universität
uulm

Fakultät für Mathematik und Wirtschafts-wissenschaften
Institut für Numerische Mathematik

Gradient-Based Methods for the Training of Neural Networks

Abschlussarbeit an der Universität Ulm

Vorgelegt von:
Nils Daniel Fleischmann
nils.fleischmann@uni-ulm.de
981781

Gutachter:
Prof. Dr. Karsten Urban

Betreuer:
Prof. Dr. Karsten Urban

2021

Fassung October 15, 2021

Satz: PDF-LaTeX 2_ε

Contents

List of Figures

List of Tables

1 Introduction

Neural networks have seen a tremendous rise in the past two decades due to the growing amount of data, technological advancements, and increasing computational capacities. Meanwhile, the applications of neural networks are ubiquitous: they compose the content feeds of social networks, recommend products on e-commerce websites, and assist in making medical diagnoses. Furthermore, they also power future technologies such as autonomous driving and robotics.

Neural networks can accomplish this wide range of challenging tasks by learning from data. This happens during the training phase, where the parameters of the neural network are adjusted to the data. Training a neural network usually amounts to solving a high-dimensional and non-convex optimization problem. However, the problem is hard in general; many conventional methods from non-linear optimization turned out to be either unsuitable or too inefficient for solving this problem.

Nonetheless, the class of gradient-based methods has shown to be useful for this task. In particular, so-called stochastic gradient methods see frequent use for the training of neural networks. This bachelor thesis investigates the suitability and the behavior of gradient-based methods for the non-convex optimization problem behind the training of neural networks. The work is organized as follows.

At first, Chapter 2 deals with feed-forward neural networks and their mathematical representation. Furthermore, the principle of empirical risk minimization is introduced, which is used to derive the optimization problem behind the training of these neural networks. In Chapter 3, this optimization problem is examined for properties such as smoothness and convexity. A particular emphasis lies on the question of whether it suffices to search for a local minimum for this optimization problem. After that, Chapter 4 introduces various gradient-based methods and investigates their behavior for non-convex optimization problems such as the training of neural networks. In Chapter 5, some of the theoretical results of this thesis are revisited and extended by numerical experiments. Finally, Chapter 6 provides a summary and gives an outlook on future work.

2 Neural Networks

The scope of this chapter is to introduce feed-forward neural networks and derive the optimization problem we need to solve in order to train them. To this end, we first define the supervised learning framework in which neural networks are employed. Then we describe how neural networks are constructed and introduce the necessary notation that finally allows us to formulate the optimization problem.

2.1 Supervised Learning

Machine learning refers to a broad set of algorithms that are capable of learning patterns in data. The most common form of machine learning is *supervised learning*. Since neural networks are usually employed in the context of supervised learning, we provide a brief overview based on the work of Luxburg and Schölkopf [14] and Urban [22].

In supervised learning, the learning algorithm has access to a finite sequence of observations $\mathcal{S} = (x_1, y_1), \ldots, (x_n, y_n)$ called the *training set*. Each observation consists of an *input* $x \in \mathcal{X}$ and an *output* $y \in \mathcal{Y}$. Depending on whether the output takes on continuous or discrete values, one speaks of *regression* or *classification* problems, respectively. We assume that the observations in the training set are independent and identically distributed (i.i.d.) according to some fixed but unknown probability distribution $\mathcal{D}$ on $\mathcal{X} \times \mathcal{Y}$. This i.i.d. assumption is essential for many theoretical results; however, it might not be realistic for practical applications.

Given the training set, the learning algorithm aims to find a *prediction function* $h : \mathcal{X} \to \mathcal{Y}$ that approximates the input-output relationship as well as possible. This requires a measure of "how good" the predictions of a function h are. In this regard, we introduce a *loss function* $\mathcal{L} : \mathcal{Y} \times \mathcal{Y} \to [0, \infty)$. The *loss* $\mathcal{L}(y, h(x))$ reflects how much the actual output and the predicted output differ, given an observation (x, y). However, to measure the quality of a prediction function, we would prefer a measure that incorporates all possible observations of the distribution $\mathcal{D}$. Such a measure is given by the *risk*, which is defined as

$$R(h) := \mathbb{E}_{(x,y)\sim\mathcal{D}} \left[\mathcal{L}(y, h(x))\right]. \tag{2.1}$$

Ideally, the learning algorithm would search for the prediction function with the lowest risk. But as mentioned above, the underlying distribution $\mathcal{D}$ is unknown. Thus it is not possible to compute the risk of a prediction function. Nevertheless, the training set

contains independent realizations of the distribution $\mathcal{D}$, which can be used to estimate the expected value in (2.1). This approach leads to the *empirical risk*

$$\hat{R}(h) := \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(y_i, h(x_i)).$$

Due to the law of large numbers, the empirical risk should be a reasonable approximation if the training set is sufficiently large. Thus we hope to find a prediction function with low risk by minimizing the empirical risk. However, we have to be cautious. If the learning algorithm has too much freedom in choosing the prediction function, it might find a function that makes good predictions for the observations in the training set, but not for observations of the distribution $\mathcal{D}$ in general. To avoid this problem, we restrict the learning algorithm to a class of prediction functions $\mathcal{H}$. Various learning algorithms use different classes in order to find an appropriate prediction function. Still, most of them try to solve the same underlying minimization problem

$$h^* = \operatorname*{argmin}_{h \in \mathcal{H}} \hat{R}(h). \tag{2.2}$$

This approach is referred to as *empirical risk minimization*. As we will see in the following section, the training of neural networks follows this underlying principle.

2.2 Architecture of Neural Networks

Neural networks are powerful, flexible, and scalable; depending on their architecture, they are suitable for a variety of demanding supervised learning tasks. Reason enough to study the architecture of neural networks in more detail. This section is based on the work of Anthony and Bartlett [1], Bruhn-Fujimoto [5], Géron [10], and Urban [22].

Neural networks can be thought of as directed graphs, consisting of several units, called *neurons*, that are connected by weighted edges. Through their incoming edges, these neurons receive inputs based on which they calculate an output. In order to formalize this computation, we denote the inputs of a neuron as $z := (z_1, z_2, \ldots, z_d)^\intercal \in \mathbb{R}^d$ and the associated edge weights as $w := (\omega_1, \omega_2, \ldots, \omega_d) \in \mathbb{R}^d$. Further, let $b \in \mathbb{R}$ be a scalar, which we call the *bias* of the neuron. From a mathematical perspective, a neuron is a function

$$z \mapsto \sigma\left(\sum_{i=1}^{d} \omega_i z_i + b\right) \tag{2.3}$$

composed of an affine linear mapping and a non-linear *activation function* σ. Figure 2.1 illustrates the output calculation of a single neuron.

The weights and the bias determine how the inputs affect the output. They are parameters of the neural network; reasonable values for them have to be determined during the

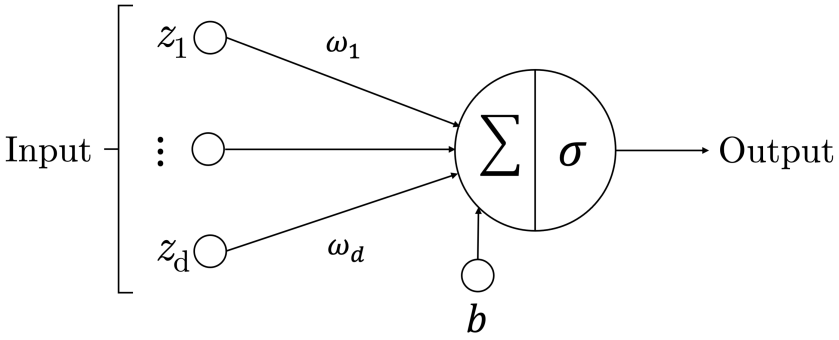


Figure 2.1: A single neuron. Adapted from Géron [10, Figure 10-4].

training phase. Although the activation function also influences the output, we typically do not treat it as a parameter. In the literature, there are different activation functions for various purposes. We will focus on three of the most common ones, which are summarized in Table 2.1. Individually, neurons remain simple functions; but organized in vast networks, they can perform complex calculations.

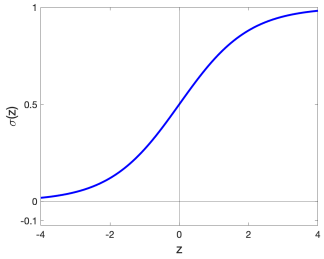
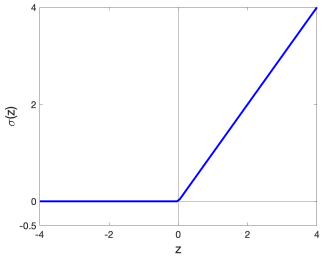
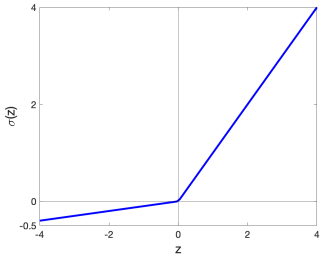
logistic function	rectified linear unit (ReLU)	leaky ReLU
$\sigma(z) = \frac{1}{1+e^{-z}}$	$\sigma(z) = \max(z, 0)$	$\sigma(z) = \max(z, 0.1z)$
		

Table 2.1: Common activation functions. Adapted from Urban [22, Table 5.2.].

There are various approaches to connect neurons into a network. However, we limit ourselves to the most common type of neural networks, namely, *feed-forward neural networks*. The neurons in such networks are arranged in layers $\ell \in \{0, \dots, L\}$. Thereby, the 0^{th} layer takes external inputs, and the L^{th} layer generates the output of the neural network. Accordingly, we call the 0^{th} layer *input layer* and the L^{th} layer *output layer* and refer to the $L-1$ layers in between as *hidden layers*.

As the name suggests, there are only connections from one layer to the subsequent layer in feed-forward neural networks. So the input travels forward through the network and is gradually processed by the neurons of each layer. We usually consider networks that are *fully connected*. This means that there is an edge from each neuron in the $(\ell-1)^{\text{th}}$ layer to every neuron in the ℓ^{th} layer. Figure 2.2 visualizes a fully connected

feed-forward neural network with two hidden layers. Since we deal exclusively with fully connected feed-forward neural networks, we will refer to them as neural networks in the following.

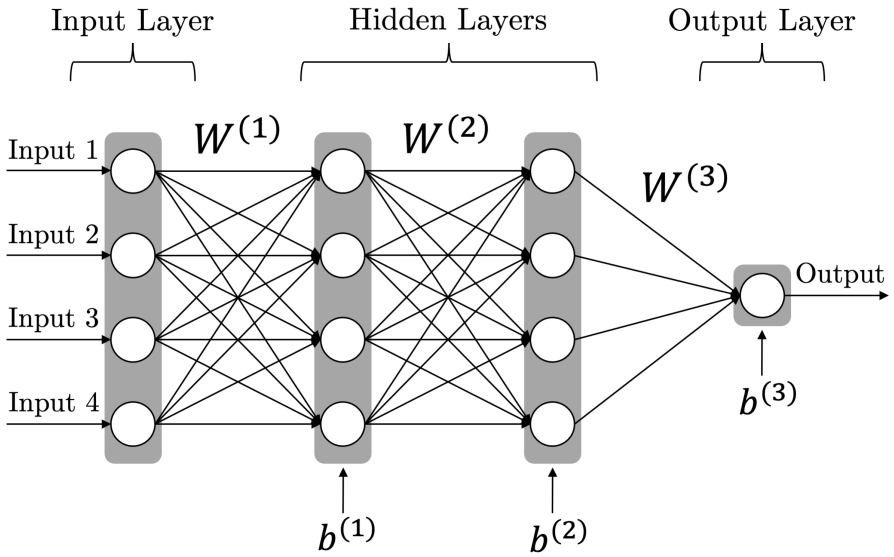


Figure 2.2: A neural network with two hidden layers. Adapted from Bruhn-Fujimoto [5, Figure 18].

Neural networks may well consist of thousands of neurons. Therefore, a rigorous notation is needed to deal with all the weights, biases, and activation functions. We refer to L as the *depth* of the neural network and denote by the *width* d_ℓ the number of neurons in the ℓ^{th} layer. The weight matrix $W^{(\ell)} \in \mathbb{R}^{d_\ell \times d_{\ell-1}}$ includes the weights of all edges between the $(\ell-1)^{\text{th}}$ and ℓ^{th} layer. To be more precise, the matrix is defined as

$$[W^{(\ell)}]_{i,j} := \omega_{i,j}^{(\ell)} \qquad i = 1, \dots d_\ell,\ j = 1, \dots, d_{\ell-1},$$

where $\omega_{i,j}^{(\ell)}$ is the weight of the edge from the j^{th} neuron in the $(\ell-1)^{\text{th}}$ layer to the i^{th} neuron in the ℓ^{th} layer. The bias vector $b^{(\ell)}$ contains the bias terms of all neurons in the ℓ^{th} layer and is defined as

$$b^{(\ell)} := \left(b_1^{(\ell)}, \dots, b_{d_\ell}^{(\ell)}\right)^\intercal \in \mathbb{R}^{d_\ell},$$

where $b_i^{(\ell)}$ is the bias of the i^{th} neuron in the ℓ^{th} layer. Altogether, the neural network has $d := \sum_{\ell=1}^{L}(d_\ell \cdot d_{\ell-1} + d_\ell)$ parameters. To refer to all of these parameters, we use the parameter vector

$$\theta := \left(\omega_{1,1}^{(1)}, \dots, \omega_{d_L,d_{L-1}}^{(L)}, b_1^{(1)}, \dots, b_{d_L}^{(L)}\right)^\intercal \in \mathbb{R}^d.$$

Usually, the same activation function is chosen for all neurons within the same layer. We denote the activation function used in the ℓ^{th} layer as $\sigma^{(\ell)}$ for $\ell = 1, \dots, L$.

Let N denote a neural network characterized by its depth, the number of neurons in each layer, and the used activation functions. With the introduced notation, we can formalize this neural network as a parameterized function of its inputs. In the following, $x^{(\ell)}$ denotes the output of the ℓ^{th} layer. Correspondingly, $x^{(0)} \in \mathcal{X}$ is the input, and $x^{(L)} \in \mathcal{Y}$ is the output of the neural network.

According to (2.3), the output of the i^{th} neuron in the ℓ^{th} layer takes the form

$$x^{(\ell-1)} \mapsto \sigma^{(\ell)} \left(\sum_{j=1}^{d_{\ell-1}} \omega_{i,j}^{(\ell)} \, x_j^{(\ell-1)} + b_i^{(\ell)} \right) := x_i^{(\ell)}, \quad i = 1, \ldots, d_\ell.$$

Using the weight matrix $W^{(\ell)}$ and the bias vector $b^{(\ell)}$, we can summarize the affine-linear mappings of all neurons in the ℓ^{th} layer as

$$A^{(\ell)}(x^{(\ell-1)}) := W^{(\ell)} x^{(\ell-1)} + b^{(\ell)}.$$

Then, we obtain the output of the ℓ^{th} layer by applying the activation function to this interim result:

$$x^{(\ell)} = \sigma^{(\ell)}(A^{(\ell)}(x^{(\ell-1)})) = \sigma^{(\ell)} \circ A^{(\ell)} \circ x^{(\ell-1)}. \tag{2.4}$$

The application of the activation function $\sigma^{(\ell)}$ is to be understood component-wise in this context.

