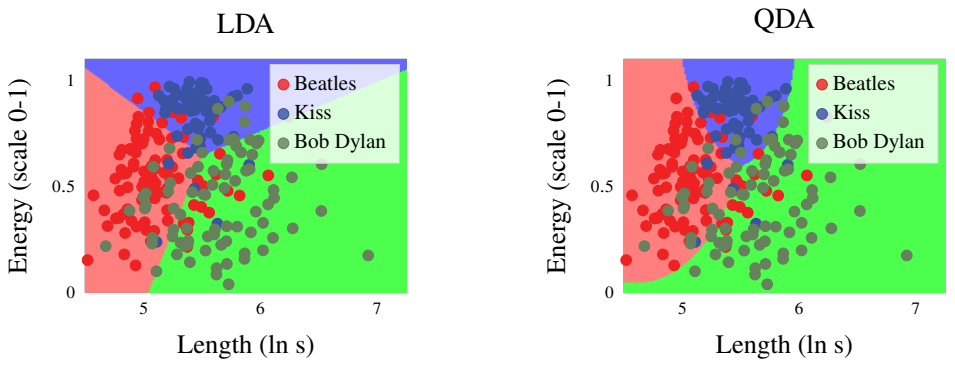



**Figure 11.3:** An illustration of QDA for $M = 3$ classes, with dimension $p = 1$ of the input $x$. At the top, the generative GMM is shown. To the left is the Gaussian model of $p(x \mid y = m)$, parameterised by $\widehat{\mu}_m$ and $\widehat{\sigma}_m^2$ (since $p = 1$, we only have a scalar variance $\sigma_m^2$, instead of a covariance matrix $\boldsymbol{\Sigma}_m$). To the right the model of $p(y)$ is shown, parameterised by $\widehat{\pi}_m$. All parameters are learned from training data, not shown in the figure. By computing the conditional distribution (11.5) the generative model is 'warped' into $p(y = m \mid x)$, shown in the bottom. The decision boundaries are shown as vertical dotted lines in the bottom plot (assuming that we classify $x$ based on the most probable class).

gained by taking this detour through a generative model compared to training a discriminative classifier (such as logistic regression) directly. A generative model contains more assumptions than a discriminative one. GMM specifically commits to the class-conditional distributions $p(\mathbf{x} \mid y)$ being Gaussian. If those assumptions are (approximately) fulfilled, this extra structure should make QDA somewhat more *data-efficient*, in the sense of requiring fewer training data points to reach a given level of performance, since the model is 'told' more about the problem in advance. If the assumptions are badly violated, on the other hand, the same extra structure can instead mislead the classifier. However, in the purely supervised setting, the difference between the two approaches is usually minor in practice. The real payoff of the generative perspective becomes evident only when we leave the fully supervised setting – which is exactly what we do next, in Section 11.2, where the model's description of $p(\mathbf{x})$ lets us learn from unlabelled data as well.


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`




**(a)** Decision boundaries for the music classification problem for an LDA classifier.

**(b)** Decision boundaries for the music classification problem for a QDA classifier.

**Figure 11.4:** We apply LDA and QDA classifiers to the music classification problem from Example 2.1 and plot the resulting decision boundaries. Note that the LDA classifier gives linear decision boundaries, whereas the QDA classifier has decision boundaries with quadratic shapes.

***Time to reflect 11.2*** *LDA and logistic regression are both linear classifiers. If the data really were Gaussian with a shared covariance, which would you expect to need fewer training points to reach a certain performance level, and why?*

## 11.2 Beyond Supervised Data: Semi-Supervised Learning

We have so far discussed how the GMM can be learned in the supervised setting, with both input and corresponding output values in the training data. This already revealed one of the attractions of the generative approach – a single model of $p(\mathbf{x}, y)$, learned with no prediction task in mind, could afterwards be turned into a classifier. But the generative perspective offers something more fundamental, which the supervised setting does not let us see: *a generative model is not limited to supervised data at all*. Because it describes the distribution of the inputs $\mathbf{x}$ as well as the outputs $y$, it can make use of data points for which no output has ever been recorded. We now take a first step in this direction by considering the so-called *semi-supervised* problem, in which some output values $y_i$ are missing.

The input values $\mathbf{x}_i$ for which the corresponding $y_i$ are missing are called *unlabelled* data points. As before, we denote the total number of training data points as $n$, out of which now only $n_l$ are labelled input–output pairs $\{\mathbf{x}_i, y_i\}_{i=1}^{n_l}$ and the remaining $n_u$ are unlabelled data points $\{\mathbf{x}_i\}_{i=n_l+1}^{n}$, where $n = n_l + n_u$. All in all we have the training data $\mathcal{T} = \{\{\mathbf{x}_i, y_i\}_{i=1}^{n_l}, \{\mathbf{x}_i\}_{i=n_l+1}^{n}\}$. For notational purposes we have, without loss of generality, ordered the data points so that the first $n_l$ are labelled and the remaining $n_u$

are unlabelled.

Semi-supervised learning is of high practical relevance. In many applications it is easy to obtain large amounts of unlabelled data, but annotating this data (that is, assigning labels $y_i$ to the training data points) can be a very costly and time-consuming procedure. This is particularly true when the labelling is done manually by a domain expert. For instance, consider learning a model for classifying images of skin lesions as either benign or malignant, to be used in a medical diagnosis support system. The training inputs $\mathbf{x}_i$ correspond to images of skin lesions, and acquiring large numbers of such images is easy. To annotate the training data with labels $y_i$, however, we need to determine whether each lesion is benign or malignant, which requires a (possibly expensive) medical examination by a dermatologist.

The simplest solution to the semi-supervised problem would be to discard the $n_u$ unlabelled data points and thereby turn the problem into a standard supervised machine learning problem. This is a pragmatic solution, but possibly very wasteful if the number of labelled data points $n_l$ is only a small fraction of the total. We illustrate this with Figure 11.5, which depicts a semi-supervised problem where we have learned a (poor) GMM by only using the few $n_l$ labelled data points.

The idea behind semi-supervised learning is to exploit the information available in the unlabelled data points to, hopefully, end up with a better model in the end. There are different ways in which the semi-supervised problem can be approached, but one principled way is to make use of a generative model. This is where the generative perspective on the GMM starts to pay off. A generative model includes a model of the marginal distribution of the inputs $p(\mathbf{x})$, so it is plausible that the unlabelled data points $\{\mathbf{x}_i\}_{i=n_l+1}^{n}$ can be used when learning the model. Intuitively, the unlabelled inputs can be used to find groups (or *clusters*) of input values with similar properties, which can then be assumed to belong to the same class. Looking at Figure 11.5 again, by considering the unlabelled data points (grey dots), it is reasonable to assume that the two apparent clusters of points correspond to the two classes (red and blue, respectively). As we will see below, by exploiting this information, we can obtain better estimates of the class-conditional distributions $p(\mathbf{x} \mid y)$ and thereby also a better classifier.

We now turn to the technical details of how to learn the GMM in this semi-supervised setting. Similarly to above, we take the maximum likelihood approach, meaning that we seek the model parameters that maximise the likelihood of the observed data. The observed data now contains both labelled and unlabelled instances, so we would like to solve

$$\widehat{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta}} \ln p(\{\{\mathbf{x}_i, y_i\}_{i=1}^{n_l}, \{\mathbf{x}_i\}_{i=n_l+1}^{n}\} \mid \boldsymbol{\theta}). \tag{11.10}$$

Unfortunately this problem has no closed-form solution for the GMM. We will discuss the reason for this intractability in more detail in Section 11.3, where we revisit the same problem in the fully unsupervised setting. Intuitively, however, we can conclude that it is not possible to compute the model parameters as in (11.4), because we do not know which classes the unlabelled data points belong to. Hence, when computing the


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

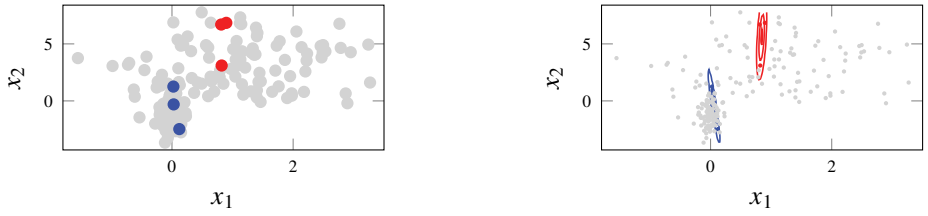



**Figure 11.5:** We consider the same situation as in Figure 11.1, except for the fact that we have 'lost' the output $y_i$ for most of the data points. The problem is now semi-supervised. The unlabelled data points $\{\mathbf{x}_i\}_{i=n_l+1}^{n}$ are illustrated in the left panel as grey dots, whereas the $n_l = 6$ labelled data points $\{\mathbf{x}_i, y_i\}_{i=1}^{n_l}$ are red or blue. In the right panel, we have learned a GMM using only the labelled data points, as if the problem were supervised with only $n_l$ data points. Clearly the missing output labels have made the problem harder, compared to Figure 11.1. We will, however, continue this story in Figure 11.6 where we instead use this as an initialisation to a semi-supervised procedure.

mean vector for the $m$th class as in (11.4b), for instance, we do not know which data points should be included in the sum.

A possible way around this issue is to first learn an initial GMM, which is then used to predict the missing values $\{y_i\}_{i=n_l+1}^{n}$, and thereafter use these predictions to update the model. Doing this iteratively results in the following algorithm:

(i) Learn the GMM from the $n_l$ labelled input–output pairs $\{\mathbf{x}_i, y_i\}_{i=1}^{n_l}$,

(ii) Use the GMM to predict (as a QDA classifier) the missing outputs of $\{\mathbf{x}_i\}_{i=n_l+1}^{n}$,

(iii) Update the GMM including the predicted outputs from step (ii),

and then repeat step (ii) and (iii) until convergence.

At first this might look like an *ad hoc* procedure, and it is far from obvious that it will converge to anything sensible. However, it turns out that it is an instance of a widely used statistical tool referred to as the *expectation–maximisation (EM)* algorithm. We will study the EM algorithm and discuss its validity in more detail in Section 11.3, where we treat the fully unsupervised case. For now we simply note that the algorithm, when applied to the maximum likelihood problem (11.10), indeed boils down to the procedure outlined above, as long as we pay attention to a few important details: from step (ii) we should return the predicted class probabilities $p(y = m \mid \mathbf{x}, \widehat{\boldsymbol{\theta}})$ (and not the hard class prediction $\widehat{y}(\mathbf{x}_\star)$) computed using the current parameter estimates $\widehat{\boldsymbol{\theta}}$, and in step (iii) we make use of the predicted class probabilities by introducing the notation

$$w_i(m) = \begin{cases} p(y = m \mid \mathbf{x}_i, \widehat{\boldsymbol{\theta}}) & \text{if } y_i \text{ is missing} \\ 1 & \text{if } y_i = m \\ 0 & \text{otherwise} \end{cases} \tag{11.11a}$$

and update the parameters as follows:

$$\widehat{\pi}_m = \frac{1}{n}\sum_{i=1}^{n} w_i(m), \tag{11.11b}$$

$$\widehat{\boldsymbol{\mu}}_m = \frac{1}{\sum_{i=1}^{n} w_i(m)}\sum_{i=1}^{n} w_i(m)\mathbf{x}_i, \tag{11.11c}$$

$$\widehat{\boldsymbol{\Sigma}}_m = \frac{1}{\sum_{i=1}^{n} w_i(m)}\sum_{i=1}^{n} w_i(m)(\mathbf{x}_i - \widehat{\boldsymbol{\mu}}_m)(\mathbf{x}_i - \widehat{\boldsymbol{\mu}}_m)^\mathsf{T}. \tag{11.11d}$$

We use the current parameter estimate $\widehat{\boldsymbol{\theta}}$ in step (ii), which is then updated in step (iii); when we go back to step (ii) for the next iteration, the class probabilities will be computed using a new value of $\widehat{\boldsymbol{\theta}}$.

When computing the parameters for class $m$ according to (11.11), the unlabelled data points contribute proportionally to the current estimates of the probabilities that they belong to this class. Note that this is a generalisation of the supervised case, as (11.4) is a special case of (11.11) when no labels $y_i$ are missing. With these modifications, it can be shown (see Section 11.3) that the procedure converges to a stationary point of (11.10) even in the semi-supervised setting. We summarise the procedure as Method 11.2 and illustrate it in Figure 11.6 by applying it to the semi-supervised data introduced in Figure 11.5.

---

**Learn** the GMM

---

**Data:** Partially labelled training data $\mathcal{T} = \{\{\mathbf{x}_i, y_i\}_{i=1}^{n_l}, \{\mathbf{x}_i\}_{i=n_l+1}^{n}\}$ (with output classes $m = 1, \ldots, M$)

**Result:** Gaussian mixture model

1 Compute $\boldsymbol{\theta} = \{\widehat{\pi}_m, \widehat{\boldsymbol{\mu}}_m, \widehat{\boldsymbol{\Sigma}}_m\}_{m=1}^{M}$ according to (11.4), using only the labelled data $\{\mathbf{x}_i, y_i\}_{i=1}^{n_l}$

2 **repeat**

3 For each $\mathbf{x}_i$ in $\{\mathbf{x}_i\}_{i=n_l+1}^{n}$, compute the prediction $p(y \mid \mathbf{x}_i, \widehat{\boldsymbol{\theta}})$ according to (11.6) using the current parameter estimates $\widehat{\boldsymbol{\theta}}$

4 Update the parameter estimates $\widehat{\boldsymbol{\theta}} \leftarrow \{\widehat{\pi}_m, \widehat{\boldsymbol{\mu}}_m, \widehat{\boldsymbol{\Sigma}}_m\}_{m=1}^{M}$ according to (11.11)

5 **until** *convergence*

---

**Predict** as QDA, Method 11.1

---

**Method 11.2:** Semi-supervised learning of the GMM

It is worth pausing to ask why we have chosen to introduce the semi-supervised problem in connection to generative models. Alternatively, we could think of using any discriminative model for iteratively predicting the missing $y_i$ and updating the


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

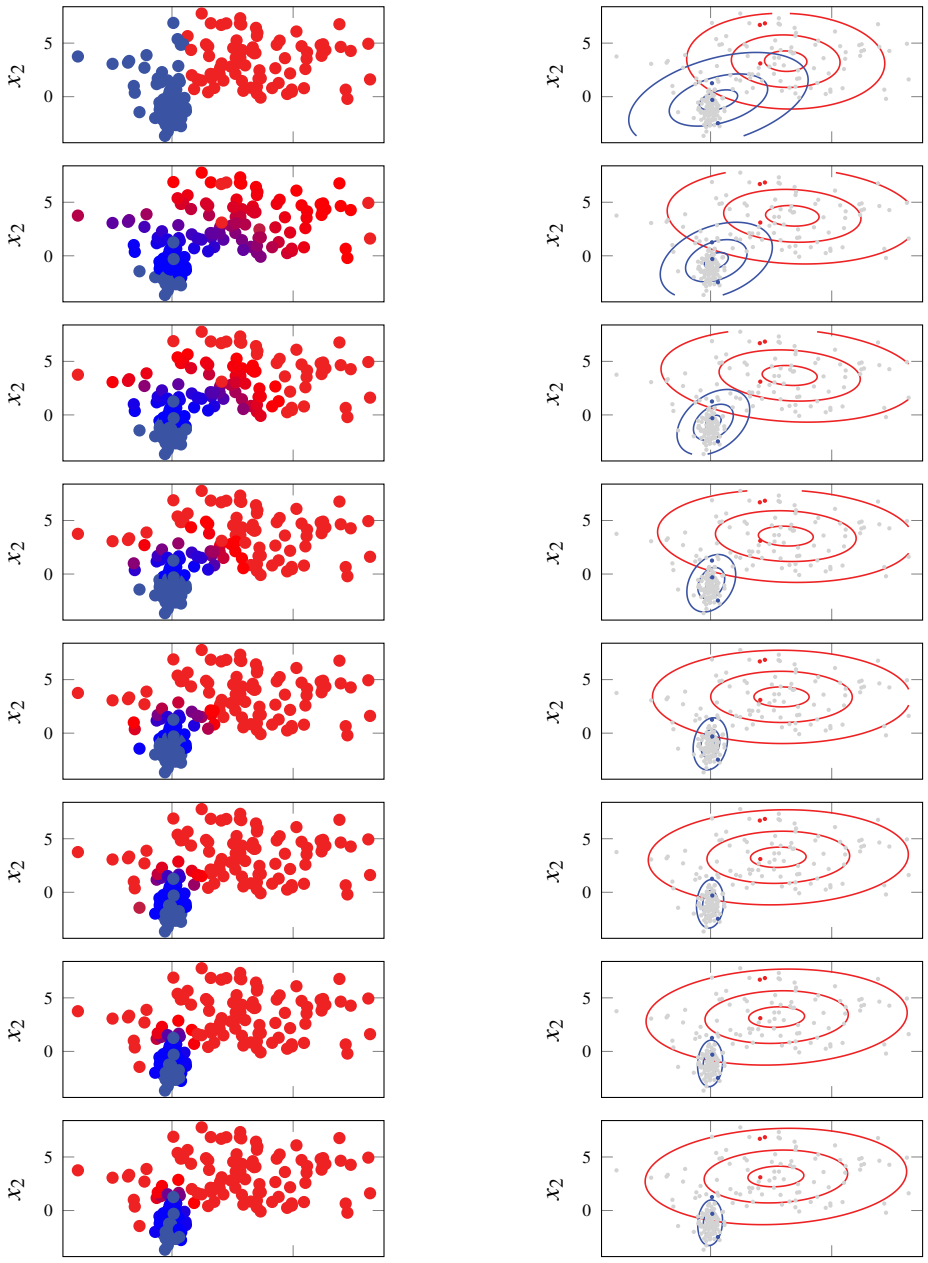



**Figure 11.6:** The iterative Method 11.2 applied to the problem from Figure 11.5. For each iteration, the left panel shows the predicted class probabilities from the previously learned model (using colour coding; purple is in the middle between red and blue). The new models learned using (11.11) (based on the predictions from the left panel) are shown in the right panel for each row. The iteration is initialised using the model shown in Figure 11.5.

model using these predictions. This is indeed possible, and such discriminative label-imputation methods can be made to work well in many challenging semi-supervised cases. However, the generative modelling paradigm provides us with a more principled and coherent framework for reasoning about missing labels. In particular, we have derived a method for the semi-supervised setting which is a direct generalisation of the corresponding supervised method. In Section 11.3, we will take this one step further and apply the same procedure to the fully unsupervised case (by simply assuming that *all* labels are missing). In all cases, the method has a strong theoretical foundation since it is solving the maximum likelihood problem.

Another explanation for why generative models can be useful in the semi-supervised setting is that they provide a richer description of the data-generating process than a discriminative model, making it easier to leverage the information contained in the unlabelled data points. A model describing only $p(y \mid \mathbf{x})$ encodes information like 'if $\mathbf{x}$, then $y$ . . . ', but it does not contain any explicit model $p(\mathbf{x})$ for the inputs themselves. The generative model, on the other hand, contains assumptions for the full distribution $p(\mathbf{x}, y)$ that is leveraged in the semi-supervised problem. For instance, the GMM encodes the information that all inputs $\mathbf{x}$ that belong to a certain class $y$ should have the same Gaussian distribution and thereby belong to the same cluster. The model parameters are then inferred from these clusters as in (11.11), where both labelled and unlabelled data points contribute. This assumption is the key enabler for Method 11.2 to work in practice, even in such a challenging situation as in Figures 11.5 and 11.6 where the vast majority of the data points are unlabelled.

Finally, it is worth emphasising that nothing has changed about *what* we end up with. Once Method 11.2 has converged, we are simply left with a learned GMM, with parameters $\widehat{\boldsymbol{\theta}}$, exactly as in the supervised case. In particular, we are free to use it as a classifier if we want to: through (11.6) it becomes a QDA (or, with a shared covariance matrix, an LDA) classifier, and we can predict the class of any new input just as in Section 11.1. The only difference is that the unlabelled data has helped us learn a better model, and hence a better classifier, than the labelled data could have produced on its own.

The generative modelling paradigm thus provides a principled approach for modelling both labelled and unlabelled data. The downside, however, is that it requires additional assumptions on the data distribution, and the result can be misleading if these assumptions are not fulfilled. Furthermore, in many contemporary machine learning problems the input $\mathbf{x}$ is extremely high-dimensional, and it can then be difficult to design and/or learn a suitable model for its distribution. For instance, if $\mathbf{x}$ is an image (as we discussed in the context of convolutional neural networks, Section 7.1), then modelling the pixel values in $\mathbf{x}$ using a (very high-dimensional) Gaussian distribution is not going to capture the characteristics of natural images in a good way. Overcoming this limitation calls for far more flexible, neural-network-based generative models, which is the topic of Chapter 12.


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

## 11.3 Cluster Analysis

In Section 11.1 we learned the GMM from fully labelled data, and in Section 11.2 from partially labelled data. In both settings the underlying task was still classification: we had a set of meaningful classes in mind and an interest in predicting them, even when most of the labels happened to be missing. We now take the procedure to its logical extreme and remove the labels altogether, so that the training data consists only of inputs, $\mathcal{T} = \{\mathbf{x}_i\}_{i=1}^n$. As we will see, this seemingly small change opens the door to a genuinely different kind of problem.

### Unsupervised Learning of the Gaussian Mixture Model

Recall the GMM (11.1), a joint model for $\mathbf{x}$ and $y$, whose marginal distribution over the inputs alone is the mixture of Gaussians (11.2). When no labels are available, we regard $y$ as a *latent* random variable – a variable that exists in the model but is never observed in the data. We still learn the joint model $p(\mathbf{x}, y)$, but from data containing only inputs $\{\mathbf{x}_i\}_{i=1}^n$. Formally, this amounts to the maximum likelihood problem

$$\widehat{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta}} \ln p(\{\mathbf{x}_i\}_{i=1}^n \mid \boldsymbol{\theta}), \tag{11.12}$$

where the parameters $\boldsymbol{\theta} = \{\boldsymbol{\mu}_m, \boldsymbol{\Sigma}_m, \pi_m\}_{m=1}^M$ are now to be inferred without any guidance from output labels. Intuitively, learning the GMM from such unlabelled data amounts to figuring out which $\mathbf{x}_i$ values come from the same class-conditional distribution $p(\mathbf{x} \mid y)$, based on their similarity.

Conveniently, we already have a tool for this. Method 11.2, which we devised for the semi-supervised case, also works for completely unlabelled data $\{\mathbf{x}_i\}_{i=1}^n$. We just need to replace the initialisation (line 1) with some pragmatic choice of initial $\{\widehat{\pi}_m, \widehat{\boldsymbol{\mu}}_m, \widehat{\boldsymbol{\Sigma}}_m\}_{m=1}^M$, since there are no labelled data points to start from. We repeat the algorithm with these minor modifications in Method 11.3 for convenience, and apply it to a set of unlabelled data in Figure 11.7.

Let us step back and consider what the algorithm in Figure 11.7 has actually accomplished. Starting from nothing but a collection of unlabelled inputs, it has partitioned them into $M$ groups of mutually similar points and fitted a Gaussian to each. We never set out to solve a specific prediction task – and yet we have ended up solving a well-defined problem in its own right, one that is of great practical interest, namely *clustering*: finding groups of similar $\mathbf{x}$-values in the data space and associating these with a discrete set of clusters.

From a mathematical and methodological point of view, clustering is intimately related to classification. We assign a discrete index to each cluster and say that all $\mathbf{x}$-values in the $m$th cluster are of class $y = m$, very much like a class label, save for the fact that there is no class label in the training data. This is exactly the role played by the latent variable $y$ above.

From a more application-oriented point of view, there are some additional differences. In classification, we usually know what the different classes correspond to. They

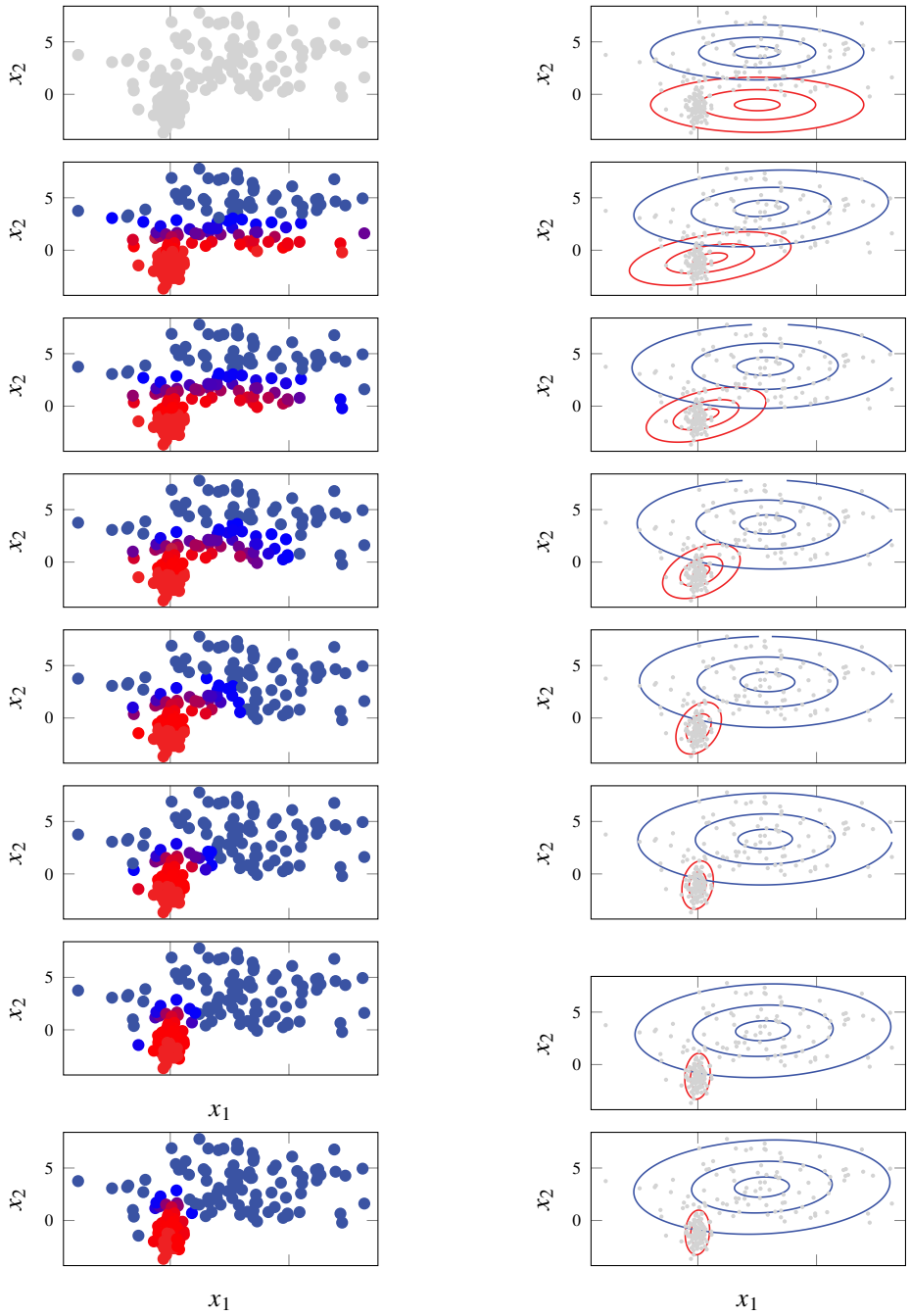



**Figure 11.7:** Method 11.3 applied to an unsupervised problem, where all training data points are unlabelled. In practice, the only difference from Figure 11.6 is the initialisation (in the upper row), which here is done arbitrarily instead of using labelled data points. The left panels show the soft assignment of each data point to the components after each iteration (colour-coded), and the right panels show the corresponding learned model.


