Statistical Machine Learning

Lecture notes on linear regression, logistic regression, deep learning & boosting

Fredrik Lindsten Andreas Svensson
Niklas Wahlström Thomas B. Schön

Version: February 2, 2018

Department of Information Technology, Uppsala University

Preface and reading instructions

These lecture notes are intended as a complement to the book by James et al. (2013) (available at http://www-bcf.usc.edu/~gareth/ISL/) for the course 1RT700 Statistical Machine Learning given at the Department of Information Technology, Uppsala University.

For an overview of these notes, the course content is listed along with its recommended reading in the following table:

Content	James et al. 2013	These lecture notes
Introduction to the topic	2.1, 4.1	
Statistics and probability background		1
Linear regression		2
Introduction to R	2.3	
Logistic regression	4.3	3.1
LDA, QDA	4.4	
kNN	2.2.3	
Bias-variance trade-off	2.2.1-2.2.2	
Cross validation	5.1	
Tree-based methods	8.1	
Bagging	8.2.1	
Boosting		5
Deep learning		3.2, 4

Recommendations on supplementary reading etc. can be found on the course website and the 'Further reading'-sections in these notes. We are very thankful for any comments you might have on these lecture notes, ranging from small typos all the way to bigger comments on the overall structure.

The authors
Uppsala
February 2018

Contents

1. Background and notation

The word *statistical* used in the title of this course refers to the fact that we will use statistical tools and probability theory to describe the methods that we work with. This is a very useful approach to machine learning since most data encountered in practice can be viewed as 'noisy' in the sense that there are variations in the data that are best described as realizations of random variables. Using statistical methods allows us to analyze the properties of the models that are learned from noisy data and to reason about the inherent *uncertainties* in these models and their predictions.

To be able to work with statistical machine learning models we need some basic concepts from statistics and probability theory. Hence, before we embark on the statistical machine learning journey in the next chapter we present some background material on these topics in this chapter. Furthermore, we discuss some of the notation used in these lecture notes and point out some notational differences from the text book James et al. 2013.

1.1. Random variables

A random variable Z is a variable that can take any value z on a certain set Z and its value depends on the outcome of a random event. For example, if Z describes the outcome of rolling a die, the possible outcomes are $\mathsf{Z} = \{1, 2, 3, 4, 5, 6\}$ and the probability of each possible outcome of a die roll is typically modeled to be $1/6$. We write $\Pr(Z = z) = 1/6$ for $z = 1, \ldots, 6$.

In these lecture notes we will primarily consider random variables where Z is continuous, for example $\mathsf{Z} = \mathbb{R}$ (Z is a scalar) or $\mathsf{Z} = \mathbb{R}^d$ (Z is a d-vector). Since there are infinitely many possible outcomes $z \in \mathsf{Z}$, we cannot speak of the probability of an outcome z—it is almost always zero! Instead, we use the *probability density function*, denoted by $p(z)$.

Remark 1.1. In this document we will use the symbol $p(\cdot)$ as a general probability density function, and we will let its argument indicate what the underlying random variable is. For instance, when writing $p(z)$ it is implicit that this is the probability density function for Z, $p(y)$ is the probability density function for Y, etc. Furthermore, we will use the word 'distribution' somewhat sloppily, also when referring to a probability density function.

The probability density function $p : \mathsf{Z} \mapsto \mathbb{R}^+$ describes the probability of Z to be within a certain set $\mathsf{C} \subseteq \mathsf{Z}$

$$\Pr(Z \in \mathsf{C}) = \int_{z \in \mathsf{C}} p(z) \mathrm{d}z. \tag{1.1}$$

For example, if Z is a random variable with the probability density function $p(z)$ describing the predicted temperature tomorrow, the chance for this temperature to be between 15° and 20° is $\Pr(15 < Z < 20) = \int_{15}^{20} p(z)\mathrm{d}z$. Note that $p(z)$ is not (in general) upper bounded by 1, however it holds that it integrates to 1: $\int_{z \in \mathsf{Z}} p(z) = 1$. For notational convenience, when the integration is over the whole domain Z we simply write $\int$ instead of $\int_{z \in \mathsf{Z}}$, for instance $\int p(z)\mathrm{d}z = 1$.

A common probability distribution is the *Gaussian* (or *Normal*) distribution, whose density is defined as

$$p(z) = \mathcal{N}(z \,|\, \mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(z-\mu)^2}{2\sigma^2}\right), \tag{1.2}$$

where we have made use of $\exp$ to denote the exponential function; $\exp(x) = e^x$. We also use the notation $Z \sim \mathcal{N}(\mu, \sigma^2)$ to say that Z has a Gaussian distribution with parameters μ and σ^2 (i.e., its probability density function is given by (1.2)). The symbol $\sim$ reads 'distributed according to'.

The *expected value* or *mean* of the random variable Z is given by

$$\mathbb{E}[Z] = \int zp(z)\mathrm{d}z. \tag{1.3}$$

We can also compute the expected value of some arbitrary function $g(z)$ applied to Z as

$$\mathbb{E}[g(Z)] = \int g(z)p(z)\mathrm{d}z. \tag{1.4}$$

For a scalar random variable with mean $\mu = \mathbb{E}[Z]$ the *variance* is defined as

$$\mathrm{Var}[Z] = \mathbb{E}[(Z-\mu)^2] = \mathbb{E}[Z^2] - \mu^2. \tag{1.5}$$

The variance measures the 'spread' of the distribution, i.e. how far a set of random number drawn from the distribution are spread out from their mean. The variance is always non-negative. For the Gaussian distribution (1.2) the mean and variance are given by the parameters μ and σ^2 respectively.

Now, consider two random variables Z_1 and Z_2 (both of which could be vectors). If we are interested in computing the probability that $Z_1 \in \mathsf{C}_1$ *and* $Z_2 \in \mathsf{C}_2$ we need their *joint* probability density function $p(z_1, z_2)$. Using this joint distribution we can compute the probability analogously to the previous case according to

$$\Pr(Z_1 \in \mathsf{C}_1,\ Z_2 \in \mathsf{C}_2) = \int_{z_1 \in \mathsf{C}_1,\ z_2 \in \mathsf{C}_2} p(z_1, z_2)\mathrm{d}z_1\mathrm{d}z_2. \tag{1.6}$$

An important property of pairs of random variables is that of *independence*. The variables Z_1 and Z_2 are said to be independent if the joint probability density function factorizes according to $p(z_1, z_2) = p(z_1)p(z_2)$. It follows from (1.6) that the probability factorizes in a similar way: $\Pr(Z_1 \in \mathsf{C}_1,\ Z_2 \in \mathsf{C}_2) = \Pr(Z_1 \in \mathsf{C}_1)\Pr(Z_2 \in \mathsf{C}_2)$. Furthermore, for independent random variables the expected value of any separable function factorizes as $\mathbb{E}[g_1(Z_1)g_2(Z_2)] = \mathbb{E}[g_1(Z_1)]\mathbb{E}[g_2(Z_2)]$.

From the joint probability density function we can deduce both its two marginal densities $p(z_1)$ and $p(z_2)$ using *marginalization*, as well as the so called conditional probability density function $p(z_2 \mid z_1)$ using *conditioning*. These two concepts will be explained below.

1.1.1. Marginalization

Consider a multivariate random variable Z which is composed of two components Z_1 and Z_2, which could be either scalars or vectors, as $Z = [Z_1^\mathsf{T},\ Z_2^\mathsf{T}]^\mathsf{T}$. If we know the (joint) probability density function $p(z) = p(z_1, z_2)$, but are interested only in the *marginal* distribution for z_1, we can obtain the density $p(z_1)$ by *marginalization*

$$p(z_1) = \int_{z_2 \in \mathsf{Z}_2} p(z_1, z_2)\mathrm{d}z_2 \tag{1.7}$$

where Z_2 is the space on which Z_2 is defined. The other marginal $p(z_2)$ is obtained analogously by integrating over z_1 instead. In Figure 1.1 a joint two-dimensional density $p(z_1, z_2)$ is illustrated along with their marginal densities $p(z_1)$ and $p(z_2)$.

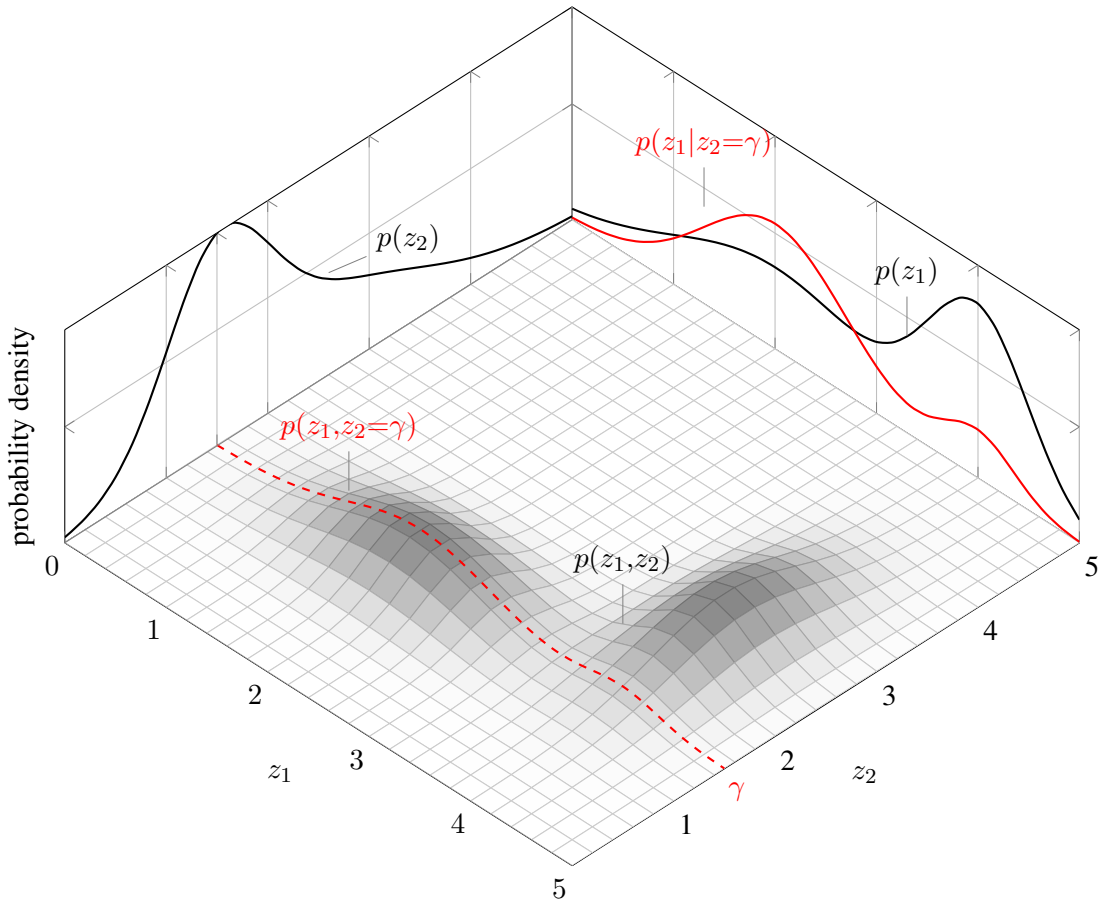


Figure 1.1.: Illustration of a two-dimensional *joint* probability distribution $p(z_1, z_2)$ (the surface) and its two *marginal* distributions $p(z_1)$ and $p(z_2)$ (the black lines). We also illustrate the *conditional* distribution $p(z_1|z_2 = \gamma)$ (the red line), which is the distribution of the random variable Z_1 conditioned on the observation $z_2 = \gamma$ ($\gamma = 1.5$ in the plot).

1.1.2. Conditioning

Consider again the multivariate random variable Z which can be partitioned in two parts $Z = [Z_1^\mathsf{T},\ Z_2^\mathsf{T}]^\mathsf{T}$. We can now define the *conditional* distribution of Z_1, conditioned on having observed a value $Z_2 = z_2$, as

$$p(z_1 \,|\, z_2) = \frac{p(z_1, z_2)}{p(z_2)}. \tag{1.8}$$

If we instead have observed a value of $Z_1 = z_1$ and want to use that to find the conditional distribution of Z_2 given $Z_1 = z_1$, it can be done analogously. In Figure 1.1 a joint two-dimensional probability density function $p(z_1, z_2)$ is illustrated along with a conditional probability density function $p(z_1 \,|\, z_2)$.

From (1.8) it follows that the joint probability density function $p(z_1, z_2)$ can be factorized into the product of a marginal times a conditional,

$$p(z_1, z_2) = p(z_2 \,|\, z_1)p(z_1) = p(z_1 \,|\, z_2)p(z_2). \tag{1.9}$$

If we use this factorization for the denominator of the right-hand-side in (1.8) we end up with the relationship

$$p(z_1 \,|\, z_2) = \frac{p(z_2 \,|\, z_1)p(z_1)}{p(z_2)}. \tag{1.10}$$

This equation is often referred to as *Bayes' rule*.

1.2. A note on notation

We end this chapter by defining some notation that we will use throughout these lecture notes. First, some general mathematical notation:

- $\log(\cdot)$ refers to the natural logarithm (base e) and it is the inverse of the exponential function $\exp(\cdot)$.
- $I(\cdot)$ is the indicator function, which is either zero or one depending on whether its argument is true or false. For instance,

$$I(x < a) = \begin{cases} 1 & \text{if } x < a, \\ 0 & \text{if } x \geq a. \end{cases}$$

- $\text{sign}(\cdot)$ is the signum function,

$$\text{sign}(x) = \begin{cases} -1 & \text{if } x < 0, \\ 0 & \text{if } x = 0, \\ 1 & \text{if } x > 0. \end{cases}$$

- $\min$ is the minimum operator which returns the smallest value attained by some function. For instance, $\min_x (x+5)^2 = 0$.
- $\arg\min$ is the mathematical operator which return the *argument* of the minimum. For instance, $\arg\min_x (x+5)^2 = -5$.
- $\max$ and $\arg\max$ are defined analogously as $\min$ and $\arg\min$, but with maximum instead of minimum.
- For a function $f : \mathbb{R}^d \mapsto \mathbb{R}$, $\nabla_x f(x)$ denotes the gradient, i.e. the vector of all partial derivatives, $\nabla_x f(x) = \left[\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_d}\right]^\mathsf{T}$.
- If $v = [v_1, \ldots, v_d]^\mathsf{T}$ is a vector of dimension d, its L^p-norm is defined as $\|v\|_p = \left(\sum_{i=1}^d |v_i|^p\right)^{1/p}$ for any $p \geq 1$. For $p = 2$ we get the usual Euclidian norm and for $p = 1$ we get the Manhattan norm.
- A^T denotes the transpose of matrix A.
- vec is the vectorization operator which stacks the columns of a matrix in a vector. For instance,

$$\text{vec}\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}^\mathsf{T}.$$

In addition to the aforementioned definitions, we also use some notational conventions which are specific to the topic under study. These conventions largely follow those used in James et al. 2013.

Upper case letters, specifically Y and X, are used when talking about 'conceptual' or 'generic' model variables. For instance, a conceptual regression model is $Y = f(X) + \varepsilon$. When talking about specific/observed values of these variables we use lower case letters. For instance, the training data points are denoted by (x_i, y_i). This is similar to the common notation that upper case is used for random variables and lower case for their realizations, but we generalize the convention to cover conceptual model variables even if they are not assumed to be random.

We use n to denote the number of training data points and the training data set is written as $\mathcal{T} = \{(x_i, y_i)\}_{i=1}^n$. We use bold face for vectors and matrices which are formed by stacking all the training data points in some way. For instance, the vector containing all training outputs is written as

$$\mathbf{y} = [y_1, \ldots, y_n]^\mathsf{T}. \tag{1.11}$$

This notation is extensively used in chapter 2.

Finally, we point out a few notational caveats that could cause confusion. A few symbols have double meanings.

- We use K to denote the number of classes in multi-class classification problems, but K is also used by James et al. 2013, Section 2.2.3 to denote the number of neighbors in the K-nearest neighbors method.
- We use p to denote the number of inputs, i.e., the dimension of the input vector X, but also as a generic probability density function as described in Section 1.1.

Furthermore, James et al. 2013 use the symbol p to denote conditional class probabilities when addressing classification problems. To avoid confusion with probability density functions we instead use the symbol q to denote these conditional class probabilities. Specifically, for binary classification problems where $Y \in \{0, 1\}$,

$$\Pr(Y = 1 \mid X = x) = \begin{cases} q(x) & \text{in these lecture notes,} \\ p(x) & \text{in James et al. 2013.} \end{cases}$$

For multi-class classification problems where $Y \in \{1, \ldots, K\}$,

$$\Pr(Y = k \mid X = x) = \begin{cases} q_k(x) & \text{in these lecture notes,} \\ p_k(x) & \text{in James et al. 2013.} \end{cases}$$

The notation $q(\cdot)$ and $q_k(\cdot)$ is used for logistic regression models in chapter 3 as well as for deep learning models in chapter 4.

2. Linear regression

Regression refers to the general statistical problem of estimating the relationships between some (qualiative or quantitative[1]) *input variables* $X = [X_1, \ldots, X_p]^\mathsf{T}$ and a quantitative *output variable* Y. Some common synonyms to input variable are predictor, regressor, feature, explanatory variable, controlled variable, independent variable and covariate. Synonyms to output variables include response, regressand, label, explained variable, predicted variable and dependent variable. Regression, in general, is about learning a model f

$$Y = f(X) + \epsilon, \tag{2.1}$$

where ϵ is some noise/error which describes everything that cannot be captured by the model. Specifically, we view ϵ as a random variable that is independent of X and has mean zero.

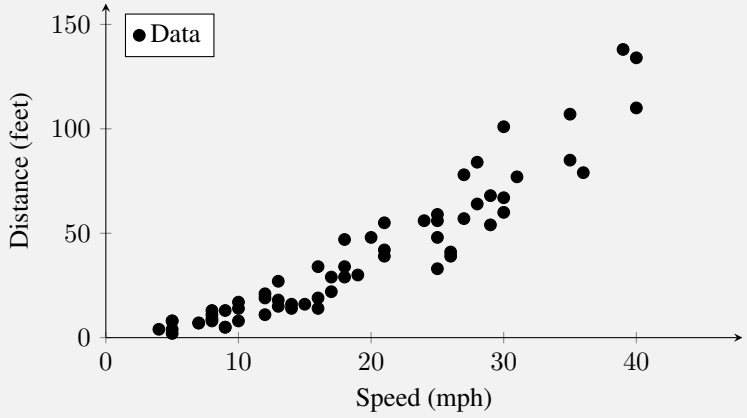
Linear regression, which is our first approach to regression, has a linear (or affine) combination of the input variables X as its model f. Even though it is a relatively simple model, it is still surprisingly useful on its own. Furthermore, it constitutes an important building block in more advanced methods such as deep learning (chapter 4). Throughout this chapter we will use Example 2.1 with car stopping distances to illustrate the ideas.

Example 2.1: Car stopping distances

Ezekiel and Fox (1959) presents a data set with 62 observations of how long distance that is needed for various cars at different initial speeds to break to a complete stop.[a] The data set has the two following variables:

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

We decide to interpret `Speed` as the **input variable** X, and `Distance` as the **output variable** Y.



Our goal is to use linear regression to estimate (that is, to *predict*) how long the stopping distance would be if the initial speed would be 33 mph or 45 mph (two speeds at which no data has been recorded).

[a] The data set is somewhat dated, so the conclusions are perhaps not applicable to modern cars. We hope, however, that it does not affect the pedagogical purpose of this example, and we believe that the reader is capable of pretending that the data comes from her/his own favorite example instead.

[1] We will start with quantiative input variables, and discuss qualitative input variables later in 2.4.

2.1. The linear regression model

The linear regression model describes the output variable Y (a scalar) as an affine combination of the input variables $X_1, X_2, \ldots, X_p$ (each a scalar) plus a noise term ϵ,

$$Y = \underbrace{\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p}_{f(X;\beta)} + \epsilon. \tag{2.2}$$

We refer to the coefficients $\beta_0, \beta_1, \ldots \beta_p$ as the *parameters* in the model, and we sometimes refer to β_0 specifically as the intercept term. The noise term ϵ accounts non-systematic, i.e. random, errors between the data and the model. The noise is assumed to have mean zero and to be independent of X. The main part of this chapter will be devoted to how to *learn* the model—that is, to learn the parameters $\beta_0, \beta_1, \ldots, \beta_p$—from some training data set $\mathcal{T} = \{(x_i, y_i)\}_{i=1}^n$. Before we dig into the details in Section 2.2, however, let us just briefly start by discussing the purpose of using linear regression. The linear regression model can namely be used for, at least, two different purposes: to *understand* relationships in the data by interpreting the parameters β, and to *predict* future outputs for inputs that we have not yet seen.

Remark 2.1. It is possible to formulate the model also for multiple outputs $Y_1, Y_2, \ldots$, see exercises 1.1d and 1.5. This is commonly referred to as multivariate linear regression.

2.1.1. Understanding relationships

An often posed question in sciences such as medicine, sociology etc., is to determine whether there is a correlation between some variables or not ('do you live longer if you only eat sea food?', etc.). Such questions can be addressed by studying the parameters β in the linear regression model. The most common question is perhaps whether it can be indicated that *some* correlation is present between two variables X_1 and Y, which can be done with the following reasoning: If $\beta_1 = 0$, it would indicate that there is no correlation between Y and X_1 (unless the other inputs also depend on X_1). By estimating β_1 together with a confidence interval (describing uncertainty of the estimate), one can rule out (with a certain significance level) that X_1 and Y are uncorrelated if 0 is not in the confidence interval for β_1. The conclusion is then instead that *some* correlation is likely to be present between X_1 and Y. This type of reasoning is referred to as *hypothesis testing* and it constitutes a important branch of classical statistics. A more elaborate treatment of hypothesis testing in linear regression models can be found in James et al. 2013, Chapter 3. However, we shall mainly be concerned with another application of the linear regression model, namely to make predictions.

2.1.2. Predicting future outputs

In machine learning, the emphasis is rather on predicting some (not yet seen) output $\widehat{y}_\star$ for some new input $x_\star$. To make a prediction for a test input $x_\star$, we insert $x_\star$ into the model (2.2). Since we (by assumption) cannot predict ϵ, we take our *prediction* $\widehat{y}_\star$ without ϵ as

$$\widehat{y}_\star = \beta_0 + \beta_1 x_{\star 1} + \beta_2 x_{\star 2} + \cdots + \beta_p x_{\star p}. \tag{2.3}$$

We use the symbol $\widehat{\ }$ on y to indicate that it is a prediction, containing no ϵ. (If we were able to somehow observe the actual output from $x_\star$, we would denote it $y_\star$.)