Starting with the output, we can derive the input-output mapping of the neural network by repeatedly applying (2.4):

$$\begin{aligned} x^{(L)} &= \sigma^{(L)} \circ A^{(L)} \circ x^{(L-1)} \\ &= \sigma^{(L)} \circ A^{(L)} \circ \sigma^{(L-1)} \circ A^{(L-1)} \circ x^{(L-2)} \\ &\vdots \\ &= \sigma^{(L)} \circ A^{(L)} \circ \sigma^{(L-1)} \circ A^{(L-1)} \circ \ldots \circ \sigma^{(2)} \circ A^{(2)} \circ \sigma^{(1)} \circ A^{(1)} \circ x^{(0)}. \end{aligned}$$

Note that the affine linear mappings $A^{(\ell)}$ depend on the weights and biases of the neural network, which are contained in the parameter vector $\theta \in \mathbb{R}^d$. Therefore we may regard the neural network N as a parametrized function $h : \mathcal{X} \times \mathbb{R}^d \to \mathcal{Y}$. Thereby each possible parametrization $\theta \in \mathbb{R}^d$ corresponds to a prediction function $h_\theta : \mathcal{X} \to \mathcal{Y}$ given by

$$h_\theta(x) = h(x, \theta) := \sigma^{(L)} \circ A^{(L)} \circ \sigma^{(L-1)} \circ A^{(L-1)} \circ \ldots \circ \sigma^{(2)} \circ A^{(2)} \circ \sigma^{(1)} \circ A^{(1)} \circ x. \tag{2.5}$$

The class of prediction functions computable by the neural network N is given by $\mathcal{H}_N := \{h_\theta : \mathcal{X} \to \mathcal{Y} \mid \theta \in \mathbb{R}^d\}$. For this specific class, we can refine the general empirical risk minimization problem (2.2). Since each prediction function of $\mathcal{H}_N$ is determined by the parameter vector θ, the minimization problem can be reformulated as

$$\theta^* = \operatorname*{argmin}_{\theta \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(y_i, h(x_i, \theta)). \tag{2.6}$$

In this way, we determine the optimal prediction function h_{θ^*} indirectly by optimizing over the parameter vector θ. This is the optimization problem we have to solve in order to train the neural network N.

2.3 Loss Functions

So far, we have treated the loss function as an arbitrary function. Nevertheless, the concrete choice of the loss function plays a crucial role in the training of neural networks and should be made carefully depending on the given task. In this section, which is adapted from Bishop [3] and Geiger [9], we derive reasonable loss functions for *regression* and *binary classification* tasks.

2.3.1 Binary Classification

For binary classification tasks, the output only takes the values 0 and 1. Common neural networks for this task have one output neuron with the logistic activation function. This choice ensures that the neural network exclusively predicts values in the interval $(0, 1)$. Hence, we can interpret the prediction $h_\theta(x)$ of such a neural network as the conditional probability $\mathbb{P}(y = 1 \,|\, x; \theta)$. Consequently, we assume that the conditional distribution of the output is a Bernoulli distribution with mass function

$$p(y \,|\, x; \theta) = h_\theta(x)^y \cdot (1 - h_\theta(x))^{1-y}.$$

Figure 2.3 exemplifies the relationship between the prediction of a neural network and the assumed conditional distribution of the output.

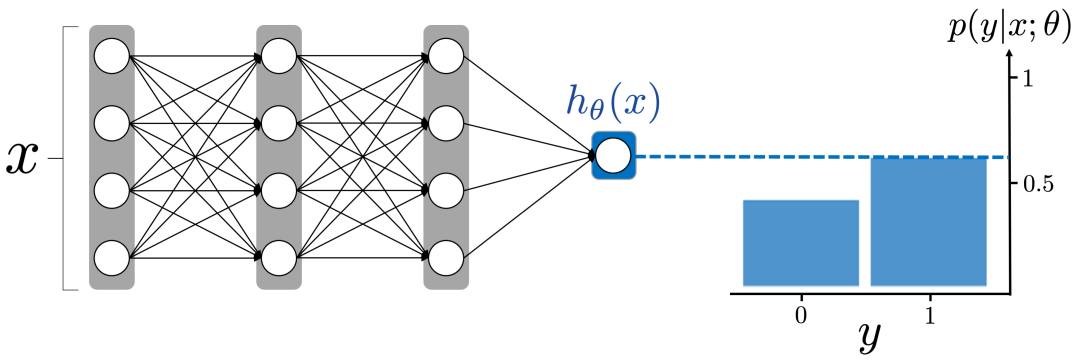


Figure 2.3: Assumed conditional distribution of the output for binary classification tasks. Adapted from Geiger [9, Figure 43].

This means that the set of parameter vectors $\{\theta \in \mathbb{R}^d\}$ corresponds to a parametric family of conditional probability distributions $\{p(\cdot \,|\, \cdot\,; \theta) \,|\, \theta \in \mathbb{R}^d\}$. Now, we can work with these distributions to find the optimal parameter vector. In particular, we can

search for the distribution that is most likely to have generated the i.i.d. observations in the training set. This can be done by maximizing the corresponding likelihood function

$$\prod_{i=1}^{n} p(y_i \mid x_i; \theta) = \prod_{i=1}^{n} h_\theta(x_i)^{y_i} \cdot (1 - h_\theta(x_i))^{1-y_i}$$

regarding θ. To transform this *maximum likelihood problem* into a minimization problem of the same format as (2.6), we first apply the logarithm and then multiply with $-\frac{1}{n}$:

$$\begin{aligned}
\theta^* &= \operatorname*{argmax}_{\theta \in \mathbb{R}^d} \prod_{i=1}^{n} h_\theta(x_i)^{y_i} \cdot (1 - h_\theta(x_i))^{1-y_i} \\
&= \operatorname*{argmax}_{\theta \in \mathbb{R}^d} \sum_{i=1}^{n} y_i \cdot \log(h_\theta(x_i)) + (1 - y_i) \cdot \log(1 - h_\theta(x_i)) \\
&= \operatorname*{argmin}_{\theta \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} -(y_i \cdot \log(h_\theta(x_i)) + (1 - y_i) \cdot \log(1 - h_\theta(x_i)))
\end{aligned}$$

Finally, the associated loss function, which is often referred to as the *log-loss*, is given by $\mathcal{L}(y, \hat{y}) := -(y \cdot \log(\hat{y}) + (1 - y) \cdot \log(1 - \hat{y}))$.

2.3.2 Regression

For most regression tasks, the output takes on continuous values $y \in \mathbb{R}$. Common network architectures for this task contain a single neuron with the identity as the activation function in the output layer. We assume that the conditional distribution of the output given the input is a Gaussian distribution. Under this assumption, we can interpret the predictions of a neural network as the mean of this Gaussian distribution. So we assume that $y \mid x, \theta$ is distributed according to $\mathcal{N}(h_\theta(x), \sigma^2)$ with unknown variance σ^2 and mass function

$$p(y \mid x; \theta) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y - h_\theta(x))^2}{2\sigma^2}\right).$$

Figure 2.4 illustrates how the prediction of a neural network influences the assumed conditional distribution of the output.

Again, starting with the maximum likelihood problem, a minimization problem of the

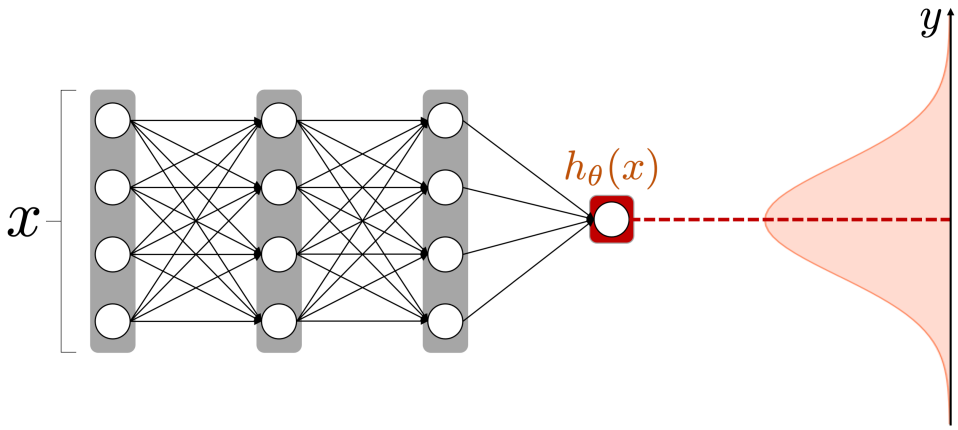


Figure 2.4: Relationship between the prediction of the neural network and the assumed conditional distribution of the output. Adapted from Geiger [9, Figure 38].

form (2.6) can be derived by applying the logarithm and then multiplying with $-\frac{1}{n}$:

$$\begin{aligned}
\theta^* &= \operatorname*{argmax}_{\theta \in \mathbb{R}^d} \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_i - h_\theta(x_i))^2}{2\sigma^2}\right) \\
&= \operatorname*{argmax}_{\theta \in \mathbb{R}^d} -\sum_{i=1}^{n} \frac{1}{2} \log(2\pi\sigma^2) - \frac{1}{2\sigma^2} \sum_{i=1}^{n} (y_i - h_\theta(x_i))^2 \\
&= \operatorname*{argmax}_{\theta \in \mathbb{R}^d} -\sum_{i=1}^{n} (y_i - h_\theta(x_i))^2 \\
&= \operatorname*{argmin}_{\theta \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} (y_i - h_\theta(x_i))^2
\end{aligned}$$

Note that we can drop the term $-\sum_{i=1}^{n} \frac{1}{2} \log(2\pi\sigma^2)$ and the factor $\frac{1}{2\sigma^2}$ because they do not depend on θ. The resulting minimization problem introduces the so-called *square loss*, which is defined as $\mathcal{L}(y, \hat{y}) := (y - \hat{y})^2$.

3 The Optimization Problem

Now that we have derived the optimization problem behind the training of neural networks, we can examine it in further detail. In particular, properties like smoothness or convexity, which play a huge role in choosing the optimizer, will be considered. Furthermore, we will discuss whether it is sufficient to search for a local minimum for this optimization problem.

3.1 Unconstrained Optimization

As we have seen, training a neural network amounts to solving an unconstrained optimization problem. Therefore, an overview of basic definitions and results of unconstrained optimization theory is given in this section. The structure and results are taken from Ulbrich and Ulbrich [21].

In general, unconstrained optimization problems are of the form

$$\text{minimize } \ f(x) \ \text{ w.r.t. } \ x \in \mathbb{R}^d,$$

where $f : \mathbb{R}^d \to \mathbb{R}$ is the *objective function*. We refer to the value of the objective function as the *cost*. A point x^* that solves the unconstrained optimization problem is called a *global minimum*. This means that for all $x \in \mathbb{R}^d$, we have $f(x) \geq f(x^*)$.

Finding a global minimum of an unconstrained optimization problem is a computationally hard problem. In cases where it is infeasible to determine a global minimum, one searches for a local minimum instead. A point x^* is called a *local minimum* of the objective function if there exists $\varepsilon > 0$ with

$$f(x) \geq f(x^*) \ \text{ for all } \ x \in \{x \in \mathbb{R}^d \mid \|x - x^*\|_2 \leq \varepsilon\}.$$

For sufficiently smooth objective functions, local minima can be characterized regarding their gradient.

Theorem 3.1 (First-order necessary condition)**.** *Let $f : U \subset \mathbb{R}^d \to \mathbb{R}$ be differentiable on the open subset $U \subset \mathbb{R}^d$ and $x^* \in U$ a local minimum of f. Then it holds*

$$\nabla f(x^*) = 0. \tag{3.1}$$

Proof. See Ulbrich and Ulbrich [21, Theorem 5.1]. □

Points that satisfy condition (3.1) are called *stationary points*. However, not every stationary point is a local minimum. It might also be a local maximum or a saddle point. Figure 3.1 depicts two different types of stationary points: a local minimum and a saddle point.

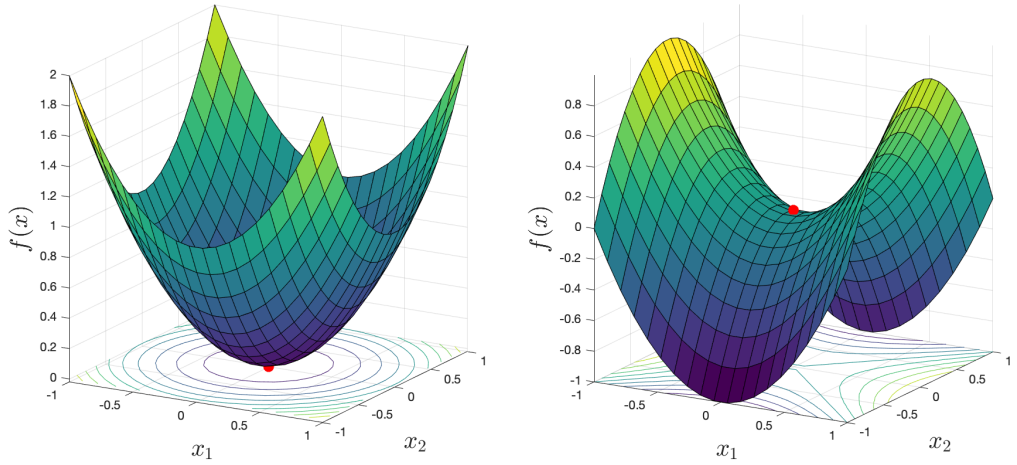


Figure 3.1: Two different types of stationary points: a local minimum (left) and a saddle point (right).

To distinguish local minima from other stationary points, one can additionally consider the *Hessian matrix* of f, which is defined as

$$[\nabla^2 f(x)]_{i,j} := \frac{\partial^2 f(x)}{\partial x_i \, \partial x_j}, \qquad i, j = 1, \ldots, d.$$

Theorem 3.2 (Second-order necessary condition)**.** *Let $f : U \subset \mathbb{R}^d \to \mathbb{R}$ be twice differentiable on the open subset $U \subset \mathbb{R}^d$ and $x^* \in U$ a local minimum of f. Then it holds*

(i) $\nabla f(x^*) = 0$.

(ii) The Hessian matrix $\nabla^2 f(x^*)$ is positive semi-definite.

Proof. See Ulbrich and Ulbrich [21, Theorem 5.4]. □

In other words, stationary points for which the Hessian matrix has at least one negative eigenvalue can not be local minima. Lee et al. [13] define stationary points with this property as *strict saddles*. If we tighten the second-order necessary condition regarding the Hessian matrix, we obtain the following sufficient condition.

Theorem 3.3 (Second-order sufficient condition)**.** *Let $f : U \subset \mathbb{R}^d \to \mathbb{R}$ be twice differentiable on the open subset $U \subset \mathbb{R}^d$ and $x^* \in U$ a point for which holds:*

(i) $\nabla f(x^*) = 0$.