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September* 1, 2026, `http://smlbook.org`

**Learn** the GMM

**Data:** Unlabelled training data $\mathcal{T} = \{\mathbf{x}_i\}_{i=1}^n$, number of clusters $M$
**Result:** Gaussian mixture model

1 Initialise $\widehat{\boldsymbol{\theta}} = \{\widehat{\pi}_m, \widehat{\boldsymbol{\mu}}_m, \widehat{\boldsymbol{\Sigma}}_m\}_{m=1}^M$
2 **repeat**
3 For each $\mathbf{x}_i$ in $\{\mathbf{x}_i\}_{i=1}^n$, compute the prediction $p(y \mid \mathbf{x}_i, \widehat{\boldsymbol{\theta}})$ according to (11.6) using the current parameter estimates $\widehat{\boldsymbol{\theta}}$.
4 Update the parameter estimates $\widehat{\boldsymbol{\theta}} \leftarrow \{\widehat{\pi}_m, \widehat{\boldsymbol{\mu}}_m, \widehat{\boldsymbol{\Sigma}}_m\}_{m=1}^M$ according to (11.11)
5 **until** *convergence*

**Method 11.3:** Unsupervised learning of the GMM

are typically specified as part of the problem formulation, and the objective is to build a predictive classifier. Clustering, on the other hand, is often applied in a more exploratory way. We might expect that there are groups of data points with similar properties, and the objective is to group them together to obtain a better understanding of the data. The clusters might not correspond to any interpretable classes. Moreover, the number of clusters is typically unknown and left to the user to decide.

There are a few important details when learning the GMM in the unsupervised setting which deserve some attention. First, the number of clusters $M$ has to be specified in order to run the algorithm. We discuss this hyperparameter choice in more detail below. Second, since there are only unlabelled data points, the indexation of the $M$ Gaussian components becomes arbitrary. Put differently, all possible permutations of the cluster labels will have the same likelihood. In Figure 11.7 this means that the colours (red and blue) are interchangeable, and the only reason for why we ended up with this particular solution is that we initialised the blue cluster in the upper part and the red in the lower part of the data space.

Related to this is that the maximum likelihood problem (11.12) is a non-convex optimisation problem. The EM algorithm is only guaranteed to converge to a stationary point, which means that a poor initialisation can result in convergence to a poor local optimum. In the semi-supervised setting, we could use the labelled training data points as a way to initialise the method, but this is not possible in a fully unsupervised setting. Hence, the initialisation becomes an important detail to consider. A pragmatic approach is to run Method 11.3 multiple times with different random initialisations.

Finally, there is a subtle issue with the maximum likelihood problem (11.12) itself, that we have so far swept under the rug. Without any constraints on the parameters $\boldsymbol{\theta}$, the unsupervised maximum likelihood problem for a GMM is in fact ill-posed, in the sense that the likelihood is unbounded. The problem is that the peak value of the Gaussian probability density function becomes larger and larger as the (co-)variance approaches zero. For any $M \geq 2$, the GMM is in principle able to exploit this fact to attain an infinite likelihood. This is possible by focusing one of the clusters on a

single data point; by centering the cluster on the data point and then shrinking the (co-)variance towards zero, the likelihood of this particular data point goes to infinity. The remaining $M - 1$ clusters then just have to cover the remaining $n - 1$ data points so that their likelihoods are bounded away from zero.[5] In practice, the EM algorithm will often get stuck in a local optimum before this 'degeneracy' shows itself, but it is nevertheless a good idea to regularise or constrain the model to make it more robust to this potential issue. One simple solution is to add a small constant value to all diagonal elements of the covariance matrices $\boldsymbol{\Sigma}_m$, thereby preventing them from degenerating to zero.

## The Expectation–Maximisation Algorithm

At several points now – first in the semi-supervised setting of Section 11.2, and again in Method 11.3 above – we have leaned on an iterative procedure that alternates between predicting the missing labels and updating the model parameters. We claimed, without justification, that this procedure solves the relevant maximum likelihood problem, and that it is an instance of the *expectation–maximisation (EM)* algorithm. It is now time to make good on that promise. The discussion applies just as well to the semi-supervised case as to the fully unsupervised case, and to keep things concrete we phrase it in terms of the latter: Method 11.3 is exactly the EM algorithm applied to the unsupervised maximum likelihood problem (11.12).

The EM algorithm is a general tool for solving maximum likelihood problems in probabilistic models with latent variables, that is, models containing both observed and unobserved random variables. In the current setting, the latent variables are $\{y_i\}_{i=1}^n$, where $y_i \in \{1, \ldots, M\}$ is the cluster index for data point $\mathbf{x}_i$. For notational brevity, we stack these latent cluster indices into an $n$-dimensional vector $\mathbf{y}$. Similarly, we stack the observed data points $\{\mathbf{x}_i\}_{i=1}^n$ into an $n \times p$ matrix $\mathbf{X}$. The task is thus to maximise the *observed data log-likelihood* $\ln p(\mathbf{X} \mid \boldsymbol{\theta})$ with respect to the model parameters $\boldsymbol{\theta} = \{\boldsymbol{\mu}_m, \boldsymbol{\Sigma}_m, \pi_m\}_{m=1}^M$.

The challenge we face is that the observed data likelihood is not readily available, due to the presence of the latent variables $\mathbf{y}$ in the model specification. Evaluating the log-likelihood requires marginalising out these variables, $\ln p(\mathbf{X} \mid \boldsymbol{\theta}) = \sum_i \ln p(\mathbf{x}_i \mid \boldsymbol{\theta}) = \sum_{i=1}^N \ln \sum_{m=1}^M p(\mathbf{x}_i, m \mid \boldsymbol{\theta})$, which has no closed-form expression for the GMM because the logarithm sits outside the sum. In the EM algorithm, we address this challenge by alternating between computing an *expected* log-likelihood and then *maximising* this expected value to update the model parameters.

Let $\widehat{\boldsymbol{\theta}}$ denote the current estimate of $\boldsymbol{\theta}$ at some intermediate iteration of Method 11.3. This can be some arbitrary parameter configuration (for instance corresponding to the initialisation at the first iteration). Then, one iteration of the EM algorithm consists of

[5] This degeneracy can happen in the semi-supervised setting as well, but only if there are $p$ or fewer labelled data points of each class.

the following two steps:

$$\text{(E)} \qquad \text{Compute } Q(\boldsymbol{\theta}) \overset{\text{def}}{=} \mathbb{E}\left[\ln p(\mathbf{X}, \mathbf{y} \mid \boldsymbol{\theta}) \mid \mathbf{X}, \widehat{\boldsymbol{\theta}}\right],$$

$$\text{(M)} \qquad \text{Update } \widehat{\boldsymbol{\theta}} \leftarrow \arg\max_{\boldsymbol{\theta}} Q(\boldsymbol{\theta}).$$

The algorithm alternates between these two steps until convergence. It can be shown that the value of the observed data log-likelihood increases at each iteration of the procedure, unless it has reached a stationary point (where the gradient of the log-likelihood is zero). Hence, it is a valid numerical optimisation algorithm for solving (11.12).

To see that this procedure boils down to Method 11.3 for the GMM model, we start by expanding the E-step. The expected value is computed with respect to the conditional distribution $p(\mathbf{y} \mid \mathbf{X}, \widehat{\boldsymbol{\theta}})$. This represents our current belief regarding the cluster assignment for all data points, given the observed data and the current parameter configuration $\widehat{\boldsymbol{\theta}}$. We thus have

$$Q(\boldsymbol{\theta}) = \mathbb{E}\left[\ln p(\mathbf{X}, \mathbf{y} \mid \boldsymbol{\theta}) \mid \mathbf{X}, \widehat{\boldsymbol{\theta}}\right] = \sum_{\mathbf{y}} \ln\left(p(\mathbf{X}, \mathbf{y} \mid \boldsymbol{\theta})\right) p(\mathbf{y} \mid \mathbf{X}, \widehat{\boldsymbol{\theta}}). \tag{11.13}$$

The first expression in the sum is referred to as the *complete data log-likelihood*. It is the log-likelihood that we would have, if the latent variables were known. Since it involves both the observed data and the latent variables (that is, the 'complete data'), it is readily available from the model:

$$\ln p(\mathbf{X}, \mathbf{y} \mid \boldsymbol{\theta}) = \sum_{i=1}^{n} \ln p(\mathbf{x}_i, y_i \mid \boldsymbol{\theta}) = \sum_{i=1}^{n} \left\{\ln \mathcal{N}\left(\mathbf{x}_i; \boldsymbol{\mu}_{y_i}, \boldsymbol{\Sigma}_{y_i}\right) + \ln \pi_{y_i}\right\}. \tag{11.14}$$

Each term in this expression depends only on one of the latent variables, so by linearity of expectation we can evaluate the expectation in (11.13) one term at a time; and because the data points are conditionally independent given $\boldsymbol{\theta}$, the expectation of the $i$th term depends on $\mathbf{y}$ only through the marginal posterior $p(y_i = m \mid \mathbf{x}_i, \widehat{\boldsymbol{\theta}})$ of its own cluster index. Plugging the complete data log-likelihood into (11.13), we therefore get

$$Q(\boldsymbol{\theta}) = \sum_{i=1}^{n} \sum_{m=1}^{M} w_i(m) \left\{\ln \mathcal{N}\left(\mathbf{x}_i; \boldsymbol{\mu}_m, \boldsymbol{\Sigma}_m\right) + \ln \pi_m\right\}, \tag{11.15}$$

where $w_i(m) = p(y_i = m \mid \mathbf{x}_i, \widehat{\boldsymbol{\theta}})$ (just like (11.11a) with all $y_i$ missing) is the probability that the data point $\mathbf{x}_i$ belongs to cluster $m$, computed based on the current parameter estimates $\widehat{\boldsymbol{\theta}}$.

Comparing this with the log-likelihood in the supervised setting (11.3b), when all labels $\{y_i\}_{i=1}^n$ are known, we note that the two expressions are very similar. The only difference is that the indicator function $\mathbb{I}\{y_i = m\}$ in (11.3b) is replaced by the weight $w_i(m)$ in (11.15). In words, instead of making a hard cluster assignment of each data point based on a given class label, we make a soft cluster assignment based on the probabilities that this data point belongs to the different clusters.

We will not go into the details, but it is hopefully not hard to believe that maximising (11.15) with respect to $\boldsymbol{\theta}$ – which is what we do in the M-step of the algorithm – gives (11.11), which is similar to the supervised setting but where the training data points are weighted by $w_i(m)$.

Putting this together, we conclude that one iteration of the EM algorithm indeed corresponds to one iteration of Method 11.3, the procedure we applied to the data in Figure 11.7. The same argument, with the weights $w_i(m)$ fixed to 0 or 1 for the labelled data points as in (11.11), shows that Method 11.2 is the EM algorithm for the semi-supervised problem, which justifies the claims we made in Section 11.2.

### $k$-Means Clustering

Before leaving this section, we will introduce an alternative clustering method known as $k$-means. This algorithm is in many ways similar to the GMM model for clustering discussed above but is derived from a different objective and lacks the generative interpretation of the GMM. The $k$ in $k$-means refers to the number of clusters, so to agree with our notation, we should perhaps refer to it as $M$-means. However, the term $k$-means is so well established that we will keep it as the name of the method, but we nevertheless let $M$ denote the number of clusters.

The key difference between the GMM and $k$-means is that in the former we model cluster assignments probabilistically, whereas in the latter we make 'hard' cluster assignments. That is, we partition the training data points $\{\mathbf{x}_i\}_{i=1}^n$ into $M$ distinct clusters $R_1, R_2, \ldots, R_M$, where each data point $\mathbf{x}_i$ is a member of exactly one cluster $R_m$. $k$-means clustering then amounts to selecting the clusters so that the sum of pairwise squared Euclidean distances *within each cluster* is minimised,

$$\arg \min_{R_1,R_2,\ldots,R_M} \sum_{m=1}^{M} \frac{1}{|R_m|} \sum_{\mathbf{x},\mathbf{x}' \in R_m} \|\mathbf{x} - \mathbf{x}'\|_2^2, \tag{11.16}$$

where $|R_m|$ is the number of data points in cluster $R_m$. The intention of (11.16) is to select the clusters such that all points within each cluster are as similar as possible. It can be shown that the problem (11.16) is equivalent to selecting the clusters such that the distances to the cluster centres, summed over all data points, is minimised,

$$\arg \min_{R_1,R_2,\ldots,R_M} \sum_{m=1}^{M} \sum_{\mathbf{x} \in R_m} 2\|\mathbf{x} - \widehat{\boldsymbol{\mu}}_m\|_2^2. \tag{11.17}$$

Here $\widehat{\boldsymbol{\mu}}_m$ is the centre of cluster $m$, that is the mean of all data points $\mathbf{x}_i \in R_m$.

Unfortunately both (11.16) and (11.17) are combinatorial problems, meaning that we cannot expect to solve them exactly if the number of data points $n$ is large. However, an approximate solution can be found as follows:

(i) Set the cluster centers $\widehat{\boldsymbol{\mu}}_1, \widehat{\boldsymbol{\mu}}_2, \ldots, \widehat{\boldsymbol{\mu}}_M$ to some initial values;

(ii) Determine which cluster $R_m$ each $\mathbf{x}_i$ belongs to, that is, find the cluster center $\widehat{\boldsymbol{\mu}}_m$ that is closest to $\mathbf{x}_i$ for all $i = 1, \ldots, n$;


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

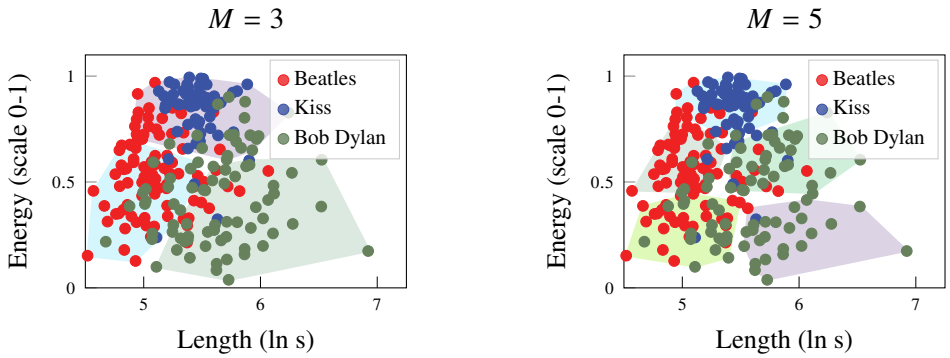



**Figure 11.8:** $k$-means applied to the music classification data Example 2.1. In this example, we actually have labelled data, so the purpose of applying a clustering algorithm to the inputs (without considering the corresponding labels) is purely for illustrative purposes. We try $M = 3$ (left) and $M = 5$ (right).

(iii) Update the cluster centers $\widehat{\boldsymbol{\mu}}_m$ as the average of all $\mathbf{x}_i$ that belongs to $R_m$;

and then iterate steps (ii) and (iii) until convergence.

This procedure is an instance of Lloyd's algorithm, but it is often simply called 'the $k$-means algorithm'. Comparing this with the EM algorithm in Method 11.3, we see a clear resemblance. As pointed out above, the key difference is that the EM algorithm uses a soft cluster assignment based on the estimated cluster probabilities, whereas $k$-means makes a hard cluster assignment in step (ii). Another difference is that $k$-means measures similarity between data points using Euclidean distance, whereas the EM algorithm applied to the GMM model takes the covariance of the clusters into account.[6]

Similarly to the EM algorithm, Lloyd's algorithm will converge to a stationary point of the objective (11.17), but it is not guaranteed to find the global optimum. In practice it is common to run it multiple times, each with a different initialisation in step (i), and pick the result of the run for which the objective in (11.16)/(11.17) attains the smallest value. As an illustration of $k$-means, we apply it to the input data from the music classification problem in Example 2.1; see Figure 11.8.

The name of the algorithm, $k$-means, is reminiscent of another method studied in this book, namely $k$-NN. The two methods do have some similarities, in particular that they both use Euclidean distance to define similarities in the input space. This implies that $k$-means, just as $k$-NN, is sensitive to the normalisation of the input values. However, the two methods should not be confused. While $k$-NN is a supervised learning method (applicable to classification and regression problems), $k$-means is a method for solving the (unsupervised) clustering problem. Note in particular that the '$k$' in the name has a quite different meaning for the two methods.

[6] Put differently, the EM algorithm for the GMM model uses the Mahalanobis distance instead of Euclidean distance.

### Choosing the Number of Clusters

In both the GMM model and the $k$-means algorithm for clustering, we need to select the number of clusters $M$ before running the corresponding algorithm. Hence, unless there is some application-specific prior knowledge regarding how to select $M$, this becomes a hyperparameter to choose. Unlike many other model selection problems, it is not possible to optimise $M$ simply by taking the value that gives the smallest training cost (negative of (11.12) for GMM, or (11.17) for $k$-means). The reason is that increasing $M$ to $M + 1$ will always give more flexibility to the model, and this increased flexibility can only decrease the value of the cost function. In the extreme case when $M = n$, we would end up with the trivial (and uninteresting) solution where each data point is assigned to its own cluster, which would be a sort of overfitting.

Validation techniques such as hold-out and cross-validation (see Chapter 4) can be used to guide the model selection, but they need to be adapted to the unsupervised setting (specifically, there is no new data error $E_{\text{new}}$ for the clustering model). For instance, for the GMM, which is a probabilistic generative model, it is possible to use the likelihood of a held-out validation data set to find a suitable value for $M$. That is, we set aside some validation data $\{\mathbf{x}'_j\}_{j=1}^{n_v}$ which is not used to learn the clustering model. We then train different models for different values of $M$ on the remaining data and evaluate the held-out likelihood $p(\{\mathbf{x}'_j\}_{j=1}^{n_v} \mid \widehat{\boldsymbol{\theta}}, M)$ for each candidate model. The model with the largest held-out likelihood is then selected as the final clustering model.

This provides us with a systematic way of selecting $M$; however, in the unsupervised setting, such validation methods should be used with care. In the context of supervised learning of a predictive model, minimising the new data (prediction) error is often the ultimate goal of the model, so it makes sense to base the evaluation on this. But in the context of clustering it is not necessarily the case that minimising the 'clustering loss' on new data is what we are really after. Instead, clustering is often applied to gain insights regarding the data, by finding a *small number* of clusters where data points within each cluster have similar characteristics. Thus, as long as the model results in a coherent and meaningful grouping of the data points, we might favour a smaller number of clusters, even if a larger number would result in a better validation loss.

One heuristic approach to handling this is to fit models of different orders, $M = 1, 2, \ldots, M_{\text{max}}$. We then plot the loss (either the training loss, the validation loss, or both) as a function of $M$. Based on this plot, the user can make a subjective decision about when the decrease in the objective appears to level off, so that it is unjustified to increase the model complexity further. That is, we select $M$ such that the gain in going from $M$ to $M + 1$ clusters is insignificant. If the dataset indeed has a few distinct clusters, this graph will typically look like an elbow, and this method for selecting $M$ is thus sometimes called the *elbow method*. We illustrate it for the $k$-means method in Figure 11.9.


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

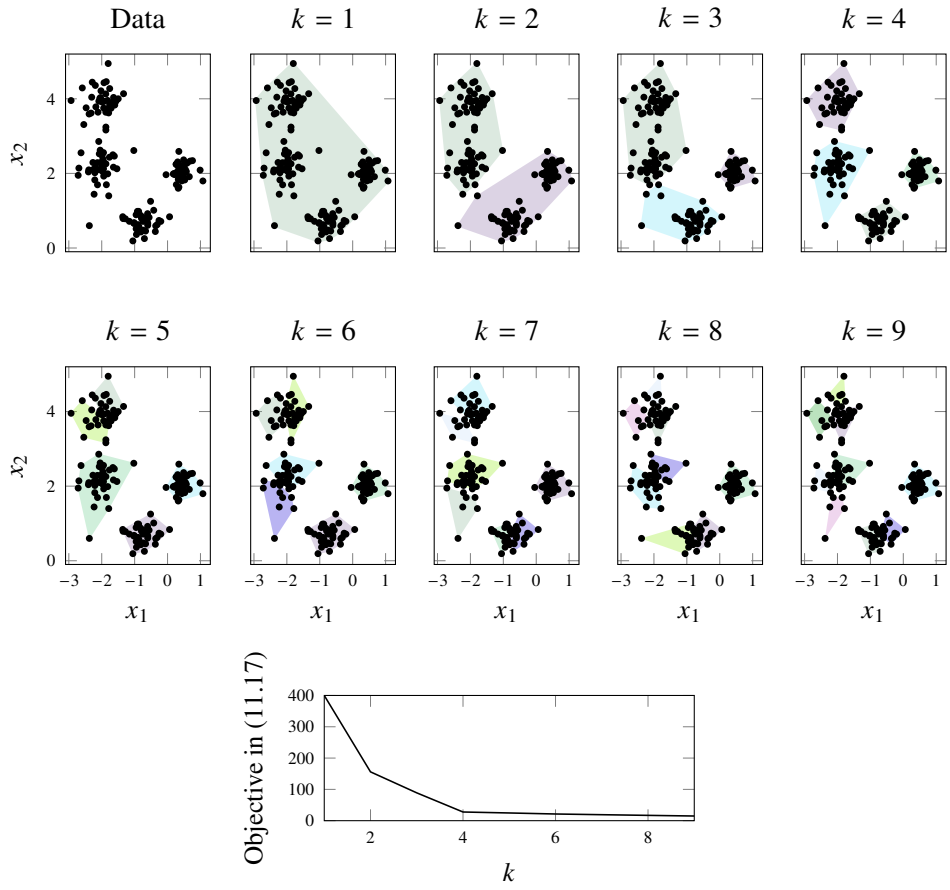



**Figure 11.9:** For selecting $M$ in $k$-means, we can use the so-called elbow method, which amounts to trying different values of $M$ (the upper panels) and recording the objective in (11.16) (the bottom panel). To select $M$, we look for a 'bend' in the bottom panel. In the ideal case, there is a very distinct kink, but for this particular data, we could either draw the conclusion that $M = 2$ or $M = 4$, and it is up to the user to decide. Note that in this example the data has only two dimensions, and we can therefore show the clusters themselves and compare them visually. If the data has more than two dimensions, however, we have to select $M$ based only on the 'elbow plot' in the bottom panel.

## 11.4 Further Reading

The topics covered in this chapter – generative modelling, discriminant analysis, and clustering – are treated in many textbooks on machine learning, including C. M. Bishop (2006), Hastie et al. (2009, Chapter 14), and Murphy (2012).

A longer discussion of the GMM, and the related $k$-means, is found in C. M. Bishop (2006, Chapter 9), which also covers the EM algorithm in detail. For a more detailed discussion of the LDA and QDA classifiers, see Hastie et al. (2009, Section 4.3) or Mardia et al. (1979, Chapter 10).

For more discussion on the fundamental choice between generative and discriminative models, see C. M. Bishop and Lasserre (2007), Liang and Jordan (2008), Ng and Jordan (2001), and Xue and Titterington (2008) and the textbook by Jebara (2004).

The generative perspective extends far beyond the Gaussian mixture model. In the next chapter we discuss how deep neural networks can be used to build generative models of much more complex and high-dimensional data, and we give further reading on those methods there.


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September* 1, 2026, `http://smlbook.org`

# 12 Deep Generative Models and Representation Learning

In Chapter 11 we introduced generative modelling through the Gaussian mixture model. It was the generative framework that made the model so versatile: learnable from labelled, partially labelled, or completely unlabelled data, and available for drawing samples as well as possible to convert into a classifier whenever we wished. But part of the story is also the fact that it was built from particularly well-behaved pieces. A categorical distribution over the classes and a Gaussian distribution for each class-conditional density are *algebra-friendly*: we can multiply them, marginalise them, and condition on them, and the result is again something we can write down in closed form. This is why, for instance, forming the conditional distribution (11.5) turned the generative GMM into the QDA classifier in a single line, and why the learning steps reduced to simple averages.

In this chapter we keep the generative point of view but replace those simple building blocks with *deep neural networks*, in order to model far more complex and high-dimensional data such as natural images, audio, or text. Conceptually, nothing changes: we still wish to model the distribution of the data, sample from it, and use it for downstream tasks. In practice, however, a great deal changes, because neural networks are emphatically *not* algebra-friendly. The tidy manipulations that the GMM permitted (writing down the joint distribution, integrating out a latent variable, conditioning to obtain a predictive distribution, and so on) no longer yield closed-form expressions once a deep network sits in the middle of the model. The required integrals and inverses are intractable, and we can no longer simply plug the model into the probability calculus and expect something usable to fall out. Care has to be taken. As we will see, each of the deep generative models in this chapter can be understood as a different strategy for coping with precisely this intractability while combining the generative approach with the powerful deep neural networks. In Section 12.1 we will introduce three such approaches, namely normalising flows, generative adversarial networks, and diffusion models.

There is also a second, closely related, theme in this chapter, namely *representation learning*. With representation learning, the focus shifts to the latent representation of the data that resides within many models, and in particular deep neural networks. In most other approaches, the latent representations are merely seen as an artifact, whereas they are the primary goal in representation learning. We will introduce two such frameworks, *contrastive learning* and the *auto-encoder*, and thereafter combine the latter with the generative approach into the *variational auto-encoder* which simultaneously is a representation learner and a deep generative model.

Finally, in Section 12.3, we study the special case of a linear auto-encoder, which retrieves the classical *Principal Component Analysis* method for dimensionality reduction.

## 12.1 Deep Generative Modelling

Key to the generative modelling paradigm is that we model the input $\mathbf{x}$ as a random variable, and it thus requires making some assumptions regarding its distribution. In the GMM of Chapter 11, we used a Gaussian to model the (class-conditional) distribution of $\mathbf{x}$. This assumption does not mean that we truly believe that the data is Gaussian, but rather that it is close enough to being Gaussian so that we can obtain a useful model. In many situations, however, the Gaussian assumption is an oversimplification, which can limit the performance of the resulting model.

One way to relax the assumption is to manually design some alternative distribution that is believed to better correspond to the properties of the data. However, in many situations it is very challenging to come up with a suitable distribution 'by hand', not least when $\mathbf{x}$ is high-dimensional and with complex dependencies between its individual coordinates. In this section, we will consider an alternative approach, which is to view $\mathbf{x}$ as a (highly nonlinear) transformation of some simple random variable $\mathbf{z}$. With a high degree of flexibility in the transformation, we can model very complex distributions over $\mathbf{x}$ in this way. Specifically, we will discuss how deep neural networks (Chapter 6) can be used in this context, resulting in so-called *deep generative models*. The random variable $\mathbf{z}$ that we feed through the transformation can be thought of as a *latent representation* of $\mathbf{x}$ – the same notion of representation that we will study from the opposite direction in Section 12.2.