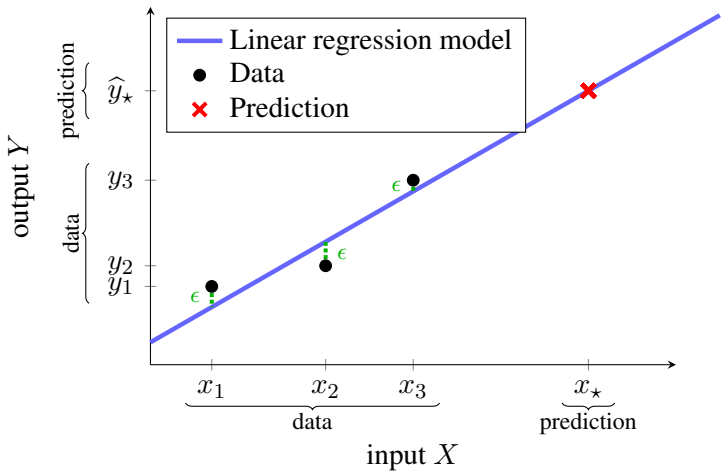


Figure 2.1.: Linear regression with $p = 1$: The black dots represent $n = 3$ data points, from which a linear regression model (blue line) is learned. The model, however, cannot fit the data exactly, but there is an error/noise ϵ (green) left. The model can be used to *predict* (red cross) the output $\widehat{y}_\star$ for a test input point $x_\star$.

2.2. Learning the model from training data

To use the linear regression model, we first need to learn the unknown parameters $\beta_0, \beta_1, \ldots, \beta_p$ from a training data set $\mathcal{T}$. The training data consists of n samples of the output variable Y, we call them y_i ($i = 1, \ldots, n$), and the corresponding n samples x_i ($i = 1, \ldots, n$) (each a column vector) of the input variable $X = [X_1\ X_2\ \cdots\ X_p]^\mathsf{T}$. We write the data set on the matrix form

$$\mathbf{X} = \begin{bmatrix} 1 & -x_1^\mathsf{T}- \\ 1 & -x_2^\mathsf{T}- \\ \vdots & \vdots \\ 1 & -x_n^\mathsf{T}- \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, \quad \text{where each } x_i = \begin{bmatrix} x_{i1} \\ x_{i2} \\ \vdots \\ x_{ip} \end{bmatrix}. \tag{2.4}$$

Note that $\mathbf{X}$ is a $n \times (p+1)$ matrix, and $\mathbf{y}$ a $n \times 1$ matrix. The first column of $\mathbf{X}$, with only ones, correspond to the intercept term β_0 in the linear regression model (2.2). If we also stack the unknown parameters $\beta_0, \beta_1, \ldots, \beta_p$ into a $(p+1)$ vector

$$\beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_p \end{bmatrix}, \tag{2.5}$$

we can express the linear regression model as a matrix multiplication

$$\mathbf{y} = \mathbf{X}\beta + \boldsymbol{\epsilon}, \tag{2.6}$$

where $\boldsymbol{\epsilon}$ is a vector of errors/noise.

Learning the unknown parameters β amounts to finding values such that *the model fits the data well*. There are multiple ways to define what 'well' actually means. We will take a statistical perspective and choose the value of β which makes the observed training data $\mathbf{y}$ as likely as possible under the model—the so-called *maximum likelihood* solution.

Example 2.2: Car stopping distances

We will continue Example 2.1, and form 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,

$$\mathbf{X} = \begin{bmatrix} 1 & 4 \\ 1 & 5 \\ 1 & 5 \\ 1 & 5 \\ 1 & 5 \\ 1 & 7 \\ 1 & 7 \\ 1 & 8 \\ \vdots & \vdots \\ 1 & 39 \\ 1 & 39 \\ 1 & 40 \end{bmatrix}, \qquad \beta = \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix}, \qquad \mathbf{y} = \begin{bmatrix} 4 \\ 2 \\ 4 \\ 8 \\ 8 \\ 7 \\ 7 \\ 8 \\ \vdots \\ 138 \\ 110 \\ 134 \end{bmatrix}. \tag{2.7}$$

2.2.1. Maximum likelihood

Our strategy to learn the unknown parameters β from the training data $\mathcal{T}$ will be the *maximum likelihood* method. The word 'likelihood' refers to the statistical concept of the likelihood function, and maximizing the likelihood function amounts to finding the value of β that makes $\mathbf{y}$ as likely as possible to have been observed. That is, we want to solve

$$\underset{\beta}{\text{maximize}} \;\; p(\mathbf{y} \mid \mathbf{X}, \beta), \tag{2.8}$$

where $p(\mathbf{y} \mid \mathbf{X}, \beta)$ is the probability density of the data $\mathbf{y}$ given a certain value of the parameters β. We denote the solution to this problem—the learned parameters—with $\widehat{\beta} = [\widehat{\beta}_0 \; \widehat{\beta}_1 \; \cdots \; \widehat{\beta}_p]^\mathsf{T}$. More compactly, we write this as

$$\widehat{\beta} = \arg\max_{\beta} p(\mathbf{y} \mid \mathbf{X}, \beta). \tag{2.9}$$

In order to have a notion of what 'likely' means, and thereby specify $p(\mathbf{y} \mid \mathbf{X}, \beta)$ mathematically, we need to make assumptions about the noise term ϵ. A common assumption is that ϵ follows a Gaussian distribution with zero mean and variance σ_ϵ^2,

$$\epsilon \sim \mathcal{N}(0, \sigma_\epsilon^2). \tag{2.10}$$

This implies that the conditional probability density function of the output Y for a given value of the input $X = x$ is given by

$$p(y \mid x, \beta) = \mathcal{N}(y \mid \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p, \sigma_\epsilon^2). \tag{2.11}$$

Furthermore, the n observed training data points are assumed to be *independent* realizations from this statistical model. This implies that the likelihood of the training data factorizes as

$$p(\mathbf{y} \mid \mathbf{X}, \beta) = \prod_{i=1}^{n} p(y_i \mid x_i, \beta). \tag{2.12}$$

Putting (2.11) and (2.12) together we get,

$$p(\mathbf{y} \mid \mathbf{X}, \beta) = \frac{1}{(2\pi\sigma_\epsilon^2)^{n/2}} \exp\left(-\frac{1}{2\sigma_\epsilon^2} \sum_{i=1}^{n} (\beta_0 + \beta_1 x_{i1} + \cdots + \beta_p x_{ip} - y_i)^2 \right). \tag{2.13}$$

Recall from (2.8) that we want to maximize the likelihood w.r.t. β. However, since (2.13) only depends on β via the sum in the exponent, and since the exponential is an increasing function, maximizing (2.13) is equivalent to minimizing

$$\sum_{i=1}^{n} \left(\beta_0 + \beta_1 x_{i1} + \cdots + \beta_p x_{ip} - y_i\right)^2. \tag{2.14}$$

This is the sum of the squares of differences between each output data y_i and the model's prediction of that output, $\widehat{\beta}_i = \beta_0 + \beta_1 x_{i1} + \cdots + \beta_p x_{ip}$. For this reason, minimizing (2.14) is usually referred to as *least squares*.

We will come back on how the values $\widehat{\beta}_0, \widehat{\beta}_1, \ldots, \widehat{\beta}_p$ can be computed. Let us just first mention that it is also possible—and sometimes a very good idea—to assume that ϵ is distributed as something else than a Gaussian distribution. One can, for instance, assume that ϵ instead has a Laplace distribution, which instead would yield the cost function

$$\sum_{i=1}^{n} |\beta_0 + \beta_1 x_{i1} + \ldots \beta_p x_{ip} - y_i|, \tag{2.15}$$

to minimize. It contains the sum of the absolute values of all differences (rather than their squares). The major benefit with the Gaussian assumption (2.10) is that there is a closed-form solution available for $\widehat{\beta}_0, \widehat{\beta}_1, \ldots, \widehat{\beta}_p$, whereas other assumptions on ϵ usually require more computationally expensive methods.

Remark 2.2. With the terminoloy we will introduce in the next chapter, we could refer to (2.13) as the likelihood function, and use the notation $\ell(\beta)$ for it.

Remark 2.3. It is not uncommon in the literature to skip the maximum likelihood motivation, and just state (2.14) as a (somewhat arbitrary) cost function for optimization.

2.2.2. Least squares and the normal equations

By assuming that the noise/error ϵ has a Gaussian distribution (2.10), the maximum likelihood parameters $\widehat{\beta}$ are the solution to the optimization problem (2.14). We illustrate this by Figure 2.2, and write the least squares problem using the compact matrix and vector notation (2.6) as

$$\underset{\beta_0,\beta_1,\ldots,\beta_p}{\text{minimize}} \ \|\mathbf{X}\beta - \mathbf{y}\|_2^2, \tag{2.16}$$

where $\|\cdot\|_2$ denotes the usual Euclidean vector norm, and $\|\cdot\|_2^2$ its square. From a linear algebra point of view, this can be seen as the problem of finding the closest (in an Euclidean sense) vector to $\mathbf{y}$ 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 $\widehat{\beta}$ can be shown (appendix A) to be

$$\widehat{\beta} = (\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}\mathbf{X}^\mathsf{T}\mathbf{y}. \tag{2.17}$$

Equation (2.17) is often referred to as the *normal equation*, and it is the solution to the least squares problem (2.14, 2.16). The fact that this closed-form solution exists is important, and is perhaps the reason for why least squares has become very popular and widely used. As discussed, other assumptions on ϵ than Gaussian leads to other problems than least squares, such as (2.15) (where no closed-form solution exists).

Remark 2.4. Note that if the columns of $\mathbf{X}$ are linearly independent and $p = n - 1$, $\mathbf{X}$ spans the entire $\mathbb{R}^n$, and a unique solution exists such that $\mathbf{y} = \mathbf{X}\beta$ exactly, i.e., the model fits the training data perfectly. In this situation (2.17) reduces to $\beta = \mathbf{X}^{-1}\mathbf{y}$. It may appear as a desireable behavior that the model fits the data perfectly, but it is often tightly connected to the problem of *overfitting*, as we will discuss in Section 2.5.

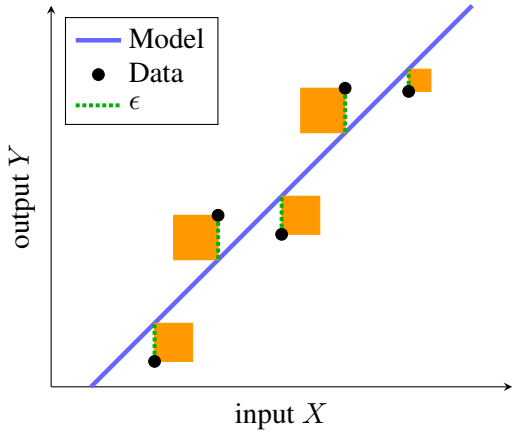
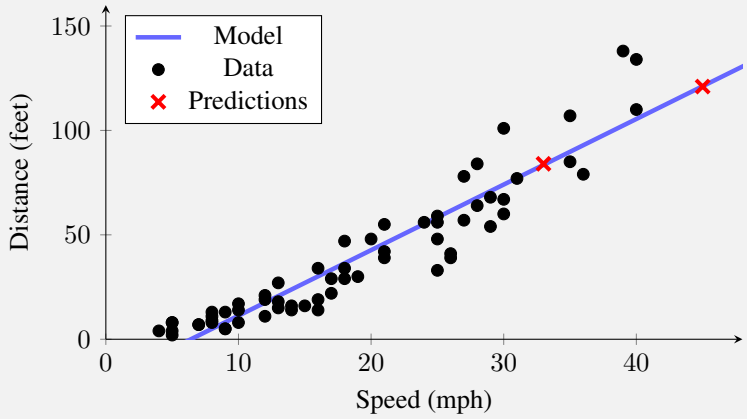


Figure 2.2.: A graphical explanation of the least squares criterion: the goal is to choose the model (blue line) such that the sum of the square (orange) of each error ϵ (green) is minimized. That is, the blue line is to be chosen so that the amount of orange color is minimized. This motivates the name *least squares*.

Example 2.3: Car stopping distances

By inserting the matrices (2.7) from Example 2.2 into the normal equations (2.6), we obtain $\widehat{\beta}_0 = -20.1$ and $\widehat{\beta}_1 = 3.1$. If we plot the resulting model, it looks like this:



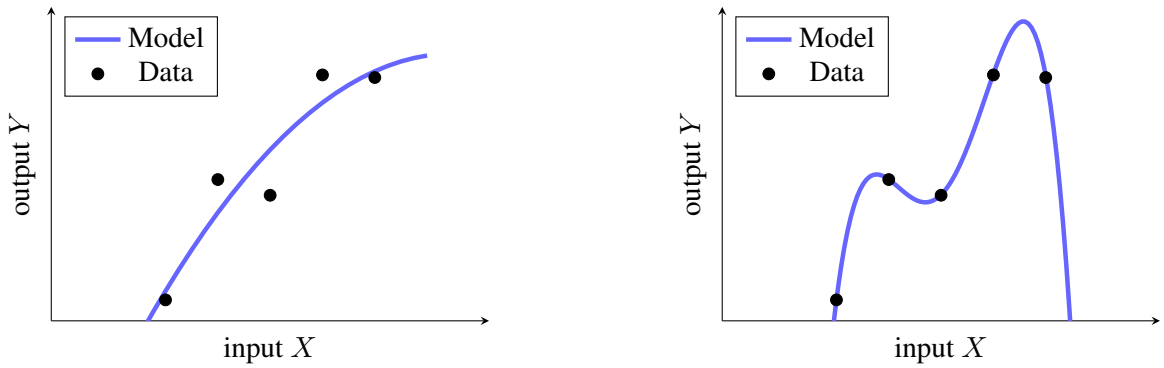
With this model, the predicted stopping distance for $x_\star = 33$ mph is $\widehat{y}_\star = 84$ feet, and for $x_\star = 45$ mph it is $\widehat{y}_\star = 121$ feet.

2.3. Nonlinear transformations of the inputs

The reason for the word *linear* in the name 'linear regression' is that the output is modelled as a linear combination of the inputs.[2] We have, however, not made a clear definition of what an input is: if the speed is an input, then why could not also the kinetic energy—it's square—be considered as another input? The answer is that it can. We can in fact make use of arbitrary *nonlinear* transformations of the "original" input variables as inputs in the linear regression model. If we, for example, only have a one-dimensional input X, a simple linear regression model is

$$Y = \beta_0 + \beta_1 X + \epsilon. \tag{2.18}$$

[2] And also the constant 1, corresponding to the offset β_0, which is why the name affine is sometimes used rather than linear.



(a) The maximum likelihood solution with a 2nd order polynomial in the linear regression model. As discussed, the line is no longer straight (cf Figure 2.1). This is, however, merely an artefact of the plot: in a three-dimensional plot with each 'new' input (here, X and X^2) on the axes, it would still be an affine set.

(b) The maximum likelihood solution with a 4th order polynomial in the linear regression model. Note that a 4th order polynomial contains 5 unknown coefficients, which roughly means that we can expect the learned model to fit 5 data ponts exactly (cf. Remark 2.4, $p = n - 1$).

Figure 2.3.: A linear regression model with 2nd and 4th order polynomials in the input X, as (2.19).

However, we can also extend the model with, for instance, $X^2, X^3, \ldots, X^p$ as inputs, and thus obtain a linear regression model which is a polynomial in X,

$$Y = \beta_0 + \beta_1 X + \beta_2 X^2 + \cdots + \beta_p X^p + \epsilon. \tag{2.19}$$

Note that this is still a linear regression model *since the unknown parameters appear in a linear fashion* but with $X, X^2, \ldots, X^p$ as inputs. The parameters $\widehat{\beta}$ are still learned with the same methods, but the $\mathbf{X}$ matrix is different for model (2.18) and (2.19).

Figure 2.3 shows an example of two linear regression model with transformed (polynomial) inputs. When studying the figure one may ask how a *linear* regression model can result in a *curved* line? Are not a linear regression models restricted to linear (or affine) straight lines? The answer is that it depends on the plot: What is shown in Figure 2.3(a) is the two-dimensional plot with X, Y (the 'original' input), but a three-dimensional plot with X, X^2, Y (all transformed inputs) would still be an affine set. The same holds true also for Figure 2.3(b) but in that case we would need a 5-dimensional plot.

Even though the model in Figure 2.3(b) is able to fit all data points exactly, it also suggests that higher order polynomials might not always be very useful: the behavior of the model in-between and outside the data points is rather peculiar, and not very well motivated by the data. High-order polynomials are for this reason rarely used in practice in machine learning. An alternative and much more common nonlinear transformation is the so-called radial basis function (RBF) kernel

$$K_c(x) = \exp\left(-\frac{\|x - c\|_2^2}{\ell}\right), \tag{2.20}$$

i.e., a Gauss bell centered around c. It can be used, instead of polynomials, in the linear regression model as

$$Y = \beta_0 + \beta_1 K_{c_1}(X) + \beta_2 K_{c_2}(X) + \cdots + \beta_p K_{c_p}(X) + \epsilon. \tag{2.21}$$

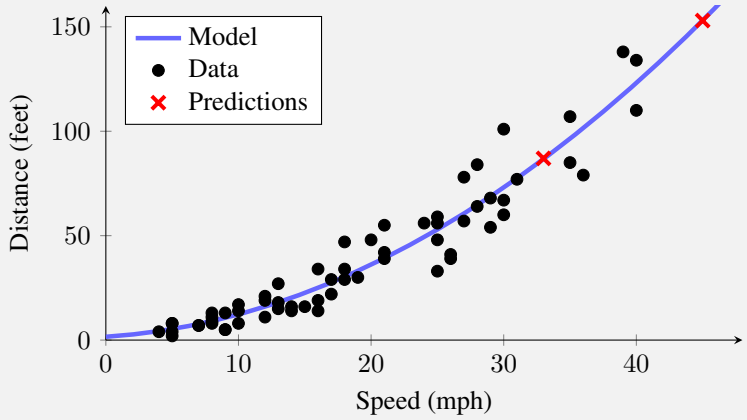
This model is can be seen as p 'bumps' located at $c_1, c_2, \ldots, c_p$, respectively. Note that the locations $c_1, c_2, \ldots, c_p$ as well as the length scale ℓ have to be decided by the user, and it is only the parameters $\beta_0, \beta_2, \ldots, \beta_p$ which are learned from data in linear regression. This is illustrated in Figure 2.4. RBF kernels are in general preferred over polynomials since they have 'local' properties, meaning that a small change in one parameter mostly affects the model only locally around that kernel, whereas a small change in one parameter in a polynomial model affects the model everywhere.

Example 2.4: Car stopping distances

We continue with Example 2.1, but this time we also consider the squared speed as an input, i.e., the inputs are now X and X^2. This gives the new matrices (cf. (2.7))

$$\mathbf{X} = \begin{bmatrix} 1 & 4 & 16 \\ 1 & 5 & 25 \\ 1 & 5 & 25 \\ \vdots & \vdots & \vdots \\ 1 & 39 & 1521 \\ 1 & 40 & 1600 \end{bmatrix}, \qquad \beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{bmatrix}, \qquad \mathbf{y} = \begin{bmatrix} 4 \\ 2 \\ 4 \\ \vdots \\ 110 \\ 134 \end{bmatrix}, \tag{2.22}$$

and when we insert them into the normal equations (2.17), the new parameter estimates are $\widehat{\beta}_0 = 1.58$, $\widehat{\beta}_1 = 0.42$ and $\widehat{\beta}_2 = 0.07$. (Note that also the value of $\widehat{\beta}_0$ and $\widehat{\beta}_1$ has changed, compared to the Example 2.3.) This new model looks as



With this model, the predicted stopping distance is now $\widehat{y}_\star = 87$ feet for $x_\star = 33$ mph, and $\widehat{y}_\star = 153$ for $x_\star = 45$ mph. This can be compared to Example 2.3, which gives different predictions. Based on the data alone we can not say that this is the "true model", but by visually comparing this model with Example 2.3, the extended model seems to follow the data slightly better. A systematic method to choose between different models (other than just visually comparing plots) is cross-validation, which is covered by James et al. (2013, Section 5.1).

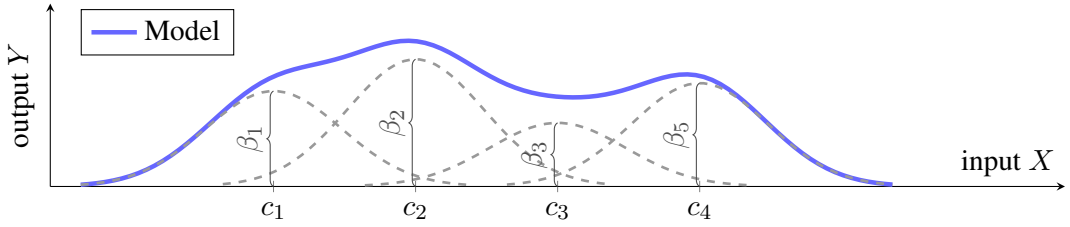


Figure 2.4.: A linear regression model with RBF kernels (2.21). Each kernel (dashed gray lines) is located at c_1, c_2, c_3 and c_4, respectively. When the model is learned from data, the parameters $\beta_0, \beta_1, \ldots, \beta_p$ are chosen such that the sum of all kernels (solid blue line) is fitted to the data in, e.g., a least square sense.

Polynomials and RBF kernels are just two particular special cases, we can of course consider any nonlinear transformation of the inputs.To distinguish the 'original' inputs from the 'new' transformed inputs, the term *features* is often used for the latter. To decide which features to use one approach is to compare competing models (with different features) using cross-validation; see James et al. 2013, Section 5.1.

2.4. Qualitative input variables

The regression problem is characterized by a quantitative output[3] Y, but the nature of the inputs X is arbitrary. We have so far only discussed the case of quantitative inputs X, but qualitative inputs are perfectly possible as well. Before we discuss how to handle qualitative input variables, let us have a look at when a variable in general is to be considered as quantiative or qualitative, respectively:

Variable type	Example	Handle as
Numeric and continuous-valued	32.23 km/h, 12.50 km/h, 42.85 km/h	Quantitative
Numeric, discrete-valued but has a natural ordering	0 children, 1 child, 2 children	Quantitative
Numeric, discrete-valued but lacks a natural ordering	1 = Sweden, 2 = Denmark, 3 = Norway	Qualitative
Non-numeric	Uppsala University, Stockholm University, Lund University	Qualitative

The distinction is, however, somewhat arbitrary, and there is not always a clear answer: one could for instance argue that having no children is something qualitatively different than having children, and use the qualitative variable "children: yes/no", instead of "0, 1 or 2 children". In a similar fashion continuous variables can be thresholded into bins (positive/negative, e.g.), and thereby be transformed into qualitative ones. In the end, it is a design choice which variables are considered as qualitative and quantitative, respectively.