(ii) The Hessian matrix $\nabla^2 f(x^*)$ *is positive definite.*

Then x^* *is a local minimum of* f.

Proof. See Ulbrich and Ulbrich [21, Theorem 5.5]. □

Convex functions play an essential role in unconstrained optimization theory. A function $f : \mathbb{R}^d \to \mathbb{R}$ is called *convex* if for all $x, y \in \mathbb{R}^d$ and all $\lambda \in [0, 1]$ it holds

$$f((1 - \lambda)x + \lambda y) \leq (1 - \lambda)f(x) + \lambda f(y).$$

One can show that all local minima of convex functions are global minima (see Ulbrich and Ulbrich [21, Theorem 6.5]). Therefore it is desirable to model problems such that the objective function is convex.

3.2 Properties of the Objective Function

Now that we have introduced the basics of unconstrained optimization theory, we can shift our attention back to the optimization problem behind the training of neural networks. Let us start by examining the corresponding objective function

$$f(\theta) := \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(y_i, h(x_i, \theta)),$$

which measures the average loss of the neural network over the training set. Since the observations $(x_1, y_1), \ldots, (x_n, y_n)$ are fixed, these losses only depend on the parameter vector $\theta \in \mathbb{R}^d$ of the neural network. To emphasize this, we abbreviate the loss on the i^{th} observation of the training set with $\mathcal{L}_i(\theta) := \mathcal{L}(y_i, h(x_i, \theta))$. By employing the definition of the parameterized prediction function of the neural network, we obtain the following representation for the losses:

$$\mathcal{L}_i(\theta) = \mathcal{L}(y_i, \cdot) \circ \sigma^{(L)} \circ A^{(L)} \circ \ldots \circ \sigma^{(1)} \circ A^{(1)} \circ x_i, \qquad i = 1, \ldots, n.$$

This representation allows us to analyze the losses by their components. Recall that the weights and biases contained in the parameter vector θ define the linear mappings $A^{(\ell)}$. Thus, the losses depend on θ in a nested manner.

By design, the loss function $\mathcal{L}(y_i, \cdot)$ maps only to non-negative values. Consequently, the losses and the objective function are bounded from below by 0. Therefore we can confidently make the following assumption that will play an essential role in analyzing gradient-based methods.

Assumption 3.4 (Bounded objective function)**.** *The objective function* f *is bounded below by a scalar* f_{inf}.

As can be seen from Tables 3.1 and 3.2, most activation and loss functions are continuous. The only exception is the log loss, which is only defined if the second argument stays on the interval $(0, 1)$. However, the log loss is always coupled with the logistic activation function, ensuring that the second argument remains in this domain. Moreover, the affine linear mappings $A^{(\ell)}$ are continuous regarding their input as well as their parameters. Consequently, the losses $\mathcal{L}_i(\theta)$ are continuous as a concatenation of continuous functions.

	$\mathcal{L}(y, z)$	$\frac{\partial}{\partial z}\,\mathcal{L}(y, z)$
square loss	$(y - z)^2$	$-2(y - z)$
log loss	$-(y \log(z) + (1 - y) \log(1 - z))$	$-(y - z)/(z - z^2)$

Table 3.1: Partial derivatives w.r.t. the second argument of common loss functions.

Furthermore, all components, except the ReLU and the leaky ReLU, are differentiable with respect to their inputs and parameters. Thus, the losses of neural networks that use neither of those activation functions are differentiable due to the chain rule (a more elaborated justification of this property will be given in Section 4.1). In contrast, the losses of neural networks that use the (leaky) ReLU are not differentiable. Nonetheless, since the (leaky) ReLU is only non-differentiable in $z = 0$, we treat it as a differentiable function in practice. In the rare case that the derivative of the (leaky) ReLU has to be evaluated at $z = 0$, one can return the right derivative given by 1. In this way, we treat the losses of such neural networks as if they were differentiable. This approach seems tolerable since numerical optimization is subject to imprecise number representations and rounding errors anyway.

These smoothness properties carry over to the objective function since it is the average over the losses. Therefore, the objective function is continuous and treated as a differentiable function. So we might use gradient-based methods to optimize it.

	$\sigma(z)$	$\sigma'(z)$
logistic function	$1/(1 + e^{-z})$	$e^{-z}/(1 + e^{-z})^2$
ReLU	$\max(z, 0)$	$\begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z < 0 \end{cases}$
leaky ReLU	$\max(z, 0.1z)$	$\begin{cases} 1 & \text{if } z > 0 \\ 0.1 & \text{if } z < 0 \end{cases}$

Table 3.2: Derivatives of common activation functions.

As already mentioned, convexity is a crucial property for optimization problems. Although some components, like the loss functions, are convex, no general statement can

be made concerning their concatenation. Moreover, considering that the losses depend on θ in a highly non-linear fashion, it seems unlikely that they are convex regarding θ. Therefore the objective function of neural networks is considered to be non-convex in general.

3.3 Local Minima

Since the objective function is non-convex, there could be local minima with high costs compared to the global minima. This would pose a significant problem, as optimization algorithms could get trapped at a poor local minimum. In order to examine whether this problem occurs, we study the local minima of the objective function in this section. The contents of this section are mainly based on the work of Baldi and Hornik [2], Bruhn-Fujimoto [5], Dauphin et al. [6], Goodfellow et al. [11], and Soudry and Carmon [20].

The objective function of neural networks usually has many local minima because of the *model identifiability* problem. A model is called *identifiable* if its parameters can be uniquely determined using a sufficiently large training set. This condition is not met for most neural networks. Consider, for instance, the neural network in Figure 3.2. The two neurons in the hidden layer receive the same input and forward their output to the same neuron. If we permute them along with their weights and biases, the corresponding prediction function stays the same, whereas the parameter vector θ changes.

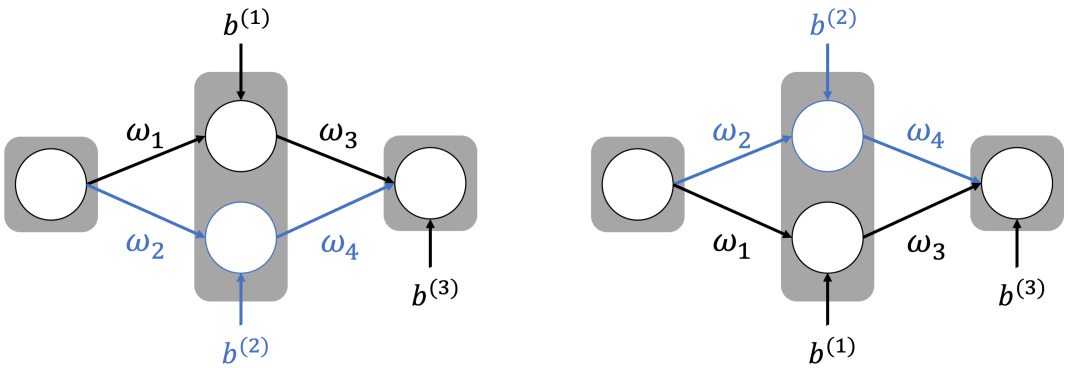


Figure 3.2: Two parameterizations of a neural network representing the same prediction function.

For deeper and wider neural networks, there are even more possibilities to permute the neurons. If a neural network has L layers with width d, there exist $(d!)^L$ ways to arrange the neurons. This kind of non-identifiability is also called *weight space symmetry*.

The scaling of the weights and biases constitutes another source of non-identifiability for neural networks that use the (leaky) ReLU. For any neuron with such an activation

function, we can multiply all incoming weights and the bias with $\alpha > 0$ if we also scale the outgoing weights with $1/\alpha$. This way, any prediction function of the neural network can be expressed by an infinite set of parameter vectors.

While each local minimum accounts for many other local minima due to the non-identifiability, they are all equivalent in cost. So the fact that there are many local minima does not necessarily imply that there exist local minima with high costs compared to global minima. The following result of Baldi and Hornik [2] exemplifies this consideration for a specific class of neural networks.

Let us consider neural networks with m inputs, a single hidden layer of $p \leq m$ neurons, and m output neurons. For simplicity, we omit the bias terms and use the identity as the activation function for all neurons. Such neural networks are called *linear neural networks* because their prediction functions correspond to a matrix-vector product

$$h_\theta(x) = h(x, W^{(1)}, W^{(2)}) = W^{(2)}W^{(1)}x.$$

While such networks are not practical since they can only express linear input-output relations, they are still helpful as a model of non-linear neural networks because their objective function

$$f(W^{(1)}, W^{(2)}) = \frac{1}{n}\sum_{i=1}^{n}\|y_i - W^{(2)}W^{(1)}x_i\|_2^2$$

is a non-convex function of the parameters $W^{(1)}$ and $W^{(2)}$. Moreover, linear neural networks are also affected by the model identifiable problem because for any invertible $p \times p$ matrix C, we have

$$W^{(2)}W^{(1)} = \underbrace{W^{(2)}C}_{:=\hat{W}^{(2)}}\underbrace{C^{-1}W^{(1)}}_{:=\hat{W}^{(1)}} = \hat{W}^{(2)}\hat{W}^{(1)}.$$

To formulate the main result of Baldi and Hornik, we need some more notation. We define $\Sigma_{XX} := \sum_{i=1}^n x_i x_i^\intercal$, $\Sigma_{XY} := \sum_{i=1}^n x_i y_i^\intercal$, $\Sigma_{YX} := \sum_{i=1}^n y_i x_i^\intercal$, and $\Sigma_{YY} := \sum_{i=1}^n y_i y_i^\intercal$. Provided that Σ_{XX} is invertible, we denote $\Sigma := \Sigma_{YX}\Sigma_{XX}^{-1}\Sigma_{XY}$.

Theorem 3.5. *Assume that Σ_{XX} is invertible and that Σ has full rank and m distinct eigenvalues $\lambda_1 > \ldots > \lambda_m$. If $\mathcal{I} = \{i_1, \ldots, i_p\}$ $(1 \leq i_1 < \ldots, i_p \leq m)$ is any ordered p-index set, let $U_\mathcal{I} = [u_{i_1}, \ldots, u_{i_p}]$ denote the matrix formed by the orthonormal eigenvectors of Σ associated with the eigenvalues $\lambda_{i_1}, \ldots, \lambda_{i_p}$. Then two full rank matrices $W^{(1)}$ and $W^{(2)}$ define a stationary point of f if and only if there exists an ordered p-index set $\mathcal{I}$ and an invertible $p \times p$ matrix C such that*

$$W^{(2)} = U_\mathcal{I}\, C$$
$$W^{(1)} = C^{-1}U_\mathcal{I}^\intercal\, \Sigma_{YX}\Sigma_{XX}^{-1}.$$

For such a stationary point, we have

$$f(W^{(1)}, W^{(2)}) = \frac{1}{n}\left(\operatorname{tr}(\Sigma_{YY}) - \sum_{i\in\mathcal{I}} \lambda_i\right).$$

For the p-index set $\mathcal{I} = \{1, \ldots, p\}$, the full rank matrices $W^{(1)}$ and $W^{(2)}$ constitute a global minimum of the objective function. The remaining $\binom{m}{p} - 1$ p-index sets correspond to saddle points. Additional stationary points defined by matrices $W^{(1)}$ and $W^{(2)}$, which are not of full rank, are also saddle points.

Proof. See Baldi and Hornik [2, Fact 4]. □

Although the objective function of these linear neural networks has many local minima due to the model identifiable problem, they are all global minima. Stationary points whose cost is higher than that of the global minima correspond to saddle points. Without proof, Baldi and Hornik stated that this result could be extended for linear neural networks with more than one hidden layer.

Soudry and Carmon [20] provide a similar result for a more relevant class of neural networks. Consider a neural network with d_0 inputs, one hidden layer with d_1 neurons, and a single output neuron. Thereby the activation function of the hidden layer is the leaky ReLU, and we use the identity as the activation function of the output neuron. For simplicity, we omit the bias terms. Overall, we can write the output of this neural network on the i^{th} input of the training set as

$$h_\theta(x_i) = W^{(2)}\sigma^{(1)}\left(W^{(1)}x_i\right) = W^{(2)}\ \operatorname{diag}(a_i)\ W^{(1)}x_i\ ,$$

with

$$(a_i)_h := \begin{cases} 1 & \text{if } (W^{(1)}x_i)_h \geq 0 \\ 0.1 & \text{otherwise} \end{cases}.$$

Now we do something that is usually not part of the training. We apply a perturbation to the activation function. To be more precise, we replace a_i with

$$(a_i)_h = (\varepsilon_i)_h \begin{cases} 1 & \text{if } (W^{(1)}x_i)_h \geq 0 \\ 0.1 & \text{otherwise} \end{cases}.$$

We collect all perturbation terms in the matrix $\mathcal{E} = [\varepsilon_1, \ldots, \varepsilon_n]$ and assume that the entries of this matrix are i.i.d. according to a Gaussian distribution. Further, we collect all inputs into a matrix $X = [x_1, \ldots, x_n]$ and assume that this matrix is subject to arbitrarily small Gaussian i.i.d. noise.

By employing the square loss as the loss function, we obtain the objective function

$$f(\theta) = \frac{1}{n}\sum_{i=1}^{n}(y_i - W^{(2)}\ \operatorname{diag}(a_i)\ W^{(1)}x_i)^2.$$

As discussed in Section 3.2, this objective function is not differentiable everywhere since the neural network uses the leaky ReLU. Therefore let us call

$$\theta = \left(\omega_{1,1}^{(1)}, \ldots, \omega_{d_1,d_0}^{(1)}, \omega_{1,1}^{(2)} \ldots, \omega_{1,d_1}^{(2)}\right)^\intercal$$

a *differentiable local minimum* if f is differentiable at θ, and if θ is a local minimum of f. In addition, we say that some statement $\phi(x)$ that depends on $x \in \mathbb{R}^d$ holds for *almost all* $x \in \mathbb{R}^d$ if the set of exceptions

$$\{x \in \mathbb{R}^d : \phi(x) \text{ false}\}$$

has Lebesgue measure zero. Having defined these notions, we can finally state the result.

Theorem 3.6. *If $n \leq d_1 d_0$, then all differentiable local minima of the objective function are global minima with $f(\theta) = 0$ for almost all $(X, \mathcal{E})$.*

Proof. See Soudry and Carmon [20, Theorem 4]. □

In other words, if we only use enough neurons in the hidden layer, we expect every differentiable local minimum to be a global minimum with zero cost. Note that we introduced the perturbation $\mathcal{E}$ for technical reasons. In practice, we expect this property to hold even if we use the leaky ReLU without any perturbation. Altogether, the presented theoretical results suggest that poor local minima are not necessarily a severe problem. Empirical studies on this topic come to similar conclusions.

Dauphin et al. [6] studied neural networks with a single hidden layer and found that the Hessian matrix of the objective function tends to have more positive eigenvalues for stationary points with a lower cost. Based on this observation, they draw a connection between the objective function of neural networks and random Gaussian error functions. The theory of random Gaussian error functions might help us better understand the objective function of neural networks.