Since the key challenge in developing such a flexible non-Gaussian generative model is to construct the distribution over $\mathbf{x}$, we will throughout this section drop the class, or cluster, label $y$ from the model. That is, we will try to learn the distribution $p(\mathbf{x})$ in an unsupervised way, without assuming that there are any clusters in the data. The purpose of this is twofold. First, learning generative models for high-dimensional data can be useful even in the absence of distinct clusters in the distribution. Second, it simplifies the notation in the presentation below. If we *do expect* that the data contains clusters, then the methods presented below can easily be generalised to model the class-conditional distribution $p(\mathbf{x} \mid y)$ instead.

The problem we are concerned with in this section can thus be formulated as: given a training data set $\{\mathbf{x}_i\}_{i=1}^n$ of $n$ independent samples from some distribution $p(\mathbf{x})$, learn a parametric model of this distribution.

### Invertible Non-Gaussian Models and Normalising Flows

To lay the foundation for non-Gaussian deep generative models, let us for the time being stick with a simple Gaussian model for the data we want to model:

$$p(\mathbf{x}) = \mathcal{N}(\mathbf{x}; \mu, \Sigma) \,. \tag{12.1}$$


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026, http://smlbook.org*

The parameters of this model are the mean vector and the covariance matrix, $\boldsymbol{\theta} = \{\mu, \Sigma\}$, which can be learned from the training data $\{\mathbf{x}_i\}_{i=1}^n$ by maximum likelihood. As we discussed in Chapter 11, this boils down to estimating $\mu$ by the sample mean and $\Sigma$ by the sample covariance (analogously to (11.4), but with a single class).

Since a linear transformation of a Gaussian random vector is also Gaussian, an equivalent representation of (12.1) is to introduce a random variable $\mathbf{z}$ of the same dimension $p$ as $\mathbf{x}$, following a standard Gaussian distribution,[1]

$$p_{\mathbf{z}}(\mathbf{z}) = \mathcal{N}(\mathbf{z}; 0, I)\,, \tag{12.2a}$$

and then expressing $\mathbf{x}$ by a linear change of variables,

$$\mathbf{x} = \mu + L\mathbf{z}. \tag{12.2b}$$

Here $L$ is any matrix (for example the Cholesky factorisation) such that $LL^\mathsf{T} = \Sigma$. Note that, in this representation, the distribution $p_{\mathbf{z}}(\mathbf{z}) = \mathcal{N}(\mathbf{z}; 0, I)$ takes a very simple and generic form, which is independent of the model parameters. Instead, the parameters are shifted to the transformation (12.2b). Specifically, we can use the alternative re-parameterisation $\boldsymbol{\theta} = \{\mu, L\}$, which directly defines the linear transformation.

The models (12.1) and (12.2) are equivalent, so we have not yet accomplished anything with this reformulation. However, the latter form suggests a non-linear generalisation. Specifically, we can model the *distribution* of $\mathbf{x}$ indirectly as the transformation of a simple Gaussian random variable $\mathbf{z}$,

$$p_{\mathbf{z}}(\mathbf{z}) = \mathcal{N}(\mathbf{z}; 0, I)\,, \tag{12.3a}$$

$$\mathbf{x} = f_{\boldsymbol{\theta}}(\mathbf{z}), \tag{12.3b}$$

for some arbitrary parametric function $f_{\boldsymbol{\theta}}$. Even though we start from a Gaussian, the implied distribution of $\mathbf{x}$ is going to be non-Gaussian due to the non-linear transformation. Indeed, we can model arbitrarily complex distributions in this way by considering complex and flexible non-linear transformations. We illustrate this idea in Example 12.1.

**Example 12.1 A two-dimensional flow by hand**

To see how a latent simple Gaussian variable $\mathbf{z}$ can connect to a non-Gaussian distribution, consider a case with dimension $p = 2$. We start from a standard Gaussian latent variable $\mathbf{z} = (z_1, z_2) \sim \mathcal{N}(0, I)$ and define $\mathbf{x} = f_{\boldsymbol{\theta}}(\mathbf{z})$ by

$$x_1 = z_1, \qquad x_2 = z_2\, e^{z_1} + z_1^2.$$

In a real normalising flow model this mapping would be learnable networks; here we fix them so we can follow every step by hand. Although $\mathbf{z}$ is a perfectly ordinary

[1] We use a subscript on $p_{\mathbf{z}}(\mathbf{z})$ to emphasise that this is the distribution of $\mathbf{z}$, and to distinguish it from $p(\mathbf{x})$.

Gaussian (left figure below) $p(\mathbf{x})$ is decidedly not: the $z_1^2$ shift bends the distribution along the parabola $x_2 = x_1^2$, producing a curved, 'banana'-shaped density (right figure). Here, the banana-shaped data would correspond to the real data that for which we want a generative model.

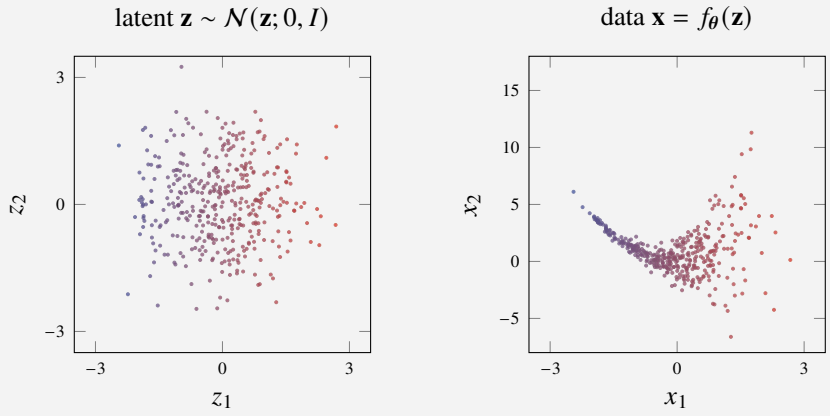



**Fig. 12.1**

The point of a normalising flow model is to learn the mapping $\mathbf{x} = f_{\boldsymbol{\theta}}(\mathbf{z})$, often a neural network, from data. There is however a challenge in learning such model parameters from data. Following the maximum likelihood approach, we would like to solve

$$\widehat{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta}} p(\{\mathbf{x}_i\}_{i=1}^n \,|\, \boldsymbol{\theta}) = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^n \ln p(\mathbf{x}_i \,|\, \boldsymbol{\theta}). \tag{12.4}$$

We thus need to *evaluate* the likelihood $p(\{\mathbf{x}_i\}_{i=1}^n \,|\, \boldsymbol{\theta})$in order to learn the model parameters, but this likelihood is not explicitly given in the model specification (12.3).

To make progress, we start by assuming that $f_{\boldsymbol{\theta}} : \mathbb{R}^p \to \mathbb{R}^p$ is an *invertible* function, with inverse $h_{\boldsymbol{\theta}}(\mathbf{x}) = f_{\boldsymbol{\theta}}^{-1}(\mathbf{x}) = \mathbf{z}$. This implies that $\mathbf{z}$ is of the same dimension as $\mathbf{x}$. Under this assumption, we can make use of the change of variables formula for probability density functions to write

$$p(\mathbf{x} \,|\, \boldsymbol{\theta}) = |\nabla h_{\boldsymbol{\theta}}(\mathbf{x})| \, p_{\mathbf{z}}(h_{\boldsymbol{\theta}}(\mathbf{x})), \tag{12.5a}$$

where

$$\nabla h_{\boldsymbol{\theta}}(\mathbf{x}) = \begin{pmatrix} \frac{\partial h_{\theta,1}(\mathbf{x})}{\partial x_1} & \cdots & \frac{\partial h_{\theta,1}(\mathbf{x})}{\partial x_p} \\ \vdots & \ddots & \vdots \\ \frac{\partial h_{\theta,p}(\mathbf{x})}{\partial x_1} & \cdots & \frac{\partial h_{\theta,p}(\mathbf{x})}{\partial x_p} \end{pmatrix} \tag{12.5b}$$

is the $p \times p$ matrix of all partial derivatives of $h_{\boldsymbol{\theta}}(\mathbf{x})$, referred to as the Jacobian matrix, and $|\nabla h_{\boldsymbol{\theta}}(\mathbf{x})|$ is the absolute value of its determinant. We illustrate this in Example 12.2.

**Example 12.2 A two-dimensional flow by hand: the likelihood**

Let us continue from Example 12.1. The Jacobian for the mapping is triangular,

$$\nabla h_{\theta}(\mathbf{x}) = \begin{pmatrix} 1 & 0 \\ * & e^{-x_1} \end{pmatrix},$$

so we do not even need the off-diagonal entry $*$: the determinant becomes the product of the diagonal, $|\nabla h_{\theta}(\mathbf{x})| = e^{-x_1}$. Plugging this into the change-of-variables formula (12.5a) gives the likelihood for $\mathbf{x}$ explicitly,

$$\begin{aligned} p(\mathbf{x}) &= e^{-x_1}\, \mathcal{N}(x_1; 0, 1)\, \mathcal{N}\Big((x_2 - x_1^2)e^{-x_1}; 0, 1\Big) \\ &= \frac{1}{2\pi} \exp\Big(-x_1 - \tfrac{1}{2}x_1^2 - \tfrac{1}{2}\big((x_2 - x_1^2)e^{-x_1}\big)^2\Big). \end{aligned}$$

Had this toy example contained learnable parameters, this likelihood would have served as the learning objective.

This is the whole idea of a normalising flow in a nutshell: a simple latent distribution, an invertible map with a cheap (here triangular) Jacobian, yielding an exact density for $\mathbf{x}$ that we could evaluate and maximise. Using flexible models for $f_{\theta}$ is what turns this toy into a model applicable to real high-dimensional data.

Plugging this expression into the maximum likelihood problem (12.4), we can thus learn the model as:

$$\widehat{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{n} \left(\ln |\nabla h_{\boldsymbol{\theta}}(\mathbf{x}_i)| + \ln p_{\mathbf{z}}(h_{\boldsymbol{\theta}}(\mathbf{x}_i))\right), \tag{12.6}$$

where both terms of the loss function are now given by the model specification (12.3). This provides us with a practical approach for learning the transformation-based generative model from data, although we make the following observations:

(i) The inverse mapping $h_{\theta}(\mathbf{x}) = f_{\theta}^{-1}(\mathbf{x})$ needs to be explicitly available, since it is part of the loss function.

(ii) The Jacobian determinant $|\nabla h_{\theta}(\mathbf{x}_i)|$ needs to be tractable. In the general case, the computational cost associated with (practical algorithms for) computing the determinant of a $p \times p$ matrix scales cubically with $p$. For high-dimensional problems, this easily results in a prohibitively large computational bottleneck, unless the Jacobian has some special structure that can be exploited for faster computation.

(iii) The forward mapping $f_{\theta}(\mathbf{z})$ *does not* enter the loss function, so in principle we can learn the model without explicitly evaluating this function (it is enough to know that it exists). However, if we want to use the model to generate samples from $p(\mathbf{x})$, then explicit evaluation of the forward mapping is also needed. Indeed, the way to sample from the model (12.3) is to first sample a standard

Gaussian vector $\mathbf{z}$ and then propagate this sample through the mapping to obtain a sample $\mathbf{x} = f_{\boldsymbol{\theta}}(\mathbf{z})$.

Designing parametric functions that satisfy these conditions while still being flexible enough to accurately describe complex high-dimensional probability distributions is non-trivial. Models based on neural networks are often used, but to satisfy the requirements on invertibility and computational tractability, special-purpose network architectures are needed. This involves, for instance, restricting the mapping $f_{\boldsymbol{\theta}}$ so that the Jacobian of its inverse becomes a triangular matrix, in which case the determinant is easily computable.

An interesting observation when designing this type of transform-based generative model using neural networks is that it is enough to ensure invertibility and tractability of each layer of the network independently. Assume that $f_{\boldsymbol{\theta}}(\mathbf{z})$ is a network with $L$ layers, where the $l$th layer corresponds to a function $f_{\boldsymbol{\theta}}^{(l)} : \mathbb{R}^p \to \mathbb{R}^p$. We can then write $f_{\boldsymbol{\theta}}(\mathbf{z}) = f_{\boldsymbol{\theta}}^{(L)} \circ f_{\boldsymbol{\theta}}^{(L-1)} \circ \cdots \circ f_{\boldsymbol{\theta}}^{(1)}(\mathbf{z})$, where $\circ$ denotes the composition of functions. The inverse of $f_{\boldsymbol{\theta}}$ is obtained by applying the layer-wise inverse functions in reverse order, $h_{\boldsymbol{\theta}}(\mathbf{x}) = h_{\boldsymbol{\theta}}^{(1)} \circ h_{\boldsymbol{\theta}}^{(2)} \circ \cdots \circ h_{\boldsymbol{\theta}}^{(L)}(\mathbf{x})$, where $h_{\boldsymbol{\theta}}^{(l)}$ is the inverse of $f_{\boldsymbol{\theta}}^{(l)}$. Furthermore, by the chain rule of differentiation and the multiplicativity of determinants, we can express the Jacobian determinant as a product:

$$|\nabla h_{\boldsymbol{\theta}}(\mathbf{x})| = \prod_{l=1}^{L} \left| \nabla h_{\boldsymbol{\theta}}^{(l)}(\mathbf{x}^{(l)}) \right|, \quad \text{where} \quad \mathbf{x}^{(l)} = h_{\boldsymbol{\theta}}^{(l+1)} \circ h_{\boldsymbol{\theta}}^{(l+2)} \circ \cdots \circ h_{\boldsymbol{\theta}}^{(L)}(\mathbf{x}).$$

This means that it is enough to design each $f_{\boldsymbol{\theta}}^{(l)}$ so that it is invertible and has a computationally tractable Jacobian determinant. While this still puts restrictions on the architecture and activation functions used in $f_{\boldsymbol{\theta}}^{(l)}$, there are many ways in which this can be accomplished. We can then build more complex models by stacking multiple such layers after each other, with a computational cost growing only linearly with the number of layers. Models exploiting this property are referred to as *normalising flows*. The idea is that a data point $\mathbf{x}$ 'flows' through a sequence of transformations, $h_{\boldsymbol{\theta}}^{(L)}$, $h_{\boldsymbol{\theta}}^{(L-1)}$, $\ldots$, $h_{\boldsymbol{\theta}}^{(1)}$, and after $L$ such transformations, the data point has been 'normalised'. That is, the result of the sequence of mappings is that the data point has been transformed into a standard Gaussian vector $\mathbf{z}$.

A concrete and widely used way to meet both the Jacobian tractability and the invertibility requirements at once is the *coupling layer*. We split the $p$ coordinates of a layer's input into two groups, $\mathbf{z} = (\mathbf{z}_1, \mathbf{z}_2)$ with $\mathbf{z}_1 \in \mathbb{R}^k$ and $\mathbf{z}_2 \in \mathbb{R}^{p-k}$, and let the layer copy the first group unchanged while applying an element-wise affine transformation to the second, whose scale and shift are computed from the first:

$$\mathbf{x}_1 = \mathbf{z}_1, \qquad \text{and} \qquad \mathbf{x}_2 = \mathbf{z}_2 \odot \exp\big(\boldsymbol{s}(\mathbf{z}_1)\big) + \boldsymbol{t}(\mathbf{z}_1).$$

Here $\boldsymbol{s}, \boldsymbol{t} : \mathbb{R}^k \to \mathbb{R}^{p-k}$ may be arbitrary functions – in particular, deep networks with *no* invertibility requirement of their own, since they are only ever evaluated at $\mathbf{z}_1$ (equivalently $\mathbf{x}_1$), which the layer leaves untouched. This buys us exactly the

two properties demanded above. The inverse is immediate and explicit, $\mathbf{z}_1 = \mathbf{x}_1$ and $\mathbf{z}_2 = (\mathbf{x}_2 - \boldsymbol{t}(\mathbf{x}_1)) \odot \exp\big(-\boldsymbol{s}(\mathbf{x}_1)\big)$, satisfying observation (i); and because $\mathbf{x}_1$ does not depend on $\mathbf{z}_2$, the Jacobian of the inverse is triangular, so its determinant is simply the product of the diagonal entries, $\nabla h_{\boldsymbol{\theta}}(\mathbf{x}) \prod_j = \exp\big(-s_j(\mathbf{z}_1)\big)$, computable in $O(p)$ rather than $O(p^3)$ time, satisfying observation (ii). In fact Example 12.1 was such a coupling model, although it contains no learnable parameters. The price is that each such layer transforms only half of the coordinates; this is remedied by alternating which group is copied between successive layers, so that after a few layers every coordinate has been transformed conditioned on the others.

The normalising flow idea can be extended and adapted to various real-world problems, and has found applications in fields such as physics and finance. We will not pursue the idea any further here, but instead turn to an alternative way of learning deep generative models that circumvents the architectural restrictions of normalising flows, namely so-called *generative adversarial networks*.

### Generative Adversarial Networks

The idea of transforming a Gaussian vector $\mathbf{z}$ by a deep generative model (12.3) to parameterise a complex distribution over data $\mathbf{x}$ is very powerful. However, we noted above that evaluating the data likelihood $p(\mathbf{x} \mid \boldsymbol{\theta})$ implied by the model is non-trivial and imposes certain restrictions on the mapping $f_{\boldsymbol{\theta}}$. Hence, without these restrictions, learning the model by explicit likelihood maximisation is not possible. However, motivated by this limitation, we can ask ourselves: is there some other way of learning the model, which *does not* require evaluating the likelihood?

To answer this question, we note that one useful property of the deep generative model is that *sampling* from the distribution $p(\mathbf{x} \mid \boldsymbol{\theta})$ is trivial, as long as the forward mapping $f_{\boldsymbol{\theta}}(\mathbf{z})$ is available. This is true even in situations when we are unable to evaluate the corresponding probability density function. That is, we can generate 'synthetic' data points from the model, simply by sampling a Gaussian vector $\mathbf{z} \sim \mathcal{N}(0, I)$ and then feeding the obtained sample through the parametric function, $f_{\boldsymbol{\theta}}(\mathbf{z})$. This does not impose any specific requirements on the mapping, such as invertibility. In fact, we do not even require that the dimension of $\mathbf{z}$ is the same as that of $\mathbf{x}$.

*Generative adversarial networks* (GANs) make use of this property for training the model, by comparing synthetic samples (generated by the model) with real samples from the training data set $\{\mathbf{x}_i\}_{i=1}^n$. The basic idea is to learn the model parameters $\boldsymbol{\theta}$ with the objective of making the synthetic samples resemble the real data points as much as possible. If it is difficult to tell the synthetic and actual training data apart, then we can conclude that the learned distribution is a good approximation of the true data distribution. To illustrate the idea, assume that the data we are working with consists of natural images of some type, say pictures of human faces. This is indeed a typical example where these models have shown remarkable capabilities. A data point $\mathbf{x}$ is thus an image of dimension $p = w \times h \times 3$ (width in pixels $\times$ height in pixels $\times$ three colour channels), $\mathbf{z}$ is a Gaussian vector of dimension $q$, and the mapping $f_{\boldsymbol{\theta}} : \mathbb{R}^q \to \mathbb{R}^{w \times h \times 3}$ takes this Gaussian vector and transforms it into the

shape of an image. Without going into details, such mappings can be constructed using deep neural networks in various ways, for instance using upsampling layers and deconvolutions (inverse convolutions). Such networks are reminiscent of convolutional neural networks (see Section 7.1) but go in the other direction – instead of taking an image as input and transforming this to a vector of class probabilities, say, we now take a vector as input and transform this into the shape of an image.

To learn a model $p(\mathbf{x} \mid \boldsymbol{\theta})$ for the distribution of the observed data, we will play a type of game, which goes as follows. At each iteration of the learning algorithm:

(i) 'Flip a coin', that is, set $y = 1$ with probability 0.5 and $y = -1$ with probability 0.5:

   (a) If $y = 1$, then generate a synthetic sample from the model $\mathbf{x}' \sim p(\mathbf{x} \mid \boldsymbol{\theta})$. That is, we sample $\mathbf{z}' \sim \mathcal{N}(0, I)$ and compute $\mathbf{x}' = f_{\boldsymbol{\theta}}(\mathbf{z}')$.

   (b) If $y = -1$, then pick a random sample from the training data set instead. That is, we set $\mathbf{x}' = \mathbf{x}_i$ for some index $i$ sampled uniformly at random from $\{1, \ldots, n\}$.

(ii) Ask a critic to determine if the sample is real or fake. For instance, in the example with pictures of faces, we would ask the question: does $\mathbf{x}'$ look like a real face, or is it synthetically generated?

(iii) Use the critic's reply as a signal for updating the model parameters $\boldsymbol{\theta}$. Specifically, update the parameters with the goal of making the critic as 'confused as possible' regarding whether or not the sample that is presented is real or fake.

The first point is easy to implement, but when we get to the second point, the procedure becomes more abstract. What do we mean by 'critic'? In a practical learning algorithm, using a human-in-the-loop to judge the authenticity of the sample $\mathbf{x}'$ is not feasible. Instead, the idea behind generative adversarial networks is to *learn an auxiliary classifier* alongside the generative model, which plays the role of the critic in the game. Specifically, we design a binary classifier $g_{\boldsymbol{\eta}}(\mathbf{x})$ which takes a data point (for example an image of a face) as input and estimates the probability that this is synthetically generated, that is,

$$g_{\boldsymbol{\eta}}(\mathbf{x}) \approx p(y = 1|\mathbf{x}). \tag{12.7}$$

Here, $\boldsymbol{\eta}$ denotes the parameters of the auxiliary classifier, which are distinct from the parameters $\boldsymbol{\theta}$ of the generative model.

The classifier is learned as usual to minimise some classification loss $L$,

$$\widehat{\boldsymbol{\eta}} = \arg\min_{\boldsymbol{\eta}} \mathbb{E}\big[L(y, g_{\boldsymbol{\eta}}(\mathbf{x}'))\big], \tag{12.8}$$

where the expected value is with respect to the random variables $y$ and $\mathbf{x}'$ generated by the process described above. Note that this becomes a *supervised* binary classification problem, but where the labels $y$ are automatically generated as part of the 'game'.


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

Indeed, since these labels correspond to the flip of a fair coin, we can express the optimisation problem as

$$\min_{\boldsymbol{\eta}} \left\{ \tfrac{1}{2}\mathbb{E}\big[L(1, g_{\boldsymbol{\eta}}(f_{\boldsymbol{\theta}}(\mathbf{z}')))\big] + \tfrac{1}{2}\mathbb{E}\big[L(-1, g_{\boldsymbol{\eta}}(\mathbf{x}_i))\big] \right\}. \tag{12.9}$$

Moving on to the third step of the procedure, we wish to update the mapping $f_{\boldsymbol{\theta}}(\mathbf{z})$, defining the generative model, to make the generated samples as difficult as possible for the critic to reject as being fake. This is in some sense the most important step of the procedure, since this is where we learn the generative model. This is done in a competition with the auxiliary classifier, where the objective for the generative model is to *maximise* the classification loss (12.9) with respect to $\boldsymbol{\theta}$,

$$\max_{\boldsymbol{\theta}} \min_{\boldsymbol{\eta}} \left\{ \tfrac{1}{2}\mathbb{E}\big[L(1, g_{\boldsymbol{\eta}}(f_{\boldsymbol{\theta}}(\mathbf{z}')))\big] + \tfrac{1}{2}\mathbb{E}\big[L(-1, g_{\boldsymbol{\eta}}(\mathbf{x}_i))\big] \right\}. \tag{12.10}$$

This results in a so-called minimax problem, where two adversaries compete for the same objective, one trying to minimise it and the other trying to maximise it. Typically, the problem is approached by alternating between updating $\boldsymbol{\theta}$ and updating $\boldsymbol{\eta}$ using stochastic gradient optimisation. We provide pseudo-code for one such algorithm in Method 12.1.

From an optimisation point of view, solving the minimax problem is more challenging than solving a pure minimisation problem, due to the competing forces that can result in oscillative behaviour. Many modifications and variations of the procedure outlined above have been developed, among other things to stabilise the optimisation and obtain efficient learning algorithms. Still, this is one of the drawbacks with generative adversarial networks compared to, for instance, normalising flows that can be learned by direct likelihood maximisation. Related to this is that, even if we successfully learn the generative model $f_{\boldsymbol{\theta}}(\mathbf{z})$, which implicitly defines the distribution $p(\mathbf{x} \mid \boldsymbol{\theta})$, it cannot be used to evaluate the likelihood $p(\mathbf{x}_\star \mid \boldsymbol{\theta})$ for some newly observed data point $\mathbf{x}_\star$. Whether the absence of an explicit likelihood is an issue or not depends on the application at hand.

### Diffusion Models

A more recent class of deep generative models, which has produced some of the most striking image-generation results to date, is the class of *diffusion models*. The idea of a diffusion model can be described in two parts. First, a *forward* process gradually adds Gaussian noise to a data point $\mathbf{x}_0$ over many steps $t = 1, 2, \ldots, T$, resulting in a sequence $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_T$. The noise schedule is chosen so that $\mathbf{x}_T$ is approximately distributed as a standard Gaussian, regardless of $\mathbf{x}_0$. The forward process is fixed and contains no learnable parameters; it can be thought of as a recipe for slowly destroying all structure in the data.

Second, a *reverse* process is learned, which attempts to undo the forward process one step at a time. Concretely, a neural network is trained to predict the noise that was added, where for each training example a step $t$ is selected at random. This reduces

**Learn** a generative adversarial network

**Data:** Training data $\mathcal{T} = \{\mathbf{x}_i\}_{i=1}^{n}$, initial parameters $\boldsymbol{\theta}$ and $\boldsymbol{\eta}$, learning rate $\gamma$ and batch size $n_b$, critic iterations per generator iteration $T_{\text{critic}}$

**Result:** Deep generative model $f_{\boldsymbol{\theta}}(\mathbf{z})$

**repeat**
  **for** $t = 1, \ldots, T_{\text{critic}}$ **do**
    Sample mini-batch $\{\mathbf{x}_i\}_{i=1}^{n_b}$ from training data
    Sample mini-batch $\{\mathbf{z}_i\}_{i=1}^{n_b}$ independently from $\mathcal{N}(0, I)$
    Compute gradient $\widehat{\mathbf{d}}_{\text{critic}} = \frac{1}{2n_b} \sum_{i=1}^{n_b} \nabla_{\boldsymbol{\eta}} \left\{ L(1, g_{\boldsymbol{\eta}}(f_{\boldsymbol{\theta}}(\mathbf{z}_i))) + L(-1, g_{\boldsymbol{\eta}}(\mathbf{x}_i)) \right\}$
    Update critic: $\boldsymbol{\eta} \leftarrow \boldsymbol{\eta} - \gamma \widehat{\mathbf{d}}_{\text{critic}}$
  **end**
  Sample mini-batch $\{\mathbf{z}_i\}_{i=1}^{n_b}$ independently from $\mathcal{N}(0, I)$
  Compute gradient $\widehat{\mathbf{d}}_{\text{gen.}} = \frac{1}{2n_b} \sum_{i=1}^{n_b} \nabla_{\boldsymbol{\theta}} L(1, g_{\boldsymbol{\eta}}(f_{\boldsymbol{\theta}}(\mathbf{z}_i)))$
  Update generator: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \gamma \widehat{\mathbf{d}}_{\text{gen.}}$
**until** *convergence*

**Sample** from a generative adversarial network

**Data:** Generator model $f_{\boldsymbol{\theta}}$

**Result:** Synthetic sample $\mathbf{x}'$

Sample $\mathbf{z}' \sim \mathcal{N}(0, I)$
Output $\mathbf{x}' = f_{\boldsymbol{\theta}}(\mathbf{z}')$

**Method 12.1:** Training a generative adversarial network.

to a supervised regression problem – predicting the added noise from the noisy data point – which is reminiscent of the self-supervised perspective from Chapter 2. When we want to use the model for generating new samples, we start from a pure Gaussian sample $\mathbf{x}_T \sim \mathcal{N}(0, I)$ and iteratively denoise to produce $\mathbf{x}_{T-1}, \mathbf{x}_{T-2}, \ldots, \mathbf{x}_0$.