Assume that we have a qualitative input variable that only takes two different values (or levels, classes), which we call type A and type B. We can then create a *dummy variable* X as

$$X = \begin{cases} 0 & \text{if type A} \\ 1 & \text{if type B} \end{cases} \tag{2.23}$$

and use this variable in the linear regression model. This effectively gives us a linear regression model which looks like

$$Y = \beta_0 + \beta_1 X + \epsilon = \begin{cases} \beta_0 + \epsilon & \text{if type A} \\ \beta_0 + \beta_1 + \epsilon & \text{if type B} \end{cases} \tag{2.24}$$

The choice is somewhat arbitrary, and type A and B can of course be switched. Other choices, such as $X = 1$ or -1, are also possible. This approach can be generalized to qualitative input variables which take more than two values, let us say type A, B, C and D. With four different values, we create four minus one dummy variables as

$$X_1 = \begin{cases} 1 & \text{if type B} \\ 0 & \text{if not type B} \end{cases}, \quad X_2 = \begin{cases} 1 & \text{if type C} \\ 0 & \text{if not type C} \end{cases}, \quad X_3 = \begin{cases} 1 & \text{if type D} \\ 0 & \text{if not type D} \end{cases} \tag{2.25}$$

which, altogether, gives the linear regression model

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_3 + \epsilon = \begin{cases} \beta_0 + \epsilon & \text{if type A} \\ \beta_0 + \beta_1 + \epsilon & \text{if type B} \\ \beta_0 + \beta_2 + \epsilon & \text{if type C} \\ \beta_0 + \beta_3 + \epsilon & \text{if type D} \end{cases} \tag{2.26}$$

Qualitative inputs can be handled similarly in other problem and models as well, such as logistic regression, LDA, kNN, deep learning, etc.

[3] If the output variable is qualitative, then we have a classification—and not a regression—problem.

2.5. Regularization

Even though the linear regression model at a first glance (cf. Figure 2.1) may seem as a fairly rigid and non-flexible model, it is not necessarily so. If the inputs are extended with nonlinear transformations as in Figure 2.3 or 2.4, or the number of inputs p is large and the number of data points n is small, one may experience *overfitting*. If considering data as consisting of 'signal' (the actual information) and 'noise' (measurement errors, irrelevant effects, etc), the term *overfitting* indicates that the model is fitted not only to the 'signal' but also to the 'noise'. An example of overfitting is given in Example 2.5, where a linear regression model with $p = 8$ RBF kernels is learned from $n = 9$ data points. Even though the model follows all data points very well, we can intuitively judge that the model is nor particularly useful: neither the interpolation (between the data points) nor the extrapolation (outside the data range) appears sensible. Note that using $p = n - 1$ is an extreme case, but the conceptual problem with overfitting is often present also in less extreme situations.

A useful approach to handle overfitting is *regularization*. The idea behind regularization can be introduced as 'keeping the parameters β small unless the data really convinces us otherwise', or alternatively 'if a model with small values of the parameters β 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 mathematically, which leads to slightly different solutions. We will focus on the so-called *ridge regression* and *LASSO*.

The concept of regularization extends well beyond linear regression and it can be used when working with other types of problems and models as well.

2.5.1. Ridge regression

In *ridge regression* (also known as *Tikhonov regularization*, $L2$ *regularaization*, or *weight decay*) the least squares criterion (2.16) is replaced with the modified minimization problem

$$\underset{\beta_0,\beta_1,\ldots,\beta_p}{\text{minimize}} \ \|\mathbf{X}\beta - \mathbf{y}\|_2^2 + \gamma\|\beta\|_2^2. \tag{2.27}$$

The value $\gamma \geq 0$ is referred to as a regularization parameter and this has to be chosen by the user. For $\gamma = 0$ we recover the original least squares problem (2.16), whereas if we let $\gamma \to \infty$ we will force all parameters β_j to approach 0. A good choice of γ is in most cases somewhere in between, and depends on the actual problem. It can either be found by manual tuning, or in a more systematic fashion using cross-validation.

It is actually possible to derive a closed-form solution for (2.27), akin to (2.17), namely

$$\widehat{\beta} = (\mathbf{X}^\mathsf{T}\mathbf{X} + \gamma I_{p+1})^{-1}\mathbf{X}^\mathsf{T}\mathbf{y}, \tag{2.28}$$

where I_{p+1} is the identity matrix of size $p + 1 \times p + 1$.

2.5.2. LASSO

With *LASSO* (an abbreviation for Least Absolute Shrinkage and Selection Operator), or equivalently $L1$ *regularization*, the least squares criterion (2.16) is replaced with

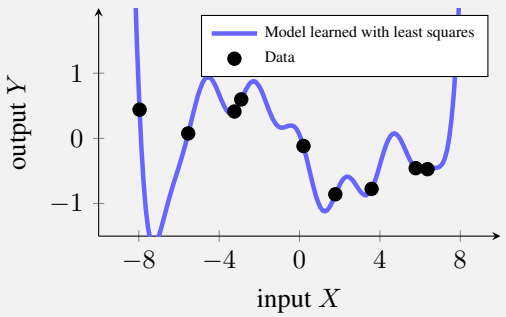
$$\underset{\beta_0,\beta_1,\ldots,\beta_p}{\text{minimize}} \ \|\mathbf{X}\beta - \mathbf{y}\|_2^2 + \gamma\|\beta\|_1, \tag{2.29}$$

where $\|\cdot\|_1$ is the Manhattan norm. Contrary to ridge regression, there is no closed-form solution available for (2.29), but it is a convex problem can be solved efficiently by numerical optimization.

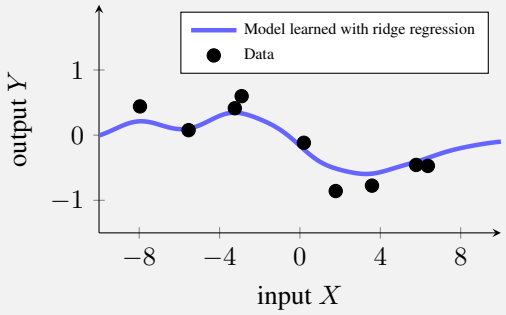
As for ridge regression, the regularization parameter γ has to be chosen by the user: $\gamma = 0$ gives the least squares problem and $\gamma \to \infty$ gives $\beta = 0$. Between these extremes, however, LASSO and ridge regression will result in different solutions: whereas ridge regression pushes all parameters $\beta_0, \beta_1, \ldots, \beta_p$ towards small values, LASSO tends to favor so-called sparse solutions where only a few of the parameters are non-zero, and the rest are exactly 0. Thus, the LASSO solution can effectively 'switch some of the inputs off' by setting the corresponding parameter to zero and it can therefore be used as an input (or feature) selection method.

Example 2.5: Regularization in a linear regression RBF model

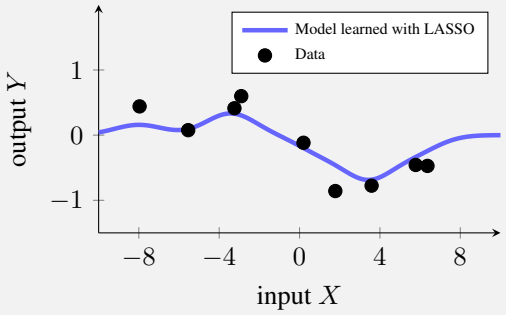
We have the problem of fitting a linear regression model with $p = 8$ radial basis function kernels (blue line) to $n = 9$ data points (black dots). Since we have $p = n-1$, we can expect the model to fit the data perfectly. However, as we see in (a) to the right, the model *overfits*, meaning that the model adapts *too* much to the data and has a 'strange' behavior between the data points. As a remedy to this, we can use ridge regression (b) or LASSO (c). Even though the final models with ridge regression and LASSO look rather similar, their parameters $\widehat{\beta}$ are different: the LASSO solution effectively only makes use of 5 (out of 8) radial basis functions. This is referred to as a *sparse* solution. Which approach to be preferred depends, of course, on the specific problem.



(a) The model learned with **least squares** (2.16). Even though the model follows the data exactly, we should typically not be happy with this model: neither the behavior between the data points nor outside the range is plausible, but is only an effect of *overfitting*, in that the model is adapted 'too well' to the data. The parameter values $\widehat{\beta}$ are around 30 and -30.



(b) The same model, this time learned with **ridge regression** (2.27) with a certain value of γ. Despite not being perfectly adapted to the training data, this model appears to give a more sensible trade-off between fitting the data and avoiding overfitting than (a), and is probably more useful in most situations. The parameter values $\widehat{\beta}$ are now roughly evenly distributed in range from -0.5 to 0.5.



(c) The same model again, this time learned with **LASSO** (2.29) with a certain value of γ. Again, this model is not perfectly adapted to the training data, but appears to have a more sensible trade-off between fitting the data and avoiding overfitting than (a), and is probably also more useful than (a) in most situations. In contrast to (b), however, 3 (out of 9) parameters are in this model exactly 0, and the rest are in the range from -1 to 1.

2.6. Linear regression in R

The main linear regression command in R is `lm()`. It takes a formula and a data frame as input, and returns an object which can be used by the `predict` command. We illustrate its use in Example 2.6.

R example 2.6: Linear regression using the `lm()` command

We consider the problem of predicting the carbon dioxide (CO_2) emission per capita in different countries, using the gross domestic product (GDP) per capita and the internet access as inputs. We use the 2015 data from

- The European Union Emission database, http://edgar.jrc.ec.europa.eu/overview.php?v=CO2ts_pc1990-2015,
- United Nations Statistics Division, https://unstats.un.org/unsd/snaama/dnlList.asp,
- International Telecommunications Union, http://data.un.org/DocumentData.aspx?id=374,

which we have compiled into a single file available at the course homepage as `emissiondata.csv`.
We start by loading the data into R as a data frame using the command `read.table()`

```
> emissiondata <- read.table("emissiondata.csv", header = TRUE)
```

We can look at the data frame by simply typing its name in the terminal,

```
> emissiondata
                    Country    Emission         GDP Internet
1                 Argentina  4.40380219  14564.5013     54.1
2                   Armenia  1.53652039   3489.1276     41.9
3                 Australia 18.62191971  51352.1972     83.5
4                   Austria  8.68896479  44117.6911     80.6
⋮
84           United_Kingdom  6.15806813  44162.3629     89.8
85 United_States_of_America 16.07446115  56053.8412     74.7
86                  Uruguay  2.15499169  15573.8085     61.5
87                Venezuela  5.74024779  11068.8722     49.1
```

that is, `emissiondata` contains 87 data points of the variables `Country` (the name of the country), `Emission` (metric tons CO_2 emmited per capita and year), `GDP` (Gross domestic product per capita in USD) and `Internet` (percentage of individuals using internet). Before we start using the `lm()` command, we first divide the data randomly into one training data set (to be used for learning) and one test data set (to be used for evaluation), so that we can evaluate and get an idea of how good our model is.

```
> train <- sample(x=1:nrow(emissiondata), size=60, replace=FALSE)
> emissiondata.train <- emissiondata[train,]
> emissiondata.test <- emissiondata[-train,]
```

The country name itself is probably not of any help in predicting the CO_2 emissions, but we start with using `GDP` and `Internet` as inputs in a linear regression model

```
> model <- lm(formula=Emission~GDP+Internet,data=emissiondata.train)
```

The argument `formula` specifies the output (left of ~) and inputs (right of ~) separated by +. The variable names here have to be the same as the ones used in the data frame provided as `data`, which specifies what data set to be used to learn the model. To evaluate the model, we try to predict the emission for the test data set,

```
> model.prediction <- predict(model,newdata=emissiondata.test)
> model.rmse <- sqrt(mean((model.prediction-emissiondata.test$Emission)^2))
```

and also compute the root mean square error (RMSE) for the predictions `model1.rmse`, where we compare the predictions `model1.prediction` to the emissions in the data `emissiondata.test$Emission`. This gives an idea of how good the model is for prediction.

2.6.1. Nonlinear transformation of inputs

If we want to try different features in the linear regression model, we can simply write it as the `formula` argument to the `lm()` command. If the data frame has three variables `Y`, `X1` and `X2`, encoding Y, X_1 and X_2 respectively, one can write:

- `formula=Y~X1+X2` to get $Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \epsilon$.
- `formula=Y~X1+X2-1` to get $Y = \beta_1 X_1 + \beta_2 X_2 + \epsilon$ (no intercept term).
- `formula=Y~X1+X2+X1:X2` to get $Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_1 X_2 + \epsilon$.
- `formula=Y~X1*X2` to also get $Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_1 X_2 + \epsilon$.
- `formula=Y~X1+log(X2)` to get $Y = \beta_0 + \beta_1 X_1 + \beta_2 \log(X_2) + \epsilon$.
- `formula=Y~X1+I(X2^2)` to get $Y = \beta_0 + \beta_1 X_1 + X_2^2 + \epsilon$
  (the `I()` is needed since `^2` otherwise has another meaning).

2.6.2. Qualitative inputs

The `lm()` command has built-in support for qualitative input variables and automatically creates dummy variables for non-numeric inputs (Section 2.4). To study which dummy variables are used, use the function `contrasts()` like `contrasts(data$X)`. To force a numeric input to be handled as a qualitative input (if the meaning of the numbers are different classes rather than a natural ordering), use the command `as.factor()` like `formula=Y~as.factor(X)`.

2.6.3. Regularization

The `lm()` command has no support for regularization. Some commands for regularized linear regression are instead

- `lm.ridge()` for ridge regression, which works similarly to the `lm()` command.
- `glmnet()` for ridge regression as well as LASSO. It has, however, a different syntax than `lm()`, and determines the regularization parameter automatically using cross-validation.

2.7. Further reading

Linear regression has now been used for well over 200 years. It was first introduced independently by Adrien-Marie Legendre in 1805 and Carl Friedrich Gauss in 1809 when they discovered the method of least squares. The topic of linear regression is due to its importance described in many textbooks in statistics and machine learning. Just to mention a few we have Bishop (2006), Gelman et al. (2013), Hastie, Tibshirani, and Friedman (2009), and Murphy (2012). While the basic least squares technique has been around for a long time, its regularized versions a much younger. Ridge regression was introduced independently in statistics by Hoerl and Kennard (1970) and in numerical analysis under the name of Tikhonov regularization. The LASSO was first introduced in 1996 by Tibshirani (1996). The recent monograph by Hastie, Tibshirani, and Wainwright (2015) covers the development relating to the use of sparse models and the LASSO.

3. Logistic regression

Logistic regression is a parametric model for classification. Since we usually talk about regression and classification as two different types of problems, the name "logistic regression" might be confusing at first. The reason for this name is that the method makes use of a linear regression model, the output of which is transformed to the interval $(0, 1)$ by the so called logistic function. This value is then interpreted as class probability.

This chapter contains a brief presentation of logistic regression and is to be viewed as a complement to James et al. 2013, Chapter 4.3. Specifically, in Section 3.1 we consider the binary classification setting and this material is mostly covered in James et al. 2013, Chapter 4.3, who also provide a more elaborate treatment. The exceptions are equations (3.6–3.9) which are not given in James et al. 2013. Note that we use the symbol q to denote the conditional class probabilities—see (3.1) and (3.10)—for which James et al. 2013 use the symbol p. We recommend reading James et al. 2013, Chapter 4.3 first and then continuing with Section 3.1 of these lecture notes. Section 3.2 then generalizes the logistic regression model to the the multi-class setting. This material is not covered by James et al. 2013. We use the multi-class logistic regression model primarily as a stepping stone to later derive a deep learning classification model in chapter 4.

3.1. Logistic regression for binary classification

In a binary classification problem the output has two possible values, which we can encode as 0 and 1, i.e. $Y \in \{0, 1\}$. Recall that the (optimal) Bayes classifier is based on the conditional class probabilities $\Pr(Y = 1 \mid X)$ and $\Pr(Y = 0 \mid X)$. Naturally, these probabilities depend on the input X. For notational convenience we define the function $q(X)$ to be the class-1 probability:

$$q(X) \stackrel{\text{def}}{=} \Pr(Y = 1 \mid X), \tag{3.1}$$

and consequently $\Pr(Y = 0 \mid X) = 1 - q(X)$. A classifier can thus be constructed based on a model for the function $q(X)$.

Since $q(X)$ corresponds to a probability it is constrained to the interval $(0, 1)$[1]. It is therefore natural to require that any model that we construct for $q(X)$ is also constrained to this interval. To accomplish this it is useful to consider a transformation of $q(X)$ which enforces the constraint. To this end, we start by defining the *odds* as the ratio between the two class probabilities,

$$\frac{q(X)}{1 - q(X)} \in (0, \infty). \tag{3.2}$$

Note that the image of the interval $(0, 1)$ by this transformation is the positive real line. Thus, if we then take the logarithm of this expression we obtain the *log-odds*

$$\log \frac{q(X)}{1 - q(X)} \in (-\infty, \infty) \tag{3.3}$$

which takes values on the whole real line.

The logistic regression model is based on using a linear regression model for the log-odds. That is, we model

$$\log \frac{q(X; \beta)}{1 - q(X; \beta)} = \beta_0 + \sum_{j=1}^{p} \beta_j X_j, \tag{3.4}$$

[1] We exclude the boundary points 0 and 1 to avoid having to work with the extended real line (including the points $\pm\infty$) below. The probability of class 1 under a logistic regression model will never be exactly zero or one, so this restriction does not matter in practice.

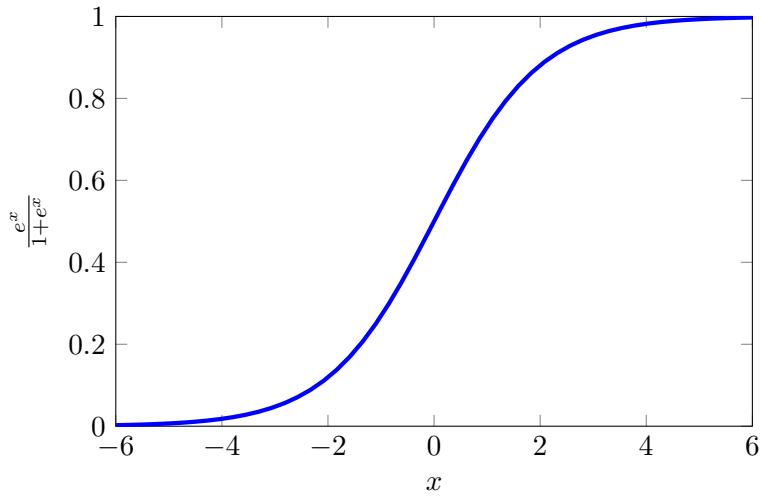


Figure 3.1.: Logistic function.

where we have added an explicit dependence on the unknown parameters β in the model. In the sequel we will use the notational convention $X = [1,\ X_1,\ \ldots,\ X_p]^\mathsf{T}$, i.e. the input vector X is assumed to contain the constant 1 in its first position (cf. the notation (2.4) used for linear regression). This means that we can write

$$\beta_0 + \sum_{j=1}^{p} \beta_j X_j = \beta^\mathsf{T} X.$$

Since the log-odds transformation is invertible, we can equivalently write the model (3.4) as

$$q(X;\beta) = \frac{e^{\beta^\mathsf{T} X}}{1 + e^{\beta^\mathsf{T} X}}, \tag{3.5}$$

which is precisely the *logistic function* applied to the linear activation $\beta^\mathsf{T} X$. The logistic function, shown in Figure 3.1 maps the real line to the interval $(0, 1)$ as needed.

Learning a logistic regression model, i.e. estimating the parameters β, is often done by maximum likelihood. The logistic regression model presented above directly provides a model of the data likelihood. Given a training data set $\mathcal{T} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ we can thus write the likelihood of the observed data as

$$\begin{aligned}
\ell(\beta) &\overset{\text{def}}{=} \Pr(Y_1 = y_1, \ldots, Y_n = y_n \,|\, X_1 = x_1, \ldots, X_n = x_n; \beta) \\
&= \prod_{i=1}^{n} \Pr(Y_i = y_i \,|\, X_i = x_i; \beta) && \text{independence} \\
&= \prod_{i:y_i=1} q(x_i;\beta) \prod_{i:y_i=0} (1 - q(x_i;\beta)) && \text{definition of } q(X;\beta)
\end{aligned} \tag{3.6}$$

Taking the logarithm we get the log-likelihood function

$$\begin{aligned}
\log \ell(\beta) &= \sum_{i:y_i=1} \log q(x_i;\beta) + \sum_{i:y_i=0} \log(1 - q(x_i;\beta)) \\
&= \sum_{i=1}^{n} \{y_i \log q(x_i;\beta) + (1 - y_i) \log(1 - q(x_i;\beta))\} && \text{using } I(y_i = 1) = y_i \\
&= \sum_{i=1}^{n} \left\{ y_i \beta^\mathsf{T} x_i - \log\left(1 + e^{\beta^\mathsf{T} x_i}\right) \right\} && \text{logistic regression model}
\end{aligned} \tag{3.7}$$

We then seek to compute the maximum likelihood estimate of β as

$$\widehat{\beta} = \arg\max_{\beta} \log \ell(\beta). \tag{3.8}$$

As usual we work with the logarithm of the likelihood rather than with the likelihood itself for improved numerical stability. This is possible due to the monotonicity of the logarithm, implying that the maximizing argument $\widehat{\beta}$ is the same for the log-likelihood and for the likelihood.

To maximize the log-likelihood we set the gradient to zero,

$$\nabla_\beta \log \ell(\beta) = \sum_{i=1}^{n} x_i \left(y_i - \frac{e^{\beta^\mathsf{T} x_i}}{1 + e^{\beta^\mathsf{T} x_i}} \right) = 0. \tag{3.9}$$

Note that we use the convention $x_i = \begin{pmatrix} 1 & x_{i1} & \cdots & x_{ip} \end{pmatrix}^\mathsf{T}$ and that the equation above is vector-valued, i.e. we have a system of $p+1$ equations to solve (with $p+1$ unknown elements of the vector β). Contrary to the linear regression model (with Gaussian noise) discussed in Section 2.2.1, this maximum likelihood problem results in a *nonlinear* system of equations, lacking a general closed form solution. Instead, we are forced to use a numerical solver. The standard choice is to use the Newton–Raphson algorithm (equivalent to the *iteratively reweighted least squares* algorithm), see e.g. Hastie, Tibshirani, and Friedman 2009, Chapter 4.4.