In high-dimensional spaces, stationary points of random Gaussian error functions are most likely to be saddle points. To gain an intuition for this property, recall that the Hessian matrix at a local minimum has only non-negative eigenvalues. Conversely, if the Hessian matrix has both positive and negative eigenvalues at a stationary point, it is a saddle point. As the dimension grows, it becomes more and more unlikely that all eigenvalues have the same sign, considering that the dimension coincides with the number of eigenvalues. Furthermore, random Gaussian error functions have another intriguing property: in regions with low error, the eigenvalues of the Hessian matrix are more likely to be positive. Together, these properties imply that stationary points with large errors are most likely saddle points, whereas local minima tend to have a low error.

Based on the empirical evidence and this consideration, Dauphin et al. argue that high-cost saddle points pose a bigger problem for the training of neural networks than bad local minima. For this reason, gradient-based methods come at hand for the training of neural networks, as they are known to move away from saddle points, whereas second-order methods like Newton's method get attracted by saddle points.

4 Gradient-Based Optimization

As pointed out in the last chapter, gradient-based methods seem to be a convenient choice for training neural networks. Therefore we present various gradient-based methods and analyze their behavior for non-convex optimization problems such as the training of neural networks.

4.1 Backpropagation

Before we can apply gradient-based methods, we need a practical method to compute the respective gradients. For this purpose, we derive the *backpropagation* algorithm, which enables an efficient gradient computation for the nested objective function of neural networks. This section is based on the work of Bruhn-Fujimoto [5], Nielsen [17], and Urban [22].

First, let us simplify the gradient of the objective function with the sum and the product rule:

$$\nabla f(\theta) = \nabla \left(\frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(y_i, h(x_i, \theta)) \right) = \frac{1}{n} \sum_{i=1}^{n} \nabla \mathcal{L}(y_i, h(x_i, \theta)).$$

As can be seen, the gradient of the objective function corresponds to the average of the loss gradients $\nabla \mathcal{L}(y_i, h(x_i, \theta))$. Therefore, it suffices to focus on the computation of these loss gradients. Accordingly, we need a method that can compute the gradient $\nabla \mathcal{L}(y, h(x, \theta))$ for any observation $(x, y) \in \mathcal{X} \times \mathcal{Y}$. Note that the gradient is taken with respect to the parameter vector θ and thus contains all quantities

$$\frac{\partial \mathcal{L}(y, h(x, \theta))}{\partial \omega_{i,j}^{(\ell)}} \quad \text{and} \quad \frac{\partial \mathcal{L}(y, h(x, \theta))}{\partial b_i^{(\ell)}}.$$

These partial derivatives describe how infinitesimal changes in the respective parameters affect the loss. Therefore, it is worthwhile to examine how changes in the weights and biases impact the loss. Consider as an example the neural network in Figure 4.1. If we change the weight $\omega_{1,1}^{(1)}$, this will initially only affect the adjacent neuron and its output. However, as an input to the following layer, this altered output also impacts the outputs of all neurons of the subsequent layer. These changes, in turn, affect the next layer and then the next until they reach the output layer and consequently the loss. So, to understand the influence of the change in $\omega_{1,1}^{(1)}$, we first need to know how

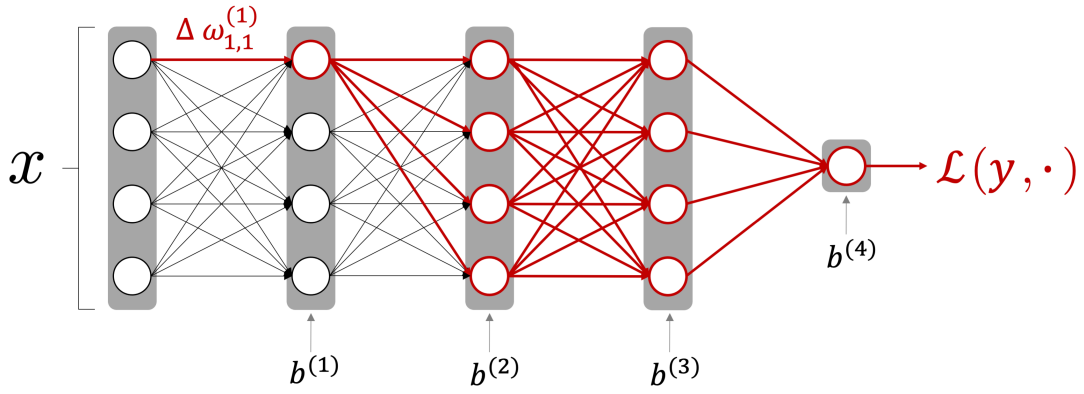


Figure 4.1: The way a change in a single weight affects the loss.

changes in the following layers affect the loss. For this reason, we will start with the partial derivatives for the output layer and then work our way backwards through the neural network layer by layer.

Before we start, let us fix some $(x, y) \in \mathcal{X} \times \mathcal{Y}$. We use $x^{(\ell)}$ to refer to the output of the ℓ^{th} layer of the neural network. In addition, we define the auxiliary variable

$$z^{(\ell)} := W^{(\ell)} x^{(\ell-1)} + b^{(\ell)}, \quad \text{i.e.} \quad x^{(\ell)} = \sigma^{(\ell)}(z^{(\ell)}).$$

For simplicity, we assume that the neural network has only a single output neuron, i.e., $x^{(L)} = x_1^{(L)}$ and $z^{(L)} = z_1^{(L)}$.

Note that the loss $\mathcal{L}(y, h(x, \theta))$, which we will abbreviate with $\mathcal{L}$, can be written as a function of $x^{(L)}$. Further, recall that $x^{(L)}$ is a function of $z^{(L)}$, which is, in turn, a function of the weights and biases of the output layer:

$$\mathcal{L}(y, h(x, \theta)) = \mathcal{L}(y, x^{(L)}), \quad x^{(L)} = \sigma^{(L)}(z^{(L)}) \text{ and } z^{(L)} = W^{(L)} x^{(L-1)} + b^{(L)}.$$

Consequently, the partial derivatives of the weights and biases of the output layer can be obtained by applying the chain rule multiple times:

$$\frac{\partial \mathcal{L}}{\partial b_1^{(L)}} = \frac{\partial \mathcal{L}}{\partial x_1^{(L)}} \cdot \frac{\partial x_1^{(L)}}{\partial z_1^{(L)}} \cdot \frac{\partial z_1^{(L)}}{\partial b_1^{(L)}} \tag{4.1}$$

$$\frac{\partial \mathcal{L}}{\partial \omega_{1,j}^{(L)}} = \frac{\partial \mathcal{L}}{\partial x_1^{(L)}} \cdot \frac{\partial x_1^{(L)}}{\partial z_1^{(L)}} \cdot \frac{\partial z_1^{(L)}}{\partial \omega_{1,j}^{(L)}}. \tag{4.2}$$

Observe that (4.1) and (4.2) share a factor. We denote this factor as

$$\delta_1^{(L)} := \frac{\partial \mathcal{L}}{\partial z_1^{(L)}} = \frac{\partial \mathcal{L}}{\partial x_1^{(L)}} \cdot \frac{\partial x_1^{(L)}}{\partial z_1^{(L)}} = \frac{\partial \mathcal{L}}{\partial x_1^{(L)}} \cdot \sigma'^{(L)}(z_1^{(L)}). \tag{4.3}$$

The factor can be computed using the formulas for the derivatives of the loss and the activation functions from Tables 3.1 and 3.2. The remaining factors of (4.1) and (4.2) are given by

$$\frac{\partial z_1^{(L)}}{\partial \omega_{1,j}^{(L)}} = x_j^{(L-1)} \text{ and } \frac{\partial z_1^{(L)}}{\partial b_1^{(L)}} = 1.$$

Altogether, we obtain the formulas

$$\frac{\partial \mathcal{L}}{\partial \omega_{1,j}^{(L)}} = \delta_1^{(L)} \cdot x_j^{(L-1)} \text{ and } \frac{\partial \mathcal{L}}{\partial b_1^{(L)}} = \delta_1^{(L)} \cdot 1.$$

Fortunately, the partial derivatives of the other layers can be computed similarly. We proceed by broadening the notion of δ:

$$\delta^{(\ell)} := \left(\frac{\partial \mathcal{L}}{\partial z_1^{(\ell)}}, \dots, \frac{\partial \mathcal{L}}{\partial z_{d_\ell}^{(\ell)}} \right)^\intercal.$$

With this notation, we can express the partial derivatives of the parameters of the ℓ^{th} layer as

$$\frac{\partial \mathcal{L}}{\partial \omega_{i,j}^{(\ell)}} = \frac{\partial \mathcal{L}}{\partial z_i^{(\ell)}} \cdot \frac{\partial z_i^{(\ell)}}{\partial \omega_{i,j}^{(\ell)}} = \delta_i^{(\ell)} \cdot \frac{\partial z_i^{(\ell)}}{\partial \omega_{i,j}^{(\ell)}} = \delta_i^{(\ell)} \cdot x_j^{(\ell-1)} \tag{4.4}$$

$$\frac{\partial \mathcal{L}}{\partial b_i^{(\ell)}} = \frac{\partial \mathcal{L}}{\partial z_i^{(\ell)}} \cdot \frac{\partial z_i^{(\ell)}}{\partial b_i^{(\ell)}} = \delta_i^{(\ell)} \cdot \frac{\partial z_i^{(\ell)}}{\partial b_i^{(\ell)}} = \delta_i^{(\ell)} \cdot 1. \tag{4.5}$$

So far, we omitted how $\delta^{(\ell)}$ can be computed. By applying the multivariate chain rule, we obtain a recursive formula that allows us to compute $\delta_i^{(\ell)}$ given $\delta^{(\ell+1)}$:

$$\delta_i^{(\ell)} = \frac{\partial \mathcal{L}}{\partial z_i^{(\ell)}} = \sum_{k=1}^{d_{\ell+1}} \frac{\partial \mathcal{L}}{\partial z_k^{(\ell+1)}} \cdot \frac{\partial z_k^{(\ell+1)}}{\partial z_i^{(\ell)}} = \sum_{k=1}^{d_{\ell+1}} \delta_k^{(\ell+1)} \cdot \frac{\partial z_k^{(\ell+1)}}{\partial z_i^{(\ell)}}$$

Considering that $z^{(\ell+1)} = W^{(\ell+1)} x^{(\ell)} + b^{(\ell+1)} = W^{(\ell+1)} \sigma^{(\ell)}(z^{(\ell)}) + b^{(\ell+1)}$, it can be seen that

$$\frac{\partial z_k^{(\ell+1)}}{\partial z_i^{(\ell)}} = \omega_{k,i}^{(\ell+1)} \cdot \sigma'^{(\ell)}(z_i^{(\ell)}).$$

Together, we obtain

$$\delta_i^{(\ell)} = \sum_{k=1}^{d_{\ell+1}} \delta_k^{(\ell+1)} \cdot \omega_{k,i}^{(\ell+1)} \cdot \sigma'^{(\ell)}(z_i^{(\ell)}). \tag{4.6}$$

Note that we require $x^{(\ell)}$ and $z^{(\ell)}$ for the formulas (4.4) and (4.6), respectively. Therefore the computation of the gradient consists of two steps:

(i) Let the input x run forward through the network in order to obtain all $x^{(\ell)}$ and $z^{(\ell)}$.

(ii) Run backward through the neural network in order to obtain all $\delta_i^{(\ell)}$.

Finally, we collect the formulas (4.3), (4.4), (4.5), and (4.6) in Algorithm 1:

Algorithm 1: Backpropagation

input : A neural network N, a loss function $\mathcal{L}(y, \hat{y})$, and a single observation $(x, y) \in \mathcal{X} \times \mathcal{Y}$.

Compute all terms $x^{(\ell)}$ and $z^{(\ell)}$;

$\delta_1^{(L)} = \frac{\partial \mathcal{L}}{\partial x_1^{(L)}} \cdot \sigma'^{(L)}(z_1^{(L)})$;

Set $\frac{\partial \mathcal{L}}{\partial b_1^{(L)}} = \delta_1^{(L)}$;

Set $\frac{\partial \mathcal{L}}{\partial \omega_{1,j}^{(L)}} = \delta_1^{(L)} \cdot x_j^{(L-1)}$ for $j = 1, \ldots, d_{L-1}$;

for $\ell = L-1, \ldots, 1$ **do**

  $\delta_i^{(\ell)} = \sum_{k=1}^{d_{\ell+1}} \delta_k^{(\ell+1)} \cdot \omega_{k,i}^{(\ell+1)} \cdot \sigma'^{(\ell)}(z_i^{(\ell)})$ for $i = 1, \ldots, d_\ell$;

  Set $\frac{\partial \mathcal{L}}{\partial b_i^{(\ell)}} = \delta_i^{(\ell)}$ for $i = 1, \ldots, d_\ell$;

  Set $\frac{\partial \mathcal{L}}{\partial \omega_{i,j}^{(\ell)}} = \delta_i^{(\ell)} \cdot x_j^{(\ell-1)}$ for $i = 1, \ldots, d_\ell,\ j = 1, \ldots, d_{\ell-1}$;

end

output: The loss gradient $\nabla \mathcal{L}(y, h(x, \theta))$.

Whenever we mention the computation of the gradients $\nabla f(\theta)$ or $\nabla \mathcal{L}_i(\theta)$ in the following sections, this will be done with the help of the backpropagation algorithm.

4.2 Gradient Descent

Now that we know how to compute the gradient of the objective function, we can shift our attention to gradient-based methods. To begin with, let us consider gradient descent as it is the most basic method of this class. This section rests on the work of Bottou et al. [4], Géron [10], Lee et al. [13], Poliak [18], and Ulbrich and Ulbrich [21].

Gradient descent is a numerical method for unconstrained optimization problems with a differentiable objective function. Starting from an initial point θ_1, the method defines an iteration sequence

$$\theta_{k+1} = \theta_k - \alpha \nabla f(\theta_k), \tag{4.7}$$

where $\alpha > 0$ is the so-called *step size*. In order to motivate this sequence, consider the first-order Taylor polynomial

$$f(\theta_k + v) \approx f(\theta_k) + \nabla f(\theta_k)^\intercal v, \tag{4.8}$$

which approximates $f \in C^1(\mathbb{R}^d)$ locally in θ_k. Regarding this approximation, a convenient direction to move would be the direction of steepest descent, i.e., the direction $v \in \mathbb{R}^d$ that solves

$$\min_{\|v\|=1} f(\theta_k) + \nabla f(\theta_k)^\intercal v. \tag{4.9}$$

If the objective function is differentiable and $\nabla f(\theta_k) \neq 0$, the unique solution of (4.9) is given by

$$v = -\frac{\nabla f(\theta_k)}{\|\nabla f(\theta_k)\|}$$

(see Ulbrich and Ulbrich [21, Theorem 7.4]). This consideration motivates the use of the negative gradient in iteration sequence (4.7). Formally, we define *gradient descent* as Algorithm 2.