Diffusion models share with normalising flows the property that there is a likelihood-style objective for training (a property they also share with the variational auto-encoders we will meet in Section 12.2), and they share with GANs the property that the generator can be a very flexible mapping without any invertibility constraints. They are also natural to combine with conditional information, for instance by conditioning the denoising network on a text embedding produced by a model such as CLIP (Example 12.4), which is the basis for many of the recent text-to-image systems.


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

**Example 12.3 Denoising Diffusion Probabilistic Model (DDPM)**

This example introduces the *denoising diffusion probabilistic model* (DDPM) design, which is the foundation for many modern diffusion models. Recall the two ingredients: a fixed forward process that gradually turns data into noise, and a learned reverse process that turns noise back into data. The forward process in DDPM adds Gaussian noise to a clean data point $\mathbf{x}_0$ over many small steps, governed by a fixed, small, increasing *noise schedule* $\beta_1, \ldots, \beta_T$. A convenient property of Gaussians is that the cumulative effect of all these steps can be written down directly, so that we can jump straight to the noisy data point at any step $t$ without passing through the intermediate ones:

$$\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\,\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\,\boldsymbol{\varepsilon}, \qquad \boldsymbol{\varepsilon} \sim \mathcal{N}(0, \mathbf{I}). \tag{12.11}$$

Here $\boldsymbol{\varepsilon}$ is a single Gaussian noise sample representing the *total* corruption accumulated up to step $t$, and $\bar{\alpha}_t = \prod_{s=1}^{t}(1-\beta_s)$ is a cumulative factor that decreases from $\bar{\alpha}_0 = 1$ (clean data) to $\bar{\alpha}_T \approx 0$ (pure noise). In words, $\mathbf{x}_t$ is the clean data point scaled down and with the noise $\boldsymbol{\varepsilon}$ added; as $t$ grows, $\bar{\alpha}_t$ shrinks and the data is progressively drowned out, until $\mathbf{x}_T$ is indistinguishable from pure noise, $\mathbf{x}_T \sim \mathcal{N}(0, \mathbf{I})$.

Now to the learning. Reversing the process means removing the noise that the forward process added, so the natural quantity to learn is exactly that noise. We therefore introduce a single neural network $\boldsymbol{\varepsilon}_\theta(\mathbf{x}_t, t)$ that takes a noisy data point $\mathbf{x}_t$ and the step index $t$, and tries to predict the total noise $\boldsymbol{\varepsilon}$ that was added to it. It can be shown – we omit the derivation, which rests entirely on the Gaussian structure of (12.11) – that maximising a bound on the likelihood of the data reduces, after simplification, to the strikingly simple objective

$$L(\theta) = \mathbb{E}_{\mathbf{x}_0,\,\boldsymbol{\varepsilon},\,t}\left[\; \left\| \boldsymbol{\varepsilon} - \boldsymbol{\varepsilon}_\theta\big(\sqrt{\bar{\alpha}_t}\,\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\,\boldsymbol{\varepsilon},\; t\big) \right\|^2 \;\right], \tag{12.12}$$

where the expectation is over a clean data point $\mathbf{x}_0$ from the training set, a noise sample $\boldsymbol{\varepsilon} \sim \mathcal{N}(0, \mathbf{I})$, and a step $t$ drawn uniformly from $\{1, \ldots, T\}$. That is the entire training objective.

It is worth pausing on what (12.12) says, since it is the heart of the matter. To produce one training example we (i) take a clean data point $\mathbf{x}_0$ from the training set; (ii) pick a step $t$ at random; (iii) draw a noise sample $\boldsymbol{\varepsilon}$ and form the noisy point $\mathbf{x}_t$ directly via (12.11); and (iv) ask the network to predict $\boldsymbol{\varepsilon}$ from $\mathbf{x}_t$, penalising it by the squared error. In other words, we have manufactured an ordinary supervised *regression* problem – input $\mathbf{x}_t$, target $\boldsymbol{\varepsilon}$ – out of unlabelled data, by corrupting it ourselves with noise of our own choosing. This is another flavour of the self-supervised idea of Chapter 2, and it is why no labels, no likelihood evaluations, and no adversarial game appear anywhere in the training.

Once the network is trained, generating a new data point amounts to running the process in reverse. We start from pure noise $\mathbf{x}_T \sim \mathcal{N}(0, \mathbf{I})$ and, for $t = T, T-1, \ldots, 1$, iterate the update

$$\mathbf{x}_{t-1} = \frac{1}{\sqrt{1-\beta_t}}\left(\mathbf{x}_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\,\boldsymbol{\varepsilon}_\theta(\mathbf{x}_t, t)\right) + \boldsymbol{\zeta}, \qquad \boldsymbol{\zeta} \sim \mathcal{N}(0, \sigma_t^2\mathbf{I}), \tag{12.13}$$

After $T$ steps, $\mathbf{x}_0$ is a fresh sample from the learned distribution. The noise $\boldsymbol{\zeta}$ is added according to the schedule $\sigma_t^2$, with $\sigma_1^2 = 0$, since the end goal after all is to generate random – yet realistic – data samples, instead of performing a deterministic procedure.

The coefficient in front of the network's prediction repays a closer look, because it reveals how diffusion sampling really works. At every step the network outputs its best estimate of the *entire* accumulated noise, $\boldsymbol{\varepsilon}_\theta(\mathbf{x}_t, t)$ – equivalently, its best guess of a fully denoised data point. Yet the update in (12.13) does not remove all of it: the factor $\beta_t/\sqrt{1-\bar{\alpha}_t}$ scales the prediction down to just the sliver of noise that belongs to the current step. The model thus repeatedly predicts the whole denoised result but only takes a small step towards it.

This "predict the whole, apply a fraction" strategy is what gives diffusion models two of their most useful properties. Early in the reverse process (large $t$), the fractional steps shape the coarse, global structure of the sample; later (small $t$), they refine fine detail and texture, giving a natural coarse-to-fine trajectory. And because each step trusts only a fraction of the prediction, an inaccurate guess at one step is not fatal: the network re-examines the slightly cleaner point at the next step and corrects course.

It is worth reflecting on the fact that DDPM formulates the generative modeling into a series of ordinary supervised learning problems. The sophistication of such a modern image generator lies almost entirely in the architecture of the single network $\boldsymbol{\varepsilon}_\theta$ and in the scale of the training data; the training objective itself just remains the squared error.

## 12.2 Representation Learning

One of the recurring observations of modern machine learning is that the way data is *represented* is at least as important as the choice of model itself. Throughout the book, we have seen this idea appear in different perspectives. For example, in Chapter 9, kernel methods rely on a fixed (but possibly infinite) non-linear transformation $\boldsymbol{\phi}(\mathbf{x})$ that re-expresses the input in a more powerful form. The intuition is clear: by representing the input data in another well-chosen space, problems that seem hard in the input space become easier in the transformed space, as illustrated by Example 9.1. The same idea appears also with deep neural networks, Chapter 6, where a nonlinear transformation is *learned* as part of the supervised training, in such a way that the final layers can solve the actual classification or regression task using a simple linear model on top of it. Within deep learning the transformed input data is often referred to as *latent representations* or, particularly in the context of language models, as *embeddings*. We will use the terms interchangeably.

Having a useful latent representation can be valuable in itself, even when no downstream supervised task has been specified yet. A good representation captures the structure of the data in a compact form, and can subsequently be used as a starting point for many different applications: classification, retrieval, visualisation, generation, and so on. In this section, we will look at two methods aimed at learning such latent representations as a primary goal rather than as a by-product of solving a supervised problem. The first, *contrastive learning*, learns embeddings by comparing data points against each other. The second, the *auto-encoder*, learns embeddings by passing the data through a low-dimensional bottleneck. In both cases, we end up training

a model in a way that uses the machinery of supervised learning – gradient-based optimisation of a loss function – but the eventual product is the representation, not the predictions. We will then close the section with the *variational auto-encoder*, a model that is at once an auto-encoder and a deep generative model of the kind discussed in Section 12.1.

### Contrastive Learning

Recall the cross-entropy loss for multi-class classification (3.44). The key idea behind that loss is to maximise the predicted probability of the correct class. Since the predicted probabilities are normalised to sum to one, this implicitly also pushes down the predicted probabilities of all incorrect classes. Now imagine that, instead of predicting a probability, we map each input $\mathbf{x}_i$ to an *embedding* vector $\mathbf{z}_i = f_\theta(\mathbf{x}_i)$, and measure how *close* this embedding is to some target vector $\mathbf{z}'_i$. This is in itself nothing new – it is the basic premise of all supervised learning. However, suppose that in addition to wanting the embedding $\mathbf{z}_i$ to be close to its correct target $\mathbf{z}'_i$, we also want it to be *far away* from all of the other possible targets, $\{\mathbf{z}'_j\}_{j\neq i}$, in the training set. We are thus contrasting the embedding against many alternatives at once. This idea is captured by the following loss function, referred to as the *contrastive loss*:

$$J(\theta) = -\sum_{i=1}^{n} \ln \frac{\exp\left(s(\mathbf{z}_i, \mathbf{z}'_i)\right)}{\sum_{j=1}^{n} \exp\left(s(\mathbf{z}_i, \mathbf{z}'_j)\right)}. \tag{12.14}$$

Here $s(\cdot,\cdot)$ is a similarity score (for example a scalar product, or a negative distance). The expression inside the logarithm has the same softmax-like form as the multi-class cross-entropy, but where the role of the 'classes' is played by the entire collection of training targets $\{\mathbf{z}'_j\}_{j=1}^n$. Maximising the numerator pulls $\mathbf{z}_i$ towards the matched target $\mathbf{z}'_i$, while the normalising denominator pushes it away from all the other targets $\{\mathbf{z}'_j\}_{j\neq i}$.

So far, (12.14) amounts to a slightly unusual but otherwise standard supervised learning problem: the targets $\mathbf{z}'_i$ are given, and we minimise the loss over the parameters $\theta$ of $f_\theta(\mathbf{x})$. The step that pushes this idea into territory beyond ordinary supervised learning is the realisation that

| the targets $\mathbf{z}'_i$ do not have to be pre-defined fixed vectors. |
|---|

They can themselves be *generated by a second model*, which is trained jointly with the first one. By doing so, we can use the contrastive loss to learn representations from data that has no labels in any traditional sense, but that comes equipped with some notion of natural pairing.

This is best illustrated by an example.

### Example 12.4 Contrastive Language–Image Pre-training (CLIP)

A widely cited application of contrastive learning in deep learning is *Contrastive Language–Image Pre-training* (CLIP), for jointly learning representations of images *and* text. The training data for CLIP consists of large amounts of *pairs* of images and their associated captions, $\mathbf{x}_i = (\text{image}_i, \text{text}_i)$ and there is no explicit output variable $y$. The training data for CLIP is more or less scraped directly from the web, typically giving rather low-quality captions, but the performance of the learned model has nevertheless proven to be both impressive and robust.

CLIP uses two distinct neural networks: an *image encoder* $f_\theta$ that maps every image to an embedding vector $P_i = f_\theta(\text{image}_i)$, and a *text encoder* $g_\theta$ that maps any piece of text to another embedding vector $T_i = g_\theta(\text{text}_i)$. The two embedding vectors live in the same high-dimensional embedding space, typically of dimension 512 or 1024. The model is trained by minimising a contrastive loss that uses one encoder's output to retrieve the matching output of the other encoder:

$$J(\theta) = -\sum_{i=1}^{n} \ln \frac{\exp\big(s(P_i, T_i)/\tau\big)}{\sum_{j=1}^{n} \exp\big(s(P_i, T_j)/\tau\big)}, \tag{12.15}$$

where the similarity is taken to be the normalised scalar product

$$s(P_i, T_j) = \frac{P_i^\mathsf{T} T_j}{\|P_i\| \, \|T_j\|}. \tag{12.16}$$

Here $\tau > 0$ is a *temperature* parameter, that can either be set manually or learned during training, that scales the similarities before the softmax. The two encoders are trained jointly (the parameters $\theta$ include both sets of network parameters), and the loss thus pulls each image embedding towards the embedding of its associated caption, while pushing it away from the embeddings of all other captions in the training batch. Connecting to the general formulation (12.14), $P_i$ plays the role of the embedding $\mathbf{z}_i$ produced by the (image) model, and $T_i$ plays the role of the target $\mathbf{z}'_i$, which is itself the output of the second, jointly trained text encoder. (In fact (12.15) is asymmetric because the denominator normalises over $T_j$ and not $P_j$. In practice a second term is typically added to achieve a symmetric formulation.)

The overall idea of CLIP is summarised below. The light green colour in the matrix highlights the diagonal elements, for which the model is optimised to be as close as possible to 1, whereas the off-diagonal elements should become as small as possible, thanks to the contrastive loss.


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September* 1, 2026, `http://smlbook.org`

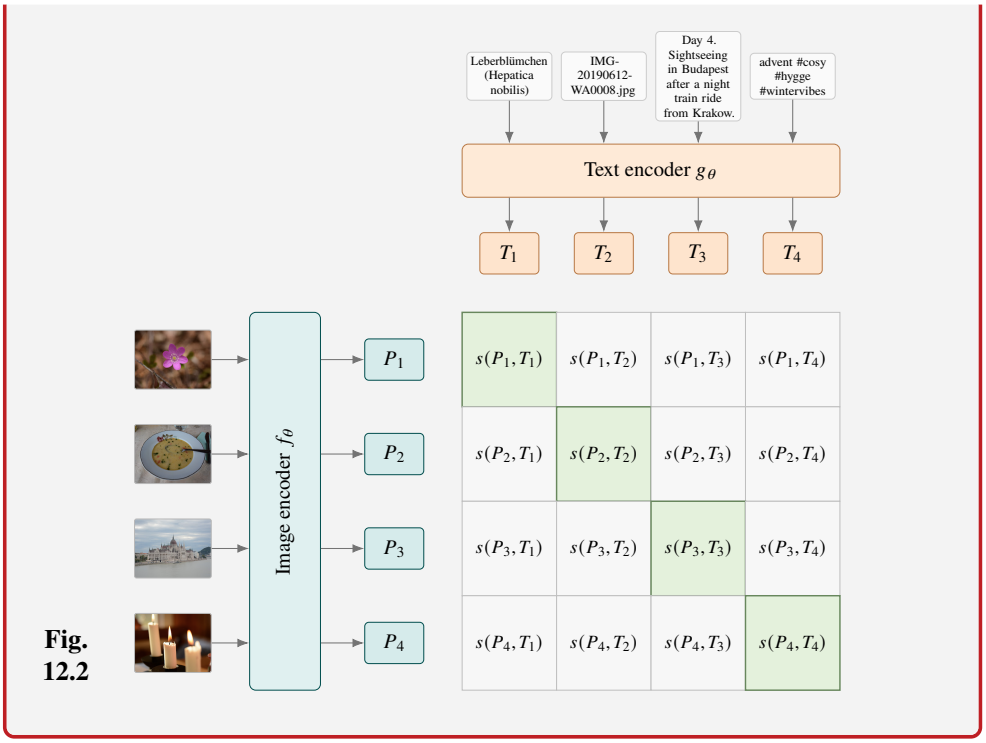



**Fig. 12.2**

The example highlights what is so powerful about contrastive learning. By the end of training, the two encoders produce embeddings such that images and captions describing similar concepts are mapped to nearby vectors in the embedding space, even if that specific image–caption pair was never seen during training. This in turn enables a range of capabilities: we can for example perform image retrieval by encoding a text query and searching for images whose embedding lies close to it, and we can perform classification by encoding an image and comparing it against a set of textual class descriptions of our choice. In contrast to ordinary classification it is also possible to use text strings at prediction time that were not seen during training, which enables so-called *zero-shot classification*.

It can be illuminating to compare (12.14) with the multi-class cross-entropy loss. Conceptually, the two are very similar – both are softmax-style losses that contrast a correct option against alternatives. Numerically, however, the contrastive loss is somewhat trickier to optimise, since the denominator in (12.14) contains a sum over all training data points (or, in practice, over a large mini-batch). The size of the mini-batch directly affects the difficulty of the contrastive task – with more data points in the denominator, there are more 'distractors' for the model to push away from, and the learned representation tends to become more discriminative.

Many variations of contrastive learning exist, with different conventions for how the pairs are generated and how the similarity score is defined. In CLIP, the pairs come from a natural alignment between the two modalities text and images. In other contrastive methods, such as SimCLR for images, the positive pair is constructed by taking a single image, applying two random augmentations (cropping, colour jitter,

etc.), and treating the two resulting views as a matched pair. The contrastive loss then encourages the embeddings of two augmentations of the same image to be close, while embeddings of different images are pushed apart. In all of these variants, the end product is a representation of the data that can subsequently be reused for downstream tasks.

### Auto-Encoders

The second method we will consider for learning a latent representation takes a rather different starting point. The *auto-encoder* forces the model to compress each data point through a low-dimensional bottleneck and then to reconstruct it as accurately as possible from the compressed code. To do so it has to make the bottleneck representation as informative about the input as possible – which is precisely what we mean by a useful representation.

For many high-dimensional problems, it is reasonable to assume that the *effective dimension* of the data is smaller than the observed dimension. That is, most of the information contained in the $p$-dimensional variable $\mathbf{x}$ can be retained even if we compress the data into a $q$-dimensional representation $\mathbf{z}$ with $q \ll p$. The typical example is when $\mathbf{x}$ is an image: the raw pixel values are extremely high-dimensional, but natural images live on a much lower-dimensional manifold of 'image-like' configurations. Similarly, a sequence of characters is a high-dimensional object, but the set of meaningful sentences in a natural language is only a tiny subset of all possible character sequences.

In an auto-encoder, we explicitly try to capture such a low-dimensional structure by jointly learning two mappings:

Encoder $h_{\boldsymbol{\theta}} : \mathbb{R}^p \to \mathbb{R}^q$, mapping a data point to a latent representation,

Decoder $f_{\boldsymbol{\theta}} : \mathbb{R}^q \to \mathbb{R}^p$, mapping a latent representation back to a data point.

Note that while $f_{\boldsymbol{\theta}}$ is intended to become close to the inverse of $h_{\boldsymbol{\theta}}$, they are – unlike normalising flow – not the exact mathematical inverse to each other. In the auto-encoder, $h_{\boldsymbol{\theta}}$ and $f_{\boldsymbol{\theta}}$ are two separate models. Importantly, the dimension $q$ of the latent representation is selected to be smaller than $p$, which makes $h_{\boldsymbol{\theta}}$ non-invertible. The encoder and decoder mappings are often parameterised as deep neural networks. For brevity, we will group the encoder and decoder parameters into a joint parameter vector $\boldsymbol{\theta}$.

Given a data point $\mathbf{x}$, its latent representation is $\mathbf{z} = h_{\boldsymbol{\theta}}(\mathbf{x})$, and feeding this representation back through the decoder yields a *reconstruction* $\widehat{\mathbf{x}} = f_{\boldsymbol{\theta}}(\mathbf{z})$ of the data point. In general, $\widehat{\mathbf{x}}$ will not be identical to $\mathbf{x}$, because we have forced the encoder to compress the data into a lower-dimensional representation in the first step, and the decoder is unable to fully compensate for this loss of information. However, we can train the model to make the reconstruction as accurate as possible by minimising the reconstruction error over the training data. Assuming the data $\mathbf{x}$ is continuous and

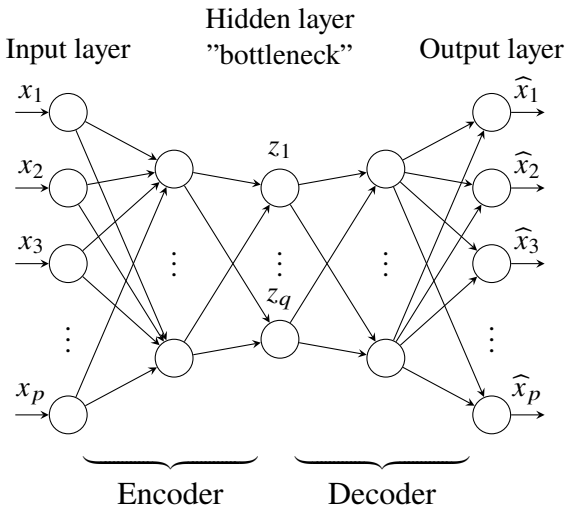



**Figure 12.3:** The auto-encoder can be viewed as a neural network with a bottleneck layer in the middle. The first part of the network corresponds to the encoder, the second part to the decoder, and the latent representation is given by the hidden units at the bottleneck.

using the squared error loss, the training objective becomes

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \sum_{i=1}^{n} \|\mathbf{x}_i - f_{\boldsymbol{\theta}}(h_{\boldsymbol{\theta}}(\mathbf{x}_i))\|^2. \tag{12.17}$$

It is important that $q < p$ for this problem to be interesting, otherwise we would just end up learning an identity mapping. When $q$ is indeed smaller than $p$, the objective encourages the encoder $h_\theta$ to compress the data into a lower-dimensional vector $\mathbf{z}$ while retaining as much information as possible, so as to enable accurate reconstruction by the decoder $f_\theta$. This means that the most essential information about the input $\mathbf{x}$ will be contained by the latent representation $\mathbf{z}$.

When using neural networks to parameterise the encoder and decoder mappings, the complete auto-encoder can also be viewed as a single neural network with a 'bottleneck layer' in the middle corresponding to the latent code. We illustrate this in Figure 12.3.

One issue to be aware of when using auto-encoders is the risk of learning a memorisation of the training data. To illustrate the point, assume that $q = 1$ so that we have a scalar latent code $z$. For any realistic problem, this should be insufficient to represent the actual information content in some complex data $\mathbf{x}$. However, conceptually, the auto-encoder could learn to map any training data point $\mathbf{x}_i$ to the value $z_i = i$, and then learn to reconstruct the data point exactly based on this unique identifier. This will never happen exactly in practice, but we can still suffer to some extent from this memorisation effect. Put differently, the model might learn to store information about the *training data* in the parameter vector $\boldsymbol{\theta}$, instead of learning a useful and generalisable representation. This is a particular concern when the model

is very flexible (very high-dimensional $\boldsymbol{\theta}$) so that it has the capacity of memorising the data.

Various extensions to the basic auto-encoder have been proposed in the literature to combat this memorisation effect, among other things. Regularisation is one useful approach. For instance, it is possible to add a probabilistic prior on the distribution of the latent representation, effectively bridging the gap between auto-encoders and generative models, an idea we develop in Section 12.2 just below. Another approach is to limit the capacity of the encoder and decoder mappings. Taking this to the extreme, we can restrict both mappings to be linear functions, which results in a well-known dimensionality reduction method referred to as principal component analysis, the subject of Section 12.3.

### Variational Auto-Encoders

The *variational auto-encoder* (VAE) is a generative take on the auto-encoder we have just introduced. The VAE re-interprets the decoder as a *deep generative model* of exactly the kind we studied in Section 12.1. We use the same terminology as the auto-encoder, with an encoder that maps a data point $\mathbf{x}$ to a low-dimensional representation $\mathbf{z}$ and a decoder that maps it back. In the VAE, however, the encoder as well as the decoder are probability distributions instead of plain functions.

The purpose of a VAE, in the first place, is to build a model for generating more realistic-looking data, that is, to build the *decoder* $f_{\boldsymbol{\theta}}(\mathbf{z})$. In the VAE we associate the latent representation with a simple Gaussian distribution

$$p(\mathbf{z}) = \mathcal{N}(\mathbf{z}; 0, I)\,, \tag{12.18}$$

ust like in (12.2a). We consider the case with continuous $\mathbf{x}$ and place a Gaussian noise model around the decoder output,

$$p(\mathbf{x} \mid \mathbf{z}, \boldsymbol{\theta}) = \mathcal{N}\Big(\mathbf{x}; f_{\boldsymbol{\theta}}(\mathbf{z}), \sigma^2 I\Big)\,. \tag{12.19}$$

(We will consider the output variance $\sigma^2$ to be a fixed hyperparameter, although it is possible to also learn it from data with some additions to the derivation we will make.) Together this defines a generative model

$$p(\mathbf{x}, \mathbf{z} \mid \boldsymbol{\theta}) = p(\mathbf{x} \mid \mathbf{z}, \boldsymbol{\theta})\, p(\mathbf{z}) = \mathcal{N}\Big(\mathbf{x}; f_{\boldsymbol{\theta}}(\mathbf{z}), \sigma^2 I\Big)\, \mathcal{N}(\mathbf{z}; 0, I)\,. \tag{12.20}$$

This is where the kind of difficulty we warned about in the chapter introduction surfaces. Learning the model by maximising the marginal likelihood $p(\mathbf{x} \mid \boldsymbol{\theta}) = \int p(\mathbf{x} \mid \mathbf{z}, \boldsymbol{\theta})\, p(\mathbf{z})\, d\mathbf{z}$ is intractable, because the integral over $\mathbf{z}$ cannot be evaluated in closed form once $f_{\boldsymbol{\theta}}$ is a deep neural network. We could try to approximate the integral by drawing samples $\mathbf{z}$ from the prior[2] $p(\mathbf{z})$, so-called Monte Carlo sampling, but in

[2]We use the term *prior* loosely for the latent distribution $p(\mathbf{z})$, borrowing it from the Bayesian terminology of Chapter 10; the VAE is not otherwise treated as a Bayesian model here.


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

high dimensions the vast majority of those samples land in regions where the decoder produces something completely unlike the given data point $\mathbf{x}$ (that is, $p(\mathbf{x} \mid \mathbf{z}, \boldsymbol{\theta})$ would be very small), and the estimate is hopelessly inefficient. What we would really like is to concentrate the samples on the values of $\mathbf{z}$ that could plausibly have produced *this* $\mathbf{x}$ – that is, to draw $\mathbf{z}$ from the conditional distribution $p(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\theta})$. Unfortunately, that conditional distribution is itself intractable, for the very same reason.