3.2. Logistic regression for multi-class problems

The logistic regression model can be extended to the multi-class classification problem by modeling the log-odds w.r.t. a chosen reference class as linear regressions. Assume that there are K possible classes, $Y \in \{1, \ldots, K\}$, and define the conditional class probabilities

$$q_k(X) \overset{\text{def}}{=} \Pr(Y = k \,|\, X) \qquad\qquad k = 1, \ldots, K. \tag{3.10}$$

If we select class K as the reference class, then the model is defined via the $K-1$ log-odds between the first $K-1$ classes and class K, i.e.

$$\begin{aligned} \log \frac{q_1(X;\theta)}{q_K(X;\theta)} &= \beta_{01} + \sum_{j=1}^{p} \beta_{j1} X_j, \\ \log \frac{q_2(X;\theta)}{q_K(X;\theta)} &= \beta_{02} + \sum_{j=1}^{p} \beta_{j2} X_j, \\ &\vdots \\ \log \frac{q_{K-1}(X;\theta)}{q_K(X;\theta)} &= \beta_{0(K-1)} + \sum_{j=1}^{p} \beta_{j(K-1)} X_j. \end{aligned} \tag{3.11}$$

The choice of reference class is in fact arbitrary and we obtain an equivalent model if the log-odds are defined using any of the K classes as reference. Note that the model has in total $(K-1) \times (p+1)$ parameters, $\beta_{01}, \ldots, \beta_{p1}, \ldots, \beta_{0(K-1)}, \ldots, \beta_{p(K-1)}$ which we collect in the parameter vector θ.

Just as in the 2-class setting we can invert (3.11) to compute the class probabilities, using the fact that $\sum_{k=1}^{K} q_k(X;\theta) = 1$, which results in

$$\begin{aligned} q_k(X;\theta) &= \frac{\exp\left(\beta_{0k} + \sum_{j=1}^{p} \beta_{jk} X_j\right)}{1 + \sum_{l=1}^{K-1} \exp\left(\beta_{0l} + \sum_{j=1}^{p} \beta_{jl} X_j\right)}, \quad k = 1, \ldots, K-1 \\ q_K(X;\theta) &= \frac{1}{1 + \sum_{l=1}^{K-1} \exp\left(\beta_{0l} + \sum_{j=1}^{p} \beta_{jl} X_j\right)}. \end{aligned} \tag{3.12}$$

Based on these expressions we can derive an expression for the log-likelihood of the observed training data analogously to (3.7), which can then be maximized using numerical optimization.

Before deriving the expression for the log-likelihood, however, we discuss an alternative parameterization of the model which is also common in practice, in particular as a component of deep neural networks (see chapter 4). This parameterization is based on the so called *softmax* function which is a mapping from $\mathbb{R}^K$ to $(0, 1)^K$, such that its elements sum to one[2]. Specifically, the softmax function applied to the vector $Z = [Z_1, \ldots, Z_K]^\mathsf{T}$ is given by

$$\mathrm{softmax}(Z) = \frac{1}{\sum_{l=1}^{K} e^{Z_l}} [e^{Z_1}, \ldots, e^{Z_K}]^\mathsf{T}. \tag{3.13}$$

Using the softmax function we can model the class probabilities $q_k(X)$ as

$$q_k(X; \theta) = [\mathrm{softmax}(Z)]_k \tag{3.14}$$

where

$$Z_k = \beta_{0k} + \sum_{j=1}^{p} \beta_{jk} X_j, \qquad k = 1, \ldots, K \tag{3.15}$$

and $[\mathrm{softmax}(Z)]_k = \frac{e^{Z_k}}{\sum_{l=1}^{K} e^{Z_l}}$ is the kth output from the softmax function.

A difference between the two parameterizations, one based on the log-odds as in (3.11) and one based on the softmax function as in (3.14), is that the latter is an *over-parameterization* of the K probabilities. Indeed, counting the total number of parameters in (3.15) we see that the dimension of θ in this parameterization is $K \times (p + 1)$. Hence, the latter paramterization has an additional $p + 1$ parameters which are "not needed". The explanation lies in the fact that the softmax function is shift-invariant, i.e. $\mathrm{softmax}(Z + c) = \mathrm{softmax}(Z)$ for any constant c which is added to all elements of the vector Z. Consequently, we obtain the same model as in (3.14) if we shift all elements of the vector Z by, for instance, the value Z_K. This effectively turns class K into a "reference class" and (3.14) reduces to (3.11) (with appropriately redefined parameters). In practice, however, the over-parameterization of the softmax model is typically not an issue and it is common to use model defined via (3.14) and (3.15) directly.

For either parameterization of the class probabilities q_k, the model parameters can be found by maximizing the likelihood of the observed training data $\mathcal{T} = \{(x_i, y_i)\}_{i=1}^n$. The log-likelihood is derived analogously to (3.7) for the binary classification problem and is given by

$$\log \ell(\theta) = \sum_{i=1}^{n} \log q_{y_i}(x_i; \theta) = \sum_{i=1}^{n} \sum_{k=1}^{K} I(y_i = k) \log q_k(x_i; \theta). \tag{3.16}$$

Remark 3.1 (One-hot encoding). For the binary problem we used the convenient encoding $y_i \in \{0, 1\}$ which means that $I(y_i = 1) = y_i$. This allowed us to express the likelihood on a simple form, without having to use the indicator function explicitly; see (3.7). It is common to use a similar "trick" also for the K-class problem in order to express the log-likelihood (3.16) on a simpler form. However, this requires an alternative encoding of the K classes, referred to as *one-hot encoding*. Instead of letting Y be an integer value in the range $\{1, \ldots, K\}$, one-hot encoding represents the kth class by a vector $Y = [Y_1, \ldots, Y_K]^\mathsf{T}$ where $Y_k = 1$ and $Y_j = 0$ for $j \neq k$. That is, all elements of the vector are zero except one and the position of the non-zero element determines the class. For example, if there are three possible classes we encode the first class as $Y = [1, 0, 0]^\mathsf{T}$, the second class as $Y = [0, 1, 0]^\mathsf{T}$, and the third class as $Y = [0, 0, 1]^\mathsf{T}$. Note that there is a one-to-one relationship between the one-hot encoding and the previously used integer-based encoding and we will use both representations interchangeably. With the one-hot encoding of the training outputs we can write the log-likelihood (3.16) as

$$\log \ell(\theta) = \sum_{i=1}^{n} \sum_{k=1}^{K} y_{ik} \log q_k(x_i; \theta), \tag{3.17}$$

where $q_k(X; \theta)$ is given by either (3.12) or (3.14).

[2] In mathematical terms, the softmax function maps $\mathbb{R}^K$ to the $(K - 1)$-dimensional probability simplex.

4. Deep learning and neural networks

Neural networks can be used for both regression and classification, and they can be seen as an extension of linear regression and logistic regression, respectively. Traditionally neural networks with *one* so-called hidden layer have been used and analysed, and several success stories came in the 1980s and early 1990s. In the 2000s it was, however, realized that *deep* neural networks with *several* hidden layers, or simply *deep learning*, are even more powerful. With the combination of new software, hardware, parallel algorithms for training and a lot of training data, deep learning has made a major contribution to machine learning. Deep learning has excelled in many applications, including image classification, speech recognition and language translation. New applications, analysis, and algorithmic developments to deep learning are published literally every day.

We will start in Section 4.1 by generalizing linear regression to a two-layer neural network (i.e., a neural network with one hidden layer), and generalize it further to a deep neural network. We thereafter leave regression and look at the classification setting in Section 4.2. In Section 4.3 we present a special neural network tailored for images and finally, we turn to the training of neural networks in Section 4.4.

4.1. Neural networks for regression

A neural network is a nonlinear function that describes the output variable Y as a nonlinear function of its input variables

$$Y = f(X_1, \ldots, X_p; \theta) + \epsilon, \tag{4.1}$$

where ϵ is an error term and the function f is parametrized by θ. Such a nonlinear function can be parametrized in many ways. In a neural network, the strategy is to use several *layers* of linear regression models and nonlinear *activation functions*. We will explain this carefully in turn below. For the model description it will be convenient to define Z as the output without the noise term ϵ,

$$Z = f(X_1, \ldots, X_p; \theta). \tag{4.2}$$

4.1.1. Generalized linear regression

We start the description with a graphical illustration of the linear regression model

$$Z = \beta_0 1 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p, \tag{4.3}$$

which is shown in Figure 4.1a. Each input variable X_i is represented with a node and each parameter β_i with a link. Furthermore, the output Z is described as the sum of all terms $\beta_i X_i$. Note that we use 1 as the input variable corresponding to the offset term β_0.

To describe *nonlinear* relationships between X and Z we introduce a nonlinear scalar function called the *activation function* $\sigma : \mathbb{R} \to \mathbb{R}$. The linear regression model (4.3) is now modified into a *generalized* linear regresssion model where the linear combination of the inputs is squashed through the (scalar) activation function

$$Z = \sigma\big(\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p\big). \tag{4.4}$$

This extension to the generalized linear regression model is visualized in Figure 4.1b.

Common choices for activation function are the *logistic function* and the *rectified linear unit* (ReLU). These are illustrated in Figure 4.2a and Figure 4.2b, respectively. The logistic function has already been used in (3.5)

(a) (b)

Figure 4.1.: Graphical illustration of a linear regression model (Figure 4.1a), and a generalized linear regression model (Figure 4.1b). In Figure 4.1a, the output Z is described as the sum of all terms β_0 and $\{\beta_i X_i\}_{i=1}^p$, as in (4.3). In Figure 4.1b, the circle denotes addition and also transformation through the activation function σ, as in (4.4).

(a) (b)

Figure 4.2.: Two common activation functions used in neural networks. The sigmoid function (Figure 4.2a), and the rectified linear unit (Figure 4.2b).

in the context of logistic regression. The logistic function is affine close to $x = 0$ and saturates at 0 and 1 as x decreases or increases. The ReLU is even simpler. The function is the identity function for positive inputs and equal to zero for negative inputs.

The logistic function used to be the standard choice of activation function in neural networks for many years, whereas the ReLU has gained in popularity (despite its simplicity!) during recent years and it is now the standard choice in most neural network models.

The generalized linear regression model (4.4) is very simple and is itself not capable of describing very complicated relationships between the input X and the output Z. Therefore, we make two further extensions to increase the generality of the model: We will first make use of *several* generalized 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, or simply *deep learning*).

4.1.2. Two-layer neural network

In (4.4), the output is constructed by one scalar regression model. To increase its flexibility and turn it into a two-layer neural network, we instead let the output be a sum of M generalized linear regression models, each of which has its own parameters. The parameter for the ith regression model are $\beta_{0i}, \ldots, \beta_{pi}$ and we denote its output by H_i,

$$H_i = \sigma\left(\beta_{0i} + \beta_{1i}X_1 + \beta_{2i}X_2 + \cdots + \beta_{pi}X_p\right), \qquad i = 1, \ldots, M. \tag{4.5}$$

These intermediate outputs H_i are so-called *hidden units*, since they are not the output of the whole model. The M different hidden units $\{H_i\}_{i=1}^M$ instead act as input variables to an additional linear regression model

$$Z = \beta_0 + \beta_1 H_1 + \beta_2 H_2 + \cdots + \beta_M H_M. \tag{4.6}$$

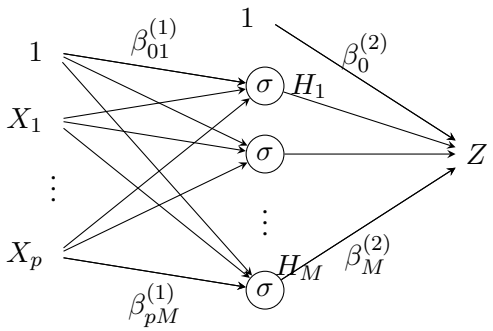


Figure 4.3.: A two-layer neural network, or equivalently, a neural network with one intermediate layer of hidden units.

To distinguish the parameters in (4.5) and (4.6) we add the superscripts (1) and (2), respectively. The equations describing this two-layer neural network (or equivalently, neural network with one layer of hidden units) are thus

$$\begin{aligned} H_1 &= \sigma\left(\beta_{01}^{(1)} + \beta_{11}^{(1)}X_1 + \beta_{21}^{(1)}X_2 + \cdots + \beta_{p1}^{(1)}X_p\right), \\ H_2 &= \sigma\left(\beta_{02}^{(1)} + \beta_{12}^{(1)}X_1 + \beta_{22}^{(1)}X_2 + \cdots + \beta_{p2}^{(1)}X_p\right), \\ &\vdots \\ H_M &= \sigma\left(\beta_{0M}^{(1)} + \beta_{1M}^{(1)}X_1 + \beta_{2M}^{(1)}X_2 + \cdots + \beta_{pM}^{(1)}X_p\right), \end{aligned} \tag{4.7a}$$

$$Z = \beta_0^{(2)} + \beta_1^{(2)}H_1 + \beta_2^{(2)}H_2 + \cdots + \beta_M^{(2)}H_M. \tag{4.7b}$$

Extending the graphical illustration from Figure 4.1, this model can be depicted as a graph with two-layers of links (illustrated using arrows), see Figure 4.3. As before, each link has a parameter associated with it. Note that we include an offset term not only in the input layer, but also in the hidden layer.

4.1.3. Matrix notation

The two-layer neural network model in (4.7) can also be written more compactly using matrix notation, where the parameters in each layer are stacked in a *weight matrix* W and an *offset vector*[1] b as

$$b^{(1)} = \begin{bmatrix}\beta_{01}^{(1)} & \cdots & \beta_{0M}^{(1)}\end{bmatrix}, \quad W^{(1)} = \begin{bmatrix}\beta_{11}^{(1)} & \cdots & \beta_{1M}^{(1)} \\ \vdots & \cdots & \vdots \\ \beta_{p1}^{(1)} & \cdots & \beta_{pM}^{(1)}\end{bmatrix}, \quad b^{(2)} = \begin{bmatrix}\beta_0^{(2)}\end{bmatrix}, \quad W^{(2)} = \begin{bmatrix}\beta_1^{(2)} \\ \vdots \\ \beta_M^{(2)}\end{bmatrix}. \tag{4.8}$$

The full model can then be written as

$$H = \sigma\left(W^{(1)\mathsf{T}}X + b^{(1)\mathsf{T}}\right), \tag{4.9a}$$

$$Z = W^{(2)\mathsf{T}}H + b^{(2)\mathsf{T}}, \tag{4.9b}$$

where we have also stacked the components in X and H as $X = [X_1, \ldots, X_p]^\mathsf{T}$ and $H = [H_1, \ldots, H_M]^\mathsf{T}$. The activation function σ acts element-wise. The two weight matrices and the two offset vectors will be the

[1] The word "bias" is often used for the offset vector in the neural network literature, but this is really just a model parameter and not a bias in the statistical sense. To avoid confusion we refer to it as an offset instead.

parameters in the model, which can be written as

$$\theta = \begin{bmatrix} \text{vec}(W^{(1)})^\mathsf{T} & \text{vec}(W^{(2)})^\mathsf{T} & b^{(1)} & b^{(2)} \end{bmatrix}^\mathsf{T}. \tag{4.10}$$

By this we have described a nonlinear regression model on the form $Y = f(X;\theta) + \epsilon$ where $f(X;\theta) = Z$ according to above. Note that the predicted output Z in (4.9b) depends on all the parameters in θ even though it is not explicitly stated in the notation.

4.1.4. 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 for it. However, the real descriptive power of a neural network is realized when we stack multiple such layers of generalized linear regression models, and thereby achieve a *deep* neural network. Deep neural networks can model complicated relationships (such as the one between an image and its class), and is one of the state-of-the-art methods in machine learning as of today.

We enumerate the layers with index l. Each *layer* is parametrized with a weight matrix $W^{(l)}$ and an offset vector $b^{(l)}$, as for the two-layer case. For example, $W^{(1)}$ and $b^{(1)}$ belong to layer $l = 1$, $W^{(2)}$ and $b^{(2)}$ belong to layer $l = 2$ and so forth. We also have multiple *layers of hidden units* denoted by $H^{(l-1)}$. Each such layer consists of M_l hidden units $H^{(l)} = [H_1^{(l)}, \ldots, H_{M_l}^{(l)}]^\mathsf{T}$, where the dimensions M_1, M_2, $\ldots$ can be different for different layers.

Each layer maps a hidden layer $H^{(l-1)}$ to the next hidden layer $H^{(l)}$ as

$$H^{(l)} = \sigma(W^{(l)\mathsf{T}} H^{(l-1)} + b^{(l)\mathsf{T}}). \tag{4.11}$$

This means that the layers are stacked such that the output of the first layer $H^{(1)}$ (the first layer of hidden units) is the input to the second layer, the output of the second layer $H^{(2)}$ (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 mathematically be described as (cf. (4.9))

$$\begin{aligned} H^{(1)} &= \sigma(W^{(1)\mathsf{T}} X + b^{(1)\mathsf{T}}), \\ H^{(2)} &= \sigma(W^{(2)\mathsf{T}} H^{(1)} + b^{(2)\mathsf{T}}), \\ &\vdots \\ H^{(L-1)} &= \sigma(W^{(L-1)\mathsf{T}} H^{(L-2)} + b^{(L-1)\mathsf{T}}), \\ Z &= W^{(L)\mathsf{T}} H^{(L-1)} + b^{(L)\mathsf{T}}. \end{aligned} \tag{4.12}$$

A graphical representation of this model is represented in Figure 4.4.

The weight matrix $W^{(1)}$ for the first layer $l = 1$ has the dimension $p \times M_1$ and the corresponding offset vector $b^{(1)}$ the dimension $1 \times M_1$. In deep learning it is common to consider applications where also the output is multi-dimensional $Z = [Z_1, \ldots, Z_K]^\mathsf{T}$. This means that for the last layer the weight matrix $W^{(L)}$ has the dimension $M_{L-1} \times K$ and the offset vector $b^{(L)}$ the dimension $1 \times K$. For all intermediate layers $l = 2, \ldots, L-1$, $W^{(l)}$ has the dimension $M_{l-1} \times M_l$ and the corresponding offset vector $1 \times M_l$.

The number of inputs p and the number of outputs K are given by the problem, but the number of layers L and the dimensions M_1, M_2, $\ldots$ are user design choices that will determine the flexibility of the model.

4.1.5. Learning the network from data

Analogously to the parametric models presented earlier (e.g. linear regression and logistic regression) we need to learn all the parameters in order to use the model. For deep neural networks the parameters are

$$\theta = \begin{bmatrix} \text{vec}(W^{(1)})^\mathsf{T} & \text{vec}(W^{(2)})^\mathsf{T} & \cdots \text{vec}(W^{(L)})^\mathsf{T} & b^{(1)\mathsf{T}} & b^{(2)} & \cdots & b^{(L)} \end{bmatrix}^\mathsf{T} \tag{4.13}$$

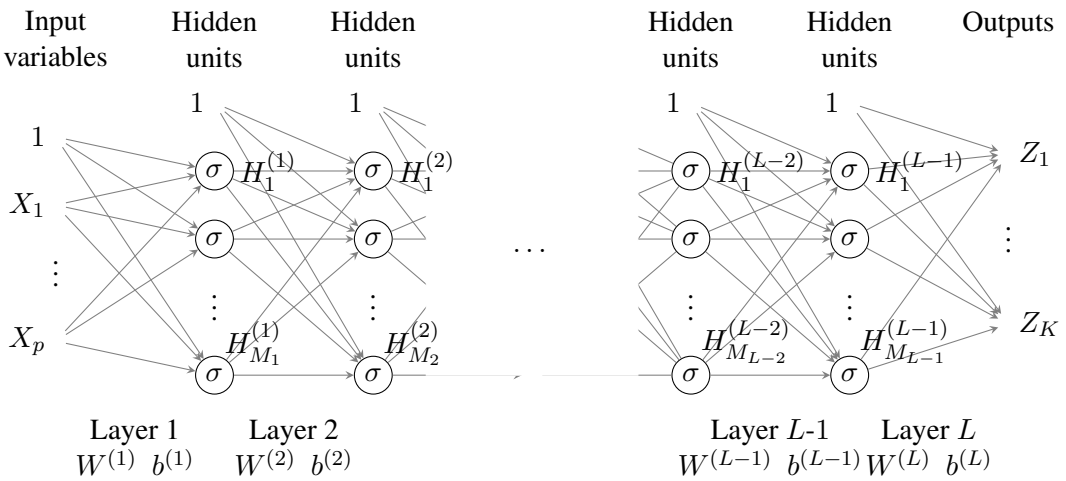


Figure 4.4.: A deep neural network with L layers. Each layer is parameterized with $W^{(l)}$ and $b^{(l)}$.

The wider and deeper the network is, the more parameters there are. Practical deep neural networks can easily have in the order of millions of parameters and these models are therefore also extremely flexible. Hence, some mechanism to avoid overfitting is needed. Regularization such as ridge regression is common (cf. Section 2.5), but there are also other techniques specific to deep learning; see further Section 4.4.4. Furthermore, the more parameters there are, the more computational power is needed to train the model. As before, the training data $\mathcal{T}$ consists of n samples of the input X and the output Y, $\{(x_i, y_i)\}_{i=1}^n$.

For a regression problem we typically start with maximum likelihood and assume Gaussian noise $\epsilon \sim \mathcal{N}(0, \sigma_\epsilon^2)$, and thereby obtain the square error loss function as in Section 2.2.1,

$$\widehat{\theta} = \arg\min_\theta \frac{1}{n} \sum_{i=1}^n L(x_i, y_i, \theta) \qquad \text{where} \qquad L(x_i, y_i, \theta) = \|y_i - f(x_i; \theta)\|^2 = \|y_i - z_i\|^2. \tag{4.14}$$

This problem can be solved with numerical optimization, and more precisely stochastic gradient descent. This is described in more detail in Section 4.4.