Algorithm 2: Gradient Descent

```
Choose the step size α and the initial iterate θ_1;
for k = 1, 2, ... do
    Compute the gradient ∇f(θ_k);
    Set the new iterate as θ_{k+1} = θ_k − α∇f(θ_k);
end
```

To illustrate the behavior of gradient-based methods, we use a simple regression problem. The training set consists of 100 randomly generated inputs $x \in \mathbb{R}^2$ for which the corresponding output was generated by the formula

$$y = x_1 + x_2 + \varepsilon \quad \text{with } \varepsilon \sim \mathcal{N}(0, 1).$$

Since we can only represent two dimensions, we employ a neural network with no hidden layer and a single output neuron with no bias. Consequently, the parameter vector of this neural network is given by $\theta = (\omega_1, \omega_2)^\intercal$. Figure 4.2 depicts an iteration sequence of gradient descent, which exhibits the characteristic zig-zag behavior.

For gradient descent to be meaningful, the first-order Taylor polynomial (4.8) should approximate the objective function appropriately, at least in a small neighborhood of θ_k. In order to ensure this, the objective function has to be sufficiently smooth. Therefore we typically assume that the gradients of the objective function are Lipschitz-continuous.

Assumption 4.1 (Lipschitz-continuous gradients)**.** *The objective function $f : \mathbb{R}^d \to \mathbb{R}$ is continuously differentiable, and the gradient function of f, $\nabla f : \mathbb{R}^d \to \mathbb{R}^d$, is Lipschitz-continuous with Lipschitz constant $L > 0$, i.e.,*

$$\|\nabla f(\theta) - \nabla f(\bar{\theta})\|_2 \leq L\|\theta - \bar{\theta}\|_2 \quad \textit{for all } \theta, \bar{\theta} \in \mathbb{R}^d.$$

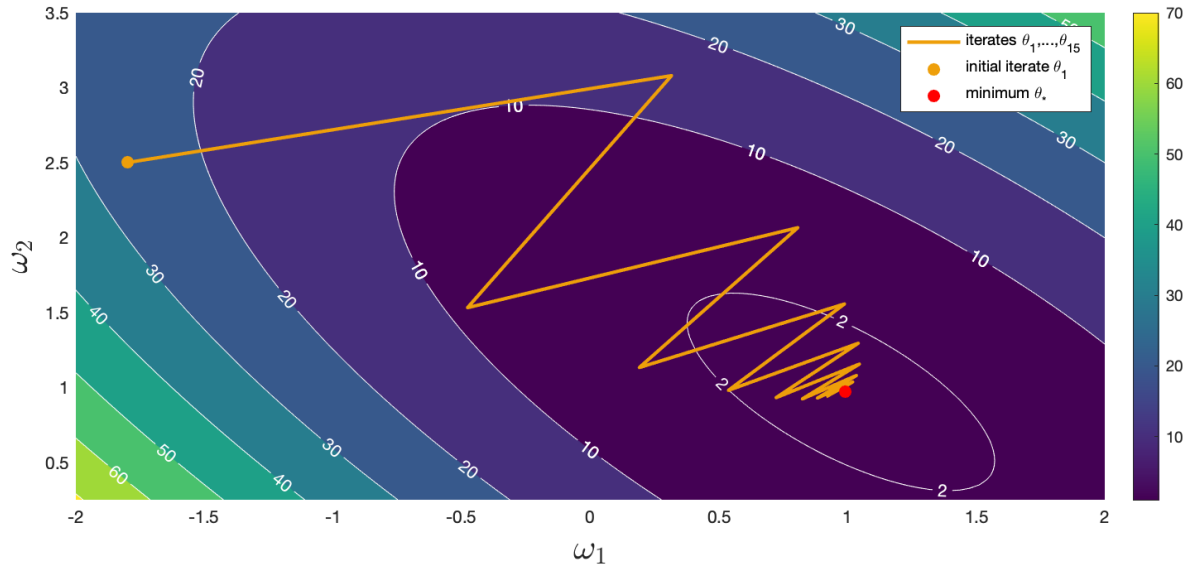


Figure 4.2: A typical iteration sequence of gradient descent. Adapted from Deisenroth et al. [7, Figure 7.3].

This assumption ensures that the gradient does not change arbitrarily fast. Such an assumption plays a crucial role in the convergence analysis of gradient-based methods. As an immediate consequence of Assumption 4.1, we have Lemma 4.2.

Lemma 4.2. *Under Assumption 4.1, it holds*

$$f(\theta) \leq f(\bar{\theta}) + \nabla f(\bar{\theta})^\intercal (\theta - \bar{\theta}) + \tfrac{1}{2} L \|\theta - \bar{\theta}\|_2^2 \quad \text{for all } \theta, \bar{\theta} \in \mathbb{R}^d. \tag{4.10}$$

Proof. See Bottou et al. [4, Appendix B]. □

However, even for a smooth objective function, the first-order Taylor polynomial (4.8) remains a local approximation. Thus, to guarantee that Algorithm 2 descends in each iteration, the step size must be chosen sufficiently small. This consideration leads to the following convergence result by Poliak [18, Theorem 1].

Theorem 4.3. *Under Assumptions 3.4 and 4.1, suppose that Algorithm 2 is run with a step size α satisfying the condition*

$$0 < \alpha < \frac{2}{L}. \tag{4.11}$$

Then, in Algorithm 2, the gradient tends to zero, i.e.,

$$\lim_{k \to \infty} \nabla f(\theta_k) = 0 \,,$$

and it holds $f(\theta_{k+1}) \leq f(\theta_k)$ for all $k \in \mathbb{N}$.

Proof. If we insert $\bar{\theta} = \theta_k$ and $\theta = \theta_{k+1} = \theta_k - \alpha \nabla f(\theta_k)$ into inequality (4.10), we obtain

$$f(\theta_{k+1}) \leq f(\theta_k) - \alpha \|\nabla f(\theta_k)\|_2^2 + \tfrac{1}{2} L \alpha^2 \|\nabla f(\theta_k)\|_2^2 \tag{4.12}$$
$$\leq f(\theta_k) - \alpha(1 - \tfrac{1}{2} L \alpha) \|\nabla f(\theta_k)\|_2^2. \tag{4.13}$$

Summing both sides of this inequality for $k \in \{1, \ldots, K\}$ and setting $\gamma := \alpha(1 - \frac{1}{2} L \alpha)$, we get

$$f(\theta_{K+1}) \leq f(\theta_1) - \gamma \sum_{k=1}^{K} \|\nabla f(\theta_k)\|_2^2.$$

Since $\gamma > 0$ due to the step size condition (4.11) and f is bounded from below by f_{inf}, it follows that

$$\sum_{k=1}^{K} \|\nabla f(\theta_k)\|_2^2 \leq \frac{f(\theta_1) - f(\theta_{K+1})}{\gamma} \leq \frac{f(\theta_1) - f_{\text{inf}}}{\gamma}$$

for all $K \in \mathbb{N}$, i.e., $\sum_{k=1}^{\infty} \|\nabla f(\theta_k)\|_2^2 < \infty$, yielding $\|\nabla f(\theta_k)\|_2^2 \to 0$. By considering inequality (4.13) and the step size condition (4.11), it can be seen that we have $f(\theta_{k+1}) \leq f(\theta_k)$ for all $k \in \mathbb{N}$. □

In other words, gradient descent converges against a stationary point of the objective function. Considering that the objective function may exhibit high-cost saddle points or local maxima, the use of gradient descent for the training of neural networks seems questionable at first glance. However, if the Hessian matrix has at least one negative eigenvalue at a certain stationary point, i.e., the point is a strict saddle, the following result of Lee et al. [13] states that gradient descent will not converge against that point almost surely.

Theorem 4.4. *In addition to Assumption 4.1, assume that f is twice continuously differentiable and that θ^* is a strict saddle. Suppose that the initial point θ_1 is chosen randomly and that Algorithm 2 is run with a step size α satisfying the condition $0 < \alpha < \frac{1}{L}$, then*

$$\mathbb{P}\left(\lim_{k \to \infty} \theta_k = \theta^*\right) = 0.$$

Proof. See Lee et al. [13, Theorem 4.1]. □

Intuitively, gradient descent does not converge against strict saddles because they have at least one direction with strictly negative curvature along which gradient descent can escape. Figure 4.3 exemplifies this consideration for a saddle point with a strictly negative curvature along the direction of the x-axis. Further, assume that each stationary point of f is either a local minimum or a strict saddle and that the set of all strict saddles has at most countably infinite cardinality. In this case, gradient descent converges to a

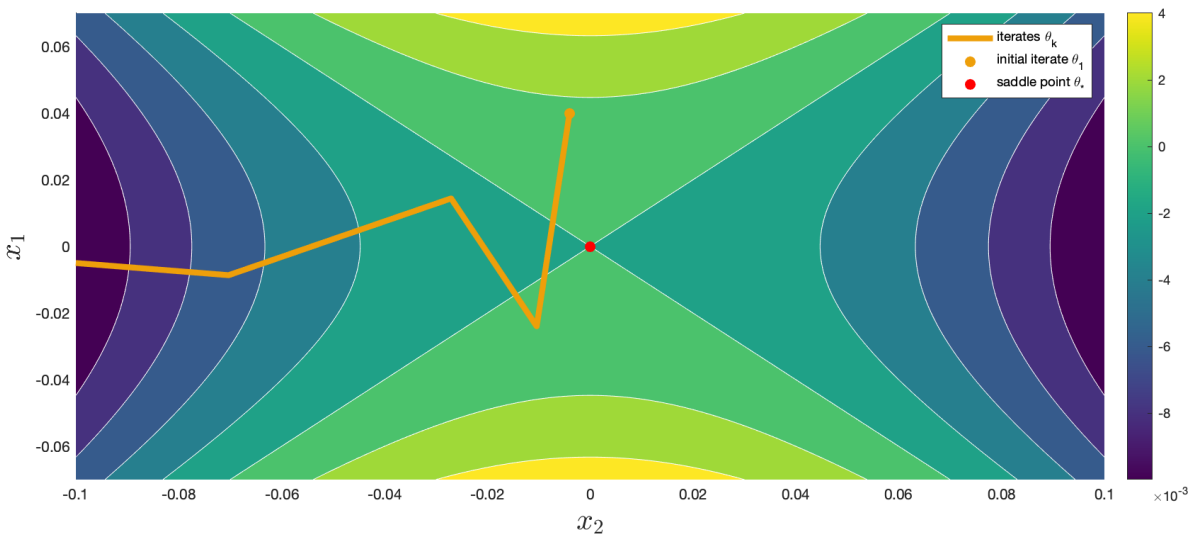


Figure 4.3: Gradient descent escapes a saddle point.

local minimum almost surely under the conditions of Theorem 4.4 and Assumption 3.4 (see Lee et al. [13, Corollary 4.6]).

In the context of machine learning, gradient descent is often referred to as *batch gradient descent* since the whole training set is involved in each iteration. Indeed, to compute the gradient

$$\nabla f(\theta) = \frac{1}{n} \sum_{i=1}^{n} \nabla \mathcal{L}_i(\theta)$$

in Algorithm 2, the loss gradients of all observations in the training set have to be computed with the backpropagation algorithm. Since the training of neural networks often involves extensive training sets, the iterations of gradient descent can become very costly.

Moreover, the iterations of the gradient descent process the training set inefficiently. Consider, for example, a training set $\mathcal{S}$ consisting of ten copies of a set $\mathcal{S}_{\text{sub}}$. In this case, computing the gradient over the entire training set would be ten times more expensive than computing the gradient over the set $\mathcal{S}_{\text{sub}}$ while both computations yield the same result. Admittedly, this example was somewhat contrived; in practice, the training set will not contain exact duplicates of sample data. Still, large training sets usually exhibit a high degree of redundancy, making gradient descent inefficient.

4.3 Stochastic Gradient Methods

Stochastic gradient methods represent a promising alternative to gradient descent because they avoid the expensive gradient computation by replacing $\nabla f(\theta)$ with a cheaper

approximation. The following section is adapted from Bottou et al. [4], Bruhn-Fujimoto [5], Géron [10], and Ruder [19].

4.3.1 Stochastic Gradient Descent

The most basic implementation of this approach is given by the *stochastic gradient descent* algorithm. In each iteration $k \in \mathbb{N}$ of stochastic gradient desent, a random index i_k, which corresponds to an observation in the training set, is uniformly drawn from $\{1, \dots, n\}$. Then the respective loss gradient $\nabla \mathcal{L}_{i_k}(\theta_k)$ is used to replace $\nabla f(\theta_k)$, which leads to the update rule

$$\theta_{k+1} = \theta_k - \alpha \nabla \mathcal{L}_{i_k}(\theta_k).$$

Each iteration of stochastic gradient descent is very cheap since we only have to compute the loss gradient of a single observation. Further, the computational cost per iteration is now independent of the training set size.

However, this speed-up per iteration comes at a price; the iteration sequence $\{\theta_k\}$ is a stochastic process whose behavior is determined by the random sequence $\{i_k\}$. Therefore, we can no longer guarantee a descent in each iteration. Nonetheless, the gradient $\nabla \mathcal{L}_{i_k}(\theta)$ is an unbiased estimate for the gradient $\nabla f(\theta)$. Using the formula of the expected value with respect to the discrete uniform distribution $\mathcal{U}\{1, n\}$, we obtain

$$\begin{aligned} \mathbb{E}_{i_k \sim \mathcal{U}\{1,n\}}[\nabla \mathcal{L}_{i_k}(\theta)] &= \sum_{i=1}^{n} \underbrace{\mathbb{P}(i_k = i)}_{=1/n} \nabla \mathcal{L}_i(\theta) \\ &= \frac{1}{n} \sum_{i=1}^{n} \nabla \mathcal{L}_i(\theta) = \nabla f(\theta). \end{aligned}$$

4.3.2 Mini-Batch Gradient Descent

Mini-batch gradient descent represents a compromise between batch and stochastic gradient descent. In each iteration, a set of indices, the so-called *mini-batch* B, is uniformly drawn from $\{1, \dots, n\}$. Thereupon, the average of the corresponding gradients $\nabla \mathcal{L}_i(\theta_k)$ is used to replace $\nabla f(\theta_k)$, which leads to the update rule

$$\theta_{k+1} = \theta_k - \frac{\alpha}{|B|} \sum_{i \in B} \nabla \mathcal{L}_i(\theta_k).$$

Since more training observations are involved in each iteration, we expect to obtain better estimates for the gradient $\nabla f(\theta)$ with this mini-batch approach. For example, consider the j^{th} component of the gradient estimate. Because the indices in the mini-batch are i.i.d., the variance of this component is reduced by the factor $\frac{1}{|B|}$ in comparison

to stochastic gradient descent:

$$\operatorname{Var}\left[\frac{1}{|B|}\sum_{i\in B}(\nabla\mathcal{L}_i(\theta_k))_j\right] = \frac{1}{|B|^2}\operatorname{Var}\left[\sum_{i\in B}(\nabla\mathcal{L}_i(\theta_k))_j\right] = \frac{1}{|B|}\operatorname{Var}[(\nabla\mathcal{L}_i(\theta_k))_j].$$

Figure 4.4 illustrates this theoretical consideration; it can be seen that the iterates of stochastic gradient descent (left) underlie a much larger variation than the iterates of mini-batch gradient descent (right).