The key idea of the VAE is to realise that $p(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\theta})$ is, in fact, the job of an *encoder*. Although the encoder might not mathematically *equal* that distribution (it is not the exact probabilistic inverse of the decoder), it can nevertheless serve as an approximation of it. Because the encoder now stands in for a probability distribution over $\mathbf{z}$ rather than producing a single value of $\mathbf{z}$, we let it output the parameters of a distribution that approximates $p(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\theta})$. To align with the standard VAE notation, we denote the encoder distribution[3] $q_{\boldsymbol{\phi}}(\,\cdot\,)$, and the standard choice is a Gaussian with diagonal covariance,

$$q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x}) = \mathcal{N}\Big(\mathbf{z}; \boldsymbol{\mu}_{\boldsymbol{\phi}}(\mathbf{x}),\, \mathrm{diag}\big(\boldsymbol{\sigma}^2_{\boldsymbol{\phi}}(\mathbf{x})\big)\Big), \tag{12.21}$$

where a neural network with parameters $\boldsymbol{\phi}$ takes $\mathbf{x}$ as input and returns the mean vector $\boldsymbol{\mu}_{\boldsymbol{\phi}}(\mathbf{x})$ and the (per-coordinate) variances[4] $\boldsymbol{\sigma}^2_{\boldsymbol{\phi}}(\mathbf{x})$. This network plays the same role as the deterministic encoder $h_{\boldsymbol{\theta}}(\mathbf{x})$ of the ordinary auto-encoder, except that it now describes a *spread* of plausible $\mathbf{z}$ for each input $\mathbf{x}$ rather than a single point. Both the generative (decoder) parameters $\boldsymbol{\theta}$ and the encoder parameters $\boldsymbol{\phi}$ are to be learned from the training data $\{\mathbf{x}_i\}_{i=1}^n$.

The encoder buys us a tractable training objective. To reach it, let us start by expanding $\ln p(\mathbf{x} \mid \boldsymbol{\theta})$ as

$$\ln p(\mathbf{x} \mid \boldsymbol{\theta}) = \ln \frac{p(\mathbf{x} \mid \boldsymbol{\theta})\, p(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\theta})}{p(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\theta})} = \ln \frac{p(\mathbf{x}, \mathbf{z} \mid \boldsymbol{\theta})}{p(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\theta})}. \tag{12.22a}$$

Now, let us take the expectation of $\ln p(\mathbf{x} \mid \boldsymbol{\theta})$ with respect to the distribution $q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})$ over $\mathbf{z}$. This is a mathematically valid operation for, in fact, any distribution over $\mathbf{z}$, since $\ln p(\mathbf{x} \mid \boldsymbol{\theta})$ does not depend on $\mathbf{z}$; for the same reason it may also seem like a pointless operation. It turns out, however, to be useful:

$$\begin{aligned} \ln p(\mathbf{x} \mid \boldsymbol{\theta}) &= \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})} \ln p(\mathbf{x} \mid \boldsymbol{\theta}) = \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})} \left[ \ln \frac{p(\mathbf{x}, \mathbf{z} \mid \boldsymbol{\theta})}{p(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\theta})} \right] \\ &= \underbrace{\mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})} \left[ \ln \frac{p(\mathbf{x}, \mathbf{z} \mid \boldsymbol{\theta})}{q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})} \right]}_{\stackrel{\text{def}}{=}\, \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x})} + \underbrace{\mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})} \left[ \ln \frac{q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})}{p(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\theta})} \right]}_{= \mathrm{KL}\left(q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x}) \,\|\, p(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\theta})\right)}. \end{aligned} \tag{12.22b}$$

[3] Not to be confused with the latent dimension $q$.

[4] To avoid the variance to accidentally becoming zero or even negative, it is a common trick to rather let the neural network predict the logarithm of the variances $s_{\boldsymbol{\phi}} = \ln \sigma^2_{\boldsymbol{\phi}}$

The second line multiplies and divides by $q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})$ inside the logarithm. The second term is a Kullback–Leibler divergence and is thereby always non-negative. The first term, which we denote $\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x})$, is consequently a *lower bound* on the marginal log-likelihood, and is therefore known as the *evidence lower bound* (ELBO).[5]

This lower bound turns out to express very well what we want to optimise. In fact, our underlying goal is conceptually twofold. On the one hand we want the decoder to match the data as well as possible, that is, to maximise the marginal likelihood $p(\mathbf{x} \mid \boldsymbol{\theta})$ with respect to $\boldsymbol{\theta}$. On the other hand, we want the encoder to be as close as possible to the probabilistic inverse of the decoder, which mathematically amounts to making $q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})$ as close as possible to $p(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\theta})$ and corresponds to minimising $\mathrm{KL}\big(q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x}) \,\|\, p(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\theta})\big)$. These goals are not separable – we cannot train the decoder in isolation from the encoder – but have to be pursued jointly. As (12.22) reveals, maximising the ELBO over both sets of parameters does exactly that: increasing $\mathcal{L}$ over $\boldsymbol{\phi}$ shrinks the gap $\mathrm{KL}\big(q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x}) \,\|\, p(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\theta})\big)$ (our second goal), while increasing it over $\boldsymbol{\theta}$ raises a bound on the marginal log-likelihood – and hence the likelihood itself, up to that gap (our first goal). The negative ELBO is therefore the loss function when learning a variational auto-encoder,

$$\widehat{\boldsymbol{\theta}}, \widehat{\boldsymbol{\phi}} = \arg\min_{\boldsymbol{\theta}, \boldsymbol{\phi}} \sum_{i=1}^{n} -\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}_i). \tag{12.23}$$

To solve this optimisation problem, we use stochastic gradient descent.

Although the notation may seem complicated, all components of $\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}_i)$ can be computed. The numerator in $\mathcal{L}$ is the joint model (12.20), which contains the decoder, and the denominator is the encoder (12.21). The one thing that is not immediately available is the expectation with respect to $q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})$. This is where the Monte Carlo idea enters: we approximate the expectation (an integral) by an average over $\mathbf{z}$-samples drawn from the encoder distribution $q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})$. Because a single such draw already gives an *unbiased* estimate of the expectation, and stochastic gradient optimisation is in any case driven by noisy gradients, it suffices in practice to use one sample $\mathbf{z}$ per data point.

There is, however, a catch. To run stochastic gradient optimisation we must not only evaluate $\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x})$ but also differentiate it with respect to the parameters we want to learn. For the neural-network parts this is just back-propagation, but differentiating through a *random* sample $\mathbf{z}$ is not meaningful as it stands. Thanks to the Gaussian assumption in (12.21), however, we can separate the randomness in $\mathbf{z}$ from the parameters $\boldsymbol{\phi}$,

$$\mathbf{z} \sim \mathcal{N}\Big(\mathbf{z}; \boldsymbol{\mu}_{\boldsymbol{\phi}}(\mathbf{x}), \operatorname{diag}\big(\boldsymbol{\sigma}^2_{\boldsymbol{\phi}}(\mathbf{x})\big)\Big) \Leftrightarrow \mathbf{z} = \boldsymbol{\mu}_{\boldsymbol{\phi}}(\mathbf{x}) + \boldsymbol{\sigma}_{\boldsymbol{\phi}}(\mathbf{x}) \odot \boldsymbol{\varepsilon},\ \boldsymbol{\varepsilon} \sim \mathcal{N}(0, I)\,, \tag{12.24}$$

where $\odot$ denotes element-wise multiplication. What happens is that we draw a random vector $\boldsymbol{\varepsilon}$ from a zero-mean Gaussian with unit variance, and then apply a

[5] Evidence is another term for the marginal likelihood $p(\mathbf{x} \mid \boldsymbol{\theta})$; the ELBO bounds its logarithm.

transformation that depends on $\boldsymbol{\phi}$. This is called the *reparameterisation trick*, and it makes it perfectly possible to differentiate the sample $\mathbf{z}$, and hence the entire ELBO $\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x})$, also with respect to $\boldsymbol{\phi}$.

To make our learning objective $\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x})$ concrete we rewrite it once more. Using $p(\mathbf{x}, \mathbf{z} \,|\, \boldsymbol{\theta}) = p(\mathbf{x} \,|\, \mathbf{z}, \boldsymbol{\theta})\, p(\mathbf{z})$ and splitting the logarithm make it possible to interpret $\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x})$ as the sum of a reconstruction term and a regularisation term,

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}) = \underbrace{\mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z} \,|\, \mathbf{x})}\big[\ln p(\mathbf{x} \,|\, \mathbf{z}, \boldsymbol{\theta})\big]}_{\text{reconstruction}} - \underbrace{\mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z} \,|\, \mathbf{x})}\left[\ln \frac{q_{\boldsymbol{\phi}}(\mathbf{z} \,|\, \mathbf{x})}{p(\mathbf{z})}\right]}_{\substack{= \mathrm{KL}\left(q_{\boldsymbol{\phi}}(\mathbf{z} \,|\, \mathbf{x}) \,\middle\|\, p(\mathbf{z})\right) \\ \text{regularisation}}}. \tag{12.25}$$

The purpose of the reconstruction term is the same as for the auto-encoder, namely that the encoder-decoder should reconstruct the inputs as accurately as possible. The KL-term (which is different from the KL term in (12.22b)), on the other hand, can be understood as a regularisation that penalises the distribution of the latent variables $\mathbf{z}$ when it deviates from the simple Gaussian prior $\mathbf{z}$ (12.18).

Since both expressions in the regularisation term are Gaussian, it actually has a closed form, and only the reconstruction term needs to be approximated with a sample. Inserting the Gaussian decoder (12.20) and encoder (12.21), the reparameterised sample (12.24), and the closed-form KL, and using a single sample for the reconstruction term, the (single-sample) per-data-point objective becomes

$$\begin{aligned}\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}_i) = &-\frac{1}{2\sigma^2}\Big\|\mathbf{x}_i - f_{\boldsymbol{\theta}}\big(\boldsymbol{\mu}_{\boldsymbol{\phi}}(\mathbf{x}_i) + \boldsymbol{\sigma}_{\boldsymbol{\phi}}(\mathbf{x}_i) \odot \boldsymbol{\varepsilon}\big)\Big\|^2 \\ &- \tfrac{1}{2}\sum_{j=1}^{q}\Big(\mu^2_{\boldsymbol{\phi},j}(\mathbf{x}_i) + \sigma^2_{\boldsymbol{\phi},j}(\mathbf{x}_i) - 1 - \ln \sigma^2_{\boldsymbol{\phi},j}(\mathbf{x}_i)\Big) + \text{const.}\end{aligned} \tag{12.26}$$

Every term is now an explicit, differentiable function of $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$ – the only randomness, $\boldsymbol{\varepsilon}$, no longer carries any parameters – so the gradient

$$\nabla_{\boldsymbol{\theta},\boldsymbol{\phi}}\, \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}_i) \tag{12.27}$$

can be obtained by ordinary back-propagation through the decoder $f_{\boldsymbol{\theta}}$ and the encoder networks $\boldsymbol{\mu}_{\boldsymbol{\phi}}$ and $\boldsymbol{\sigma}_{\boldsymbol{\phi}}$, exactly as for any other deep model.

We now have all pieces in place for the VAE, and we summarise it as Method 12.2.

Let us conclude by noting a few important aspects of the VAE. First, the two terms in the learning objective (12.25) explain the name 'variational auto-encoder'. With the Gaussian decoder the reconstruction term reduces essentially to the squared error $-\frac{1}{2\sigma^2}\|\mathbf{x} - f_{\boldsymbol{\theta}}(\mathbf{z})\|^2$ (12.26); this is exactly the objective of the ordinary auto-encoder (12.17), only now the latent representation $\mathbf{z}$ is *sampled* from the encoder rather than read off deterministically, and the error is averaged over that sample. This is also where the sampling difficulty we started with finally resolves: the intractable integral for $p(\mathbf{x} \,|\, \boldsymbol{\theta})$ is now approximated, but not by a naive Monte Carlo approximation of the integral $\int p(\mathbf{x} \,|\, \mathbf{z}, \boldsymbol{\theta})\, p(\mathbf{z})\, d\mathbf{z}$ with samples from the prior $\mathbf{z} \sim p(\mathbf{z})$ (which

**Learn** a variational auto-encoder

**Data:** Training data $\mathcal{T} = \{\mathbf{x}_i\}_{i=1}^n$, initial parameters $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$, learning rate $\gamma$, batch size $n_b$

**Result:** Encoder $q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})$ and decoder (generative model) $f_{\boldsymbol{\theta}}(\mathbf{z})$

**repeat**

  Sample mini-batch $\{\mathbf{x}_i\}_{i=1}^{n_b}$ from the training data

  For each $i$: encode to obtain $\boldsymbol{\mu}_{\boldsymbol{\phi}}(\mathbf{x}_i)$ and $\boldsymbol{\sigma}_{\boldsymbol{\phi}}(\mathbf{x}_i)$

  For each $i$: draw $\boldsymbol{\varepsilon}_i \sim \mathcal{N}(0, I)$ and set $\mathbf{z}_i = \boldsymbol{\mu}_{\boldsymbol{\phi}}(\mathbf{x}_i) + \boldsymbol{\sigma}_{\boldsymbol{\phi}}(\mathbf{x}_i) \odot \boldsymbol{\varepsilon}_i$

  Compute the gradient of the average estimated ELBO, $\widehat{\mathbf{d}} = -\frac{1}{n_b} \sum_{i=1}^{n_b} \nabla_{\boldsymbol{\theta},\boldsymbol{\phi}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}_i)$

  Update: $(\boldsymbol{\theta}, \boldsymbol{\phi}) \leftarrow (\boldsymbol{\theta}, \boldsymbol{\phi}) - \gamma\, \widehat{\mathbf{d}}$

**until** *convergence*

**Sample** from a variational auto-encoder

**Data:** Decoder $f_{\boldsymbol{\theta}}$ (and noise level $\sigma^2$)

**Result:** Synthetic sample $\mathbf{x}'$

Sample $\mathbf{z}' \sim \mathcal{N}(0, I)$

Output $\mathbf{x}' = f_{\boldsymbol{\theta}}(\mathbf{z}')$ (the decoder mean), or draw $\mathbf{x}' \sim \mathcal{N}\big(f_{\boldsymbol{\theta}}(\mathbf{z}'), \sigma^2 I\big)$

**Method 12.2:** Training a variational auto-encoder.

would give inefficient samples that fail to explain $\mathbf{x}$), but by re-writing the expression into an expectation over the encoder $q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})$, whose samples are much more likely to explain $\mathbf{x}$ and thereby statistically more efficient. That is why even a single sample approximation now suffices. Moreover, the regularisation term in (12.25) is the 'probabilistic prior on the latent representation' we anticipated when discussing auto-encoder regularisation: it keeps the encoded distribution $q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})$ close to the prior $p(\mathbf{z})$, which is what makes it safe to generate new data from the decoder by sampling $\mathbf{z} \sim \mathcal{N}(0, I)$ after training. The decoder variance $\sigma^2$ acts as a knob that sets the balance between the two: a small $\sigma^2$ weights reconstruction heavily (sharper but less regularised), a large $\sigma^2$ weights the prior more heavily.

The result is a model that can do both what an auto-encoder does (encode an input to a latent code, via the encoder mean $\boldsymbol{\mu}_{\boldsymbol{\phi}}(\mathbf{x})$) and what a deep generative model does (sample new inputs from a latent code): once trained, we draw $\mathbf{z} \sim \mathcal{N}(0, I)$ and feed it through the decoder $f_{\boldsymbol{\theta}}$ to generate a new data point, just as we did for the normalising flow or the GAN in Section 12.1. Compared with those alternatives, the VAE offers a different trade-off: it typically produces less sharp samples than a GAN, but the presence of an explicit (lower-bounded) likelihood objective makes it considerably easier to train and evaluate. VAEs can also be used for semi-supervised learning, in a way that is structurally similar to how we used the GMM in the semi-supervised setting in Section 11.2: the discrete class label of the GMM is reintroduced alongside

the continuous latent variable $\mathbf{z}$, and unlabelled data points contribute through the same likelihood-based objective.

> ***Time to reflect 12.1*** *The ELBO* (12.25) *balances a reconstruction term against a KL regulariser. Suppose the KL term were given an overwhelming weight, so that the encoder is forced to make $q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})$ essentially equal to the prior $\mathcal{N}(0, I)$ for* every *input $\mathbf{x}$. What could the latent representation then tell the decoder about $\mathbf{x}$, and what would the reconstructions look like? Conversely, if the reconstruction term dominated entirely, how would the model relate to the plain auto-encoder of Figure 12.3?*

## 12.3 Latent Representations for Dimensionality Reduction: PCA

The auto-encoder learns a $q$-dimensional latent representation $\mathbf{z}$ of $p$-dimensional data, with $q \ll p$. Such a representation is useful as a feature vector for downstream tasks, but also in its own right: the encoder is a recipe for compressing the data from $p$ to $q$ dimensions in a way that still allows the original data to be approximately recovered. This is *dimensionality reduction* – filtering out redundancy and noise and exposing the most informative directions in the data.

In this section we study the simplest possible auto-encoder, in which the encoder and decoder are restricted to be *linear* maps. The result turns out to be the classical method of *principal component analysis* (PCA), by far the most widely used dimensionality-reduction technique in statistics and machine learning.

Concretely, we restrict the encoder and decoder in (12.17) to be linear (technically, affine) functions,

$$\mathbf{z} = \underbrace{W_e}_{q\times p}\,\mathbf{x} + \underbrace{b_e}_{q\times 1} \qquad \text{and} \qquad \widehat{\mathbf{x}} = \underbrace{W_d}_{p\times q}\,\mathbf{z} + \underbrace{b_d}_{p\times 1}, \tag{12.28}$$

with parameters $\boldsymbol{\theta} = \{W_e, b_e, W_d, b_d\}$. In light of Figure 12.3, this is simply a two-layer neural network with a bottleneck layer and linear activation functions. The complete reconstruction is then itself a linear (affine) map,

$$\widehat{\mathbf{x}} = W_d W_e\,\mathbf{x} + (W_d b_e + b_d), \tag{12.29}$$

and we learn the parameters by minimising the squared reconstruction error over the (unlabelled) training data $\{\mathbf{x}_i\}_{i=1}^n$,

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \sum_{i=1}^{n} \left\| \mathbf{x}_i - (W_d W_e \mathbf{x}_i + W_d b_e + b_d) \right\|^2. \tag{12.30}$$

Two simple observations let us tidy up this problem before solving it. First, the reconstruction (12.29) depends on the parameters only through the product $W_d W_e$ and the combined offset $W_d b_e + b_d$. The individual matrices can therefore never be uniquely identified from (12.30) – but nor do we need them, since we only seek *one* good solution. Second, the offset merely re-centres the reconstruction, and its best value simply makes the reconstruction have the same mean as the data. We may therefore handle the offset by a pre-processing step: we subtract the mean $\bar{\mathbf{x}} = \frac{1}{n}\sum_{i=1}^{n} \mathbf{x}_i$ from every data point, work with the *centred* data $\mathbf{x}_{0,i} = \mathbf{x}_i - \bar{\mathbf{x}}$, and drop the offsets from the model. The objective then reduces to

$$\widehat{W}_e, \widehat{W}_d = \arg\min_{W_e, W_d} \sum_{i=1}^{n} \left\| \mathbf{x}_{0,i} - W_d W_e \, \mathbf{x}_{0,i} \right\|^2. \tag{12.31}$$

It is convenient to collect the centred data points as the rows of an $n \times p$ matrix,

$$\mathbf{X}_0 = \begin{bmatrix} \mathbf{x}_{0,1}^\mathsf{T} \\ \vdots \\ \mathbf{x}_{0,n}^\mathsf{T} \end{bmatrix}, \tag{12.32}$$

in terms of which the reconstructions of all data points are the rows of $\widehat{\mathbf{X}}_0 = \mathbf{X}_0 W_e^\mathsf{T} W_d^\mathsf{T}$, and (12.31) becomes $\arg\min \|\mathbf{X}_0 - \widehat{\mathbf{X}}_0\|_F^2$, where $\|\cdot\|_F$ is the Frobenius norm.[6] The crucial structural fact is that, since $W_e$ and $W_d$ have only $q$ rows and columns respectively, the reconstruction $\widehat{\mathbf{X}}_0$ has *rank at most* $q$. Learning the best linear auto-encoder is therefore exactly the problem:

Find the best rank-$q$ approximation $\widehat{\mathbf{X}}_0$ of the centred data matrix $\mathbf{X}_0$.

To solve this we need one tool from linear algebra, the *singular value decomposition* (SVD). Let us briefly recall what SVD says. Any real $n \times p$ matrix $\mathbf{X}_0$ can be factorised as

$$\mathbf{X}_0 = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^\mathsf{T}, \tag{12.33}$$

where $\mathbf{U}$ (of size $n \times n$) and $\mathbf{V}$ (of size $p \times p$) are *orthogonal* matrices – meaning their columns are mutually orthogonal unit vectors – and $\boldsymbol{\Sigma}$ (of size $n \times p$) is 'diagonal', carrying non-negative numbers $\sigma_1 \geq \sigma_2 \geq \cdots \geq 0$ on its main diagonal and zeros everywhere else. The numbers $\sigma_j$ are called the *singular values*. Geometrically, the columns of $\mathbf{V}$ form an orthonormal set of directions in the $p$-dimensional input space, ordered by importance: the data extends the most along the first direction (with 'amount' measured by $\sigma_1$), next-most along the second, and so on.[7] The SVD is

[6] The Frobenius norm of a matrix $A$ is $\|A\|_F = \sqrt{\sum_{ij} A_{ij}^2}$, i.e. the ordinary Euclidean norm applied to the matrix as if it were one long vector.

[7] If $n > p$ and the data has full rank, there are $p$ non-zero singular values. In general, the number of non-zero singular values equals the rank of $\mathbf{X}_0$.

closely tied to the eigenvalue decomposition: as we will see at the end of the section, the columns of $\mathbf{V}$ are exactly the eigenvectors of the data's covariance matrix.

With the SVD in hand, the best rank-$q$ approximation is given by a classical result, the *Eckart–Young–Mirsky theorem*: it is obtained by keeping only the $q$ largest singular values and setting the rest to zero. Writing $\mathbf{V}_1$ for the matrix formed by the first $q$ columns of $\mathbf{V}$, this best approximation amounts to *projecting each centred data point onto the subspace spanned by those $q$ columns*. Translated back to the linear auto-encoder (12.28), it is achieved by the choice

$$\widehat{W}_e = \mathbf{V}_1^\mathsf{T} \qquad \text{and} \qquad \widehat{W}_d = \mathbf{V}_1, \tag{12.34}$$

so that the encoder projects $\mathbf{x}$ onto the first $q$ directions of $\mathbf{V}$, and the decoder maps the result back.[8] We summarise the resulting procedure in Method 12.3.

**Perform** PCA

**Data:** Training data $\mathcal{T} = \{\mathbf{x}_i\}_{i=1}^n$
**Result:** Principal axes $\mathbf{V}$ and principal components $\mathbf{Z}_0$

1 Compute the mean vector $\bar{\mathbf{x}} = \frac{1}{n}\sum_{i=1}^n \mathbf{x}_i$
2 Centre the data, $\mathbf{x}_{0,i} = \mathbf{x}_i - \bar{\mathbf{x}}$, for $i = 1, \ldots, n$
3 Construct the data matrix $\mathbf{X}_0$ according to (12.32)
4 Perform SVD on $\mathbf{X}_0$ to obtain the factorisation $\mathbf{X}_0 = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\mathsf{T}$
5 Compute principal components $\mathbf{Z}_0 = \mathbf{U}\mathbf{\Sigma}$

**Method 12.3:** Principal Component Analysis

A pleasant feature of this solution is that it requires nothing more than a single SVD of the centred data matrix, and that computation does not depend on $q$. In contrast with the non-linear auto-encoder, where the bottleneck dimension must be fixed before training, here one factorisation gives the optimal representation *simultaneously for every $q$*: to reduce to $q$ dimensions, we simply keep the first $q$ columns. The columns of $\mathbf{V}$ are called the *principal axes*, and the coordinates of the data expressed in this new basis,

$$\mathbf{Z}_0 = \mathbf{X}_0\mathbf{V} = \mathbf{U}\mathbf{\Sigma}, \tag{12.35}$$

are called the *principal components* (or scores). We illustrate PCA in Figure 12.4.

> ***Time to reflect 12.2*** *We have defined the principal components in terms of the centred data. However, we can also compute the non-centred principal components in the same way, $\mathbf{Z} = \mathbf{X}\mathbf{V}$ (note that $\mathbf{V}$ is still computed from the*

[8] That this choice reproduces the truncated SVD follows from the orthogonality of $\mathbf{V}$; we omit the short verification.

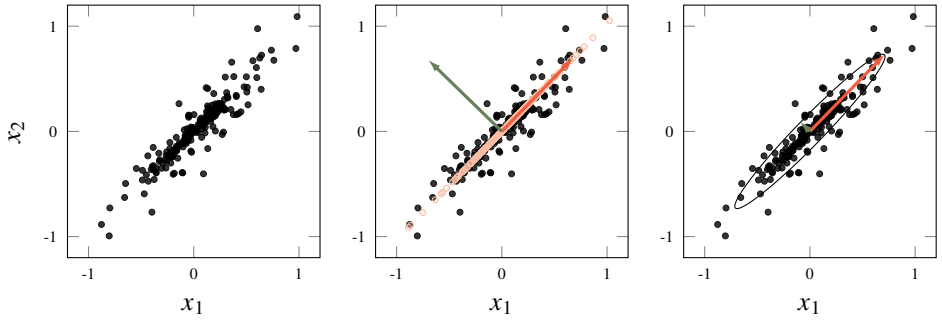



**Figure 12.4:** An illustration of PCA in $\mathbb{R}^2$. In the left panel, some data $\{\mathbf{x}_i\}_{i=1}^n$ is shown. The middle panel shows the first (red) and second (green) principal axes, given by the first and second columns of $\mathbf{V}$. It also shows the data points projected onto the first principal axis (pink), which are exactly the reconstructions produced by a linear auto-encoder with $q = 1$ latent dimensions. The right panel shows an ellipse fitted to the covariance matrix $\frac{1}{n}\mathbf{X}_0^\mathsf{T}\mathbf{X}_0$ of the data: its axes agree with the principal axes, and its width along each axis is the standard deviation of the data in that direction. PCA thus finds a new basis for $\mathbb{R}^p$, rotated to align with the covariance of the data.

*SVD of the centred data matrix). How is* $\mathbf{Z}$ *related to* $\mathbf{Z}_0$*? How does this relate to the encoder mapping* $\mathbf{z} = W_e\mathbf{x}$ *that we started the derivation from?*

It is also possible to understand PCA as *variance maximisation*. The (sample) covariance matrix of the centred data is

$$\frac{1}{n}\mathbf{X}_0^\mathsf{T}\mathbf{X}_0 = \frac{1}{n}\mathbf{V}\boldsymbol{\Sigma}^\mathsf{T}\mathbf{U}^\mathsf{T}\mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^\mathsf{T} = \mathbf{V}\boldsymbol{\Lambda}\mathbf{V}^\mathsf{T}, \tag{12.36}$$

where $\boldsymbol{\Lambda}$ is diagonal with entries $\sigma_i^2/n$. This is precisely an eigenvalue decomposition of the covariance matrix: the principal axes (columns of $\mathbf{V}$) are its eigenvectors, and the variance of the data along the $i$th axis is the corresponding eigenvalue $\sigma_i^2/n$. Hence the first principal axis is the direction of greatest variance, the second is the direction of greatest variance among those orthogonal to the first, and so on. This is the geometry visualised by the covariance ellipse in the right panel of Figure 12.4.