From the model, the parameters θ, and the inputs $\{x_i\}_{i=1}^n$ we can compute the predicted outputs $\{z_i\}_{i=1}^n$ using the model $z_i = f(x_i; \theta)$. For example, for the two-layer neural network presented in Section 4.1.2 we have

$$h_i^\mathsf{T} = \sigma(x_i^\mathsf{T} W^{(1)} + b^{(1)}), \tag{4.15a}$$
$$z_i^\mathsf{T} = h_i^\mathsf{T} W^{(2)} + b^{(2)}. \tag{4.15b}$$

In (4.15) the equations are transposed in comparison to the model in (4.9). This is a small trick such that we easily can extend (4.15) to include multiple data points i. Similar to (2.4) we stack all data points in matrices, where each data point represents one row

$$\mathbf{Y} = \begin{bmatrix} y_1^\mathsf{T} \\ \vdots \\ y_n^\mathsf{T} \end{bmatrix}, \qquad \mathbf{X} = \begin{bmatrix} x_1^\mathsf{T} \\ \vdots \\ x_n^\mathsf{T} \end{bmatrix}, \quad \mathbf{Z} = \begin{bmatrix} z_1^\mathsf{T} \\ \vdots \\ z_n^\mathsf{T} \end{bmatrix}, \quad \text{and} \quad \mathbf{H} = \begin{bmatrix} h_1^\mathsf{T} \\ \vdots \\ h_n^\mathsf{T} \end{bmatrix}. \tag{4.16}$$

We can then write (4.15) as

$$\mathbf{H} = \sigma(\mathbf{X} W^{(1)} + b^{(1)}), \tag{4.17a}$$
$$\mathbf{Z} = \mathbf{H} W^{(2)} + b^{(2)}, \tag{4.17b}$$

where we also have stacked the predicted output and the hidden units in matrices. This is also how the model would be implemented in code. In Tensorflow for R, which will be used in the laboratory work in the course, it can be written as

```
H <- tf$sigmoid(tf$matmul(X, W1) + b1)
Z <- tf$matmul(H, W2) + b2
```

Note that in (4.17) the offset vectors b_1 and b_2 are added and broadcasted to each row. See more details regarding implementation of a neural network in the instructions for the laboratory work.

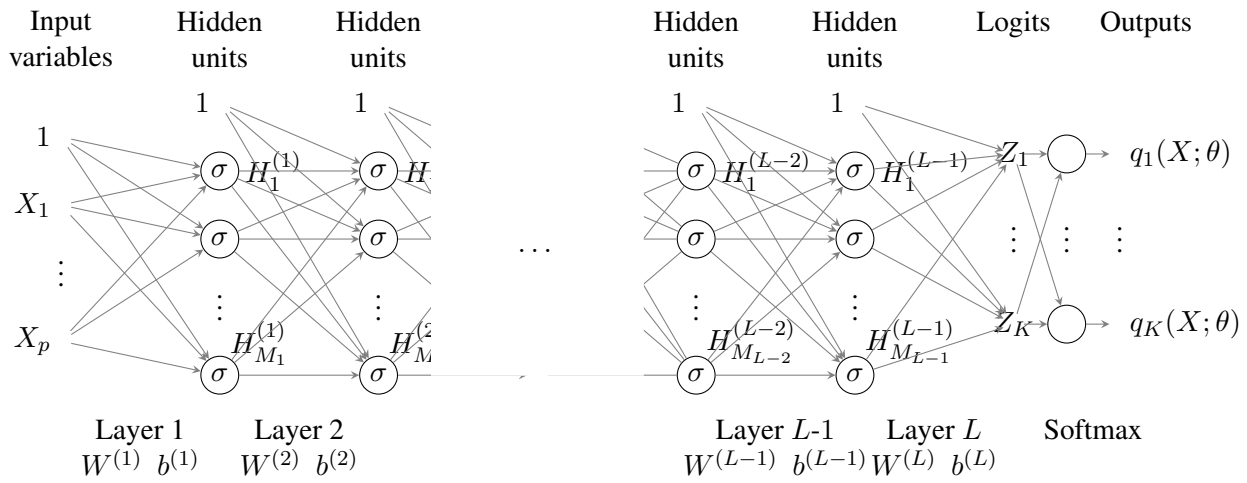


Figure 4.5.: A deep neural network with L layers for classification. The only difference to regression (Figure 4.4) is the softmax transformation after layer L.

4.2. Neural networks for classification

Neural networks can also be used for classification where we have qualitative outputs $Y \in \{1, \ldots, K\}$ instead of quantitative. In Chapter 3 we extended linear regression to logistic regression by adding the logistic function to the output. In the same manner we can extend the neural network presented in the last section to a neural network for classification. In doing this extension, we will use the multi-class version of logistic regression presented in Section 3.2, and more specifically the softmax parametrization given in (3.13), repeated here for convenience

$$\mathrm{softmax}(Z) = \frac{1}{\sum_{j=1}^{K} e^{Z_j}} \begin{bmatrix} e^{Z_1} & \cdots & e^{Z_K} \end{bmatrix}^{\mathsf{T}}. \tag{4.18}$$

The softmax function acts as an additional activation function on the final layer of the neural network. In addition to the regression network in (4.12) we add the softmax function at the end of the network as

$$\begin{aligned} &\vdots \\ Z &= W^{(L)\mathsf{T}} H^{(L-1)} + b^{(L)\mathsf{T}}, && \text{(4.19a)} \\ q_1(X;\theta) &= [\mathrm{softmax}(Z)]_1, \\ &\vdots && \text{(4.19b)} \\ q_K(X;\theta) &= [\mathrm{softmax}(Z)]_K. \end{aligned}$$

The softmax function maps the output of the last layer $Z_1, \ldots, Z_K$ to the class probabilities $q_1(X;\theta), \ldots, q_K(X;\theta)$, see also Figure 4.5 for a graphical illustration. The inputs to the softmax function, i.e. the variables $Z_1, \ldots, Z_K$, are referred to as *logits*. Each element $q_k(X;\theta)$ serves as a model for the conditional class probability $\Pr(Y = k \mid X)$, as for logistic regression in Section 3.2.

Note that the softmax function does not come as a layer with additional parameters, it only transforms the previous output $[Z_1, \ldots, Z_K] \in \mathbb{R}^K$ to $[q_1(X;\theta), \ldots, q_K(X;\theta)] \in [0,1]^K$ such that they can be interpreted as probabilities. Also note that by construction of the softmax function, these values will sum to 1 regardless of the values of $[Z_1, \ldots, Z_K]$.

4.2.1. Learning classification networks from data

As before, the training data consists of n samples of inputs and outputs $\{(x_i, y_i)\}_{i=1}^{n}$. For the classification problem we use the one-hot encoding for the output y_i. This means that for a problem with K different classes,

	$k=1$	$k=2$	$k=3$
y_{ik}	0	**1**	0
$q_k(x_i;\theta_A)$	0.1	**0.8**	0.1

Cross-entropy:
$L(x_i, y_i, \theta_A) = -1 \cdot \log 0.8 = 0.22$

	$k=1$	$k=2$	$k=3$
y_{ik}	0	**1**	0
$q_k(x_i;\theta_B)$	0.8	**0.1**	0.1

Cross-entropy:
$L(x_i, y_i, \theta_B) = -1 \cdot \log 0.1 = 2.30$

Figure 4.6.: Illustration of the cross-entropy between a data point y_i and two different prediction outputs.

y_i consists of K elements $y_i = \begin{bmatrix} y_{i1} & \dots & y_{iK} \end{bmatrix}^\mathsf{T}$. If a data point i belongs to class k then $y_{ik} = 1$ and $y_{ij} = 0$ for all $j \neq k$. See more about the one-hot encoding in Section 3.2.

For a neural network with the softmax activation function on the final layer we typically use the negative log-likelihood, which is also commonly referred to as the *cross-entropy* loss function, to train the model (cf. (3.17))

$$\widehat{\theta} = \arg\min_\theta \frac{1}{n} \sum_{i=1}^{n} L(x_i, y_i, \theta) \qquad \text{where} \qquad L(x_i, y_i, \theta) = -\sum_{k=1}^{K} y_{ik} \log q_k(x_i;\theta). \tag{4.20}$$

The cross-entropy is close to its minimum if the predicted probability $q_k(x_i;\theta)$ is close to 1 for the k for which $y_{ik} = 1$. For example, if the ith data point belongs to class $k = 2$ out of in total $K = 3$ classes we have $y_i = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}^\mathsf{T}$. Assume that we have a sets of parameters of the network denoted θ_A, and with these parameters we predict $q_1(x_i;\theta_A) = 0.1$, $q_2(x_i;\theta_A) = 0.8$ and $q_3(x_i;\theta_A) = 0.1$ meaning that we are quite sure that data point i actually belongs to class $k = 2$. This would generate a low cross-entropy $L(x_i, y_i, \theta_A) = -(0 \cdot \log 0.1 + 1 \cdot \log 0.8 + 0 \cdot \log 0.1) = 0.22$. If we instead predict $q_1(x_i;\theta_B) = 0.8$, $q_2(x_i;\theta_B) = 0.1$ and $q_3(x_i;\theta_B) = 0.1$, the cross-entropy would be much higher $L(x_i, y_i, \theta_B) = -(0 \cdot \log 0.8 + 1 \cdot \log 0.1 + 0 \cdot \log 0.1) = 2.30$. For this case, we would indeed prefer the parameters θ_A over θ_B. This is summarized in Figure 4.6.

Computing the loss function explicitly via the logarithm could lead to numerical problems when $q_k(x_i;\theta)$ is close to zero since $\log(x) \to -\infty$ as $x \to 0$. This can be avoided since the logarithm in the cross-entropy loss function (4.20) can "undo" the exponential in the softmax function (4.18),

$$\begin{aligned} L(x_i, y_i, \theta) &= -\sum_{k=1}^{K} y_{ik} \log q_k(x_i;\theta) = -\sum_{k=1}^{K} y_{ik} \log[\mathrm{softmax}(z_i)]_k \\ &= -\sum_{k=1}^{K} y_{ik} \left(z_{ik} - \log\left\{ \textstyle\sum_{j=1}^{K} e^{z_{ij}} \right\} \right), \qquad (4.21) \\ &= -\sum_{k=1}^{K} y_{ik} \left(z_{ik} - \max_j z_{ij} - \log\left\{ \textstyle\sum_{j=1}^{K} e^{z_{ij} - \max_j z_{ij}} \right\} \right), \qquad (4.22) \end{aligned}$$

where z_{ik} are the logits.

4.3. Convolutional neural networks

Convolutional neural networks (CNN) are a special kind neural networks tailored for problems where the input data has a grid-like topology. In this text we will focus on images, which have a 2D-topology of pixels. Images as also the most common type of input data in applications where CNNs are applied. However, CNNs can be used for any input data on a grid, also in 1D (e.g. audio waveform data) and 3D (volumetric data e.g. CT scans or video data). We will focus on grayscale images, but the approach can easily be extended to color images as well.

4.3.1. Data representation of an image

Digital grayscale 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 gray.

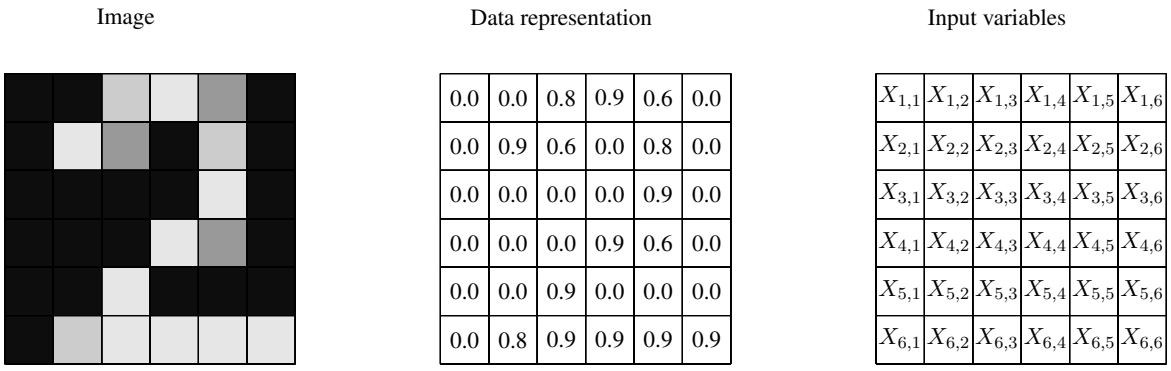


Figure 4.7.: Data representation of a grayscale image with 6×6 pixels. Each pixel is represented with a number describing the grayscale color. We denote the whole image as X (a matrix), and each pixel value is an input variable $X_{j,k}$ (element in the matrix X).

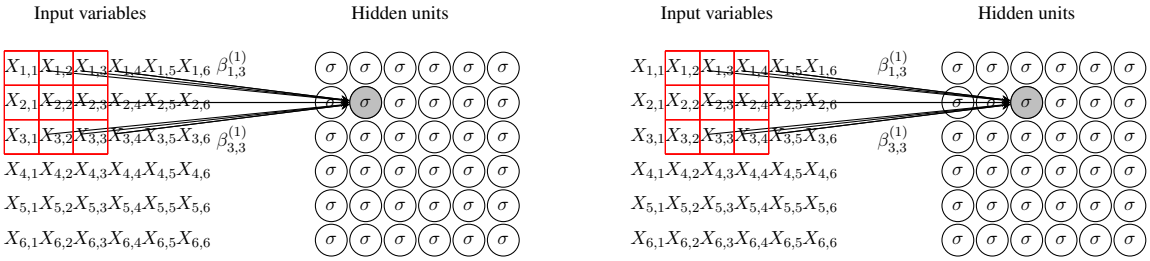


Figure 4.8.: An illustration of the interactions in a convolutional layer: Each hidden unit (circle) is only dependent on the pixels in a small region of the image (red boxes), here of size 3×3 pixels. The location of the hidden unit corresponds to the location of the region in the image: if we move to a hidden unit one step to the right, the corresponding region also moves one step to the right, compare Figure 4.8a and Figure 4.8b. Furthermore, the nine parameters $\beta_{1,1}^{(1)}, \beta_{1,2}^{(1)}, \ldots, \beta_{3,3}^{(1)}$ are the same for all hidden units in the layer.

In Figure 4.7 this is illustrated for an image with 6×6 pixels. In an image classification problem, an image is the input X and the pixels in the image are the input variables $X_{1,1}, X_{1,2}, \ldots, X_{6,6}$. The two indices j and k determine the position of the pixel in the image, as illustrated in Figure 4.7.

If we put all input variables representing the images pixels in a long vector, we can use the network architecture presented in Section 4.1 and 4.2 (and that is what we will do in the laboratory work to start with!). However, by doing that, a lot of the structure present in the image data will be lost. For example, we know that two pixels close to each other have more in common than two pixels further apart. This information would be destroyed by such a vectorization. In contrast, CNNs perverse this information by representing the input variables as well as the hidden layers as matrices. The core component in a CNN is the convolutional layer, which will be explained next.

4.3.2. The convolutional layer

Following the input layer, we use a hidden layer with equally many hidden units as input variables. For the image with 6×6 pixels we consequently have $6 \times 6 = 36$ hidden units. We choose to order the hidden units in a 6×6 matrix, i.e. in the same manner as we did for the input variables, see Figure 4.8a.

The network layers presented in earlier sections (like the one in Figure 4.3) have been *dense layer*. This means that each input variable is connected to each hidden unit in the following layer, and each such connection has a unique parameter β_{jk} associated with it. These layers have empirically been found to provide too much flexibility for images and we might not be able to capture the patterns of real importance, and hence not generalize and perform well on unseen data. Instead, a convolutional layer appears to exploit the structure present in images to

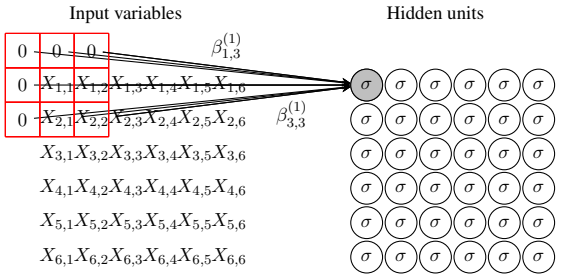


Figure 4.9.: An illustration of zero-padding. If the region is partly is outside the image. With zero-padding, the size of the image can be preserved in the following layer.

find a more efficiently parameterized model. In contrast to a dense layer, a convolutional layer leverages two important concepts – *sparse interactions* and *parameter sharing* – to achieve such a parametrization.

Sparse interactions

With sparse interactions we mean that most of the parameters in a dense layer are forced to be equal to zero. More specifically, a hidden unit in a convolutional layer only depends on the pixels in a small region of the image and not on all pixels. In Figure 4.8 this region is of size 3×3. The position of the region is related to the position of the hidden unit in its matrix topology. If we move to the hidden unit one step to the right, the corresponding region in the image also moves one step to the right, as displayed by comparing Figure 4.8a and Figure 4.8b. For the hidden units on the border, the corresponding region is partly located outside the image. For these border cases, we typically use zero-padding where the missing pixels are replaced with zeros. Zero-padding is illustrated in Figure 4.9.

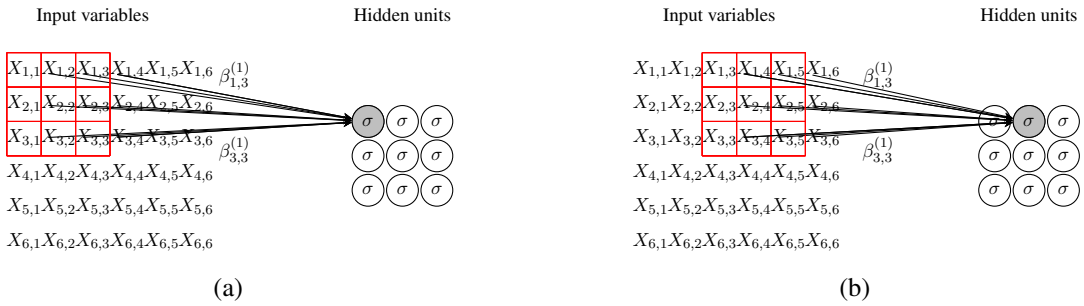
Parameter sharing

In a dense layer each link between an input variable and a hidden unit has its 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 unit are all the same. For example, in Figure 4.8a we use the same set of parameters to map the 3×3 region of pixels to the hidden unit as we do in Figure 4.8b. Instead of learning separate sets of parameters for every position we only learn one such 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 *kernel*. The mapping between the input variables and the hidden units can be interpreted as a convolution between the input variables and the kernel, hence the name convolutional neural network.

The sparse interactions and parameter sharing in a convolutional layer makes the CNN fairly invariant to translations of objects in the image. If the parameters in the kernel are sensitive to a certain detail (such as a corner, an edge, etc.) a hidden unit will react to this detail (or not) *regardless of where in the image that detail is present*! Furthermore, a convolutional layer uses a lot fewer parameters than the corresponding dense layer. In Figure 4.8 only $3 \cdot 3 + 1 = 10$ parameters are required (including the offset parameter). If we had used a dense layer $(36 + 1) \cdot 36 = 1332$ parameters would have been needed! Another way of interpreting this is: with the same amount of parameters, a convolutional layer can encode more properties of an image than a dense layer.

4.3.3. Condensing information with strides

In the convolutional layer presented above we have equally many hidden units as we have pixels in the image. As we add more layers to the CNN we usually want to condense the information by reducing the number of hidden units at each layer. One way of doing this is by not applying the kernel to every pixel but to say every two pixels. If we apply the kernel 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 6×6 image we get 3×3 hidden units. This concept is illustrated in Figure 4.10.



(a) (b)

Figure 4.10.: A convolutional layer with stride [2,2] and kernel of size 3×3.

The *stride* controls how many pixels the kernel shifts over the image at each step. In Figure 4.8 the stride is [1,1] since the kernel moves by one pixel both row- and column-wise. In Figure 4.10 the stride is [2,2] since it moves by two pixels row- and column-wise. Note that the convolutional layer in Figure 4.10 still requires 10 parameters, as the convolutional layer in Figure 4.8 does. Another way of condensing the information after a convolutional layer is by subsampling the data, so-called *pooling*. The interested can read further about pooling in Goodfellow, Bengio, and Courville 2016.

4.3.4. Multiple channels

The networks presented in Figure 4.8 and 4.10 only have 10 parameters each. Even though this parameterization comes with a lot of advantages, one kernel is probably not sufficient to encode all interesting properties of the images in our data set. To extended the network, we add multiple kernels, each with their own set of kernel parameters. Each kernel produces its own set of hidden units—a so-called *channel*—using the same convolution operation as explained in Section 4.3.2. Hence, each layer of hidden units in a CNN are organized into a tensor with the dimensions (rows $\times$ columns $\times$ channels). In Figure 4.11, 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 kernel 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 4.11. As a consequence, each kernel is a tensor of dimension (kernel rows $\times$ kernel columns $\times$ input channels). For example, each kernel in the second convolutional layer in Figure 4.11 is of size $3 \times 3 \times 4$. If we collect all kernels parameters in one weight tensor W, that tensor will be of dimension (kernel rows $\times$ kernel columns $\times$ input channels $\times$ output channels). In the second convolutional layer in Figure 4.11, the corresponding weight matrix $W^{(2)}$ is a tensor of dimension $3 \times 3 \times 4 \times 6$. With multiple kernels in each convolutional layer, each of them can be sensitive to different features in the image, such as certain edges, lines or circles enabling a rich representation of the images in our training data.

4.3.5. Full CNN architecture

A full CNN architecture consists of multiple convolutional layers. Typically, we decrease the number of rows and columns in the hidden layers as we proceed though the network, but instead increase the number of channels to enable the network to encode more high level features. After a few convolutional layers we usually end a CNN with one or more dense layers. If we consider 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 in the regression and classification networks explained earlier, depending on which type of problem we have at hand. In Figure 4.11 a small example of a full CNN architecture is displayed.

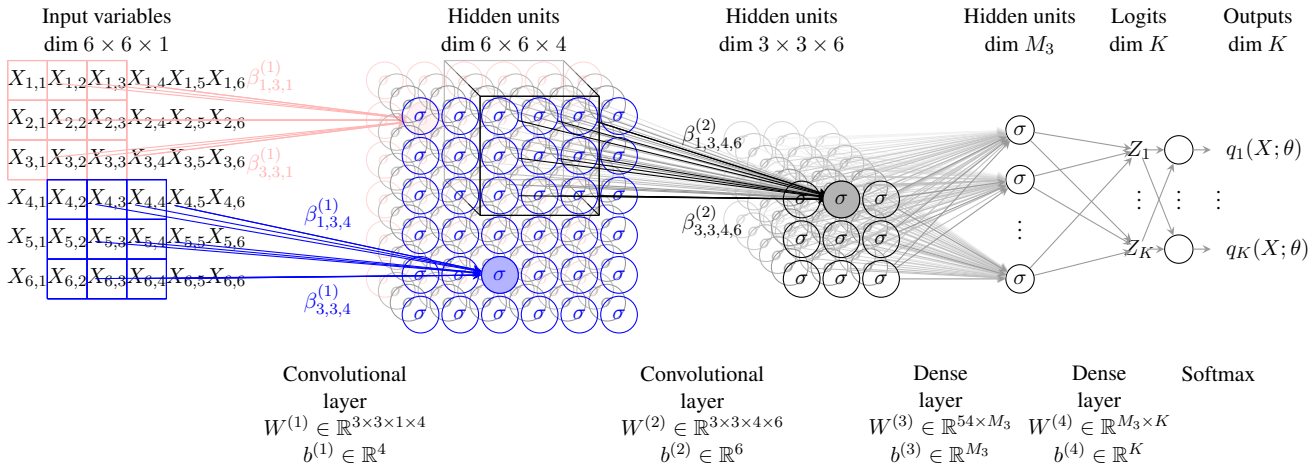


Figure 4.11.: A full CNN architecture for classification of grayscale 6×6 images. In the first convolutional layer four kernels, each of size 3×3, produce a hidden layer with four channels. The first channel (in the bottom) is visualized in red and the forth (on the top in blue). We use the stride [1,1] which maintains the number of rows and columns. In the second convolutional layer, six kernels of size $3 \times 3 \times 4$ and the stride [2,2] 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 the third layer of hidden units where all links have their unique parameters. We add an additional dense layer mapping down to the K logits. The network ends with a softmax function to provide predicted class probabilities as output.