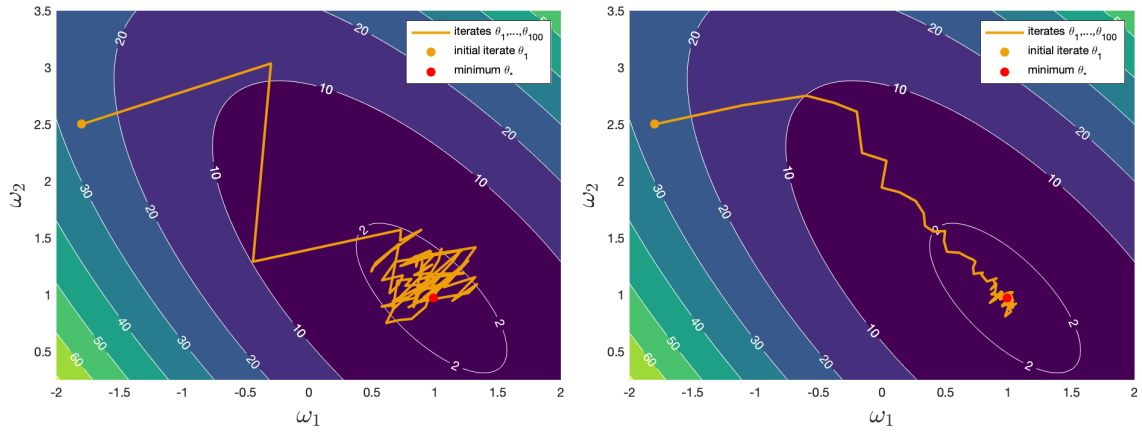


Figure 4.4: Comparison of stochastic gradient descent (left) and mini-batch gradient descent with mini-batch size 5 (right) applied with the same step size.

On the other hand, mini-batch gradient descent also requires the calculation of multiple loss gradients per iteration. Therefore the iterates are more expensive. Fortunately, modern hardware can quite efficiently handle the computation of these gradients in parallel. In a sense, computing only one loss gradient per iteration (as for stochastic gradient descent) would even waste machine efficiency.

4.4 Analysis of Stochastic Gradient Methods

Since stochastic gradient descent and mini-batch gradient descent are closely related, we will analyze them together. For this purpose, we consider a generalized stochastic gradient method that represents both of these methods. The structure and results in this section are adapted from Bottou et al. [4]. While the analysis of Bottou et al. also includes second-order stochastic methods, we will focus exclusively on gradient-based methods and make stronger assumptions.

4.4.1 Generalized Stochastic Gradient Method

Recall that stochastic gradient descent and mini-batch gradient descent differ in the number of indices drawn and the computation of the estimate for $\nabla f(\theta)$. Therefore, we

have to formulate the respective steps in a more generic sense. We define the *generalized stochastic gradient method* as Algorithm 3.

Algorithm 3: Stochastic Gradient Method

1 Choose an initial iterate θ_1;
2 **for** $k = 1, 2, \ldots$ **do**
3 | Generate a realization of the random variable ξ_k;
4 | Compute a stochastic gradient $g(\theta_k, \xi_k)$;
5 | Choose a step size $\alpha_k > 0$;
6 | Set the new iterate as $\theta_{k+1} = \theta_k - \alpha_k g(\theta_k; \xi_k)$;
7 **end**

Algorithm 3 may represent a variety of methods. In the context of stochastic and mini-batch gradient descent, we can understand it as follows:

(i) The realization of the random variable ξ_k in line 3 might be an index i_k (as in stochastic gradient descent) or a set of indices B (as in mini-batch gradient descent) uniformly drawn from $\{1, \ldots, n\}$. We assume that $\{\xi_k\}_{k \in \mathbb{N}}$ is a sequence of independent random variables.

(ii) Depending on the realization of ξ_k, the computation of the stochastic gradient in line 4 could follow a stochastic or a mini-batch approach:

$$g(\theta_k, \xi_k) = \begin{cases} \nabla \mathcal{L}_{i_k}(\theta_k) \\ \frac{1}{|B|} \sum_{i \in B} \nabla \mathcal{L}_i(\theta_k) \end{cases} .$$

(iii) Line 5 allows us to use various step size sequences. Our analysis focuses on two choices, one involving a fixed step size and one involving diminishing step sizes.

An essential part of the analysis of gradient descent was to derive an upper bound on the decrease in the objective function for each iteration. However, since a stochastic gradient is involved in the update rule, it is infeasible to determine such a deterministic bound for Algorithm 3. Instead, we attempt to derive a bound for the expected decrease in each iteration. In this context, we use $\mathbb{E}_{\xi_k}[\cdot]$ to denote the expected value taken with respect to the distribution of the random variable ξ_k given θ_k. A first bound for the expected decrease in each iteration is given by Lemma 4.5.

Lemma 4.5. *Under Assumption 4.1, the iterates of Algorithm 3 satisfy the following inequality for all $k \in \mathbb{N}$:*

$$\mathbb{E}_{\xi_k}[f(\theta_{k+1})] - f(\theta_k) \leq -\alpha_k \nabla f(\theta_k)^\intercal \mathbb{E}_{\xi_k}[g(\theta_k, \xi_k)] + \tfrac{1}{2}\alpha_k^2 L \mathbb{E}_{\xi_k}[\|g(\theta_k, \xi_k)\|_2^2]. \tag{4.14}$$

Proof. See Bottou et al. [4, Lemma 4.2]. □

So regardless of how Algorithm 3 arrived at θ_k, the expected decrease yielded by the k^{th} step depends on two properties:

(i) The expected directional derivative of f at θ_k along $-g(\theta_k, \xi_k)$.

(ii) The second moment of $g(\theta_k, \xi_k)$.

In order to deal with these properties, we state additional requirements on the first and second moments of the stochastic gradients.

Assumption 4.6 (First and second moment)**.** *Algorithm 3 satisfies the following conditions:*

(i) For all $k \in \mathbb{N}$, it holds

$$\mathbb{E}_{\xi_k}[g(\theta_k, \xi_k)] = \nabla f(\theta_k). \tag{4.15}$$

(ii) There exist scalars $M \geq 0$ and $M_G \geq 0$ such that, for all $k \in \mathbb{N}$,

$$\mathbb{E}_{\xi_k}[\|g(\theta_k, \xi_k)\|_2^2] \leq M + M_G \|\nabla f(\theta_k)\|_2^2. \tag{4.16}$$

The first assumption states that $g(\theta_k, \xi_k)$ is an unbiased estimate of $\nabla f(\theta_k)$. As we already discussed, stochastic and mini-batch gradient descent meet this assumption. The second assumption provides an upper bound for the second moment of $g(\theta_k, \xi_k)$. With these assumptions, we can refine Lemma 4.5.

Lemma 4.7. *Under Assumptions 4.1 and 4.6, the iterates of Algorithm 3 satisfy the following inequality for all $k \in \mathbb{N}$*

$$\mathbb{E}_{g_k}[f(\theta_{k+1})] - f(\theta_k) \leq \underbrace{-(1 - \tfrac{1}{2}\alpha_k L M_G)\alpha_k}_{\text{(I)}} \|\nabla f(\theta_k)\|_2^2 + \underbrace{\tfrac{1}{2}\alpha_k^2 L M}_{\text{(II)}}. \tag{4.17}$$

Proof. By inserting the assumptions (4.15) and (4.16) into inequality (4.14), we obtain

$$\begin{aligned} \mathbb{E}_{\xi_k}[f(\theta_{k+1})] - f(\theta_k) &\leq -\alpha_k \nabla f(\theta_k)^\intercal \mathbb{E}_{\xi_k}[g(\theta_k, \xi_k)] + \tfrac{1}{2}\alpha_k^2 L \mathbb{E}_{\xi_k}[\|g(\theta_k, \xi_k)\|_2^2] \\ &\leq -\alpha_k \|\nabla f(\theta_k)\|_2^2 + \tfrac{1}{2}\alpha_k^2 L (M + M_G \|\nabla f(\theta_k)\|_2^2). \end{aligned}$$

Rearranging yields inequality (4.17). □

Lemma 4.7 provides a deterministic upper bound for the expected decrease in the objective function for each iteration. Note that the term (I) is strictly negative for small α_k and suggests a decrease in the objective function by a magnitude proportional to $\|\nabla f(\theta_k)\|_2^2$. However, the term (II) could be large enough to allow the objective value to increase. Balancing these terms is critical for the choice of the step size.

4.4.2 Fixed Step Size

Having done this preliminary work, we can examine the asymptotic behavior of Algorithm 3 for fixed step sizes. In the following result, we use $\mathbb{E}[\cdot]$ to denote the expected value taken with respect to the joint distribution of all random variables $\{\xi_k\}_{k\in\mathbb{N}}$.

Theorem 4.8. *Under Assumptions 3.4, 4.1, and 4.6, suppose that Algorithm 3 is run with a fixed step size $\alpha_k = \alpha$ for all $k \in \mathbb{N}$, satisfying*

$$0 < \alpha \leq \frac{1}{LM_G}. \tag{4.18}$$

Then, the expected average squared norm of the gradients of f corresponding to the iterates of Algorithm 3 satisfies the following inequality for all $K \in \mathbb{N}$:

$$\mathbb{E}\left[\frac{1}{K}\sum_{k=1}^{K}\|\nabla f(\theta_k)\|_2^2\right] \leq \alpha LM + \frac{2(f(\theta_1) - f_{inf})}{K\alpha} \tag{4.19}$$

$$\xrightarrow{K\to\infty} \alpha LM. \tag{4.20}$$

Proof. Taking the total expectation of inequality (4.17) and using the step size condition (4.18), we have

$$\begin{aligned}\mathbb{E}[f(\theta_{k+1})] - \mathbb{E}[f(\theta_k)] &\leq -(1 - \tfrac{1}{2}\alpha LM_G)\alpha\mathbb{E}[\|\nabla f(\theta_k)\|_2^2] + \tfrac{1}{2}\alpha^2 LM \\ &\leq -\tfrac{1}{2}\alpha\mathbb{E}[\|\nabla f(\theta_k)\|_2^2] + \tfrac{1}{2}\alpha^2 LM.\end{aligned}$$

Summing both sides of this inequality for $k \in \{1, \ldots, K\}$ and using that f is bounded from below by f_{inf} gives

$$f_{\text{inf}} - f(\theta_1) \leq \mathbb{E}[f(\theta_{K+1})] - f(\theta_1) \leq -\tfrac{1}{2}\alpha\sum_{k=1}^{K}\mathbb{E}[\|\nabla f(\theta_k)\|_2^2] + \tfrac{1}{2}K\alpha^2 LM.$$

Rearranging then yields

$$\mathbb{E}\left[\sum_{k=1}^{K}\|\nabla f(\theta_k)\|_2^2\right] \leq K\alpha LM + \frac{2(f(\theta_1) - f_{inf})}{\alpha}.$$

Finally, dividing by K yields inequality (4.19). □

In the special case $M = 0$, where the noise reduces proportionally to $\|\nabla f(\theta_k)\|_2^2$, (4.20) implies that $\lim_{k\to\infty}\mathbb{E}[\|\nabla f(\theta_k)\|_2^2] = 0$. However, this is a unrealistic scenario. If $M > 0$, we can not prove convergence to a stationary point for the generalized stochastic gradient method with a fixed step size. Nevertheless, inequality (4.19) bounds the average squared norm of the gradients of f observed during the first K iterations. This quantity gets smaller when K increases, indicating that the stochastic gradient method

spends increasingly more time in regions where the objective function has a relatively small gradient.

As can be seen from the asymptotic result (4.20), the noise term M inhibits the stochastic gradient method from making further progress. This result motivates the use of mini-batch gradient descent. Consider the variance of the stochastic gradient $g(\theta_k, \xi_k)$, which we define as

$$\mathrm{Var}_{\xi_k}[g(\theta_k, \xi_k)] := \mathbb{E}_{\xi_k}[\|g(\theta_k, \xi_k)\|_2^2] - \|\mathbb{E}_{\xi_k}[g(\theta_k, \xi_k)]\|_2^2. \tag{4.21}$$

Taken together, Assumption 4.6, combined with (4.21), requires that the variance of $g(\theta_k, \xi_k)$ satisfies

$$\mathrm{Var}_{\xi_k}[g(\theta_k, \xi_k)] \leq M + (M_G - 1)\|\nabla f(\theta_k)\|_2^2. \tag{4.22}$$

In fact, our analysis would also hold if this bound on the variance were to be assumed directly (see Bottou et al. [4, p. 246]). If we use mini-batch gradient descent with mini-batch size $|B|$, the variance of the stochastic gradient $g(\theta_k, \xi_k)$ is reduced by a factor of $1/|B|$ compared to stochastic gradient descent (see Bottou et al. [4, p. 252]). That is, with respect to our analysis, the terms M and $M_G - 1$ in (4.22) are reduced by the same factor, becoming $M/|B|$ and $(M_G - 1)/|B|$ for mini-batch gradient descent. Therefore we expect the average squared norm of the gradients to decrease as we increase the mini-batch size.

4.4.3 Diminishing Step Size

Another property that plays a vital role in Theorem 4.8 is the step size. While we could make the average squared norm of the gradients arbitrarily small by shrinking the step size further and further, this would also reduce the rate at which the squared norm of the gradient approaches its limiting distribution. In this regard, diminishing step size sequences represent a compromise. They allow the stochastic gradient method to make rapid progress in the first iterations starting with large step sizes. When the iterates reach a region where the objective function has a small gradient, the diminishing step sizes enable even further progress. The following result reflects this consideration.