As a final practical remark, when the input variables are measured on very different scales it is usually wise to standardise them before applying PCA, as otherwise a variable with a large numerical range would dominate the principal directions for purely artificial reasons. When the scales are genuinely meaningful, however, standardising can mask the very variance structure that PCA is meant to uncover, so the choice should be made case by case.

## 12.4 Further Reading

The topics covered in this chapter – deep generative modelling and representation learning – are at the centre of much contemporary machine learning research, and the

literature is correspondingly vast. We comment briefly on each.

For *contrastive learning*, the CLIP model is described by Radford et al. (2021), and a closely related image-only method is SimCLR (Ting Chen et al. 2020). Le-Khac et al. (2020) provides an overview of contrastive representation learning more broadly.

The book by Goodfellow, Bengio, et al. (2016) has more in-depth discussions about *deep generative models* (Chapter 20), *auto-encoders* (Chapter 14), and other approaches for *representation learning* (Chapter 15). Generative adversarial networks were introduced by Goodfellow, Pouget-Abadie, et al. (2014) and reviewed by Creswell et al. (2018). An overview of normalising flows is given by Kobyzev et al. (2020). The variational auto-encoder was introduced by Diederik P. Kingma and Welling (2014) and Rezende et al. (2014) and is described in detail by Diederik P. Kingma and Welling (2019); it has also been used for semi-supervised learning (Diederik P. Kingma, Rezende, et al. 2014) in a way which is similar to how we used the GMM in the semi-supervised setting in Section 11.2. *Diffusion models* have a longer history but the formulation that underlies most current systems was introduced by J. Ho et al. (2020), which is the topic of Example 12.3; an overview is given by Yang et al. (2023).

Finally, *principal component analysis* is a classical technique that is treated in essentially every textbook on statistics and machine learning, including C. M. Bishop (2006), Hastie et al. (2009, Chapter 14), and Murphy (2012); its formulation as a linear auto-encoder, as presented here, also makes clear its close relationship to the singular value decomposition and the Eckart–Young–Mirsky theorem from linear algebra.

# 13 User Aspects of Machine Learning

Dealing with supervised machine learning problems in practice is to a great extent an engineering discipline where many practical issues have to be considered and where the available amount of work-hours to undertake the development is often the limiting resource. To use this resource efficiently, we need to have a well-structured procedure for how to develop and improve the model. Multiple actions can potentially be taken. How do we know which action to take and if it is really worth spending the time implementing it? Is it, for example, worth spending an extra week collecting and labelling more training data, or should we do something else? These issues will be addressed in this chapter. Note that the layout of this chapter is thematic and does not necessarily represent the sequential order in which the different issues should be addressed.

## 13.1 Defining the Machine Learning Problem

Solving a machine learning problem in practice is an iterative process. We train the model, evaluate the model, and from there suggest an action for improvement and then train the model again, and so on. To do this efficiently, we need to be able to tell whether a new model is an improvement over the previous one or not. One way to evaluate the model after each iteration would be to put it into production (for example running a traffic-sign classifier in a self-driving car for a few hours). Besides the obvious safety issues, this evaluation procedure would be very time consuming and cumbersome. It would most likely also be inaccurate since it could still be hard to tell whether the proposed change was an actual improvement or not.

A better strategy is to automate this evaluation procedure without the need to put the model into production each time we want to evaluate its performance. We do this by putting aside a *validation* dataset and a *test* dataset and evaluate the performance using a scalar *evaluation metric*. The validation and test datasets together with the evaluation metric will define the machine learning problem that we are solving.

### Training, Validation, and Test Data

In Chapter 4, we introduced the strategy of splitting the available data into training data, validation data, and test data, as repeated in Figure 13.1.

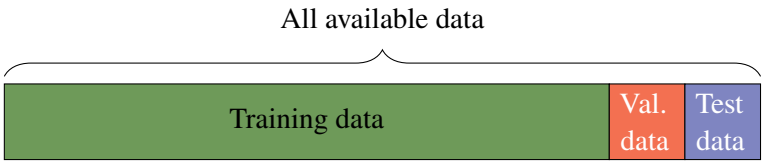



**Figure 13.1:** Splitting the data into training data, hold-out validation data, and test data.

- **Training data** is used for training the model.
- **Hold-out validation data** is used for comparing different model structures, choosing hyperparameters of the model, feature selection, and so on.
- **Test data** is used to evaluate the performance of the final model.

If the amount of available data is small, it is possible to perform $k$-fold cross-validation instead of putting aside hold-out validation data; the idea of how to use it in the iterative procedure is unchanged. To get a final estimate of the performance, test data is used.

In the iterative procedure, the hold-out validation data (or $k$-fold cross-validation) is used to judge if the new model is an improvement over the previous model. During this validation stage, we can also choose to train several new models. For example, if we have a neural network model and are interested in the number of hidden units to use in a certain layer, we can train several models, each with a different choice of hidden units. Afterwards, we pick the one that performs best on the validation data. The number of hidden units in a certain layer is one example of a hyperparameter. If we have more hyperparameteters that we want to evaluate, we can perform a grid search over these parameters. In Section 5.6, we discuss hyper-parameter optimisation in more detail.

Eventually, we will effectively have used the validation data to compare many models. Depending on the size of the validation data, we might risk picking a model that does particularly well on the validation data in comparison to completely unseen data. To detect this and to get a fair estimate of the actual performance of a model, we use the test data, which has been used neither during training nor validation. If the performance on the validation data is substantially better than the performance on the test data, we have overfitted on the validation data. The easiest solution in that case would be to extend the size of the validation data. The test data should not be used repeatedly as a part of the training and model selection procedure. If we start making major decisions based on our test data, then the model will be adapted to the test data, and we can no longer trust that the test performance is an objective measure of the actual performance of our model.

It is important that both the validation data and the test data always come from the same data distribution, namely the data distribution that we are expecting to see when we put the model into production. If they do not stem from the same distribution, we are validating and improving our model towards something that is not represented in


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026, http://smlbook.org*

the test data and hence are 'aiming for the wrong target'. Usually, the training data is also expected to come from the same data distribution as the test and validation data, but this requirement can be relaxed if we have good reasons to do so. We will discuss this further in Section 13.2.

When splitting the data into training, validation, and test, *group leakage* could be a potential problem. Group leakage can occur if the data points are not really stochastically independent but are ordered into different groups. For example, in the medical domain many X-ray images may belong to the same patient. In this case, if we do a random split over the images, different images belonging to the same patient will most likely end up both in the training and the validation set. If the model learns the properties of a certain patient, then the performance on the validation data might be better than what we could expect in production.

The solution to the group leakage problem is to do group partitioning. Instead of doing a random split over the data points, we do the split over the groups that the data points belong to. In the medical example above, that would mean that we do a random split of the patients rather than of the medical images. By this, we make sure that the images for a certain patient only end up in one of the datasets, and the leakage of unintentional information from the training data to the validation and test data is avoided.

Even though we advocate the use of validation and test data to improve and assess the performance of the model, we should eventually also evaluate the performance of the model in production. If we realise that the model is performing systematically worse in production than on the test data, we should try to find the reason for why this is the case. If possible, the best way of improving the model is to update the test data and validation data such they actually represent what we expect to see in production.

### Size of Validation and Test Datasets

How much data should we set aside as hold-out validation data and test data, or should we even avoid setting aside hold-out validation data and use $k$-fold cross-validation instead? This depends on how much data we have available, what performance difference we plan to detect, and how many models we plan to compare. For example, if we have a classification model with a 99.8% accuracy and want to know if a new model is even better, a validation dataset of 100 data points will not be able to tell that difference. Also, if we plan to compare many (say, hundreds or more) different hyperparameter values and model structures using 100 validation data points, we will most likely overfit to that validation data.

If we have, say, 500 data points, one reasonable split could be 60%–20%–20% (that is 300–100–100 data points) for training–validation–test. With such a small validation dataset, we cannot afford to compare several hyperparameter values and model structures or to detect an improvement in accuracy of 0.1%. In this situation, we are probably better off using $k$-fold cross-validation to decrease the risk of overfitting to the validation data. Be aware, however, that the risk of overfitting the training data still exists even with $k$-fold cross-validation. We also still need to set aside test data if

we want a final unbiased estimate of the performance.

Many machine learning problems have substantially larger datasets. Assume we have a dataset of 1 000 000 data points. In this scenario, one possible split could be 98%–1%–1%, that is, leaving 10 000 data points for validation and test, respectively, unless we really care about the very last decimals in performance. Here, $k$-fold cross-validation is of less use in comparison to the scenario with just 500 data points, since having all 99% = 98% + 1% (training + validation) available for training would make a small difference in comparison to using 'only' 98%. Also, the price for training $k$ models (instead of only one) with this amount of data would be much higher.

Another advantage of having a separate validation dataset is that we can allow the training data to come from a slightly different distribution than the validation and test dataset, for example if that would enable us to find a much larger training dataset. We will discuss this more in Section 13.2.

### Single Number Evaluation Metric

In Section 4.5, we introduced additional metrics besides the misclassification rate, such as precision, recall, and F1-score for evaluating binary classifiers. There is no unique answer to which metric is the most appropriate. What metric to pick is rather a part of the problem definition. To improve the model quickly and in a more automated fashion, it is advisable to agree on a single number evaluation metric, especially if a larger team of engineers is working on the problem.

The single number evaluation metric together with the validation data are what defines the supervised machine learning problem. Having an efficient procedure in place where we can evaluate the model on the hold-out validation data (or by $k$-fold cross-validation) using the metric allows us to speed up the iterations since we can quickly see if a proposed change to the model improves the performance or not. This is important in order to manage an efficient workflow of trying out and accepting or rejecting new models.

That being said, beside the single number evaluation metric, it is useful to monitor other metrics as well to reveal the tradeoffs being made. For example, we might develop the model with different end users in mind who care more or less about different metrics, but for practical reasons, we only train one model to accommodate them all. If we, based on these tradeoffs, realise that the single number evaluation metric we have chosen does not favour the properties we want a good model to have, we can always change that metric.

### Baseline and Achievable Performance Level

Before working with the machine learning problem, it is a good idea to establish some reference points for the performance level of the model. A baseline is a very simple model that serves as a lower expected performance level. A baseline can, for example, be to randomly pick an output value $y_i$ from the training data and use that as the prediction. Another baseline for the regression problem is to take the mean of


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026, http://smlbook.org*

all output values in the training data and use that as the prediction. A corresponding baseline for a classification problem is to pick the most common class among class labels in the training data and use that for the prediction. For example, if we have a binary classification problem with 70% of the training data belonging to one class and 30% belonging to the other class, and we have chosen the accuracy as our performance metric, the accuracy for that baseline is 70%. The baseline is a lower threshold on the performance. We know that the model has to be better than this baseline.

Hopefully, the model will perform well beyond the naive baselines stated above. In addition, it is also good to define an achievable performance which is on par with the maximum performance we can expect from the model. For a regression problem, this performance is in theory limited by the irreducible error presented in Chapter 4, and for classification problems, the analogous concept is the so-called Bayes error rate. In practice, we might not have access to these theoretical bounds, but there are a few strategies for estimating them. For supervised problems that that are easily solved by human annotators, the human-level performance can serve as the achievable performance. Consider for example an image classification problem. If humans can identify the correct class with an accuracy of 99%, that serves as a reference point for what we can expect to achieve from our model. The achievable performance can also be based on what other state-of-the-art models on the same or a similar problem achieve. To compare the performance with the achievable performance gives us a reference point to assess the quality of the model. Also, if the model is close to the achievable performance, we might not be able to improve our model further.

## 13.2 Improving a Machine Learning Model

As already mentioned, solving a machine learning problem is an iterative procedure where we train, evaluate, and suggest actions for improvement, for instance by changing some hyperparameters or trying another model. How do we start this iterative procedure?

### Try Simple Things First

A good strategy is to try simple things first. This could, for example, be to start with basic methods like $k$-NN or linear/logistic regression. Also, do not add adds-on like regularisation for the first model – this will come at a later stage when the basic model is up and running. A simple thing can also be to start with an already existing solution to the same or a similar problem, for example by using a foundation model (Section 13.6). Starting simple can also mean to consider only a subset of the available training data for the first model and then retrain the model on all the data if it looks promising. Also, avoid doing more data pre-processing than necessary for your first model, since we want to minimise the risk of introducing bugs early in the process. This first step not only involves writing code for learning your first simple model but also code for evaluating it on your validation data using your single number evaluation

metric.

Trying simple things first allows us to start early with the iterative procedure of finding a good model. This is important since it might reveal important aspects of the problem formulation that we need to re-think before it makes sense to proceed with more complicated models. Also, if we start with a low-complexity model, it also reduces the risk of ending up with a too-complicated model, when a much simpler model would have been just as good (or even better).

### Debugging your Model

Before proceeding, we should make sure that the code we have is producing what we are expecting it to do. The first obvious check is to make sure that the code runs without any errors or warnings. If it does not, use a debugging tool to spot the error. These are the easy bugs to spot.

The trickier bugs are those where the code is syntactically correct and runs without warnings but is still not doing what we expect it to do. The procedure of how to debug this depends on the model you have picked, but there are a few general tips:

- *Compare with baseline.* Compare you model performance on validation data with the baselines you have stated (see Section 13.1). If we do not manage to beat these baselines or are even worse than them, the code for training and evaluating the model might not be working as expected.
- *Overfit a small subset.* Try to overfit the model on a very small subset (e.g. as small as two data points) of the training data and make sure that we can achieve the best possible performance evaluated on that training data subset. If it is a parametric model, also aim for the lowest possible training cost.

When we have verified to the best of our ability that the code is bug-free and does what it is expected to do, we are ready to proceed. There are many actions that could be taken to improve the model – for example, changing the type of model, increasing/decreasing model complexity, changing input variables, collecting more data, correcting mislabelled data (if there is any), etc. What should we do next? Two possible strategies for guiding us to meaningful actions to improve the solution are by trading *training error and generalisation gap* or by applying *error analysis*.

### Training Error vs. Generalisation Gap

With the notation from Chapter 4, the training error $E_{\text{train}}$ is the performance of the model on training data, and the validation error $E_{\text{hold-out}}$ is the performance on hold-out validation data. In the validation step, we are interested in changing the model such that $E_{\text{hold-out}}$ is minimised. We can write the validation error as a sum of the training error and the generalisation gap:

$$E_{\text{hold-out}} = E_{\text{train}} + \underbrace{(E_{\text{hold-out}} - E_{\text{train}})}_{\approx \text{ generalisation gap}}. \tag{13.1}$$

In words, the generalisation gap is approximated by the difference between the validation error $E_{\text{hold-out}}$ and the training error $E_{\text{train}}$.[1]

We can easily compute the training error $E_{\text{train}}$ and the generalisation gap $E_{\text{hold-out}} - E_{\text{train}}$; we just have to evaluate the error on the training data and validation data, respectively. By computing these quantities, we can get good guidance for what changes we may consider for the next iteration.

As we discussed in Chapter 4, if the training error is small and the generalisation gap is big ($E_{\text{train}}$ small, $E_{\text{hold-out}}$ big), we have typically overfitted the model. The opposite situation, big training error and small generalisation gap (both $E_{\text{train}}$ and $E_{\text{hold-out}}$ big), typically indicates underfitting.

If we want to reduce the generalisation gap $E_{\text{hold-out}} - E_{\text{train}}$ (reduce overfitting), the following actions can be explored:

- *Use a less flexible model.* If we have a very flexible model, we might start overfitting to the training data, that is, $E_{\text{train}}$ is much smaller than $E_{\text{hold-out}}$. If we use a less flexible model, we also reduce this gap.

- *Use more regularisation.* Using more regularisation will reduce the flexibility of the model and hence also reduce the generalisation gap. Read more about regularisation in Section 5.3

- *Early stopping* For models that are trained iteratively, we can stop the training before reaching the minimum. One good practice is to monitor $E_{\text{hold-out}}$ during training and stop if it starts increasing; see Example 5.7.

- *Use bagging*, or use more ensemble members if we already are using it. Bagging is a method for reducing the variance of the model, which typically also means that we reduce the generalisation gap; see more in Section 8.1.

- *Collect more training data.* If we collect more training data, the model is less prone to overfit that extended training dataset and is forced to only focus on aspects which generalise to the validation data.

[1] This can be related to (4.11), if approximating $\bar{E}_{\text{train}} \approx E_{\text{train}}$ and $\bar{E}_{\text{new}} \approx E_{\text{hold-out}}$. If we use $k$-fold cross validation instead of hold-out validation data, we use $\bar{E}_{\text{new}} \approx E_{k\text{-fold}}$ when computing the generalisation gap.

If we want to reduce the training error $E_{\text{train}}$ (reduce underfitting), the following actions can be considered:

- *Use a more flexible model* that is able to fit the training data better. This can be changing a hyperparameter in the model we are considering – for example decreasing $k$ in $k$-NN– or changing the model to a more flexible one – for example by replacing a linear regression model by a deep neural network.
- *Extend the set of input variables.* If we suspect that there are more input variables that carry information, we might want to extend the data with these input variables.
- *Use less regularisation.* This can, of course, only be applied if regularisation is used at all.
- *Train the model for longer.* For models that are trained iteratively, we can reduce $E_{\text{train}}$ by training for longer.

It is usually a balancing act between reducing the training error and the generalisation gap, and measures to decrease one of them might result in an increase of the other. This balancing act is also related to the bias–variance tradeoff discussed in Example 4.3.

We summarise the above discussion in Figure 13.2. Fortunately, evaluating $E_{\text{train}}$ and $E_{\text{hold-out}}$ is cheap. We only have to evaluate the model on the training data and the validation data, respectively. Yet, it gives us good advice on what actions to take next. Besides suggesting what action to explore next, this procedure also tells us what *not* to do: If $E_{\text{train}} \gg E_{\text{hold-out}} - E_{\text{train}}$, collecting more training data will most likely not help. Furthermore, if $E_{\text{train}} \ll E_{\text{hold-out}} - E_{\text{train}}$, a more flexible model will most likely not help.

## Learning Curves

Of the different methods mentioned to reduce the generalisation gap, collecting more training data is often the simplest and most reliable strategy. However, in contrast to the other techniques, collecting and labelling more data is often significantly more time consuming. Before collecting more data, we would like to tell how much improvement we can expect. By plotting learning curves, we can get such an indication.

In a learning curve, we train models and evaluate $E_{\text{train}}$ and $E_{\text{hold-out}}$ using different sizes of training dataset. For example, we can train different models with 10%, 20%, 30%, ... of the available training data and plot how $E_{\text{train}}$ and $E_{\text{hold-out}}$ vary with the amount of training data. By extrapolating these plots, we can get an indication of the improvement on the generalisation gap that we can expect by collecting more data.

In Figure 13.3, two sets of learning curves for two different scenarios are depicted. First, note that previously we evaluated $E_{\text{train}}$ and $E_{\text{hold-out}}$ only for the rightmost point in these graphs using all our available data. However, these plots reveal more information about the impact of the training dataset size on the performance of the


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

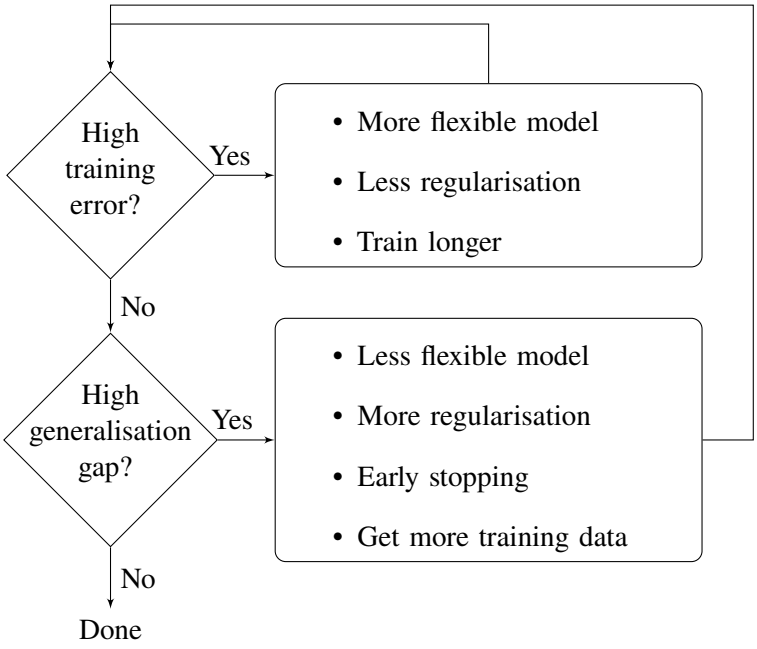
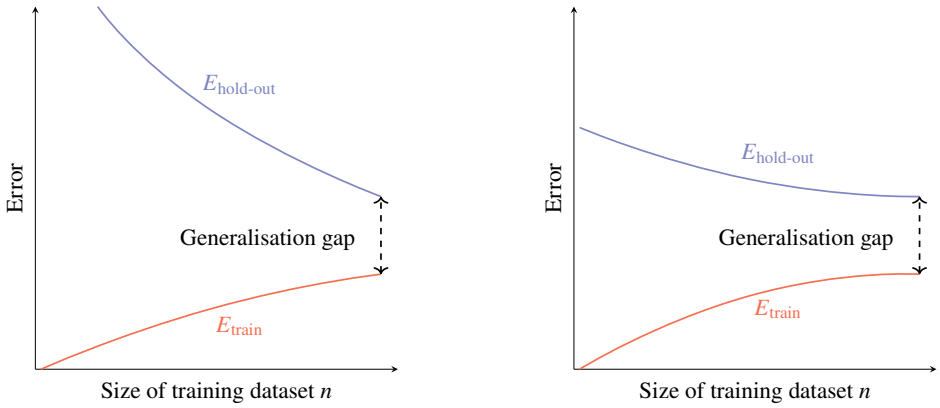



**Figure 13.2:** The iterative procedure of improving a model based on the decomposition of the validation error into the training error and generalisation gap.




**(a)** Scenario A **(b)** Scenario B

**Figure 13.3:** Learning curve for two different scenarios. In Scenario A, we can expect an improvement in the generalisation gap by collecting more training data, whereas in Scenario B, we are less likely to see an immediate improvement by adding more data.

model. In the two scenarios depicted in Figure 13.3, we have the same values for $E_{\text{train}}$ and $E_{\text{hold-out}}$ if we train the model using all the available training data. However, by extrapolating the learning curves for $E_{\text{train}}$ and $E_{\text{hold-out}}$ in these two scenarios, it is likely that in Scenario A, we can reduce the generalisation gap much more than we can in Scenario B. Hence, in Scenario A, collecting more training data is more beneficial

than in Scenario B. By extrapolating the learning curves, you can also answer the question of how much extra data is needed to reach some desired performance. Plotting these learning curves does not require much extra effort we only have to train a few more models on subsets of our training data. However, it can provide valuable insight on whether it is worth the extra effort of collecting more training data and how much extra data you should collect.

## Error Analysis

Another strategy to identify actions that can improve the model is to perform *error analysis*. Below we only describe error analysis for classification problems, but the same strategy can be applied to regression problems as well.

In error analysis, we manually look at a subset, say 100 data points, of the validation data that the model classified incorrectly. Such an analysis does not take much time but might give valuable clues as to what type of data the model is struggling with and how much improvement we can expect by fixing these issues. We illustrate the procedure with an example.

**Example 13.1 Error analysis applied to vehicle detection**

Consider a classification problem of detecting cars, bicycles, and pedestrians in an image. The model takes an image as input and outputs one of the four classes `car`, `bike`, `pedestrian`, or `other`. Assume that the model has a classification accuracy of 90% on validation data.

When looking at a subset of 100 images that were misclassified in the validation data, we make the following observations:

- All 10 images of class `pedestrian` that were incorrectly classified as `bike` were taken in dark conditions with the pedestrian being equipped with safety reflectors.
- 30 images were substantially tilted.
- 15 images were mislabelled.

From this observation we can conclude:

- If we launch a project for improving the model to classify pedestrians with safety reflectors as `pedestrian` and not incorrectly as `bike`, an improvement of at most a ~1% (a tenth of the 10% classification error rate) can be expected.
- If we improve the performance on tilted images, an improvement of at most ~3% can be expected.
- If we correct all mislabelled data, an improvement of at most ~1.5% can be expected.

Following the example, we get an indication on what improvement we can expect by tackling these three issues. These numbers should be considered as the maximal possible improvement. To prioritise which aspect to focus on, we should also consider what strategies are available for improving them, how much progress we expect to make applying these strategies, and how much effort we would have to invest fixing these issues.

For example, to improve the performance on tilted images, we could try to extend the training data by augmenting it with more tilted images. This strategy could be investigated without too much extra effort by augmenting the training data with tilted versions of the training data points that we already have. Since this could be applied fairly quickly and has a maximal performance increase of 3%, it seems to be a good thing to try out.

To improve the performance on the images of pedestrians with safety reflectors, one approach would be to collect more images in dark conditions of pedestrians with safety reflector. This obviously requires some more manual work, and it can be questioned if it is worth the effort since it would only give a performance improvement of at most 1%. However, for this application, you could also argue that this 1% is of extra importance.

Regarding the mislabelled data, the obvious action to take to improve on this issue is to manually go through the data and correct these labels. In the example above, we may say it is not quite worth the effort to get an improvement of 1.5%. However, assume that we have improved the model with other actions to an accuracy of 98.0% on validation data and that still 1.5% of the total error is due to mislabelled data; this issue is now quite relevant to address if we want to improve the model further. Remember, the purpose of the validation data is to choose between different models. This purpose is degraded when the majority of the reported error on validation is due to incorrectly labelled data rather than the actual performance of the model.

There are two levels of ambition for correcting the labels:

(i) Go through all data points in the validation/test data and correct the labels.

(ii) Go through all data points, including the training data, and correct the labels.

The advantage of approach (i), in comparison to approach (ii), is the lower amount of work it requires. Assume, for example, that we have made a 98%–1%–1% split of training-validation-test data. Then there is 50 times less data to process in comparison to approach (ii). Unless the mislabeling is systematic, correcting the labels in the training data will not necessarily pay off. Also, note that correcting labels in only test and validation data does not necessarily increase the performance of a model in production, but it will give us a fairer estimate of the actual performance of the model.

Applying the data cleaning to validation and test data only, as suggested in approach (i), will result in the training data coming from a slightly different distribution than the validation and test data. However, if we are eager to correct the mislabelled data in the training data as well, a good recommendation would still be to start correcting validation and test data only, and then use the techniques in the following section to

see how much extra performance we can expect by cleaning the training data as well before launching that substantially more labor-intensive data cleaning project.

In some domains, for example medical imaging, the labelling can be difficult, and two different labellers might not agree on the label for the very same data point. This agreement between labellers is also called inter-rater reliability. It can be wise to check this metric on a subset of your data by assigning multiple labellers for that data. If the inter-rater reliability is low, you might want to consider addressing this issue. This can, for example, be done by assigning multiple labellers to all data points in the validation and test data and, if you can afford the extra labelling cost, also to the training data. For the samples where labellers do not agree, the majority vote can be used for these labels.