4.4. Training a neural network

To use a neural network for prediction we need to find an estimate for the parameters $\widehat{\theta}$. To do that we solve an optimization problem on the form

$$\widehat{\theta} = \arg\min_{\theta} J(\theta) \qquad \text{where } J(\theta) = \frac{1}{n}\sum_{i=1}^{n} L(x_i, y_i, \theta). \tag{4.23}$$

We denote $J(\theta)$ as the *cost function* and $L(x_i, y_i, \theta)$ as the loss function. The functional form of the loss function depends on if we have regression or a classification problem at hand, see e.g. (4.14) and (4.20).

These optimization problems can not be solved in closed form, so numerical optimization has to used. In Appendix B, an introduction to numerical optimization is provided. In all numerical optimization algorithms the parameters are updated in an iterative manner. In deep learning we typically use various versions of gradient descent:

1. Pick an initialization θ_0.
2. Update the parameters as $\theta_{t+1} = \theta_t - \gamma \nabla_\theta J(\theta_t)$ for $t = 1, 2, \ldots$. (4.24)
3. Terminate when some criterion is fulfilled, and take the last θ_t as $\widehat{\theta}$.

In many applications of deep learning we cannot afford to compute the exact gradient $\nabla_\theta J(\theta_t)$ at each iteration. Instead we use approximations, which are explained in Section 4.4.2. In Section 4.4.3 strategies how to tune the learning rate γ are presented and in Section 4.4.4 a popular regularization method called dropout is described. First, however, a few words on how to initialize the training.

4.4.1. Initialization

The previous optimization problems (LASSO (2.29), logistic regression (3.8)) that we have encountered have all been convex. This means that we can guarantee global convergence regardless of what initialization θ_0 we use.

In contrast, the cost functions for training neural networks is usually non-convex. This means that the training is sensitive to the value of the initial parameters. Typically, we initialize all the parameters to small random numbers such that we ensure that the different hidden units encode different aspects of the data. If the ReLU activation functions are used, offset elements b_0 are typically initialized to a small positive value such that it operates in the non-negative range of the ReLU.

4.4.2. Stochastic gradient descent

Many problems that are addressed with deep learning contain more than a million training data points n, and the design of the neural network is typically made such that θ has more than a million elements. This provides a computational challenge.

A crucial component is the computation of the gradient required in the optimization routine (4.24)

$$\nabla_\theta J(\theta) = \frac{1}{n}\sum_{i=1}^{n} \nabla_\theta L(x_i, y_i, \theta). \tag{4.25}$$

If the number of data points n is big, this operation is costly. However, we can often assume that the data set is redundant meaning that many of the data points are similar. Then the gradient based on the first half of the dataset $\nabla_\theta J(\theta) \approx \sum_{i=1}^{\frac{n}{2}} \nabla_\theta L(x_i, y_i, \theta)$ is almost identical to the gradient based on the second half of the dataset $\nabla_\theta J(\theta) \approx \sum_{i=\frac{n}{2}+1}^{n} \nabla_\theta L(x_i, y_i, \theta)$. Consequently, it is a waste of time of compute the gradient based on the whole data set. Instead, we could compute the gradient based on the first half of the data set, update the parameters, and then get the gradient for the new parameters based on the second half of the data,

$$\theta_{t+1} = \theta_t - \gamma \frac{1}{n/2} \sum_{i=1}^{\frac{n}{2}} \nabla_\theta L(x_i, y_i, \theta_t), \tag{4.26a}$$

$$\theta_{t+2} = \theta_{t+1} - \gamma \frac{1}{n/2} \sum_{i=\frac{n}{2}+1}^{n} \nabla_\theta L(x_i, y_i, \theta_{t+1}). \tag{4.26b}$$

These two steps would only require roughly half the computational time in comparison to if we had used the whole data set for each gradient computation.

The extreme version of this strategy would be to use only a single data point each time when computing the gradient. However, most commonly when training a deep neural network we do something in between, using more than one data point but not all data points when computing the gradient We use a smaller set of training data called a *mini-batch*. Typically, a mini-batch contains $n_b = 10$, $n_b = 100$ or $n_b = 1000$ data points.

One important aspect when using mini-batches is that the different mini-batches are balanced and representative for the whole data set. For example, if we have a big training data set with a few different classes and the data set is sorted after the classes (i.e. samples belonging to class $k = 1$ are first, and so on), a mini-batch with first n_b samples would only include one class and hence not give a good approximation of the gradient for the full data set.

For this reason, we prefer to draw n_b training data points *at random* from the training data to form a mini-batch. One implementation of this is to first randomly shuffle the training data, before dividing it into mini-batches in an order manner. One complete pass through the training data is called an *epoch*. When we have completed one epoch, we do another random reshuffling of the training data and do another pass though the data set. We call this procedure *stochastic gradient descent* or *mini-batch gradient descent*. A pseudo algorithm is presented in Algorithm 1.

Since the neural network model is a composition of multiple layers, the gradient of the loss function with respect to all the parameters $\nabla_\theta L(x_i, y_i, \theta)\Big|_{\theta=\theta_t}$ can be analytically and efficiently computed by applying the chain rule of differentiation. This is called back-propagation and is not described further here. The interested reader can find more in, for example, Goodfellow, Bengio, and Courville 2016, Section 6.5.

Algorithm 1 Mini-batch gradient descent

1. Initialize all the parameters θ_0 in the network and set $t \leftarrow 1$.
2. For $i = 1$ to E
   a) Randomly shuffle the training data $\{(x_i, y_i)\}_{i=1}^{n}$.
   b) For $j = 1$ to $\frac{n}{n_b}$
      i. Approximate the gradient of the loss function using the mini-batch $\{(x_i, y_i)\}_{i=(j-1)n_b+1}^{jn_b}$, $\hat{g}_t = \frac{1}{n_b} \sum_{i=(j-1)n_b+1}^{jn_b} \nabla_\theta L(x_i, y_i, \theta)\Big|_{\theta=\theta_t}$.
      ii. Do a gradient step $\theta_{t+1} = \theta_t - \gamma \hat{g}_t$.
      iii. Update the iteration index $t \leftarrow t + 1$.

4.4.3. Learning rate

An important tuning parameter for (stochastic) gradient descent is the *learning rate* γ. The learning rate γ decides the length of the gradient step that we take at each iteration. If we use a too low learning rate, the estimate θ_t from one iteration to the next will not change much and the learning will progress slower than necessarily. This is illustrated in Figure 4.12a for a small optimization problem with only one parameter θ.

In contrast, with a too big learning rate, the estimate will pass the optimum and never converge since the step is too long, see Figure 4.12b. For a learning rate which neither is too slow nor too fast, convergence is achieved in a reasonable amount of iterations. A good strategy to find a good learning rate is:

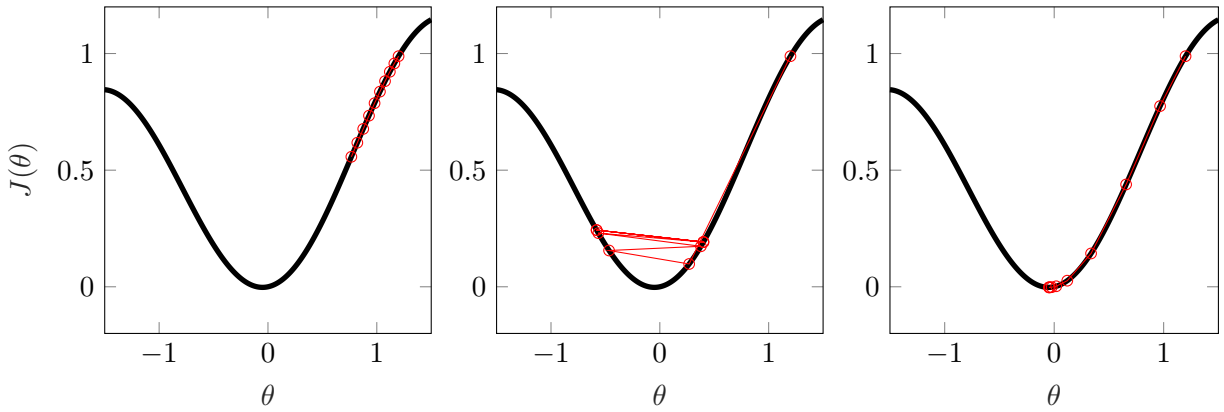
- if the error keeps getting worse or oscillates widely, reduce the learning rate
- if the error is fairly consistently but slowly increasing, increase the learning rate.

Convergence with gradient descent can be achieved with a constant learning rate since the gradient itself approaches zero when we reach the optimum, hence also the gradient step $\gamma \nabla_\theta J(\theta)|_{\theta=\theta_t}$. However, this is not true for stochastic gradient descent since the gradient $\hat{g}_t$ is only an approximation of the true gradient $\nabla_\theta J(\theta)|_{\theta=\theta_t}$, and $\hat{g}_t$ will not necessarily approach 0 as $J(\theta)$ approaches its minimum. As a consequence, we will make a too big updates as we start approaching the optimum and the stochastic gradient algorithm will not converge. In practice, we instead adjust the learning rate. We start with a fairly high learning rate and then decay the learning rate to a certain level. This can, for example, be achieved by the rule

$$\gamma_t = \gamma_{\text{min}} + (\gamma_{\text{max}} - \gamma_{\text{min}}) e^{-\frac{t}{\tau}}. \tag{4.27}$$

Here the learning rate starts at γ_{max} and goes to γ_{min} as $t \to \infty$. How to pick the parameters γ_{min}, γ_{max} and τ is a more of an art than science. As a rule of thumb γ_{min} can be chosen approximately as 1% of γ_{max}. The parameter τ depends on the size of the data set and the complexity of the problem, but should be chosen such that multiple epochs have passed before we reach γ_{min}. The strategy to pick γ_{max} can be the same as for normal gradient descent explained above.

Under certain regularity conditions and if the so called Robbins-Monro condition holds: $\sum_{t=1}^{\infty} \gamma_t = \infty$ and $\sum_{t=1}^{\infty} \gamma_t^2 < \infty$, then stochastic gradient descent converges almost surely to a local minimum. However, to be able to satisfy the Robbins-Monro condition we need $\gamma_t \to 0$ as $t \to \infty$. In practice this it typically not the case and we instead let the learning rate approach a non-zero level $\gamma_{\text{min}} > 0$ by using a scheme like the one in (4.27). This has been found to work better in practice in many situations, despite sacrificing the theoretical convergence of the algorithm.



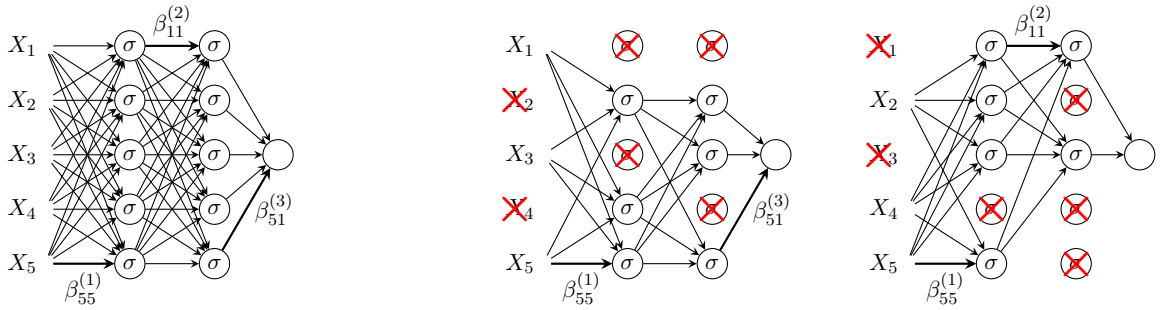
(a) Low learning rate $\gamma = 0.05$ (b) High learning rate $\gamma = 1.2$ (c) Good learning rate $\gamma = 0.3$

Figure 4.12.: Optimization using gradient descent of a cost function $J(\theta)$ where θ 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).

4.4.4. Dropout

Like all models presented in this course, neural network models can suffer from overfitting if we have a too flexible model in relation to the complexity of the data. Bagging (James et al. 2013, Chapter 8.2) is one way to reduce the variance and by that also the overfitting of the model. In bagging we train an entire *ensemble* of models. We train all models (ensemble members) on a different data set each, which has been bootstrapped (sampled with replacement) from the original training data set. To make a prediction, we first make one prediction with each model (ensemble member), and then average over all models to obtain the final prediction.

Bagging is also applicable to neural networks. However, it comes with some practical problems; a large neural network model usually takes quite some time to train and it also has quite some parameters to store. To train not just one but an entire ensemble of many large neural networks would thus be very costly, both in terms of runtime and memory. *Dropout* (Srivastava et al. 2014) is 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 4.13.: 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 neural network like the one in Figure 4.13a. In dropout we construct the equivalent to an ensemble member by randomly removing some of the hidden units. We say that we drop the units, hence the name dropout. By this we achieve a sub-network of our original network. Two such sub-networks are displayed in Figure 4.13b. We randomly sample with a pre-defined probability which units to drop, and the collection of dropped units in

one sub-network is independent from the collection of dropped units in another sub-network. When a unit is removed, also all its incoming and outgoing connections are removed. Not only hidden units can be dropped, but also input variables.

Since all sub-networks are of the very same original network, the different sub-networks share some parameters with each other. For example, in Figure 4.13b the parameter $\beta_{55}^{(1)}$ is present in both sub-networks. The fact that they share parameters with each other allow us to train the ensemble of sub-networks in an efficient manner.

Training with dropout

To train with dropout we use the mini-batch gradient descent algorithm described in Algorithm 1. In each gradient step a mini-batch of data is used to compute an approximation of the gradient, as before. 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 as if the dropped units were not present and then do a gradient step. This gradient step only updates the parameters present in the sub-network. The parameters that are not present are left untouched. In the next gradient step we grab 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.

Dropout vs bagging

This procedure to generate an ensemble of models differs from bagging in a few ways:

- 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 model is trained until convergence. In dropout each sub-network is only trained for a singe gradient step. However, since they share parameters all models will be updated also when the other networks are trained.
- Similar to bagging, in dropout we train each model on a data set that has been randomly selected from our training data. However, in bagging we usually do it on a bootstrapped version of the whole data set 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 overfittning and reducing the variance of the model.

Prediction at test time

After we have trained the sub-networks, we want to make a prediction based on an unseen input data point $X = x_*$. In bagging we evaluate all the different models in the ensemble and combine their results. This would be infeasible in dropout due to the very large (combinatorial) number of possible sub-networks. However, there is a simple trick to approximately achieve the same result. Instead of evaluating all possible sub-networks we simply evaluate the full network containing all the parameters. To compensate for the fact that the model was trained with dropout, we multiply each estimated parameter going out from a unit with the probability of that unit being included during training. This ensures that the expected value of the input to a unit is the same during training and testing, as during training only a fraction of the incoming links were active. For instance, assume that we during training kept a unit with probability p in all layers, then during testing we multiply all estimated parameters with p before we do a prediction based on network. This is illustrated in Figure 4.14. This procedure of approximating the average over all ensemble members has been shown to work surprisingly well in practice even though there is not yet any solid theoretical argument for the accuracy of this approximation.

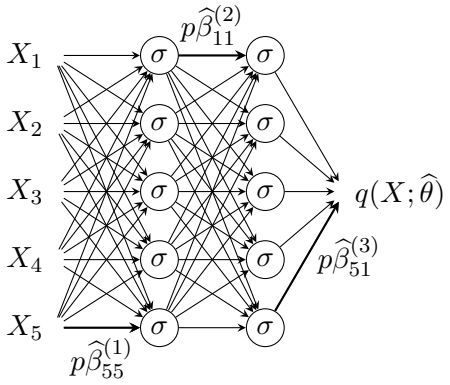


Figure 4.14.: The network used for prediction after being trained with dropout. All units and links are present (no dropout) but the weights going out from a certain unit is multiplied with the probability of that unit being included during training. This is to compensate for the fact that some of them where dropped during training. Here all units have been kept with the probability p during training (and dropped with the probability $1 - p$).

Dropout as a regularization method

As a way to reduce the variance and avoid overfitting, dropout can be seen as a regularization method. There are plenty of other regularization methods for neural networks including parameter penalties (like we did in ridge regression and LASSO in Section 2.5.1 and 2.5.2), early stopping (you stop the training before the parameters have converged, and thereby avoid overfitting) and various sparse representations (for example CNNs can be seen as a regularization method where most parameters are forced to be zero) to mention a few. Since its invention, dropout has become one of the most popular regularization techniques due to its simplicity, computationally cheap training and testing procedure and its good performance. In fact, a good practice of designing a neural network is often to extended the network until you see that it starts overfitting, extended it a bit more and add a regularization like dropout to avoid that overfitting.

4.5. Perspective and 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 so-called back-propagation algorithm. At that stage, neural networks could, for example, be used to classify handwritten digits from low-resolution images (LeCun, Boser, et al. 1990). However, in the late 1990s neural networks were largely forsaken because it was widely thought that they could not be used to solve any challenging problems in computer vision and speech recognition. In these areas, neural networks could not compete with hand-crafted solutions based on domain specific prior knowledge.

This picture has changed dramatically since the late 2000s, with multiple layers under the name deep learning. Progress in software, hardware and algorithm parallelization made it possible to address more complicated problems, which were unthinkable only a couple of decades ago. For example, in image recognition, these deep models are now the dominant methods of use and they reach almost human performance on some specific tasks (LeCun, Bengio, and Hinton 2015). Recent advances based on deep neural networks have generated algorithms that can learn how to play computer games based on pixel information only (Mnih et al. 2015), and automatically understand the situation in images for automatic caption generation (Xu et al. 2015).

A fairly recent and accessible introduction and overview of deep learning is provided by LeCun, Bengio, and Hinton (2015), and a recent textbook by Goodfellow, Bengio, and Courville (2016).

5. Boosting

In the previous chapters we have introduced some fundamental methods for machine learning. We now turn our attention to a technique referred to as *boosting*, which is of a slightly different flavor. Boosting is a meta-algorithm, in the sense that it is a method composed of other methods. We start by explaining the high-level idea in Section 5.1, followed by a detailed derivation of an early boosting method called AdaBoost in Sections 5.2 and 5.3. Finally, in Section 5.5 we discuss some improvements and extensions.

5.1. The conceptual idea

Boosting is built on the idea that even a simple (or weak) regression or classification model often can 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 turn an *ensemble* of weak models into one strong model.

Boosting shares some similarities with bagging, see James et al. 2013, Section 8.2. Both are ensemble methods, i.e. 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 throughout this chapter.

The first key difference is in the construction of the ensemble. In bagging we construct B models in parallel. These models are random (based on randomly bootstrapped datasets) but they are *identically distributed*. Consequently, there is no reason to trust one model more than another, and the final prediction of the bagged model is based on a plain average or majority vote of the individual predictions of the ensemble members.

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 data set at each iteration in order to put more emphasis on the data points for which the model (so far) has performed poorly. In the subsequent sections we will see exactly how this is done for a specific boosting algorithm known as AdaBoost (Freund and Schapire 1996). First, however, we consider a simple example of to illustrate the idea.

Example 5.1: Boosting illustration

We consider a toy binary classification problem with two input variables, X_1 and 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). A decision stump means that we select one of the input variables, X_1 or X_2, and split the input space into two half spaces, in order to minimize the training error. This results in a decision boundary that is perpendicular to one of the axes. The left panel of Figure 5.1 shows the training data, illustrated by red crosses and blue dots for the two classes, respectively. The colored regions shows the decision boundary for a decision stump $\widehat{G}^1(X)$ trained on this data.

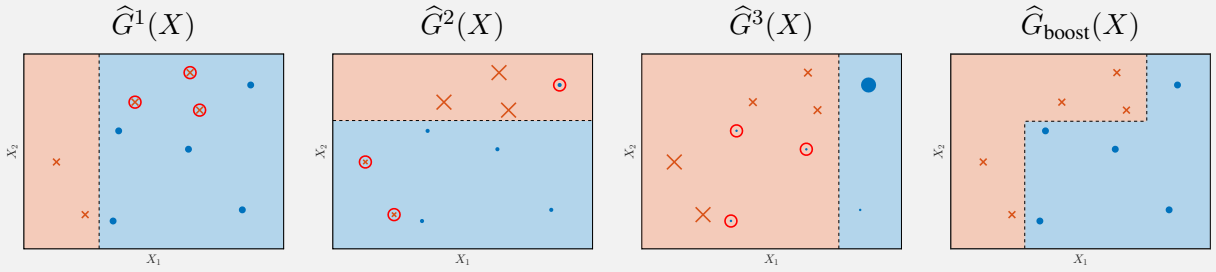


Figure 5.1.: Three iterations of AdaBoost for toy problem. See text for details.

The model $\widehat{G}^1(X)$ misclassifies three data points (red crosses falling in the blue region), which are encircled 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 this end, we train another decision stump, $\widehat{G}^2(X)$, on the same data. To put emphasis on the three misclassified points, however, we assign *weights* $\{w_i^2\}_{i=1}^n$ to the data. Points correctly classified by $\widehat{G}^1(X)$ are down-weighted, whereas the three points misclassified by $\widehat{G}^1(X)$ are up-weighted. This is illustrated in the second panel of Figure 5.1, where the marker sizes have been scaled according to the weights. The classifier $\widehat{G}^2(X)$ is then found by minimizing the *weighted* misclassification error, $\frac{1}{n}\sum_{i=1}^n w_i^2 I(y_i \neq \widehat{G}^2(x_i))$, resulting in the decision boundary shown in the second panel. This procedure is repeated for a third and final iteration: we update the weights based on the hits and misses of $\widehat{G}^2(X)$ and train a third decision stump $\widehat{G}^3(X)$ shown in the third panel. The final classifier, $\widehat{G}_{\text{boost}}(X)$ is then taken as a combination of the three decision stumps. Its (nonlinear) decision boundaries are shown in the right panel.