Theorem 4.9. *Under Assumptions 3.4, 4.1, and 4.6, suppose that Algorithm 3 is run with a step size sequence $\{\alpha_k\}_{k\in\mathbb{N}}$ satisfying*

$$\sum_{k=1}^{\infty} \alpha_k = \infty \quad \textit{and} \quad \sum_{k=1}^{\infty} {\alpha_k}^2 < \infty. \tag{4.23}$$

Then, with $A_K := \sum_{k=1}^{K} \alpha_k$ for $K \in \mathbb{N}$, we obtain

$$\lim_{K\to\infty} \mathbb{E}\left[\sum_{k=1}^{K} \alpha_k \|\nabla f(\theta_k)\|_2^2\right] < \infty \tag{4.24}$$

$$\textit{and therefore } \mathbb{E}\left[\frac{1}{A_K}\sum_{k=1}^{K} \alpha_k \|\nabla f(\theta_k)\|_2^2\right] \xrightarrow{K\to\infty} 0. \tag{4.25}$$

Proof. The second condition in (4.23) ensures that $\{\alpha_k\} \to 0$, meaning that, without loss of generality, we may assume that $\alpha_k L M_G \leq 1$ for all $k \in \mathbb{N}$. Subsequent, by taking the total expectation of inequality (4.17), we obtain

$$\begin{aligned}\mathbb{E}[f(\theta_{k+1})] - \mathbb{E}[f(\theta_k)] &\leq -(1 - \tfrac{1}{2}\alpha_k L M_G)\alpha_k \mathbb{E}[\|\nabla f(\theta_k)\|_2^2] + \tfrac{1}{2}\alpha_k^2 L M \\ &\leq -\tfrac{1}{2}\alpha_k \mathbb{E}[\|\nabla f(\theta_k)\|_2^2] + \tfrac{1}{2}\alpha_k^2 L M.\end{aligned}$$

Summing both sides of this inequality for $k \in \{1, \ldots, K\}$ and using that f is bounded from below by f_{inf} gives

$$f_{\text{inf}} - f(\theta_1) \leq \mathbb{E}[f(\theta_{K+1})] - f(\theta_1) \leq -\tfrac{1}{2}\sum_{k=1}^{K} \alpha_k \mathbb{E}[\|\nabla f(\theta_k)\|_2^2] + \tfrac{1}{2} L M \sum_{k=1}^{K} \alpha_k^2.$$

By multiplying this inequality with 2 and rearranging the terms, we obtain

$$\sum_{k=1}^{K} \alpha_k \mathbb{E}[\|\nabla f(\theta_k)\|_2^2] \leq 2(f(\theta_1) - f_{\text{inf}}) + L M \sum_{k=1}^{K} \alpha_k^2.$$

The second condition in (4.23) implies that the right-hand side of this inequality converges to a finite limit when K increases, proving (4.24). Then (4.25) follows since the first condition in (4.23) ensures that $A_K \to \infty$ as $K \to \infty$. □

So the expected weighted average of the squared norm of the gradients converges to zero even in the presence of noise. As a direct consequence of Theorem 4.9, we have the following Corollary that yields a more straightforward interpretation.

Corollary 4.10. *Under Assumptions 3.4, 4.1, and 4.6, suppose that Algorithm 3 is run with a step size sequence $\{\alpha_k\}_{k\in\mathbb{N}}$ satisfying (4.23). Then*

$$\liminf_{k\to\infty} \mathbb{E}[\|\nabla f(\theta_k)\|_2^2] = 0. \tag{4.26}$$

Proof. If (4.26) would not hold, it would contradict (4.24). □

In other words, for stochastic gradient methods with diminishing step sizes, the expected squared norm of the gradients cannot asymptotically stay bounded away from zero. Under additional assumptions, we can even state a more decisive convergence result.

Corollary 4.11. *Under the conditions of Theorem 4.9, if we further assume that the objective function f is twice differentiable and that the mapping $\theta \mapsto \|\nabla f(\theta)\|_2^2$ has Lipschitz-continuous derivatives, then*

$$\lim_{k\to\infty} \mathbb{E}[\|\nabla f(\theta_k)\|_2^2] = 0.$$

Proof. See Bottou et al. [4, Appendix B]. □

Thus, while we could solely show that stochastic gradient methods with a fixed step size spend much time in regions with a small gradient, we can achieve convergence against stationary points by employing diminishing step sizes. Still, considering that the objective function for the training of neural networks might exhibit high-cost saddle points, these theoretical guarantees seem to be relatively weak. Nonetheless, since stochastic gradient methods work with unbiased gradient estimates, they are supposed to behave similar to gradient descent, i.e., to escape such high-cost saddle points.

Under some additional assumptions, Ge et al. [8] derived a theoretical result that goes in line with this consideration. They analyzed an altered version of stochastic gradient descent, where noise is added to each update vector. Further, Ge et al. assumed that each point with a small gradient is either close to a robust local minimum or has a Hessian matrix with a significant negative eigenvalue. Under this assumption, they stated that that the noisy stochastic gradient descent applied with a sufficiently chosen fixed step size ends up close to a local minimum after a polynomial number of steps (see Gee et al. [8, Theorem 1]).

4.5 Challenges in Gradient-Based Optimization

Although gradient-based methods are widely used to train neural networks, they are by no means flawless. To emphasize this, we will address some challenges for the discussed methods in this section. The following is based on the work of Bishop [3], Bruhn-Fujimoto [5], Goodfellow et al. [11], Murphy [16], and Urban [22].

4.5.1 Parameter Initialization

Gradient-based methods (e.g., Algorithms 2 and 3) require the choice of an initial iterate θ_1. As training neural networks is a non-convex problem, this choice has a significant impact on the outcome of the gradient-based methods. It determines in which regions the methods will end up and thus influences the cost of the found solution.

How to choose good initial iterates is poorly understood. The only thing known with certainty is that the parameters should be initialized with different values. Assume, for example, that two neurons in a layer start with the same weights and biases. In this case, both neurons represent identical functions and contribute equally to the objective function. Thus their respective partial derivatives will be identical, and gradient-based methods will not distinguish between them; throughout all iterations, they will have the same parameters. To prevent this from happening, one typically randomly initializes the weights according to either the uniform or the Gaussian distribution and sets the biases to heuristically chosen constants.

4.5.2 Step Size Selection

As seen from the theoretical results in this chapter, the step size plays an essential role in gradient-based methods. For Theorems 4.3 and 4.8, we required the step size to be smaller than a threshold depending on the Lipschitz constant L. However, this Lipschitz constant is unknown in practice, making it difficult to choose an appropriate step size: a too-small step size leads to slow progress, whereas a too-large step size can cause harmful oscillations. Figure 4.5 exemplifies this consideration as it depicts stochastic gradient descent applied with a small and large step size. Therefore, there is usually nothing left to do but to determine the step size by trial and error.

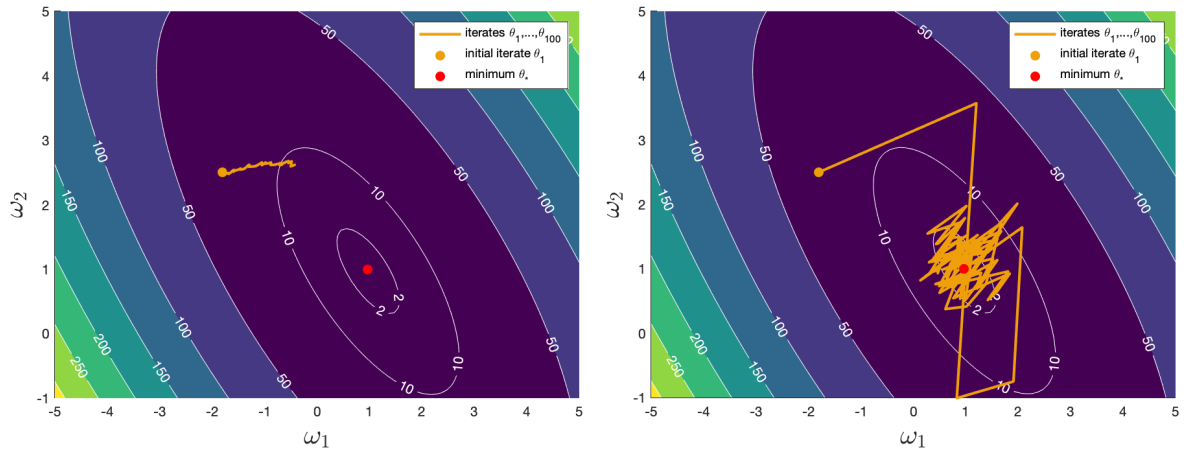


Figure 4.5: Stochastic gradient descent with step size $\alpha = 0.002$ (left) and step size $\alpha = 0.065$ (right).

In contrast, the condition on the decreasing step size sequence in Theorems 4.10 and 4.9 does not depend on the Lipschitz constant. While it is easy to construct sequences that satisfy this condition, not every sequence is practicable. Consider that we can only compute a finite number of steps, whereas the theoretical results are of asymptotic nature. A feasible sequence should make fast progress initially and slow down at the right time. To determine an appropriate step size sequence, one can use tunable step size schedules. Some of the most common ones are listed below:

(i) *piecewise constant*: $\alpha_k = \alpha_i$ if $k_i \leq k \leq k_{i+1}$
There are many possible choices for α_i. For example we may set $\alpha_i = \alpha_1 \gamma^i$, which reduces the initial learning rate by a factor γ for each threshold we pass. Figure 4.6 illustrates this for $\alpha_1 = 0.03$ and $\gamma = 0.5$. Sometimes the thresholds k_i are chosen adaptively, depending on the progress of the method.

(ii) *exponential decay*: $\alpha_k = \alpha_1 e^{-\lambda k}$
Exponential decay decreases the step size rapidly, as shown in Figure 4.6, where it is applied with $\lambda = 0.2$.

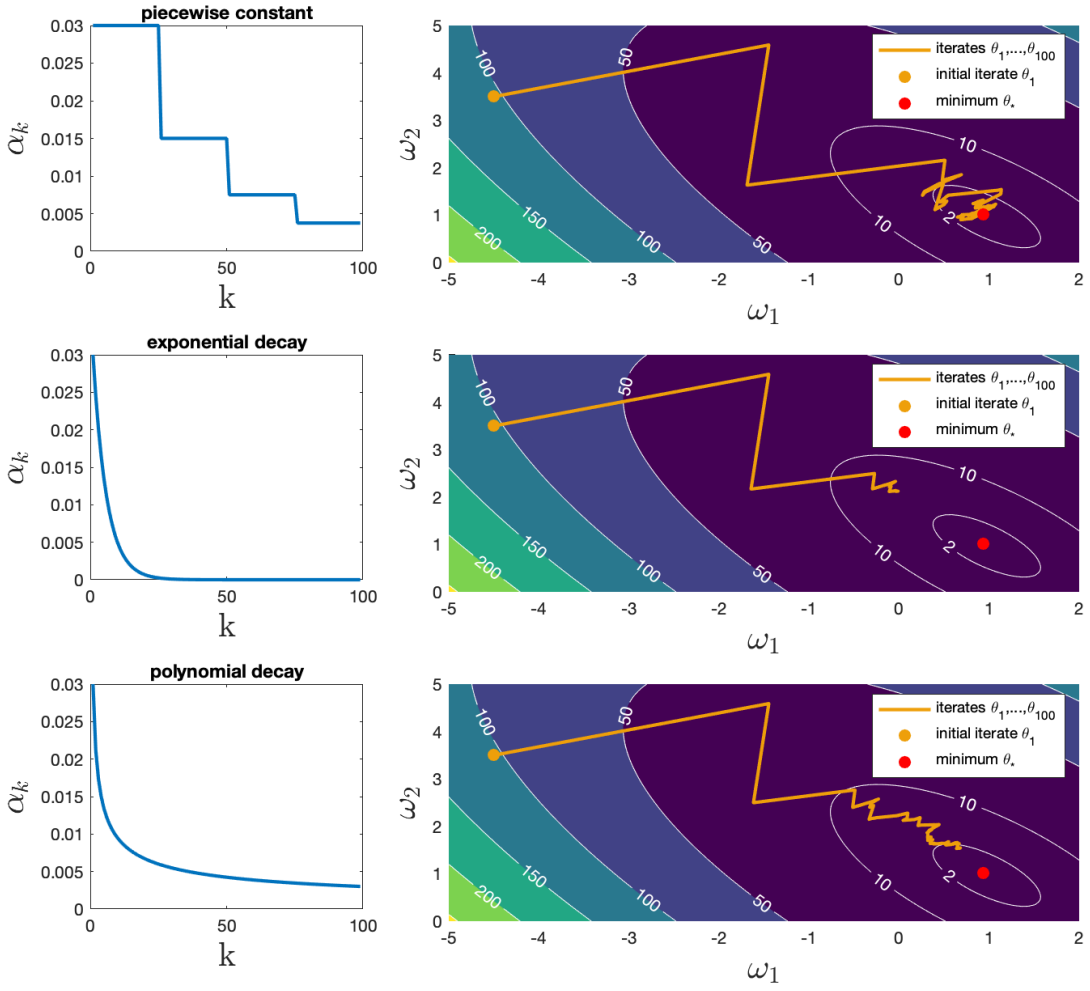


Figure 4.6: Comparison of a piecewise constant (top), an exponential decay (middle), and a polynomial decay (bottom) schedule for stochastic gradient descent. Adapted from Murphy [16].

(iii) *polynomial decay*: $\alpha_k = \alpha_1(\beta k + 1)^{-\gamma}$
A common choice for polynomial decay is $\beta = 1$ and $\gamma = 0.5$; this corresponds to a *square root schedule*. Figure 4.6 depicts such a schedule.

4.5.3 Random Sampling

Recall that the analysis of the stochastic gradient method was based on the assumption that the stochastic gradient is an unbiased estimate of the gradient ∇f. This assumption is met as long as the samples for stochastic and mini-batch gradient descent are drawn uniformly from the training set. However, to draw these random samples, we need to access the whole training set. While this is not a problem for small training sets, it can become very costly for extensive training sets, especially if they are stored in a distributed manner.

For this reason, one typically uses a different approach to get samples from the training set. Here, the training set is shuffled before the stochastic gradient method is started. Then the required samples are obtained by cycling through the shuffled training set, as shown in Figure 4.7. We expect this implementation to behave somewhat similar to the version we analyzed. However, the theoretical results do not transfer in general. Note that this modified version behaves like a deterministic method after the initial shuffling, which makes the analysis very difficult.

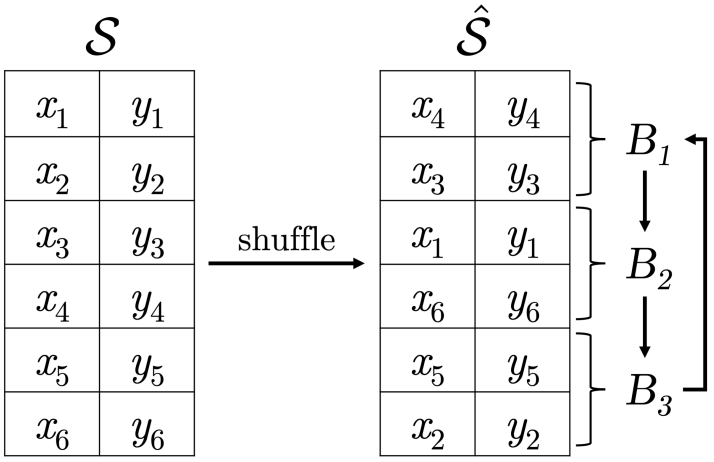


Figure 4.7: Alternative approach to generate samples from the training set.