### Mismatched Training and Validation/Test Data

As already pointed out in Chapter 4, we should strive to let the training data come from the same distribution as the validation and test data. However, there are situations where, for different reasons, we can accept the training data coming from a slightly different distribution than the validation and test data. One reason was presented in the previous section where we chose to correct mislabelled data in the validation and test data but not necessarily to invest the time to do the same correction to the training data.

Another reason for mismatched training and validation/test data is that we might have access to another, substantially larger dataset which comes from a slightly different distribution than the data we care about but is similar enough that the advantage of having a larger training data outweighs the disadvantage of that data mismatch. This scenario is further described in Section 13.3.

If we have a mismatch between training data and validation/test data, that mismatch contributes to yet another error source of the final validation error $E_{\text{hold-out}}$ that we care about. We want to estimate the magnitude of that error source. This can be done by revising the training–validation–test data split. From the training data, we can carve out a separate *training-validation* dataset, see Figure 13.4. That dataset is neither used for training nor for validation. However, we do evaluate the performance of our model on that dataset as well. As before, the remaining part of the training data is used for training, the validation data is used for comparing different model structures, and test data is used for evaluating the final performance of the model.

This modified data split also allows us to revise the decomposition in (13.1) to include this new error source:

$$E_{\text{hold-out}} = E_{\text{train}} + \underbrace{(E_{\text{train-val}} - E_{\text{train}})}_{\approx \text{ generalisation gap}} + \underbrace{(E_{\text{hold-out}} - E_{\text{train-val}})}_{\approx \text{ train-val mismatch}}, \tag{13.2}$$

where $E_{\text{train-val}}$ is the performance of the model on the new training-validation data and where, as before, $E_{\text{hold-out}}$ and $E_{\text{train}}$ are the performances on the validation and training data, respectively. With this new decomposition, the term $E_{\text{train-val}} - E_{\text{train}}$ is


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026, http://smlbook.org*

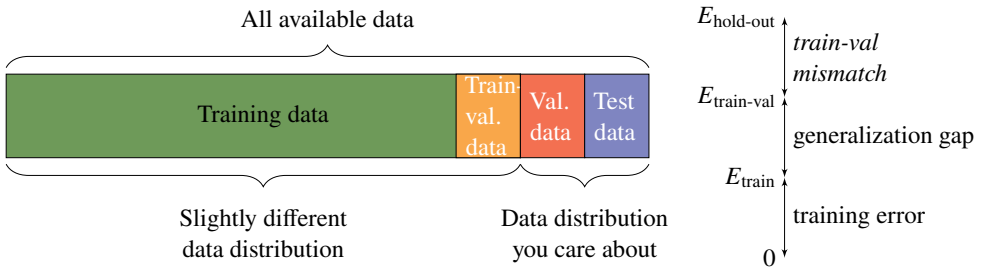



**Figure 13.4:** Revising the training–validation–test data split by carving out a separate training-validation dataset from the training data.

an approximation of the generalisation gap, that is, how well the model generalises to unseen data *of the same distribution* as the training data, whereas the term $E_{\text{hold-out}} - E_{\text{train-val}}$ is the error related to the training-validation data mismatch. If the term $E_{\text{hold-out}} - E_{\text{train-val}}$ is small in comparison to the other two terms, it seems likely that the training-validation data mismatch is not a big problem and that it is better to focus on techniques reducing the other training error and the generalisation gap as we talked about earlier. On the other hand, if $E_{\text{hold-out}} - E_{\text{train-val}}$ is significant, the data mismatch does have an impact, and it might be worth investing time reducing that term. For example, if the mismatch is caused by the fact that we only corrected labels in the validation and test data, we might want to consider correcting labels for the training data as well.

## 13.3 What If We Cannot Collect More Data?

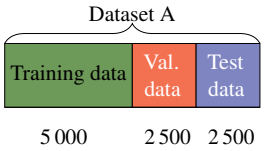
We have seen in Section 13.2 that collecting more data is a good strategy to reduce the generalisation gap and hence reduce overfitting. However, collecting labelled data is usually expensive and sometimes not even possible. What can we do if we cannot afford to collect more data but still want to benefit from the advantages that a larger dataset would give? In this section, a few approaches are presented.

### Extending the Training Data with Slightly Different Data

As already mentioned, there are situations where we can accept the training data coming from a slightly different distribution than the validation and test data. One reason to accept this is if we then would have access to a substantially larger training dataset.

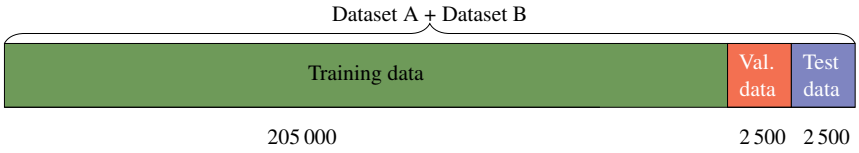
Consider a problem with 10 000 data points, representing the data that we would also expect to get when the model is deployed in production. We call this Dataset A. We also have another dataset with 200 000 data points that come from a slightly different distribution but which is similar enough that we think exploiting information from that data can improve the model. We call this Dataset B. Some options to proceed would be the following:

- **Option 1** Use only Dataset A and split it into training, validation, and test data.



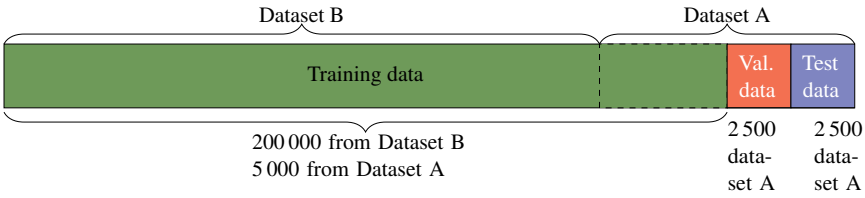

  The advantage of this option is that we only train, validate, and evaluate on Dataset A, which is also the type of data that we want our model to perform well on. The disadvantage is that we have quite few data points, and we do not exploit potentially useful information in the larger Dataset B.

- **Option 2** Use both Dataset A and Dataset B. Randomly shuffle the data and split it into training, validation, and test data.




  The advantage over option 1 is that we have a lot more data available for training. However, the disadvantage is that we mainly evaluate the model on data from Dataset B, whereas we want our model to perform well on data from Dataset A.

- **Option 3** Use both Dataset A and Dataset B. Use data points from Dataset A for validation data and test data and some in the training data. Dataset B only goes into the training data.




  Similar to option 2, the advantage is that we have more training data in comparison to option 1, and in contrast to option 2, we now evaluate the model on data from Dataset A, which is the data we want our model to perform well on. However, one disadvantage is that the training data no longer has the same distribution as the validation and test data.

From these three options, we would recommend either option 1 or 3. In option 3, we exploit the information available in the much larger Dataset B but evaluate only on the data we want the model to perform well on (Dataset A). The main disadvantage with option 3 is that the training data no longer comes from the same distribution as the validation data and test data. In order to quantify how big an impact this mismatch has on the final performance, the techniques described in Section 13.2 can be used. To push the model to do better on data from Dataset A during training, we can also


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

consider giving data from Dataset A a higher weight in the cost function than data from Dataset B, or simply upsample the data points in Dataset A that belong to the training data.

There is no guarantee that adding Dataset B to the training data will improve the model. If that data is very different from Dataset A, it can also do harm, and we might be better off just using Dataset A as suggested in option 1. Using option 2 is generally not recommended since we would then (in contrast to option 1 and 3) evaluate our model on data which is different from that which we want it to perform well on. Hence, if the data in Dataset A is scarce, prioritise putting it into the validation and test datasets and, if we can afford it, some of it in the training data.

### Data Augmentation

*Data augmentation* is another approach to extending the training data without the need to collect more data. In data augmentation, we construct new data points by duplicating the existing data with invariant transformations. This is especially common for images, where such invariant transformations can be cropping, rotation, vertical flipping, noise addition, colour shift, and contrast change. For example, if we vertically flip an image of a cat, it still displays a cat; see the examples Figure 13.5. One should be aware that some objects are not invariant to some of these operations. For example, a flipped image of a digit is not a valid transformation. In some cases such operations can even make the object resemble an object from another class. If we were to flip an image of a '6' both vertically and horizontally, that image would resemble a '9'. Hence, before applying data augmentation, we need to know and understand the problem and the data. Based on that knowledge we can identify valid invariants and suggest which transformations that can be applied to augment the data that we already have.

To apply data augmentation offline before the training would increase the required amount of storage and is hence only recommended for small datasets. For many models and training procedures, we can instead apply it online during training. For example, if we train a parametric model using stochastic gradient descent (see Section 5.5), we can apply the transformation directly on the data that goes into the current mini-batch without the need to store the transformed data.

### Transfer Learning

Transfer learning is yet another technique that allows us to exploit information from more data than the dataset we have. In transfer learning we use the knowledge from a

**Figure 13.5:** Example of data augmentation applied to images. An image of a cat has been reproduced by tilting, vertical flipping and cropping.
*source*: Image of cat is reprinted from `https://commons.wikimedia.org/wiki/File:Stray_Cat,_Nafplio.jpg` and is in the public domain.

model that has been trained on a different task with a different dataset and then apply that model in solving a different, but slightly related, problem.

Transfer learning is especially common for layered model structures such as the neural network models introduced in Chapter 6. Consider an application where we want to detect whether a certain type of skin cancer is malignant or benign, and for this task we have 100 000 labelled images of skin cancer. We call this Task A. Instead of training the full neural network from scratch on this data, we can reuse an already pretrained network from another image classification task (Task B), which preferably has been trained on a much larger dataset, not necessarily containing images even resembling skin cancer tumors. By using the weights from the model trained for Task B and only train the last few layers on the data for Task A, we can get a better model than if the whole model would had been trained on only the data for Task A. The procedure is also displayed in Figure 13.6. The intuition is that the layers closer to the input accomplish tasks that are generic for all types of images, such as extracting lines, edges and corners in the image, whereas the layers closer to the output are more specific to the particular problem.

In order for transfer learning to be applicable, we need the two tasks to have the same type of input (in the example above, images of the same dimension). Further, for transfer learning to be an attractive option, the task that we transfer from should have been trained on substantially more data than the task we transfer to.

### Learning from Unlabelled Data

We can also improve our model by learning from an additional (typically much larger) dataset without outputs, so called unlabelled data. Two families of such methods are semi-supervised learning and proxy-task learning. In our description below, we call our original dataset with both inputs and output Dataset A and our unlabelled dataset Dataset B.

In semi-supervised learning, we formulate and train a generative model for the inputs in both Dataset A and Dataset B. The generative model of the inputs and

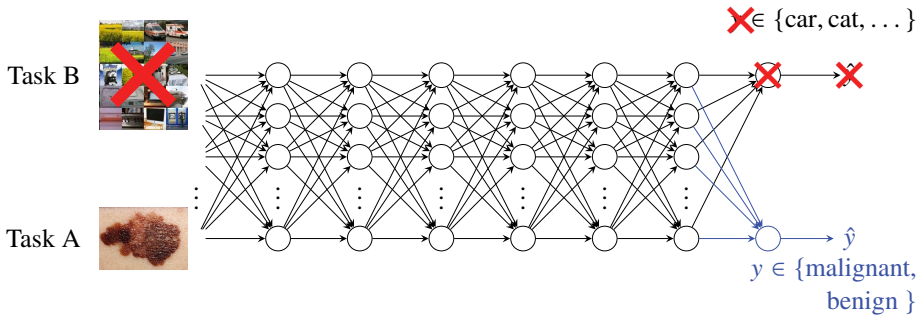



**Figure 13.6:** In transfer learning, we reuse models that have been trained on a different task than the one we are interested in. Here we reuse a model which has been trained on images displaying all sorts of classes, such as cars, cats, and computers, and later train only the last few layers on the skin cancer data which is the task we are interested in.

*source*: Skin cancer sample is reprinted from `https://visualsonline.cancer.gov/details.cfm?imageid=9186` and is in the public domain.

the supervised model for Dataset A are then trained jointly. The idea is that the generative model on the inputs, which is trained on the much larger dataset, improves the performance of the supervised task. Semi-supervised learning is further described in Chapter 11.

In proxy-task learning, we instead use Dataset B in a very similar way as we do in transfer learning described previously. Hence, we pretrain the model based on Dataset B and then fine-tune that model using Dataset A. Since Dataset B does not contain any outputs, we automatically generate outputs for Dataset B and pretrain our model with these generated outputs. The automatically generated outputs can, for example, be a subset of the input variables or a transformation thereof. As in transfer learning, the idea is that the pretrained model learns to extract features from the input data which then can be used to improve the training of the supervised task that we are interested in. Also, if we don't have an additional unlabelled Dataset B, we can also use proxy-task learning on the inputs in Dataset A as the pretraining before training on the supervised task on that dataset.

## 13.4 Practical Data Issues

Besides the amount and distribution of data, a machine learning engineer may also face other data issues. In this section, we will discuss some of the most common ones; outliers, missing data, and if some features can be removed.

### Outliers

In some applications, a common issue is *outliers*, meaning data points whose outputs do not follow the overall pattern. Two typical examples of outliers are sketched in Figure 13.7. Even though the situation in Figure 13.7 looks simple, it can be quite hard to find outliers when the data has more dimensions and is harder to visualise. The

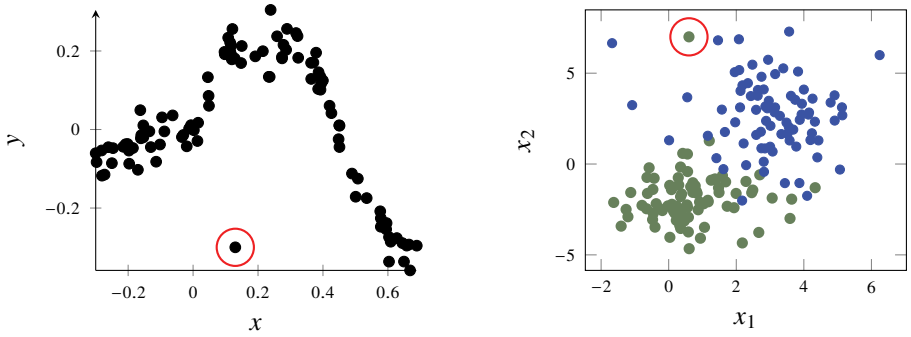



**Figure 13.7:** Two typical examples of outliers (marked with red circle) in regression (left) and classification (right), respectively.

error analysis discussed in Section 13.2, which amounts to inspecting misclassified data points in the validation data, is a systematic way to discover outliers.

When facing a problem with outliers, the first question to ask is whether the outliers are meant to be captured by the model or not. Do the encircled data points in Figure 13.7 describe an interesting phenomenon that we would like to predict, or are they irrelevant noise (possibly originating from a poor data collection process)? The answer to this question depends on the actual problem and ambition. Since outliers by definition (no matter their origin) do not follow the overall pattern, they are typically hard to predict.

If the outliers are not of any interest, the first thing we should do is to consult the data provider and identify the reason for the outliers and if something could be changed in the data collection process to avoid these outliers, for example replacing a malfunctioning sensor. If the outliers are unavoidable, there are basically two approaches one could take. The first approach is to simply delete (or replace) the outliers in the data. Unfortunately this means that one has to first find the outliers, which can be hard, but sometimes some thresholding and manual inspection (that is, look at all data points whose output value is smaller/larger than some value) can help. Once the outliers are removed from the data, one can proceed as usual. The second approach is to instead make sure that the learning algorithm is robust against outliers, for example by using a robust loss function such as absolute error instead of squared error loss (see Chapter 5 for more details). Making a model more robust amounts, to some extent, to making it less flexible. However, robustness amounts to making the model less flexible in a particular way, namely by putting less emphasis on the data points whose predictions are severely wrong.

If the outliers are of interest to the prediction, they are not really an issue but rather a challenge. We have to use a model that is flexible enough to capture the behavior (small bias). This has to be done with care since very flexible models have a high risk of also overfitting to noise. If it turns out that the outliers in a classification problem are indeed interesting and in fact are from an underrepresented class, we are rather facing an imbalanced problem; see Section 4.5.


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September* 1, 2026, `http://smlbook.org`

### Missing Data

A common practical issue is that certain values are sporadically missing in the data. Throughout this book so far, the data has always consisted of complete input-output pairs $\{\mathbf{x}_i, y_i\}_{i=1}^n$, and *missing data* refers to the situation where some (or a few) values from either the input $\mathbf{x}_i$ or the output $y_i$, for some $i$, are missing. If the output $y_i$ is missing, we can also refer to it as *unlabelled* data. It is a common practice to denote missing data in a computer with `NaN` (not a number), but less obvious codings also exists, such as 0. Reasons for missing data could, for instance, be a malfunctioning sensor or similar issues at data collection time, or that certain values for some reason have been discarded during the data processing.

As for outliers, a sensible first option is to figure out the reason for the missing data. By going back to the data provider, this issue could potentially be fixed and the missing data recovered. If this is not possible, there is no universal solution for how to handle missing data. There is, however, some common practice which can serve as a guideline. First of all, if the output $y_i$ is missing, the data point is useless for supervised machine learning[2] and can be discarded. In the following, we assume that the missing values are only in the input $\mathbf{x}_i$.

The easiest way to handle missing data is to discard the entire data points ('rows in $\mathbf{X}$') where data is missing. That is, if some feature is missing in $\mathbf{x}_i$, the entire input $\mathbf{x}_i$ and its corresponding output value $y_i$ are discarded from the data, and we are left with a smaller dataset. If the dataset that remains after this procedure still contains enough data, this approach can work well. However, if this would lead to too small a dataset, it is of course problematic. More subtle, but also important, is the situation when the data is missing in a systematic fashion, for example that missing data is more common for a certain class. In such a situation, discarding data points with missing data would lead to a mismatch between reality and training data, which may degrade the performance of the learned model further.

If missing data is common, but only for certain features, another easy option is to not use those features ('column of $\mathbf{X}$') which are suffering from missing data. Whether or not this is a fruitful approach depends on the situation.

Instead of discarding the missing data, it is possible to impute (fill in) the missing values using some heuristics. Say, for example, that the $j$th feature $x_j$ is missing from data point $\mathbf{x}_i$. A simple imputation strategy would be to take the mean or median of $x_j$ for all other data points (where it is not missing) or the mean or median of $x_j$ for all data points of the same class (if it is a classification problem). It is also possible to come up with more complicated imputation strategies, but each imputation strategy implies some assumptions about the problem. Those assumptions might or might not be fulfilled, and it is hard to guarantee that imputation will help the performance in the end. A poor imputation can even degrade the performance compared to just discarding the missing data.

[2] The 'partly labelled data' problem is a semi-supervised problem, which is introduced in Chapter 11 but not covered in depth by this book.

Some methods are actually able to handle missing data to some extent (which we have not discussed in this book), but under rather restrictive assumptions. Such an example is that the data is 'completely missing at random', meaning that which data is missing is completely uncorrelated to what value it, and other features and the output, would have had, had it not been missing. Assumptions like these are very strong and rarely met in practice, and the performance can be severely degraded if those assumptions are not fulfilled.

### Feature Selection

When working with a supervised machine learning problem, the question of whether all available input variables/features contribute to the performance is often relevant. Removing the right feature is indeed a type of regularisation which can possibly reduce overfitting and improve the performance, and the data collection might be simplified if a certain variable does not even have to be collected. Selecting between the available features is an important task for the machine learning engineer.

The connection between regularisation and feature selection becomes clear by considering $L^1$ regularisation. Since the main feature of $L^1$ regularisation is that the learned parameter vector $\widehat{\boldsymbol{\theta}}$ is sparse, it effectively removes the influence of certain features. If using a model where $L^1$ regularisation is possible, we can study $\widehat{\boldsymbol{\theta}}$ to see which features we simply can remove from the dataset. However, if we cannot or prefer not to use $L^1$ regularisation, we can alternatively use a more manual approach to feature selection.

Remember that our overall goal is to obtain a small new data error $E_{\text{new}}$, which we for most methods estimate using cross-validation. We can therefore always use cross-validation to tell whether we gain or lose by including a certain feature in $\mathbf{x}$. Depending on the amount of data, evaluating all possible combinations of removed features might not be a good idea, either due to computational aspects or the risk of overfitting. There are, however, some rules of thumb that can possibly give us some guidance on which features we should investigate more closely for whether they contribute to the performance or not.

To get a feeling for the different features, we can look at the correlation between each feature and the output and thereby get a clue about which features might be more informative about the output. If there is little correlation between a feature and the output, it is possibly a useless feature, and we could investigate further if we can remove it. However, looking one feature at a time can be misleading, and there are cases where this would lead to the wrong conclusion – for example the case in Example 9.1.

Another approach is to explore whether there are redundant features, with the reasoning that having two features that (essentially) contain the same information will lead to an increased variance compared to having only one feature with the same information. Based on this argument, one may look at the pairwise correlation between the features and investigate removing features that have a high correlation to other features. This approach is somewhat related to PCA (Section 12.3).

## 13.5 Can I Trust my Machine Learning Model?

Supervised machine learning presents a powerful family of all-purpose general black-box methods and has demonstrated impressive performance in many applications. The main argument for supervised machine learning is, frankly, that it works well empirically. However, depending on the requirements of the application, supervised machine learning also has a potential shortcoming, in that it relies on 'repeating patterns seen in training data' rather than 'deduction from a set of carefully written rules'.

### Understanding Why a Certain Prediction was Made

In some applications, there might be an interest in 'understanding' why a certain prediction was made by a supervised machine learning model, for example in medicine or law. Unfortunately, the underlying design philosophy in machine learning is to deliver good predictions rather than to explain them.

With a simpler model, like the ones in Chapters 2–3, it can to some degree be possible for an engineer to inspect the learned model and explain the 'reasoning' behind it for a non-expert. For more complicated models, however, it can be a rather hard task.

There are, however, methods at the forefront of research, and the situation may look different in the future. A related topic is that of so-called adversarial examples, which essentially amounts to finding an input $\mathbf{x}'$ which is as close as possible to $\mathbf{x}$ but gives a different prediction. In the image classification setting, it can, for example, be the problem of having a picture of a car being predicted as a dog by only changing a few pixel values.

### Worst Case Guarantees

In the view of this book, a supervised machine learning model is good if it attains a small $E_{\text{new}}$. It is, however, important to remember that $E_{\text{new}}$ is a *statistical* claim, under the assumption that the training and/or test data resembles the reality which the model will face once it is put into production. And even if that non-trivial assumption is satisfied, there are no claims about how badly the model will predict in the worst individual cases. This is indeed a shortcoming of supervised machine learning and potentially also a show-stopper for some applications.

Simpler and more interpretable models, like logistic regression and trees, for example, can be inspected manually in order to deduce the 'worst case' that could happen. By looking at the leaf nodes in a regression tree, as an example, it is possible to give an interval within which all predictions will lie. With more complicated models, like random forests and deep learning, it is very hard to give any worst case guarantees about how inaccurate the model can be in its predictions when faced with some particular input. However, an extensive testing scheme might reveal some of the potential issues.

## 13.6 Foundation Models

An increasingly common way to use machine learning in practice is to not build your own model from scratch but instead to rely on so-called *foundation models* provided by third parties. Some of these models are openly available for download, others are openly licensed but with restrictions on their use, and yet others are accessible only as paid services through an interface on the internet. Either way, the trend is clear: many contemporary machine learning systems are built around a foundation model that the engineers who built the system did not train themselves.

### What Is a Foundation Model?

A foundation model is, technically speaking, a machine learning model just like the ones we have discussed throughout this book – typically a transformer (Section 7.4) or a convolutional neural network (Section 7.2). What sets a foundation model apart is not the underlying model itself but rather two characteristics on how they are trained: First, they are trained on enormous amounts of data, with very large compute budgets. The scale is such that training a foundation model from scratch is usually beyond the reach of any organisation that is not prepared to invest substantially in this activity. Second, they are trained on rather generic tasks – such as next-token prediction for an LLM from a very broad set of training data sources – that are useful as a backbone for many downstream applications, rather than being tailored to a single very specific task.

The idea of foundation models clearly overlaps with the idea of transfer learning (Section 13.3), that is, re-using a model trained on a different dataset. What sets foundation models apart from transfer learning is not the technicalities as much as the sheer scale of the training data and their general-purpose design intent.

The most prominent foundation models today are the large language models, LLMs, introduced in Section 7.5, which are transformer-based models trained on huge amounts of text. Foundation models are, however, not limited to text. CLIP (**??**) plays a similar role in the vision-and-text domain, and modern image segmentation models (Example 1.4) act as a foundation for some image analysis tasks.

A foundation model rarely solves the end user's problem out of the box. To turn it into something useful, the engineer has to *adapt* it to the application at hand. The adaptation can take many forms, ranging from very lightweight to fairly involved:

- Designing a textual instruction to an LLM, called a *prompt*, that steers the model's behaviour towards the desired task. For LLMs, this can be a surprisingly capable approach on its own.
- Using the foundation model only as a feature extractor and learning a small downstream model on top of the resulting features (or embeddings). This is essentially the idea of transfer learning that we discussed in Section 13.3.
- Fine-tuning the foundation model on a smaller, application-specific dataset.


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

To keep the cost manageable, only a small subset of the parameters is often updated, while the rest are kept fixed.

- Combining an LLM foundation model with a retrieval system that supplies it with relevant information at the time of prediction, so-called retrieval-augmented generation (RAG).

The two examples below illustrate two rather different ways of using a foundation model in practice.

**Example 13.2 An LLM as a natural-language interface**

A pizza restaurant has been taking orders through an online form for years. The form has structured fields for the type of pizza, size, quantity, optional toppings, and any drinks. The restaurant now wants to offer customers the option to write their order in plain text instead, e.g. "two large margheritas with extra mushrooms, and a soda".

Rather than training a new model for this purpose – which would require a substantial labelled dataset of free-text orders mapped to the form fields – the restaurant uses a hosted LLM as the natural-language interface. Each customer message is wrapped into a fixed prompt that explains the available menu and the format that the answer should take, and the prompt is sent to the API of an LLM provider. The LLM returns a structured response, which is parsed and filled into the existing form. The form, the database, and the rest of the restaurant's IT infrastructure remain entirely unchanged.

In this setup, the engineering work is not on the model itself but on designing a robust prompt, validating that the structured output is well-formed, handling cases where the model's output deviates from the expected format, and gracefully reporting problems back to the customer when they occur.

**Example 13.3 CLIP for semantic image search**

A photographer has a private collection of about 50 000 photographs and would like to search through them using free-text queries, such as "photos of my dog at the beach" or "a sunset over mountains". Building such a search system from scratch would require a substantial labelled dataset and a non-trivial amount of model training.

Instead, the photographer downloads CLIP – a vision-language foundation model that maps images and text fragments into a common embedding space of latent representations, see **??** – and uses it directly. Once and for all, CLIP is run for each photograph in the collection to produce an embedding vector, and these vectors are stored alongside the photographs. To answer a text query for an image, CLIP is run on the query text to produce a query embedding, and the photographs whose embeddings are closest to it (in scalar product, say) are returned as the search result.

In this setup, no part of the photographer's collection ever leaves the local machine; the foundation model is used purely as a feature extractor. The engineering work consists of integrating CLIP into the search pipeline, indexing the embeddings efficiently, and choosing a sensible similarity threshold.