5.2. Binary classification, margins, and exponential loss

Before diving into the details of the AdaBoost algorithm we will lay the groundwork by introducing some notations and concepts that will be used in its derivation. AdaBoost was originally proposed for binary classification ($K = 2$) and we will restrict our attention to this setting. That is, the output Y can take two different values which, in this chapter, we encode as -1 and $+1$. This encoding turns out to be mathematically convenient in the derivation of AdaBoost. However, it is important to realize that the encoding that we choose for the two classes is arbitrary and all the concepts defined below can be generalized to any binary encoding (e.g. $\{0, 1\}$ which we have used before).

Let $G(X) = \text{sign}\{C(X)\}$, which can be seen as a classifier constructed by thresholding some real-valued function $C(X)$ at 0. This is a common situation and, in particular, it is the case for the AdaBoost algorithm presented below. Note that the decision boundary is given by values of X for which $C(X) = 0$. For simplicity of presentation we will assume that no data points fall exactly on the decision boundary (which always gives rise to an ambiguity), so that we can assume that $G(X)$ as defined above is always either -1 or $+1$.

Based on the function $C(X)$ we define the *margin* of the classifier as

$$Y \cdot C(X). \tag{5.1}$$

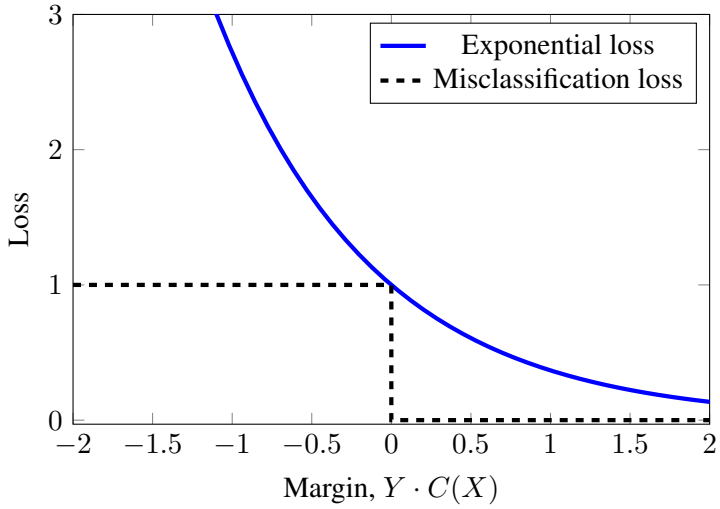


Figure 5.2.: Exponential loss.

It follows that if Y and $C(X)$ have the same sign, i.e. if the classification is correct, then the margin is positive. Analogously, for an incorrect classification Y and $C(X)$ will have different signs and the margin is negative. More specifically, since Y is either -1 or $+1$, the margin is simply $|C(X)|$ for correct classifications and $-|C(X)|$ for incorrect classifications. The margin can thus be viewed as a measure of certainty in a prediction, where values with small margin in some sense (not necessarily Euclidian!) are close to the decision boundary. The margin plays a similar role for binary classification as the residual $Y - f(X)$ does for regression.

Loss functions for classification can be defined in terms of the margin, by assigning small loss to positive margins and large loss to negative margins. One such loss function is the *exponential loss* defined as

$$L(Y, C(X)) = \exp(-YC(X)). \tag{5.2}$$

Figure 5.2 illustrates the exponential loss and compares it against the misclassification loss, which is simply $I(YC(X) < 0)$.

Remark 5.1. The misclassification loss is often used to evaluate the performance of a classifier (in particular if interest only lies in the number of correct and incorrect classification). However, it is typically not suitable to use directly during training of the model. The reason is that it is discontinuous which is problematic in a numerical minimization of the training loss. The exponential loss function, on the other hand, is both convex and (infinitely many times) differentiable. These are nice properties to have when optimizing the training loss. In fact, the exponential loss can in this way be seen as a more convenient proxy for misclassification loss during training. Other loss functions of interest are discussed in Section 5.5.

5.3. AdaBoost

We are now ready to derive a practical boosting method, the AdaBoost (Adaptive Boosting) algorithm proposed by Freund and Schapire (1996). AdaBoost was the first successful practical implementation of the boosting idea and lead the way for its popularity. Freund and Schapire were awarded the prestigious Gödel Prize in 2003 for their algorithm.

Recall from the discussion above that boosting attempts to construct a sequence of B (weak) classifiers $\widehat{G}^1(X)$, $\widehat{G}^2(X)$, …, $\widehat{G}^B(X)$. Any classification model can in principle be used to construct these so called *base classifiers*—shallow classification trees are common in practice (see Section 5.4 for further discussion). The individual predictions of the B ensemble members are then combined into a final prediction. However, 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 according to

$$\widehat{G}^B_{\text{boost}}(X) = \text{sign}\left\{\sum_{b=1}^{B} \alpha^b \widehat{G}^b(X)\right\}. \tag{5.3}$$

Note that each ensemble member votes either -1 or $+1$. The output from the boosted classifier is $+1$ if the weighted sum of the individual votes is positive and -1 if it is negative.

How, then, do we compute the individual ensemble members and their coefficients? The answer to this question depends on the specific boosting method that is used. For AdaBoost this is done by greedily minimizing the exponential loss of the boosted classifier at each iteration. Note that we can write the boosted classifier after b iterations as $\widehat{G}^b_{\text{boost}} = \text{sign}\{C^b(X)\}$ where $C^b(X) = \sum_{j=1}^{b} \alpha^j \widehat{G}^j(X)$. Furthermore, we can express the function $C^b(X)$ sequentially as

$$C^b(X) = C^{b-1}(X) + \alpha^b \widehat{G}^b(X), \tag{5.4}$$

initialized with $C^0(X) \equiv 0$. Since we assume that the ensemble members are constructed sequentially, when at iteration b of the procedure the function $C^{b-1}(X)$ is known and fixed. Thus, what remains to be computed at iteration b is the coefficient α^b and the ensemble member $\widehat{G}^b(X)$. This is done by minimizing the exponential loss of the training data,

$$(\alpha^b, \widehat{G}^b) = \underset{(\alpha,G)}{\arg\min} \sum_{i=1}^{n} L(y_i, C^b(x_i)) \tag{5.5a}$$

$$= \underset{(\alpha,G)}{\arg\min} \sum_{i=1}^{n} \exp\left(-y_i \left\{C^{b-1}(x_i) + \alpha G(x_i)\right\}\right) \tag{5.5b}$$

$$= \underset{(\alpha,G)}{\arg\min} \sum_{i=1}^{n} w_i^b \left(-y_i \alpha G(x_i)\right) \tag{5.5c}$$

where for the first equality we have used the definition of the exponential loss function (5.2) and the sequential structure of the boosted classifier (5.4). For the second equality we have defined the quantities

$$w_i^b \overset{\text{def}}{=} \exp\left(-y_i C^{b-1}(x_i)\right), \qquad i = 1, \ldots, n, \tag{5.6}$$

which can be interpreted as weights for the individual data points in the training data set. Note that these weights are independent of α and G and can thus be viewed as constants when solving the optimization problem (5.5c) at iteration b of the boosting procedure (the fact that we keep previous coefficients and ensemble members fixed is what makes the optimization "greedy").

To solve (5.5) we start by writing the objective function as

$$\sum_{i=1}^{n} w_i^b \exp\left(-y_i \alpha G(x_i)\right) = e^{-\alpha} \underbrace{\sum_{i=1}^{n} w_i^b I(y_i = G(x_i))}_{=W_c} + e^{\alpha} \underbrace{\sum_{i=1}^{n} w_i^b I(y_i \neq G(x_i))}_{=W_e}, \tag{5.7}$$

where we have used the indicator function to split the sum into two sums: the first ranging over all training data points correctly classified by G and the second ranging over all point erroneously classified by G. Furthermore, for notational simplicity we define W_c and W_e for the sum of weights of correctly classified and erroneously classified data points, respectively. Furthermore, let $W = W_c + W_e$ be the total weight sum, $W = \sum_{i=1}^{n} w_i^b$.

Minimizing (5.7) is done in two stages, first w.r.t. G and then w.r.t. α. This is possible since the minimizing argument in G turns out to be independent of the actual value of $\alpha > 0$. To see this, note that we can write the objective function (5.7) as

$$e^{-\alpha} W + (e^{\alpha} - e^{-\alpha}) W_e. \tag{5.8}$$

Since the total weight sum W is independent of G and since $e^{\alpha} - e^{-\alpha} > 0$ for any $\alpha > 0$, minimizing this expression w.r.t. G is equivalent to minimizing W_e w.r.t. G. That is,

$$\widehat{G}^b = \arg\min_G \sum_{i=1}^{n} w_i^b I(y_i \neq G(x_i)). \tag{5.9}$$

In words, the bth ensemble member should be selected by minimizing the *weighted misclassification loss*, where each data point (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.

How the problem (5.9) is solved in practice depends on the choice of base classifier that we use, i.e. on the specific restrictions that we put on the function G (for example a shallow classification tree). However, since (5.9) is essentially a standard classification objective it can be solved, at least approximately, by standard learning algorithms. Incorporating the weights in the objective function is straightforward for most base classifiers, since it simply boils down to weighting the individual terms of the loss function used when training the base classifier.

When the bth ensemble member, $\widehat{G}^b(X)$, has been found by solving (5.9) it remains to compute its coefficient α^b. Recall that this is done by minimizing the objective function (5.8). Differentiating this expression w.r.t. α and setting the derivative to zero we get the equation

$$\begin{aligned} -\alpha e^{-\alpha} W + \alpha \left(e^{\alpha} + e^{-\alpha}\right) W_e &= 0 \\ \iff W &= \left(e^{2\alpha} + 1\right) W_e \\ \iff \alpha &= \frac{1}{2} \log\left(\frac{W}{W_e} - 1\right). \end{aligned} \tag{5.10}$$

Thus, by defining

$$\overline{\text{err}}^b \overset{\text{def}}{=} \frac{W_e}{W} = \sum_{i=1}^{n} \frac{w_i^b}{\sum_{j=1}^{n} w_j^b} I(y_i \neq \widehat{G}^b(x_i)) \tag{5.11}$$

to be the weighted misclassification error for the bth classifier, we can express the optimal value for its coefficient as

$$\alpha^b = \frac{1}{2} \log\left(\frac{1 - \overline{\text{err}}^b}{\overline{\text{err}}^b}\right). \tag{5.12}$$

This completes the derivation of the AdaBoost algorithm, which is summarized in Algorithm 2. In the algorithm we exploit the fact that the weights (5.6) can be computed recursively by using the expression (5.4); see line 2. Furthermore, we have added an explicit weight normalization (line 2) which is convenient in practice and which does not affect the derivation of the method above.

Remark 5.2. One detail worth commenting is that the derivation of the AdaBoost procedure 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 (5.12), note that the function $\log((1-x)/x)$ is positive for any $0 < x < 0.5$. Thus, α^b will be positive as long as the weighted training error for the bth classifier, $\overline{\text{err}}^b$, 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 $\overline{\text{err}}^b$ is the *training* error). Indeed, if $\overline{\text{err}}^b > 0.5$, then we could simply flip the sign of all predictions made by $\widehat{G}^b(X)$ to reduce the error!

In the method discussed above we have assumed that each base classifier outputs a discrete class prediction, $\widehat{G}^b(X) \in \{-1, 1\}$. However, many classification models used in practice are in fact based on estimating the conditional class probabilities. Hence, it is possible to instead let each base model output a real number and use these numbers as the basis for the "vote". This extension of Algorithm 2 is discussed by Friedman, Hastie, and Tibshirani 2000 and is referred to as Real AdaBoost.

Algorithm 2 AdaBoost

1. Assign weights $w_i^1 = 1/n$ to all data points.
2. For $b = 1$ to B
   (a) Train a weak classifier $\widehat{G}^b(X)$ on the weighted training data $\{(x_i, y_i, w_i^b)\}_{i=1}^n$.
   (b) *Update the weights* $\{w_i^{b+1}\}_{i=1}^n$ *from* $\{w_i^b\}_{i=1}^n$*:*
      i. Compute $\overline{\text{err}}^b = \sum_{i=1}^n w_i^b I(y_i \neq \widehat{G}^b(x_i))$
      ii. Compute $\alpha^b = 0.5 \log((1 - \overline{\text{err}}^b)/\overline{\text{err}}^b)$.
      iii. Compute $w_i^{b+1} = w_i^b \exp(-\alpha^b y_i \widehat{G}^b(x_i))$, $i = 1, \ldots, n$
      iv. *Normalize.* Set $w_i^{b+1} \leftarrow w_i^{b+1} / \sum_{j=1}^n w_j^{b+1}$, for $i = 1, \ldots, n$.
3. Output $\widehat{G}^B_{\text{boost}}(X) = \text{sign}\left\{\sum_{b=1}^B \alpha^b \widehat{G}^b(X)\right\}$.

5.4. Boosting vs. bagging: base models and ensemble size

AdaBoost, an in fact any boosting algorithm, has two important design choices, *(i)* which base classifier to use, an *(ii)* how many iterations B to run the boosting algorithm for. As previously pointed out, we can use essentially any classification method as base classifier. However, the most common choice in practice is to use a shallow classification tree, or even a decision stump (i.e., a tree of depth one; see example 5.1). This choice is guided by the fact that the boosting algorithm can learn a good model despite using very weak base classifiers, and shallow trees can be trained quickly. In fact, using deep (high-variance) classification trees as base classifiers typically deteriorates performance compared to using shallow trees. More specifically, the depth of the tree should be chosen to obtain a desired degree of interactions between input variables. A tree with M terminal nodes is able to model functions depending on maximally $M - 1$ of the input variables; see Hastie, Tibshirani, and Friedman 2009, Chapter 10.11 for a more in-depth discussion on this matter.

The fact that boosting algorithms often use shallow trees as base classifiers is a clear distinction from the (somewhat similar) bagging method (see James et al. 2013, Chapter 8.2). Bagging is a pure variance reduction technique based on averaging and it can not reduce the bias of the individual base models. Hence, for bagging to be successful we need to use base models with low bias (but possibly high variance)—typically very deep decision trees. Boosting on the other hand can reduce both the variance *and* the bias of the base models, making it possible to use very simple base models as described above.

Another important difference between boosting and bagging is that the former is sequential whereas the latter is parallel. Each iteration of a boosting algorithm introduces a new base model aiming at reducing the errors made by the current model. In bagging, on the other hand, all base models are identically distributed and they all try to solve the same problem, with the final model being a simple average over the ensemble. An effect of this is that bagging *does not* overfit as the number of ensemble members B tend to infinity. Unfortunately, this is not the case for boosting. Indeed, a boosting model becomes more and more flexible as the number of iterations B increase and using too many base models can result in overfitting. 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 so called early stopping (this is also common in the context of training neural networks; see Section 4.4.4).

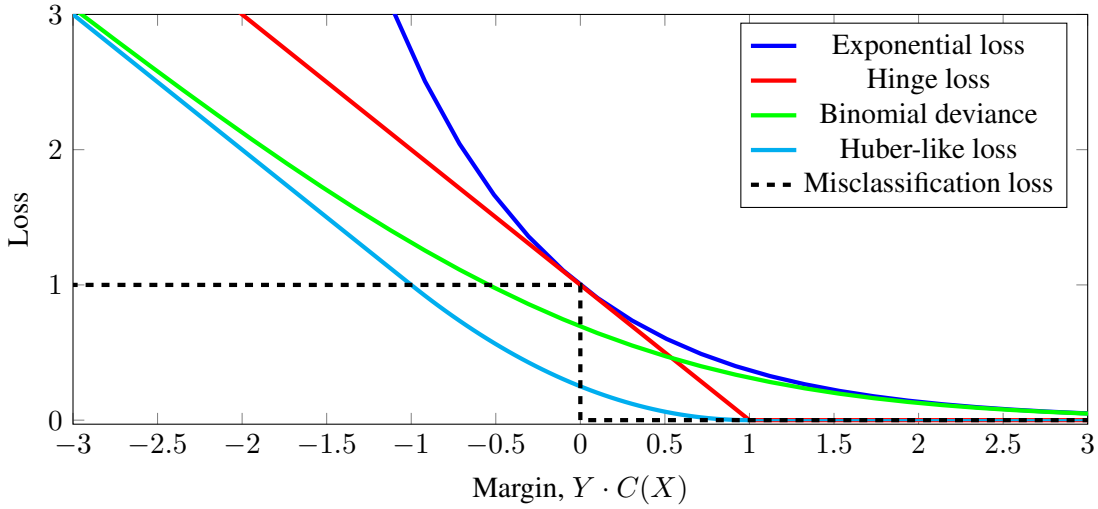


Figure 5.3.: Comparison of common loss functions for classification.

5.5. Robust loss functions and gradient boosting

As pointed out above, the margin $Y \cdot C(X)$ can be used as a measure of the error made by the classifier $G(X) = \text{sign}\{C(X)\}$, where negative margins correspond to incorrect classifications and positive margins correspond to correct classifications. It is therefore natural to use a loss function which is a decreasing function of the margin: negative margins should be penalized more than positive margins. The exponential loss function (5.2)—which was used in the derivation of the AdaBoost algorithm—satisfies this requirement, as can be seen in Figure 5.2. However, it is also evident that this loss function penalizes negative margins very heavily. This can be an issue in practical applications, making the classifier sensitive to noisy data and "outliers", such as mislabeled or atypical data points.

To address these limitations we can consider using some other, more *robust*, loss function in place of the exponential loss. A few examples of commonly used loss functions for classification are shown in Figure 5.3 (see appendix C for the mathematical definitions of these functions). An in-depth discussion of the rationale and pros and cons of these different loss functions is beyond the scope of these lecture notes and we refer the interested reader to Hastie, Tibshirani, and Friedman 2009, Chapter 10.6. However, we note that all the alternative loss functions illustrated in the figure have less "aggressive" penalties for large negative margins compared to the exponential loss, i.e., their slopes are not as sharp,[1] making them more robust to noisy data.

Why then have we not used a more robust loss function in the derivation of the AdaBoost algorithm? The reason for this is mainly computational. Using exponential loss is convenient since it leads to a closed form solution to the optimization problem in (5.5). If we instead use another loss function this analytical tractability is unfortunately lost.

However, this difficulty can be dealt with by using techniques from numerical optimization. This approach is complicated to some extent by the fact that the optimization "variable" in (5.5a) is the base classifier $G(X)$ itself. Hence, it is not possible to simply use an off-the-shelf numerical optimization algorithm to solve this problem. That being said, however, it has been realized that it is possible to approximately solve (5.5a) for rather general loss function using a method reminiscent of gradient descent (appendix B). The resulting method is referred to as *gradient boosting* Friedman 2001; Mason et al. 1999. We provide pseudo-code for one instance of a gradient boosting method in Algorithm 3. As can be seen from the algorithm, the key step involves fitting a base model to the *negative gradient of the loss function*. This can be understood via the intuitive interpretation of boosting, that each base model should try to correct the mistakes made by the ensemble thus far. The negative gradient of the loss function gives an indication of in which "direction" the model should be updated in order to reduce the loss.

[1] Hinge loss, binomial deviance, and the Huber-like loss all increase linearly for large negative margins. Exponential loss, of course, increases exponentially.

Algorithm 3 A gradient boosting algorithm

1. Initialize (as a constant), $C^0(X) \equiv \arg\min_c \sum_{i=1}^n L(y_i, c)$.
2. For $b = 1$ to B
   (a) Compute the negative gradient of the loss function,
   $$g_i^b = -\left[\frac{\partial L(y_i, c)}{\partial c}\right]_{c=C^{b-1}(x_i)}, \quad i = 1, \ldots, n.$$
   (b) Train a base *regression* model $\widehat{f}^b(X)$ to fit the gradient values,
   $$\widehat{f}^b = \arg\min_f \sum_{i=1}^n \left(f(x_i) - g_i^b\right)^2.$$
   (c) Update the boosted model,
   $$C^b(X) = C^{b-1} + \gamma \widehat{f}^b(X)$$
3. Output $\widehat{G}^B_{\text{boost}}(X) = \text{sign}\{C^B(X)\}$.

While presented for classification in Algorithm 3, gradient boosting can also be used for regression with minor modifications. In fact, an interesting aspect of the algorithm presented here is that the base models $\widehat{f}^b(X)$ are found by solving a *regression problem* despite the fact that the algorithm produces a classifier. The reason for this is that the negative gradient values $\{g_i^b\}_{i=1}^n$ are quantitative variables, even if the data $\{y_i\}_{i=1}^n$ is qualitative. Here we have considered fitting a base model to these negative gradient values by minimizing a square loss criterion.

The value γ used in the algorithm (line 2(c)) is a tuning parameter which plays a similar role to the step size in ordinary gradient descent. In practice this is usually found by line search (see appendix B), often combined with a type of regularization via shrinkage (Friedman 2001). When using trees as base models—as is common in practice—optimizing the steps size can be done jointly with finding the terminal node values, resulting in a more efficient implementation (Friedman 2001).

As mentioned above, gradient boosting requires a certain amount of smoothness in the loss function. A minimal requirement is that it is almost everywhere differentiable, 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 binomial deviance (see Figure 5.3) is in this respect a "safe choice" which is infinitely differentiable and strongly convex, while still enjoying good statistical properties. As a consequence, binomial deviance is one of the most commonly used loss functions in practice.

5.6. Implementations of boosting

There are several R packages implementing boosting algorithms. AdaBoost is available in for instance `adabag` and `gbm`. The latter package also provides some gradient boosting models. The most popular implementation of gradient boosting is perhaps XGBoost (Chen and Guestrin 2016), which has won several Kaggle competitions over the past few years. XGBoost uses various tricks for improving the speed and performance (for instance, regularization and second order optimization). It is available in R via the package `xgboost`. A recent competitor to XGBoost, developed as part of the Microsoft Distributed Machine Learning Toolkit, is LightGBM (Ke et al. 2017). This library is currently not available on CRAN, but an R implementation is available on GitHub, see https://github.com/Microsoft/lightGBM.

A. Derivation of the normal equations

The normal equations (2.17)

$$\widehat{\beta} = (\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}\mathbf{X}^\mathsf{T}\mathbf{y}. \tag{A.1}$$

can be derived from (2.16)

$$\widehat{\beta} = \underset{\beta}{\operatorname{argmin}} \ \|\mathbf{X}\beta - \mathbf{y}\|_2^2, \tag{A.2}$$

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

A.1. A calculus approach

Let

$$S(\beta) = \|\mathbf{X}\beta - \mathbf{y}\|_2^2 = (\mathbf{X}\beta - \mathbf{y})^\mathsf{T}(\mathbf{X}\beta - \mathbf{y}) = \mathbf{y}^\mathsf{T}\mathbf{y} - 2\mathbf{y}^\mathsf{T}\mathbf{X}\beta + \beta^\mathsf{T}\mathbf{X}^\mathsf{T}\mathbf{X}\beta, \tag{A.3}$$

and differentiate $S(\beta)$ with respect to the vector β,

$$\frac{\partial}{\partial\beta}S(\beta) = -2\mathbf{X}^\mathsf{T}\mathbf{y} + 2\mathbf{X}^\mathsf{T}\mathbf{X}\beta. \tag{A.4}$$

Since $S(\beta)$ is a positive quadrartic form, its minimum must be attained at $\frac{\partial}{\partial\beta}S(\beta) = 0$, which characterizes the solution $\widehat{\beta}$ as

$$\frac{\partial}{\partial\beta}S(\widehat{\beta}) = 0 \Leftrightarrow -2\mathbf{X}^\mathsf{T}\mathbf{y} + 2\mathbf{X}^\mathsf{T}\mathbf{X}\beta = 0 \Leftrightarrow \mathbf{X}^\mathsf{T}\mathbf{X}\widehat{\beta} = \mathbf{X}^\mathsf{T}\mathbf{y}. \tag{A.5}$$

If $\mathbf{X}^\mathsf{T}\mathbf{X}$ is invertible, this (uniquely) gives

$$\widehat{\beta} = (\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}\mathbf{X}^\mathsf{T}\mathbf{y}, \tag{A.6}$$

i.e., the normal equations. If $\mathbf{X}^\mathsf{T}\mathbf{X}$ is not invertible, the equation (A.5) has infinite many solutions $\widehat{\beta}$, which are all equally good solutions to the problem (A.2).

A.2. A linear algebra approach

Denote the $p+1$ columns of $\mathbf{X}$ as $\mathbf{c}_j, j = 1, \ldots, p+1$. We first show that $\|\mathbf{X}\beta - \mathbf{y}\|_2^2$ is minimzed if β is chosen such that $\mathbf{X}\beta$ is the orthogonal projection of $\mathbf{y}$ onto the (sub)space spanned by the columns $\mathbf{c}_j$ 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 $\mathbf{c}_i$, and $y_\parallel$ is in the subscpace spanned by all columns $\mathbf{c}_i$. Since $\mathbf{y}_\perp$ is orthogonal to both $\mathbf{y}_\parallel$ and $\mathbf{X}\beta$, it follows that

$$\|\mathbf{X}\beta - \mathbf{y}\|_2^2 = \|\mathbf{X}\beta - (\mathbf{y}_\perp + \mathbf{y}_\parallel)\|_2^2 = \|(\mathbf{X}\beta - \mathbf{y}_\parallel) - \mathbf{y}_\perp\|_2^2 \geq \|\mathbf{y}_\perp\|_2^2, \tag{A.7}$$

and the triangle inequality also gives us

$$\|\mathbf{X}\beta - \mathbf{y}\|_2^2 = \|\mathbf{X}\beta - \mathbf{y}_\perp - \mathbf{y}_\parallel\|_2^2 \leq \|\mathbf{y}_\perp\|_2^2 + \|\mathbf{X}\beta - \mathbf{y}_\parallel\|_2^2. \tag{A.8}$$

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

$$(\mathbf{y} - \mathbf{X}\beta)^\mathsf{T}\mathbf{c}_j = 0, j = 1, \ldots, p+1 \tag{A.9}$$

(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 $\mathbf{c}_j$ together form the matrix $\mathbf{X}$, we can write this compactly as

$$(\mathbf{y} - \mathbf{X}\widehat{\beta})^\mathsf{T}\mathbf{X} = 0, \tag{A.10}$$

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

$$\mathbf{X}^\mathsf{T}\mathbf{X}\widehat{\beta} = \mathbf{X}^\mathsf{T}\mathbf{y}, \tag{A.11}$$

from which the normal equations are obtained as from (A.5).

B. Unconstrained numerical optimization

Given a function $J(\theta)$, the *optimization* problem is about finding the value of the variable x for which the function $J(\theta)$ is either minimized or maximized. To be precise it will here be formulated as finding the value $\widehat{\theta}$ that minimizes[1] the function $J(\theta)$ according to

$$\min_{\theta} J(\theta), \tag{B.1}$$

where the vector θ is allowed to be anywhere in $\mathbb{R}^n$, motivating the name *unconstrained* optimization. The function $J(\theta)$ will be referred to as the *cost function*[2], with the motivation that the minimization problem in (B.1) is striving to minimize some cost. We will make the assumption that the cost function $J(\theta)$ is continuously differentiable on $\mathbb{R}^n$. If there are requirements on θ (e.g. that its components θ have to satisfy a certain equation $g(\theta) = 0$) the problem is instead referred to as a constrained optimization problem.

The unconstrained optimization problem (B.1) is ever-present across the sciences and engineering, since it allows us to find the best—in some sense—solution to a particular problem. One example of this arises when we are searching for the parameters in a linear regression problem by finding the parameters that make the available measurements as likely as possible by maximizing the likelihood function. For a linear model with Gaussian noise, this resulted in a least squares problem, for which there is an explicit expression (the normal equations, (2.17)) describing the solution. However, for most optimization problems that we face there are no explicit solutions available, forcing us to use approximate *numerical* methods in solving these problems. We have seen several concrete examples of this kind, for example the optimization problems arising in deep learning and logistic regression. This appendix provides a brief introduction to the practical area of unconstrained numerical optimization.

The key in assembling a working optimization algorithm is to build a simple and useful *model* of the complicated cost function $J(\theta)$ around the current value for θ. The model is often local in the sense that it is only valid in a neighbourhood of this value. The idea is then to exploit this model to select a new value for θ that corresponds to a smaller value for the cost function $J(\theta)$. The procedure is then repeated, which explains why most numerical optimization algorithms are of *iterative* nature. There are of course many different ways in which this can be done, but they all share a few key parts which we outline below. Note that we only aim to provide the overall strategies underlying practical unconstrained optimization algorithms, for precise details we refer to the many textbooks available on the subject, some of which are referenced towards the end.

B.1. A general iterative solution

What do we mean by a solution to the unconstrained minimization problem in (B.1)? The best possible solution is the *global minimizer*, which is a point $\widehat{\theta}$ such that $J(\widehat{\theta}) \leq J(\theta)$ for all $\theta \in \mathbb{R}^n$. The global minimizer is often hard to find and instead we typically have to settle for a local minimizer instead. A point $\widehat{\theta}$ is said to be a *local minimizer* if there is a neighbourhood $\mathcal{M}$ of $\widehat{\theta}$ such that $J(\widehat{\theta}) \leq J(\theta)$ for all $\theta \in \mathcal{M}$.

In our search for a local minimizer we have to start somewhere, let us denote this starting point by θ_0. Now, if θ_0 is not a local minimizer of $J(\theta)$ then there must be an increment d_0 that we can add to θ_0 such that $J(\theta_0 + d_0) < J(\theta_0)$. By the same argument, if $\theta_1 = \theta_0 + d_0$ is not a local minimizer then there must be another increment d_1 that we can add to θ_1 such that $J(\theta_1 + d_1) < J(\theta_1)$. This procedure is repeated until it is no longer possible to find an increment that decrease the value of the objective function. We have then found a local

[1] Note that it is sufficient to cover minimization problem, since any maximization problem can be considered as a minimization problem simply by changing the sign of the cost function.

[2] Throughout the course we have talked quite a lot about different loss functions. These loss functions are examples of cost functions.

minimizer. Most of the algorithms capable of solving (B.1) are iterative procedures of this kind. Before moving on, let us mention that the increment d is often resolved into two parts according to

$$d = \gamma p. \tag{B.2}$$

Here, the scalar and positive parameter γ is commonly referred to as the *step length* and the vector $p \in \mathbb{R}^n$ is referred to as the *search direction*. The intuition is that the algorithm is searching for the solution by moving in the search direction and how far it moves in this this direction is controlled by the step length.

The above development does of course lead to several questions, where the most pertinent are the following:

1. How can we compute a useful search direction p?
2. How big steps should we make, i.e. what is a good value of the step length γ?
3. How do we determine when we have reached a local minimizer, and stop searching for new directions?

Throughout the rest of this section we will briefly discuss these questions and finally we will assemble the general form of an algorithm that is often used for unconstrained minimization.

A straightforward way of finding a general characterization of all search directions p resulting in a decrease in the value of the cost function, i.e. directions p such that

$$J(\theta + p) < J(\theta) \tag{B.3}$$

is to build a local model of the cost function around the point θ. One model of this kind is provided by Taylor's theorem, which builds a local polynomial approximation of a function around some point of interest. A linear approximation of the cost function $J(\theta)$ around the point θ is given by

$$J(\theta + p) \approx J(\theta) + p^{\mathsf{T}} \nabla J(\theta). \tag{B.4}$$

By inserting the linear approximation (B.4) of the objective function into (B.3) we can provide a more precise formulation of how to find a search direction p such that $J(\theta + p) < J(\theta)$ by asking for which p it holds that $J(\theta) + p^{\mathsf{T}} \nabla J(\theta) < J(\theta)$, which can be further simplified into

$$p^{\mathsf{T}} \nabla J(\theta) < 0. \tag{B.5}$$

Inspired by the inequality above we chose a generic description of the search direction according to

$$p = -V \nabla J(\theta), \qquad V \succ 0, \tag{B.6}$$

where we have introduced some extra flexibility via the positive definite scaling matrix V. The inspiration came from the fact that by inserting (B.6) into (B.5) we obtain

$$p^{\mathsf{T}} J(\theta) = -\nabla^{\mathsf{T}} J(\theta) V^{\mathsf{T}} \nabla J(\theta) = -\|\nabla J(\theta)\|^2_{V^{\mathsf{T}}} < 0, \tag{B.7}$$

where the last inequality follows from the positivity of the squared weighted two-norm, which is defined as $\|a\|^2_W = a^{\mathsf{T}} W a$. This shows that $p = -V \nabla J(\theta)$ will indeed result in a search direction that decreases the value of the objective function. We refer to such a search direction as a *descent direction*.

The strategy summarized in Algorithm 4 is referred to as *line search*. Note that we have now introduced subscript t to clearly show the iterative nature. The algorithm searches along the line defined by starting at the current iterate θ_t and then moving along the search direction p_t. The decision of how far to move along this line is made by simply minimizing the cost function along the line

$$\min_{\gamma} J(\theta_t + \gamma p_t). \tag{B.8}$$

Note that this is a one-dimensional optimization problem, and hence simpler to deal with compared to the original problem. The *step length* γ_t that is selected in (B.8) controls how far to move along the current search

Algorithm 4 General form of unconstrained minimization

1. Set $t = 0$.
2. **while** *stopping criteria is not satisfied* **do**
   a) Compute a search direction $p_t = -V_t \nabla J(\theta_t)$ for some $V_t \succ 0$.
   b) Find a step length $\gamma_t > 0$ such that $J(\theta_t + \gamma_t p_t) < J(\theta_t)$.
   c) Set $\theta_{t+1} = \theta_t + \gamma_t p_t$.
   d) Set $t \leftarrow t + 1$.
3. **end while**

direction p_t. It is sufficient to solve this problem approximately in order to find an acceptable step length, since as long as $J(\theta_t + \gamma_t p_t) < J(\theta_t)$ it is not crucial to find the global minimizer for (B.8).

There are several different indicators that can be used in designing a suitable stopping criteria for row 2 in Algorithm 4. The task of the stopping criteria is to control when to stop the iterations. Since we know that the gradient is zero at a stationary point it is useful to investigate when the gradient is close to zero. Another indicator is to keep an eye on the size of the increments between adjacent iterates, i.e. when θ_{t+1} is close to θ_t.

In the so-called *trust region* strategy the order of step 2a and step 2b in Algorithm 4 is simply reversed, i.e. we first decide how far to step and then we chose in which direction to move.

B.2. Commonly used search directions

Three of the most popular search directions corresponds to specific choices when it comes to the positive definite matrix V_t in step 2a of Algorithm 4. The simplest choice is to make use of the identity matrix, resulting in the so-called *steepest descent* direction described in Section B.2.1. The *Newton* direction (Section B.2.2) is obtained by using the inverse of the Hessian matrix and finally we have the *quasi-Newton* direction (Section B.2.3) employing an approximation of the inverse Hessian.

B.2.1. Steepest descent direction

Let us start by noting that according to the definition of the scalar product[3], the descent condition (B.5) imposes the following requirement of the search direction

$$p^{\mathsf{T}} \nabla J(\theta_t) = \|p\|_2 \|\nabla J(\theta_t)\|_2 \cos(\varphi) < 0, \tag{B.9}$$

where φ denotes the angle between the two vectors p and $\nabla J(\theta_t)$. Since we are only interested in finding the direction we can without loss of generality fix the length of p, implying the scalar product $p^{\mathsf{T}} \nabla J(\theta_t)$ is made as small as possible by selecting $\varphi = \pi$, corresponding to

$$p = -\nabla J(\theta_t). \tag{B.10}$$

Recall that the gradient vector at a point is the direction of maximum rate of change of the function at that point. This explains why the search direction suggested in (B.10) is referred to as the *steepest descent direction*.

Sometimes, the use of the steepest descent direction can be very slow. The reason for this is that there is more information available about the cost function that the algorithm can make use of, which brings us to the Newton and the quasi-Newton directions described below. They make use of additional information about the local geometry of the cost function by employing a more descriptive local model.

[3]The scalar (or dot) product of two vectors a and b is defined as $a^{\mathsf{T}} b = \|a\| \|b\| \cos(\varphi)$, where $\|a\|$ denotes the length (magnitude) of the vector a and φ denotes the angle between a and b.

B.2.2. Newton direction

Let us now instead make use of a better model of the objective function, by also keeping the quadratic term of the Taylor expansion. The result is the following quadratic approximation $m(\theta_t, p_t)$ of the cost function around the current iterate θ_t

$$L(\theta_t + p_t) \approx \underbrace{J(\theta_t) + p_t^\mathsf{T} g_t + \frac{1}{2} p_t^\mathsf{T} H_t p_t}_{=m(\theta_t, p_t)} \tag{B.11}$$

where $g_t = \nabla J(\theta)|_{\theta=\theta_t}$ denotes the cost function gradient and $H_t = \nabla^2 J(\theta)|_{\theta=\theta_t}$ denotes the Hessian, both evaluated at the current iterate θ_t. The idea behind the Newton direction is to select the search direction that minimizes the quadratic model in (B.11), which is obtained by setting its derivative

$$\frac{\partial m(\theta_t, p_t)}{\partial p_t} = g_t + H_t p_t \tag{B.12}$$

to zero, resulting in

$$p_t = -H_t^{-1} g_t. \tag{B.13}$$

It is often too difficult or too expensive to compute the Hessian, which has motivated the development of search directions employing an approximation of the Hessian. The generic name for these are quasi-Newton directions.

B.2.3. Quasi-Newton

The quasi-Newton direction makes use of a local quadratic model $m(\theta_t, p_t)$ of the cost function according to (B.11), similarly to what was done in finding the Newton direction. However, rather than assuming that the Hessian is available, the Hessian will now instead be learned from the information that is available in the cost function values and its gradients.

Let us first denote the line segment connecting two adjacent iterates θ_t and θ_{t+1} by

$$r_t(\tau) = \theta_t + \tau(\theta_{t+1} - \theta_t), \qquad \tau \in [0, 1]. \tag{B.14}$$

From the fundamental theorem of calculus we know that

$$\int_0^1 \frac{\partial}{\partial \tau} \nabla J(r_t(\tau)) d\tau = \nabla J(r_t(1)) - \nabla J(r_t(0)) = \nabla J(\theta_{t+1}) - \nabla J(\theta_t) = g_{t+1} - g_t, \tag{B.15}$$

and from the chain rule we have that

$$\frac{\partial}{\partial \tau} \nabla J(r_t(\tau)) = \nabla^2 J(r_t(\tau)) \frac{\partial r_t(\tau)}{\partial \tau} = \nabla^2 J(r_t(\tau))(\theta_{t+1} - \theta_t). \tag{B.16}$$

Hence, in combining (B.15) and (B.16) we obtain

$$y_t = \int_0^1 \frac{\partial}{\partial \tau} \nabla J(r_t(\tau)) d\tau = \int_0^1 \nabla^2 J(r_t(\tau)) s_t d\tau. \tag{B.17}$$

where we have defined $y_t = g_{t+1} - g_t$ and $s_t = \theta_{t+1} - \theta_t$. An interpretation of the above equation is that the difference between two consecutive gradients y_t is given by integrating the Hessian times s_t for points θ along the line segment $r_t(\tau)$ defined in (B.14). The approximation underlying quasi-Newton methods is now to assume that this integral can be described by a constant matrix B_{t+1}, resulting in the following approximation

$$y_t = B_{t+1} s_t \tag{B.18}$$

of the integral (B.17), which is sometimes referred to as the *secant condition* or the quasi-Newton equation. The secant condition above is still not enough to determine the matrix B_{t+1}, since even though we know that B_{t+1} is symmetric there are still too many degrees of freedom available. This is solved using regularization and B_{t+1} is selected as the solution to

$$\begin{aligned} B_{t+1} = \min_{B} \quad & \|B - B_t\|_W^2, \\ \text{s.t.} \quad & B = B^\mathsf{T}, \quad Bs_t = y_t, \end{aligned} \tag{B.19}$$

for some weighting matrix W. Depending on which weighting matrix that is used we obtain different algorithms. The most common quasi-Newton algorithms are referred to as BFGS (named after Broyden, Fletcher, Goldfarb and Shanno), DFP (named after Davidon, Fletcher and Powell) and Broyden's method. The resulting Hessian approximation B_{t+1} is then used in place of the true Hessian.

B.3. Further reading

This appendix is heavily inspired by the solid general introduction to the topic of numerical solutions to optimization problems given by Nocedal and Wright (2006) and by Wills (2017). In solving optimization problems the initial important classification of the problem is whether it is convex or non-convex. Here we have mainly been concerned with the numerical solution of non-convex problems. When it comes to convex problems Boyd and Vandenberghe (2004) provide a good engineering introduction. A thorough and timely introduction to the use of numerical optimization in the machine learning context is provided by Bottou, Curtis, and Nocedal (2017). The focus is naturally on large scale problems and as we have explained in the deep learning chapter this naturally leads to stochastic optimization problems.

C. Classification loss functions

The classification loss functions illustrated in Figure 5.3 are:

Exponential loss: $L(y, c) = \exp(-yc)$.

Hinge loss: $L(y, c) = \begin{cases} 1 - yc & \text{for } yc < 1, \\ 0 & \text{otherwise.} \end{cases}$

Binomial deviance: $L(y, c) = \log(1 + \exp(-yc))$.

Huber-like loss: $L(y, c) = \begin{cases} -yc & \text{for } yc < -1, \\ \frac{1}{4}(1 - yc)^2 & \text{for } -1 \leq yc \leq 0, \\ 0 & \text{otherwise.} \end{cases}$

Misclassification loss: $L(y, c) = \begin{cases} 1 & \text{for } yc < 0, \\ 0 & \text{otherwise.} \end{cases}$

Bibliography

Bishop, Christopher M. (2006). *Pattern Recognition and Machine Learning*. Springer.

Bottou, L., F. E. Curtis, and J. Nocedal (2017). *Optimization methods for large-scale machine learning*. Tech. rep. arXiv:1606.04838v2.

Boyd, S. and L. Vandenberghe (2004). *Convex Optimization*. Cambridge, UK.

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*.

Ezekiel, Mordecai and Karl A. Fox (1959). *Methods of Correlation and Regression Analysis*. John Wiley & Sons, Inc.

Freund, Yoav and Robert E. Schapire (1996). "Experiments with a new boosting algorithm". In: *Proceedings of the 13th International Conference on Machine Learning (ICML)*.

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 et al. (2013). *Bayesian data analysis*. 3rd ed. CRC Press.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. http://www.deeplearningbook.org. MIT Press.

Hastie, Trevor, Robert Tibshirani, and Jerome Friedman (2009). *The elements of statistical learning. Data mining, inference, and prediction*. 2nd ed. Springer.

Hastie, Trevor, Robert Tibshirani, and Martin J. Wainwright (2015). *Statistical learning with sparsity: the Lasso and generalizations*. CRC Press.

Hoerl, Arthur E. and Robert W. Kennard (1970). "Ridge regression: biased estimation for nonorthogonal problems". In: *Technometrics* 12.1, pp. 55–67.

James, Gareth, Daniela Witten, Trevor Hastie, and Robert Tibshirani (2013). *An introduction to statistical learning. With applications in R*. Springer.

Ke, Guolin et al. (2017). "LightGBM: A Highly Efficient Gradient Boosting Decision Tree". In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon et al. Curran Associates, Inc., pp. 3149–3157. URL: http://papers.nips.cc/paper/6907-lightgbm-a-highly-efficient-gradient-boosting-decision-tree.pdf.

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton (2015). "Deep learning". In: *Nature* 521, pp. 436–444.

LeCun, Yann, Bernhard Boser, et al. (1990). "Handwritten Digit Recognition with a Back-Propagation Network". In: *Advances in Neural Information Processing Systems (NIPS)*, pp. 396–404.

Mason, Llew, Jonathan Baxter, Peter Bartlett, and Marcus Frean (1999). "Boosting Algorithms as Gradient Descent". In: *Proceedings of the 12th International Conference on Neural Information Processing Systems (NIPS)*.

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.

Mnih, Volodymyr et al. (2015). "Human-level control through deep reinforcement learning". In: *Nature* 518.7540, pp. 529–533.

Murphy, Kevin P. (2012). *Machine learning – a probabilistic perspective*. MIT Press.

Nocedal, J. and S. J. Wright (2006). *Numerical Optimization*. 2nd ed. Springer Series in Operations Research. New York, USA.

Srivastava, Nitish et al. (2014). "Dropout: A simple way to prevent neural networks from overfitting". In: *The Journal of Machine Learning Research* 15.1, pp. 1929–1958.

Tibshirani, Robert (1996). “Regression Shrinkage and Selection via the LASSO”. In: *Journal of the Royal Statistical Society (Series B)* 58.1, pp. 267–288.

Wills, A. G. (2017). “Real-time optimisation for embedded systems”. Lecture notes.

Xu, Kelvin et al. (2015). “Show, attend and tell: Neural image caption generation with visual attention”. In: *Proceedings of the International Conference on Learning representations (ICML)*.