### Practical Concerns When Using a Foundation Model

Integrating a foundation model into a larger system is to a large extent a software systems integration project, with all the challenges that this entails: how data is sent and returned (and which of that data is sensitive), where the model is deployed, cyber security, and so on. Cost and latency are also relevant in a way they often are not for in-house models. Each call to a hosted foundation model API typically costs money and takes time, which can quickly become important architectural constraints. These are important issues, but they are largely outside the scope of this book.

There are, however, a number of points that specifically require an understanding of machine learning, and which the engineer who integrates a foundation model has to take responsibility for.

- *The training data is largely unknown.* For most foundation models, you only have a partial knowledge of which data the model was trained on, and the dataset is too large for anyone (including its developers) to characterise in detail. As a consequence, it is hard to predict in advance when the model will work well and when it will fail. The intuition that comes from training your own model, "the training data covers this regime well, so predictions in that regime should be reliable", does not transfer directly to foundation models.

- *You still need your own validation and test data.* Reported benchmark scores for foundation models should be treated with caution as a guide to how the model will perform in your application. The model has been evaluated on benchmarks whose input distribution is likely different from yours; some benchmarks may even have leaked into the training data, leading to optimistically biased numbers; and benchmark scores typically measure average-case behaviour rather than the worst-case behaviour you might care about. The only evaluation you can really trust is one performed on data that is representative of your application, and that you are confident the model has not seen during training.

- *Trust and worst-case behaviour are still concerns.* The discussion in Section 13.5 of explaining individual predictions and giving worst-case guarantees applies in full to foundation models – and arguably more strongly. A foundation model is typically more complex than the models you would train yourself, and you also know less about its training data and design choices.

- *Hallucinations and other failure modes.* For LLMs in particular, the model can produce fluent and confident-sounding output that is nevertheless factually incorrect, a phenomenon called *hallucination* (Section 7.5). This is a property of the technology, not a temporary bug, and any system that relies on an LLM's output as if it were a reliable source of fact has to be designed accordingly. Other foundation models have analogous failure modes that are worth investigating before trusting their output.


*Machine Learning – A First Course for Engineers and Scientists*
*Second Edition, online draft version September 1, 2026,* `http://smlbook.org`

### Regulation and Documentation

Foundation models are increasingly subject to legal and regulatory frameworks that target them specifically. The European Union's AI Act, for example, places obligations on providers of foundation models (referred to as *general-purpose AI models*) including transparency requirements about training data and model capabilities, and additional obligations for the largest models that are deemed to pose a systemic risk. Similar legislation is being developed in other jurisdictions. As an engineer integrating a foundation model into a product, you may inherit some of these obligations, and the regulatory landscape is something to factor in already at the system design stage.

A common practice for foundation model transparency, with or without legislation requiring it, is the publication of a so-called *model card* alongside the model. A model card is a short document, analogous to the datasheet of a hardware component, that describes the model's intended use, its training data and procedure (to the extent that the developer is willing to disclose), its known limitations, evaluation results, and ethical considerations. Reading the model card carefully before integrating a foundation model into a product is good engineering practice. If no model card is available, that itself is a piece of information about the model.

## 13.7 Further Reading

The user aspects of machine learning is a fairly under-explored area, both in academic research publications and in standard textbooks on machine learning. Two exceptions are Ng (2019) and Burkov (2020), from which parts of this chapter have been inspired. Regarding data augmentation, see Shorten and Khoshgoftaar (2019) for a review of different techniques for images.

Some of the research on 'understanding' why a certain prediction was made by a machine learning method is summarised by Hassija et al. (2024). The notion of foundation models was introduced thoroughly by Bommasani et al. (2021).

# 14 Ethics and Societal Aspects of Machine Learning

*This chapter is currently work in progress*

# Bibliography


Abramson, Josh et al. (June 2024). 'Accurate structure prediction of biomolecular interactions with AlphaFold 3'. In: *Nature* 630.8016, pp. 493–500. DOI: 10.1038/s41586-024-07487-w. URL: https://doi.org.

Abu-Mostafa, Yaser S., Malik Magdon-Ismail, and Hsuan-Tien Lin (2012). *Learning From Data. A short course*. AMLbook.com.

Barber, David (2012). *Bayesian Reasoning and Machine Learning*. Cambridge University Press.

Belkin, Mikhail, Daniel Hsu, Siyuan Ma, and Soumik Mandal (2019). 'Reconciling modern machine-learning practice and the classical bias–variance trade-off'. In: *Proceedings of the National Academy of Sciences* 116.32, pp. 15849–15854.

Bishop, Christopher M. (1995). 'Regularization and Complexity Control in Feed-forward Networks'. In: *Proceedings of the International Conference on Artificial Neural Networks*, pp. 141–148.

Bishop, Christopher M. (2006). *Pattern Recognition and Machine Learning*. Springer.

Bishop, Christopher M. and Hugh Bishop (2024). *Deep Learning: Foundations and Concepts*. Springer. URL: https://www.bishopbook.com/.

Bishop, Christopher M. and Julia Lasserre (2007). 'Generative or Discriminative? Getting the Best of Both Worlds'. In: *Bayesian Statistics* 8, pp. 3–24.

Blei, David M., Alp Kucukelbir, and Jon D. McAuliffe (2017). 'Variational Inference: A Review for Statisticians'. In: *Journal of the American Statisticial Association* 112.518, pp. 859–877.

Blundell, Charles, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra (2015). 'Weight Uncertainty in Neural Network'. In: *Proceedings of the 32nd International Conference on Machine Learning*, pp. 1613–1622.

Bommasani, Rishi et al. (2021). 'On the Opportunities and Risks of Foundation Models'. In: *arXiv preprint arXiv:2108.07258*.

Bottou, Léon, Frank E. Curtis, and Jorge Nocedal (2018). 'Optimization Methods for Large-Scale Machine Learning'. In: *SIAM Review* 60.2, pp. 223–311.

Breiman, Leo (1996). 'Bagging Predictors'. In: *Machine Learning* 24, pp. 123–140.

Breiman, Leo (2001). 'Random Forests'. In: *Machine Learning* 45.1, pp. 5–32.

Breiman, Leo, Jerome Friedman, Charles J. Stone, and Richard A. Olshen (1984). *Classification And Regression Trees*. Chapman & Hall.

Brown, Tom, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. (2020). 'Language Models are Few-Shot Learners'. In: *Advances in Neural Information Processing Systems 33*, pp. 1877–1901.

Burkov, Andriy (2020). *Machine Learning Engineering*. URL: http://www.mlebook.com/.

Burkov, Andriy (2025). *The Hundred-Page Languge Models Book. hands-on with PyTorch*. True Positive Inc.

Chang, Chih-Chung and Chih-Jen Lin (2011). 'LIBSVM: A library for support vector machines'. In: *ACM Transactions on Intelligent Systems and Technology* 2 (3). Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`, 27:1–27:27.

Chen, Tianqi and Carlos Guestrin (2016). 'XGBoost: A Scalable Tree Boosting System'. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 785–794.

Chen, Ting, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton (2020). 'A Simple Framework for Contrastive Learning of Visual Representations'. In: *Proceedings of the 37th International Conference on Machine Learning (ICML)*. Vol. 119. Proceedings of Machine Learning Research, pp. 1597–1607.

Cover, Thomas M. and Peter E. Hart (1967). 'Nearest Neighbor Pattern Classification'. In: *IEEE Transactions on Information Theory* 13.1, pp. 21–27.

Cramer, Jan Salomon (2003). *The Origins of Logistic Regression*. Tinbergen Institute Discussion Papers 02-119/4, Tinbergen Institute.

Creswell, Antonia, Tom White, Vincent Dumoulin, Kai Arulkumaran, Biswa Sengupta, and Anil A. Bharath (2018). 'Generative Adversarial Networks: An Overview'. In: *IEEE Signal Processing Magazine* 35.1, pp. 53–65.

Deisenroth, M. P., A. Faisal, and C. O. Ong (2019). *Mathematics for machine learning*. Cambridge University Press.

Dheeru, Dua and Efi Karra Taniskidou (2017). *UCI Machine Learning Repository*. URL: `http://archive.ics.uci.edu/ml`.

Domingos, Pedro (2000). 'A Unified Bias-Variance Decomposition and its Applications'. In: *Proceedings of the 17th International Conference on Machine Learning*, pp. 231–238.

Duchi, J., E. Hazan, and Y. Singer (2011). 'Adaptive subgradient methods for online learning and stochastic optimization'. In: *Journal of Machine Learning Research (JMLR)* 12, pp. 2121–2159.

Dusenberry, Michael W, Ghassen Jerfel, Yeming Wen, Yi-an Ma, Jasper Snoek, Katherine Heller, Balaji Lakshminarayanan, and Dustin Tran (2020). 'Efficient and Scalable Bayesian Neural Nets with Rank-1 Factors'. In: *Proceedings of the 37nd International Conference on Machine Learning*.

Efron, Bradley and Trevor Hastie (2016). *Computer age statistical inference*. Cambridge University Press.

Ezekiel, Mordecai and Karl A. Fox (1959). *Methods of Correlation and Regression Analysis*. John Wiley & Sons, Inc.

Faber, Felix A., Alexander Lindmaa, O. Anatole von Lilienfeld, and Rickard Armiento (Sept. 2016). 'Machine Learning Energies of 2 Million Elpasolite ($ABC_2D_6$) Crystals'. In: *Phys. Rev. Lett.* 117 (13), p. 135502. DOI: `10.1103/PhysRevLett.117.135502`. URL: `https://link.aps.org/doi/10.1103/PhysRevLett.117.135502`.

Fisher, Ronald A. (1922). 'On the mathematical foundations of theoretical statistics'. In: *Philospohical Transactions of the Royal Society A* 222, pp. 309–368.

Flach, Peter and Meelis Kull (2015). 'Precision-Recall-Gain Curves: PR Analysis Done Right'. In: *Advances in Neural Information Processing Systems 28*, pp. 838–846.

Fort, Stanislav, Huiyi Hu, and Balaji Lakshminarayanan (2019). 'Deep ensembles: A loss landscape perspective'. In: *arXiv:1912.02757 preprint*.

Frazier, Peter I. (2018). 'A Tuutorial on Bayesian Optimization'. In: *arXiv:1807.02811*.

Freund, Yoav and Robert E. Schapire (1996). 'Experiments with a new boosting algorithm'. In: *Proceedings of the 13th International Conference on Machine Learning*.

Friedman, Jerome (2001). 'Greedy function approximation: A gradient boosting machine'. In: *Annals of Statistics* 29.5, pp. 1189–1232.

Friedman, Jerome, Trevor Hastie, and Robert Tibshirani (2000). 'Additive logistic regression: a statistical view of boosting (with discussion)'. In: *The Annals of Statistics* 28.2, pp. 337–407.

Gelman, Andrew, John B. Carlin, Hal S. Stern, David. B. Dunson, Aki Vehtari, and Donald B. Rubin (2014). *Bayesian data analysis*. 3rd ed. CRC Press.

Gershman, Samuel J. and David M. Blei (2012). 'A tutorial on Bayesian nonparametric models'. In: *Journal of Mathematical Psychology* 56.1, pp. 1–12.

Ghahramani, Zoubin (2013). 'Bayesian non-parametrics and the probabilistic approach to modelling'. In: *Philospohical Transactions of the Royal Society A* 371.1984.

Ghahramani, Zoubin (2015). 'Probabilistic machine learning and artificial intelligence'. In: *Nature* 521, pp. 452–459.

Gneiting, Tilmann and Adrian E. Raftery (2007). 'Strictly Propoer Scoring Rules, Prediction, and Estimation'. In: *Journal of the American Statistical Association* 102.477, pp. 359–378.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press.

Goodfellow, Ian, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio (2014). 'Generative Adversarial Nets'. In: *Advances in Neural Information Processing Systems 27*, pp. 2672–2680.

Gustafsson, Stefan et al. (May 2026). 'A deep learning ECG model for identification and localization of occlusion myocardial infarction'. In: *Nature Communications* 17, p. 4336. DOI: `10.1038/s41467-026-73023-1`. URL: `https://doi.org/10.1038/s41467-026-73023-1`.

Hamelijnck, O., T. Damoulas, K. Wang, and M. A. Girolami (2019). 'Multi-resolution multi-task Gaussian processes'. In: *Neural Information Processing Systems (NeurIPS)*. Vancouver, Canada.

Hardt, Moritz, Benjamin Recht, and Yoram Singer (2016). 'Train faster, generalize better: Stability of stochastic gradient descent'. In: *Proceedings of the 33rd International Conference on Machine Learning*.

Hassija, Vikas, Vinay Chamola, Atmesh Mahapatra, Abhinandan Singal, Divyansh Goel, Kaizhu Huang, Simone Scardapane, Indro Spinelli, Mufti Mahmud, and Amir Hussain (2024). 'Interpreting Black-Box Models: A Review on Explainable Artificial Intelligence'. In: *Cognitive Computation* 16, pp. 45–74. DOI: `10.1007/s12559-023-10179-8`. URL: `https://doi.org/10.1007/s12559-023-10179-8`.

Hastie, Trevor, Robert Tibshirani, and Jerome Friedman (2009). *The Elements of Statistical Learning. Data mining, inference, and prediction*. 2nd ed. Springer.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun (2016). 'Deep Residual Learning for Image Recognition'. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778.

Hjort, Nils Lid, Chris Holmes, Peter Müller, and Stephen G. Walker, eds. (2010). *Bayesian Nonparametrics*. Cambridge University Press.

Ho, Jonathan, Ajay Jain, and Pieter Abbeel (2020). 'Denoising Diffusion Probabilistic Models'. In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 33, pp. 6840–6851.

Ho, Tin Kam (1995). 'Random Decision Forests'. In: *Proceedings of 3rd International Conference on Document Analysis and Recognition*. Vol. 1, pp. 278–282.

Hoerl, Arthur E. and Robert W. Kennard (1970). 'Ridge regression: biased estimation for nonorthogonal problems'. In: *Technometrics* 12.1, pp. 55–67.

Ioffe, Sergey and Christian Szegedy (2015). 'Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift'. In: *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, pp. 448–456.

James, Gareth, Daniela Witten, Trevor Hastie, and Robert Tibshirani (2013). *An introduction to statistical learning. With applications in R*. Springer.

Jebara, Tony (2004). *Machine Learning: Discriminative and Generative*. Springer.

Jumper, John et al. (Aug. 2021). 'Highly accurate protein structure prediction with AlphaFold'. In: *Nature* 596.7873, pp. 583–589. DOI: 10.1038/s41586-021-03819-2. URL: https://doi.org/10.1038/s41586-021-03819-2.

Ke, Guolin, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu (2017). 'LightGBM: A Highly Efficient Gradient Boosting Decision Tree'. In: *Advances in Neural Information Processing Systems 30*, pp. 3149–3157.

Kendall, Alex and Yarin Gal (2017). 'What uncertainties do we need in Bayesian deep learning for computer vision?' In: *Advances in Neural Information Processing Systems 30*, pp. 5574–5584.

Le-Khac, Phuc H., Graham Healy, and Alan F. Smeaton (2020). 'Contrastive Representation Learning: A Framework and Review'. In: *IEEE Access* 8, pp. 193907–193934. DOI: 10.1109/ACCESS.2020.3031549.

Kingma, D. P. and J. Ba (2015). 'Adam: a method for stochastic optimization'. In: *Proceedings of the 3rd international conference on learning representations (ICLR)*. San Diego, CA, USA.

Kingma, Diederik P., Danilo Jimenez Rezende, Shakir Mohamed, and Max Welling (2014). 'Advances in Neural Information Processing Systems 27'. In: *Semi-supervised Learning with Deep Generative Models*, pp. 3581–3589.

Kingma, Diederik P. and Max Welling (2014). 'Auto-Encoding Variational Bayes'. In: *2nd International Conference on Learning Representations*.

Kingma, Diederik P. and Max Welling (2019). 'An Introduction to Variational Autoencoder'. In: *Foundations and Trends in Machine Learning* 12.4, pp. 307–392.

Kirillov, Alexander et al. (2023). 'Segment Anything'. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 4015–4026. DOI: 10.1109/ICCV51070.2023.00371.

Kobyzev, Ivan, Simon J. D. Prince, and Marcus A. Brubaker (2020). 'Normalizing Flows: An Introduction and Review of Current Methods'. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence*. To appear.

Lam, Remi, Alvaro Sanchez-Gonzalez, Alexander Willson, Peter Wirnsberger, Meire Fortunato, Alexander Pritzel, Suman Ravuri, Thomas Hasani, Sam Hudson, Jess Esquivel, et al. (2023). 'Learning skillful medium-range global weather forecasting'. In: *Science* 382.6677, pp. 1416–1421. DOI: 10.1126/science.adi2336.

LeCun, Yann, Bernhard Boser, John S. Denker, Don Henderson, Richard E. Howard, W. Hubbard, and Larry Jackel (1989). 'Handwritten Digit Recognition with a Back-Propagation Network'. In: *Advances in Neural Information Processing Systems 2*, pp. 396–404.

LeCun, Yann, Léon Bottou, Yoshua Bengio, and Patrick Haffner (1998). 'Gradient-based learning applied to document recognition'. In: *Proceedings of the IEEE* 86.11, pp. 2278–2324.

Liang, Percy and Michael I. Jordan (2008). 'An Asymptotic Analysis of Generative, Discriminative, and Pseudolikelihood Estimators'. In: *Proceedings of the 25th International Conference on Machine Learning*, pp. 584–591.

Liu, Zhuang, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie (2022). 'A ConvNet for the 2020s'. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 11976–11986.

Loh, Wei-Yin (2014). 'Fifty Years of Classification and Regression Trees'. In: *International Statistical Review* 82.3, pp. 329–348.

Long, J., E. Shelhamer, and T. Darell (2015). 'Fully convolutional networks for semantic segmentation'. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

MacKay, D. J. C. (2003). *Information theory, inference and learning algorithms*. Cambridge University Press.

Mandt, Stephan, Matthew D. Hoffman, and David M. Blei (2017). 'Stochastic Gradient Descent as Approximate Bayesian Inference'. In: *Journal of Machine Learning Research* 18, pp. 1–35.

Mardia, Kantilal Varichand, John T. Kent, and John Bibby (1979). *Multivariate Analysis*. Academic Press.

Mason, Llew, Jonathan Baxter, Peter Bartlett, and Marcus Frean (1999). 'Boosting Algorithms as Gradient Descent'. In: *Advances in Neural Information Processing Systems 12*, pp. 512–518.

McCullagh, P. and J. A. Nelder (2018). *Generalized Linear Models*. 2nd. Monographs on Statistics and Applied Probability 37. Chapman & Hall/CRC.

McCulloch, Warren S. and Walter Pitts (1943). 'A logical calculus of the ideas immanent in nervous activity'. In: *The bulletin of mathematical biophysics* 5.4, pp. 115–133.

Merchant, Amil, Simon Batzner, Samuel S. Schoenholz, Muratahan Aykol, Gowoon Cheon, and Ekin Dogus Cubuk (2023). 'Scaling Deep Learning for Materials Discovery'. In: *Nature* 624.7990, pp. 80–85. DOI: 10.1038/s41586-023-06735-9.

Mohri, Mehryar, Afshin Rostamizadeh, and Ameet Talwalkar (2018). *Foundations of Machine Learning*. 2nd ed. MIT Press.

Murphy, Kevin P. (2012). *Machine learning – a probabilistic perspective*. MIT Press.

Murphy, Kevin P. (2022). *Probabilistic machine learning: an introduction*. MIT Press. URL: http://probml.github.io/book1.

Murphy, Kevin P. (2023). *Probabilistic Machine Learning: Advanced Topics*. MIT Press. URL: http://probml.github.io/book2.

Nakkiran, Preetum, Gal Kaplun, Yamini Bansal, Tristan Yang, Boaz Barak, and Ilya Sutskever (2020). 'Deep Double Descent: Where Bigger Models and More Data Hurt'. In: *International Conference on Learning Representations*.

Neal, Radford M. (1996). *Bayesian Learning for Neural Networks*. Springer.

Neyshabur, Behnam, Srinadh Bhojanapalli, David McAllester, and Nati Srebro (2017). 'Exploring Generalization in Deep Learning'. In: *Advances in Neural Information Processing Systems 30*, pp. 5947–5956.

Ng, Andrew Y. (2019). *Machine learning yearning*. In press. URL: https://info.deeplearning.ai/machine-learning-yearning-book.

Ng, Andrew Y. and Michael I. Jordan (2001). 'On Discriminative vs. Generative Classifiers: A comparison of logistic regression and naive Bayes'. In: *Advances in Neural Information Processing Systems 14*, pp. 841–848.

Nocedal, Jorge and Stephen J. Wright (2006). *Numerical Optimization*. Springer.

Ouyang, Long, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. (2022). 'Training language models to follow instructions with human feedback'. In: *Advances in Neural Information Processing Systems 35*, pp. 27730–27744.

Owen, Art B. (2013). *Monte Carlo theory, methods and examples*. Available at https://statweb.stanford.edu/ owen/mc/.

Pelillo, Marcello (2014). 'Alhazen and the nearest neighbor rule'. In: *Pattern Recognition Letters* 38, pp. 34–37.

Poggio, Tomaso, Sayan Mukherjee, Ryan M. Rifkin, Alexander Rakhlin, and Alessandro Verri (2001). *b*. Tech. rep. AI Memo 2001-011/CBCL Memo 198. Massachusetts Institue of Technology - Artificial Intelligence Laboratory.

Prince, Simon J.D. (2023). *Understanding Deep Learning*. The MIT Press. URL: http://udlbook.com.

Quinlan, J. Ross (1986). 'Induction of Decision Trees'. In: *Machine Learning* 1, pp. 81–106.

Quinlan, J. Ross (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers.

Radford, Alec et al. (2021). 'Learning Transferable Visual Models From Natural Language Supervision'. In: *Proceedings of the 38th International Conference on Machine Learning (ICML)*. Vol. 139. Proceedings of Machine Learning Research, pp. 8748–8763.

Rasmussen, Carl E. and Christopher K. I. Williams (2006). *Gaussian processes for machine learning*. MIT press.

Reddi, S. J., S. Kale, and S. Kumar (2018). 'On the convergence of ADAM and beyond'. In: *International Conference on Learning Representations (ICLR)*.

Rezende, Danilo Jimenez, Shakir Mohamed, and Daan Wierstra (2014). 'Stochastic Backpropagation and Approximate Inference in Deep Generative Models'. In: *Proceedings of the 31st International Conference on Machine Learning*, pp. 1278–1286.

Ribeiro, A. H. et al. (2020). 'Automatic diagnosis of the 12-lead ECG using a deep neural network'. In: *Nature Communications* 11.1, p. 1760.

Robbins, Herbert and Sutton Monro (1951). 'A stochastic approximation method'. In: *The Annals of Mathematical Statistics* 22.3, pp. 400–407.

Robert, Chistian P. and George Casella (2004). *Monte Carlo Statistical Methods*. 2nd ed. Springer.

Rogers, Simon and Mark Girolami (2017). *A first course on machine learning*. CRC Press.

Ruder, Sebastian (2017). 'An overview of gradient descent optimization algorithms'. In: *arXiv:1609.04747*.

Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams (1986). 'Learning representations by back-propagating errors'. In: *Nature* 323.6088, pp. 533–536.

Schaeffer, Rylan, Zachary Robertson, Akhilan Boopathy, Mikail Khona, Kateryna Pistunova, Jason W. Rocks, Ila R. Fiete, Andrey Gromov, and Sanmi Koyejo (2024). 'Double Descent Demystified'. In: *ICLR Blogposts 2024*. URL: https://iclr-blogposts.github.io/2024/blog/double-descent-demystified/.

Schölkopf, Bernhard, Ralf Herbrich, and Alexander J. Smola (2001). 'A Generalized Representer Theorem'. In: *Lecture Notes in Computer Science, Vol. 2111*. LNCS 2111. Springer, pp. 416–426.

Schölkopf, Bernhard and Alexander J. Smola (2002). *Learning with kernels*. Ed. by Thomas Dietterich. MIT Press.

Shalev-Shwartz, S. and S. Ben-David (2014). *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press.

Shorten, Connor and Taghi M. Khoshgoftaar (2019). 'A survey on Image Data Augmentation for Deep Learning'. In: *Journal of Big Data* 6.1, p. 60.

Sjöberg, Jonas and Lennart Ljung (1995). 'Overtraining, regularization and searching for a minimum, with application to neural networks'. In: *International Journal of Control* 62.6, pp. 1391–1407.

Snoek, Jasper, Hugo Larochelle, and Ryan P. Adams (2012). 'Practical Bayesian Optimization of Machine Learning Algorithms'. In: *Advances in Neural Information Processing Systems 25*, pp. 2951–2959.

Steinwart, Ingo, Don Hush, and Clint Scovel (2011). 'Training SVMs Without Offset'. In: *Journal of Machine Learning Research* 12, pp. 141–202.

Strang, G. (2019). *Linear algebra and learning from data*. Wellesley - Cambridge Press.

Sumpter, D. (2016). *Soccermatics: mathematical adventures in the beautiful game*. Bloomsbury Sigma.

Tibshirani, Robert (1996). 'Regression Shrinkage and Selection via the LASSO'. In: *Journal of the Royal Statistical Society (Series B)* 58.1, pp. 267–288.

Vapnik, Vladimir N. (2000). *The Nature of Statistical Learning Theory*. 2nd ed. Springer.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin (2017). 'Attention is All you Need'. In: *Advances in Neural Information Processing Systems 30*, pp. 5998–6008.

Xu, Jianhua and Xuegong Zhang (2004). 'Kernels Based on Weighted Levenshtein Distance'. In: *IEEE International Joint Conference on Neural Networks*, pp. 3015–3018.

Xue, Jing-Hao and D. Michael Titterington (2008). 'Comment on 'On Discriminative vs. Generative Classifiers: A Comparison of Logistic Regression and Naive Bayes''. In: *Neural Processing Letters* 28, pp. 169–187.

Yang, Ling, Zhilong Zhang, Yang Song, Shenda Hong, Runsheng Xu, Yue Zhao, Wentao Zhang, Bin Cui, and Ming-Hsuan Yang (2023). 'Diffusion Models: A Comprehensive Survey of Methods and Applications'. In: *ACM Computing Surveys* 56.4, pp. 1–39. DOI: 10.1145/3626235.

Yu, Kai, Liang Ji, and Xuegong Zhang (2002). 'Kernel Nearest-Neighbor Algorithm'. In: *Neural Processing Letters* 15.2, pp. 147–156.

Zhang, Chiyuan, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals (2017). 'Understanding deep learning requires rethinking generalization'. In: *5th International Conference on Learning Representations*.

Zhang, Ruqi, Chunyuan Li, Jianyi Zhang, Changyou Chen, and Andrew Gordon Wilson (2020). 'Cyclical Stochastic Gradient MCMC for Bayesian Deep Learning'. In: *8th International Conference on Learning Representations*.

Zhu, Ji and Trevor Hastie (2005). 'Kernel Logistic Regression and the Import Vector Machine'. In: *Journal of Computational and Graphical Statistics* 14.1, pp. 185–205.

# Index