4.5.4 Ill Conditioning

To understand the performance of gradient-based methods better, consider the second-order Taylor polynomial

$$f(\theta) \approx f(\theta_k) + (\theta - \theta_k)^\intercal \nabla f(\theta_k) + \tfrac{1}{2}(\theta - \theta_k)^\intercal \nabla^2 f(\theta_k)(\theta - \theta_k), \tag{4.27}$$

which approximates $f \in C^2(\mathbb{R}^d)$ in a neighborhood of the point θ_k. We can use this approximation to predict how a step of gradient descent will affect the cost. By inserting $\theta_{k+1} = \theta_k - \alpha_k \nabla f(\theta_k)$ into (4.27), we get

$$f(\theta_{k+1}) - f(\theta_k) \approx \tfrac{1}{2}\alpha_k^2 (\nabla f(\theta_k))^\intercal \nabla^2 f(\theta_k)(\nabla f(\theta_k)) - \alpha_k \|\nabla f(\theta_k)\|_2^2.$$

Assuming that the Hessian matrix is positive semi-definite, this approximation predicts that the gradient descent step will decrease the objective function if

$$\frac{2}{\alpha_k} > \frac{(\nabla f(\theta_k))^\intercal \nabla^2 f(\theta_k)(\nabla f(\theta_k))}{\|\nabla f(\theta_k)\|_2^2} = R(\nabla^2 f(\theta_k); \nabla f(\theta_k)). \tag{4.28}$$

The term on the right-hand side is known as the *Rayleigh quotient*. If the Hessian matrix is ill-conditioned, this Rayleigh quotient can take on a wide range of values. Since we choose the step size sequence α_k in advance and do not adjust it for (4.28), ill-conditioning can cause fluctuations and even prohibit further descent. This issue also affects stochastic gradient methods as they can be seen as approximations of gradient descent.

To adjust for the ill-conditioning of the Hessian matrix, one typically uses second-order methods. However, most of these methods require the computation or approximation of the Hessian matrix, making them impractical for the training of neural networks. Neural networks can have up to several million parameters, which already makes it hard to keep a $d \times d$ matrix like the Hessian in memory, let alone computing it. Moreover, it is challenging to apply a stochastic approach to second-order methods since many samples are required for a reasonable estimate of the Hessian matrix.

5 Numerical Experiments

Based on the theoretical results, we conduct numerical experiments to gain further insights. The code for the experiments is written in Python and accessible on GitHub (see github.com/nfleischmann/Bachelor-Thesis). For building and training the neural networks, the Keras API of TensorFlow [15] has been used.

5.1 Empirical Analysis of Local Minima

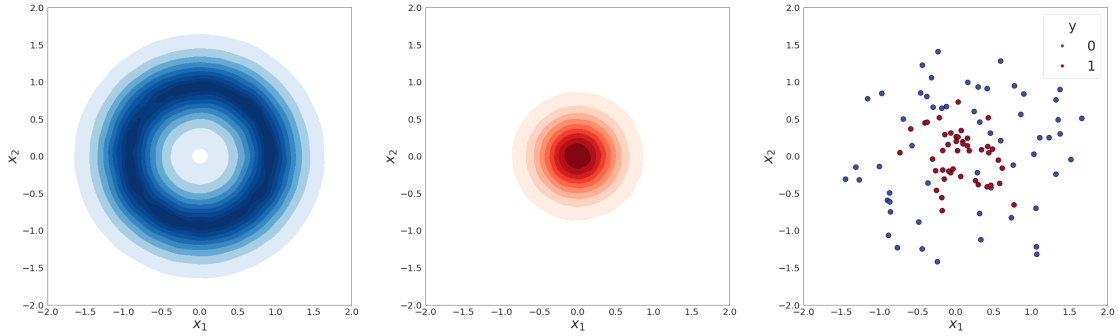
In Section 3.3, we discussed whether the objective function has local minima with high costs compared to global minima. The main result of this section was Theorem 3.6. For a class of neural networks with one hidden layer, Theorem 3.6 states that any differentiable local minimum of the objective function is a global minimum with zero cost if $n \leq d_0 d_1$. Motivated by this result, we will empirically investigate the relationship between the local minima and the number of hidden neurons in the following experiment. Soudry and Carmon [20] conducted a similar numerical experiment. However, we consider neural networks with a different activation function for the hidden layers and employ a distinct loss function. Further, we put more emphasis on neural networks with few hidden neurons.

Experimental Design

We used a small training set generated by the function make_circles of the library scikit learn throughout the experiment. It consists of 100 observations, which have a real input $x \in \mathbb{R}^2$ and a binary output $y \in \{0, 1\}$. Figure 5.1 depicts the training set and the underlying distribution of both classes.

In the experiment, we considered neural networks with two input neurons, $L - 1 \in \{1, 2, 3\}$ hidden layers, and one output neuron. Thereby all hidden layers contain the same number of neurons $d_i \in \{1,\ 2,\ 3,\ 4,\ 5,\ 10,\ 15,\ 20,\ 30,\ 40,\ 50,\ 75,\ 100\}$. For the hidden neurons, we used the leaky ReLU, whereas the logistic function was employed as the activation function for the output neuron. Further, we utilized the log loss as the loss function for this binary classification problem.

For each of these neural networks, we searched for local minima of the respective objective function and computed their cost. We made use of gradient descent to determine the local minima; considering Theorem 4.4, gradient descent should converge to a local



(a) density of x given $y = 0$ (b) density of x given $y = 1$ (c) training set

Figure 5.1: The training set generated by the function make_circles and the underlying distribution of the inputs for both classes.

minimum of the objective function if the step size is chosen appropriately. We let gradient descent make 40,000 steps with fixed step size and then adaptively lowered the step size (after 50 steps without descent, the step size was halved). The method was stopped after 10,000 steps in which the cost improved less than 1e-7 or at the latest after 120,000 steps.

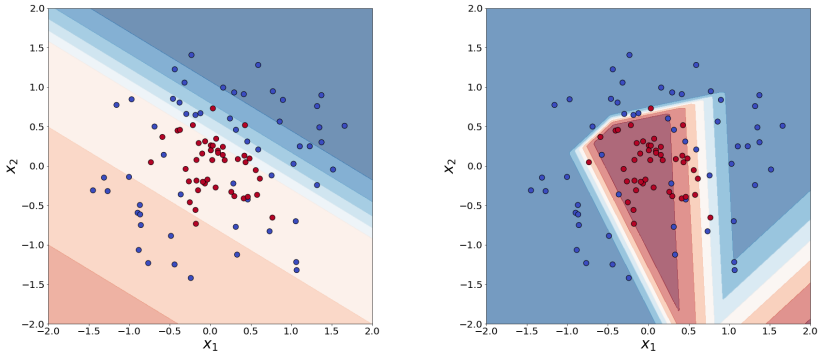
In order to find different local minima, we ran gradient descent several times. For each run, all parameters of the neural networks were initialized according to a continuous uniform distribution. This way, gradient descent was started 50 times for the neural networks with one hidden layer and 15 times for those with 2 or 3 hidden layers.

Evaluation of the Results

Let us begin by evaluating the results for the neural networks with one hidden layer. Figure 5.3 gives an overview of the results; the x-axis shows the number neurons in the hidden layer, and the y-axis shows the cost of the local minima. For each neural network, we summarized the costs of the found local minima with a boxplot. In addition, we plotted the costs of the individual local minima as dots.

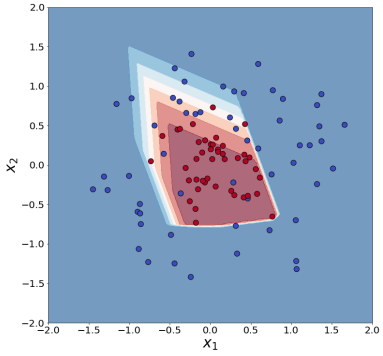
The most apparent trend, which can be observed in Figure 5.3 is that the cost of the local minima decreases for neural networks with more hidden neurons. This observation seems plausible; after all, neural networks with more hidden neurons also have more parameters and thus should be able to fit the training set better. In Figure 5.2, we have exemplarily illustrated the corresponding prediction functions of some of the found local minima. The prediction functions of neural networks with many hidden neurons seem to be more flexible, allowing them to adapt better to the training set.

As shown in Figure 5.3, the local minima of neural networks with few hidden neurons differ recognizably in terms of cost. This observation reflects the non-convexity of the objective function. However, these variations among the costs of the found local minima

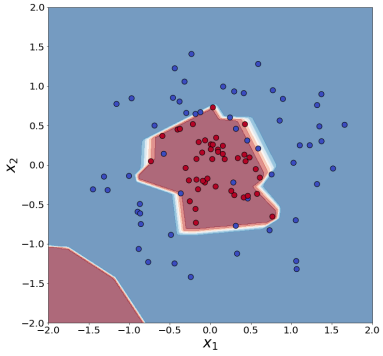


(a) 1 hidden neuron

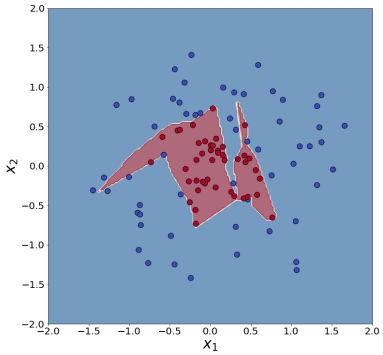
(b) 3 hidden neurons



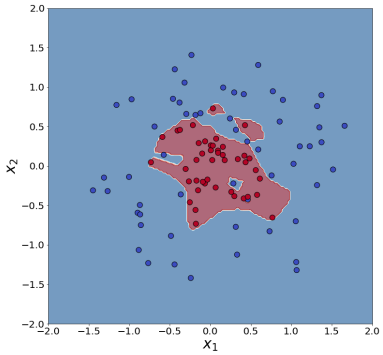
(c) 5 hidden neurons



(d) 10 hidden neurons



(e) 20 hidden neurons



(f) 50 hidden neurons

Figure 5.2: Predictions of neural networks with different numbers of hidden neurons plotted against the training set.

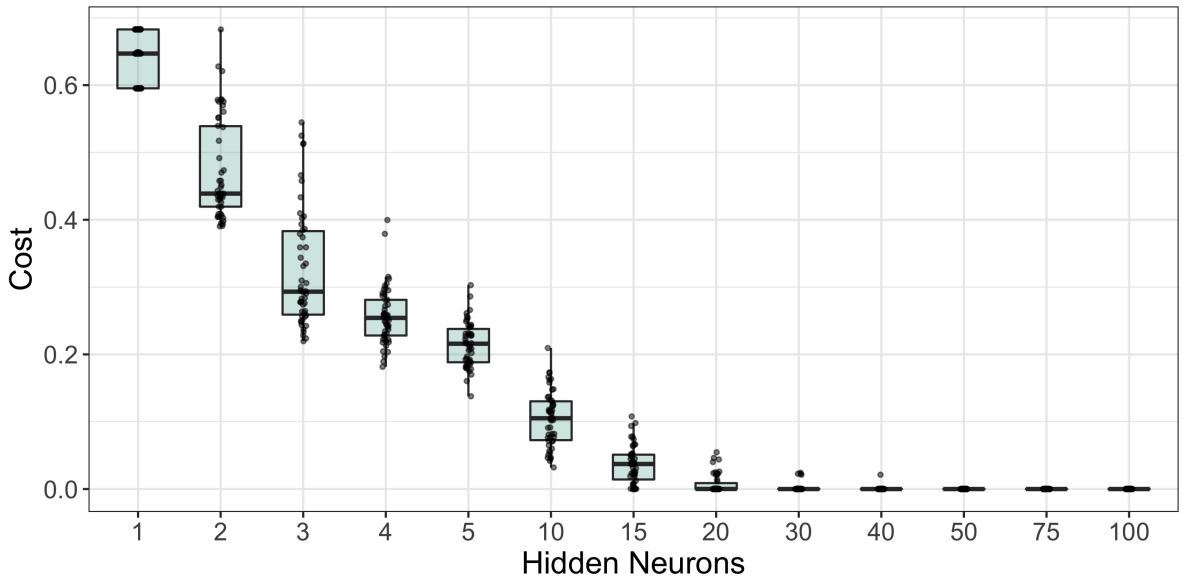


Figure 5.3: The cost of the found local minima for the neural networks with one hidden layer.

decrease the more neurons the hidden layer contains.

Remarkably, for neural networks with 50 or more hidden neurons, all found local minima have costs close to zero. Figure 5.4 depicts this in more detail, as it shows the median costs of the found local minima for each neural network with a logarithmized y-axis. As we can recognize, the cost of most local minima of neural networks with 50 or more hidden neurons remains below 1e-06. Considering that the cost is bounded from below by zero, these local minima must be close to global minima in terms of cost.

This observation goes well with Theorem 3.6. Since we have $d_0 = 2$ and $n = 100$, Theorem 3.6 would have predicted that every differentiable local minimum is a global minimum with zero cost for neural networks with $d_1 \geq 50$ hidden neurons. This is interesting because, in contrast to Theorem 3.6, we dealt with a binary classification problem instead of a regression problem in our experiment.

Let us now look at the results of the neural networks with two or three hidden layers, which are summarized in Figure 5.5. For these networks, we can observe similar trends as for the neural networks with one hidden layer. For instance, the cost of the found local minima and their variation decreases the more neurons the hidden layers contain. Figure 5.5 also shows that for neural networks with few neurons per layer, there are local minima whose costs differ significantly from the other found local minima. Furthermore, it can be seen that above a certain number of hidden neurons per layer, the cost of all found local minima are close to zero. Compared to neural networks with one hidden layer, it seems that this is already the case with fewer neurons per layer.

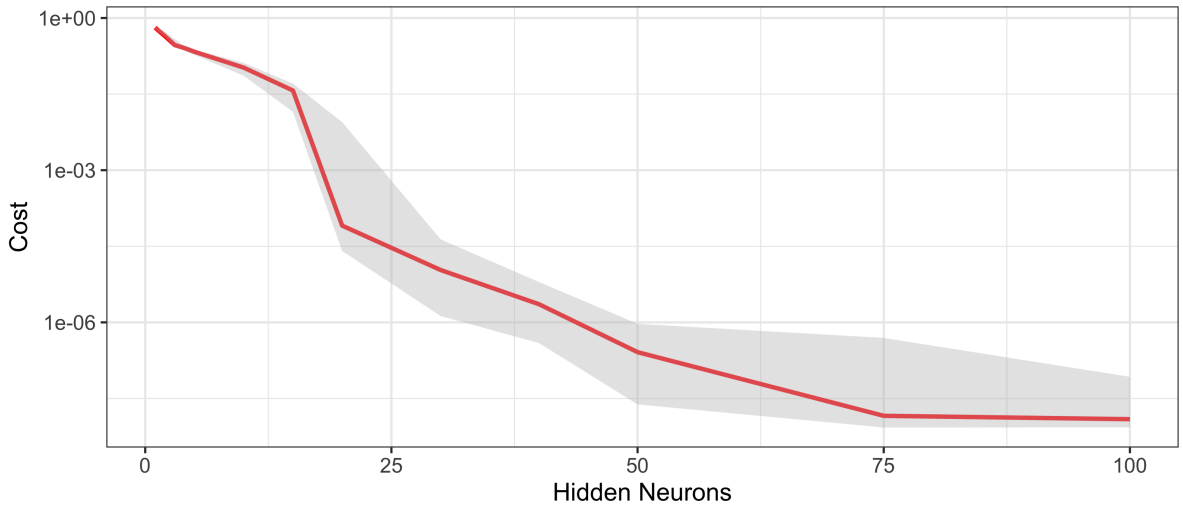


Figure 5.4: The median (red) and the first and third quartile (grey) of the cost of the found local minima for the neural networks with one hidden layer.

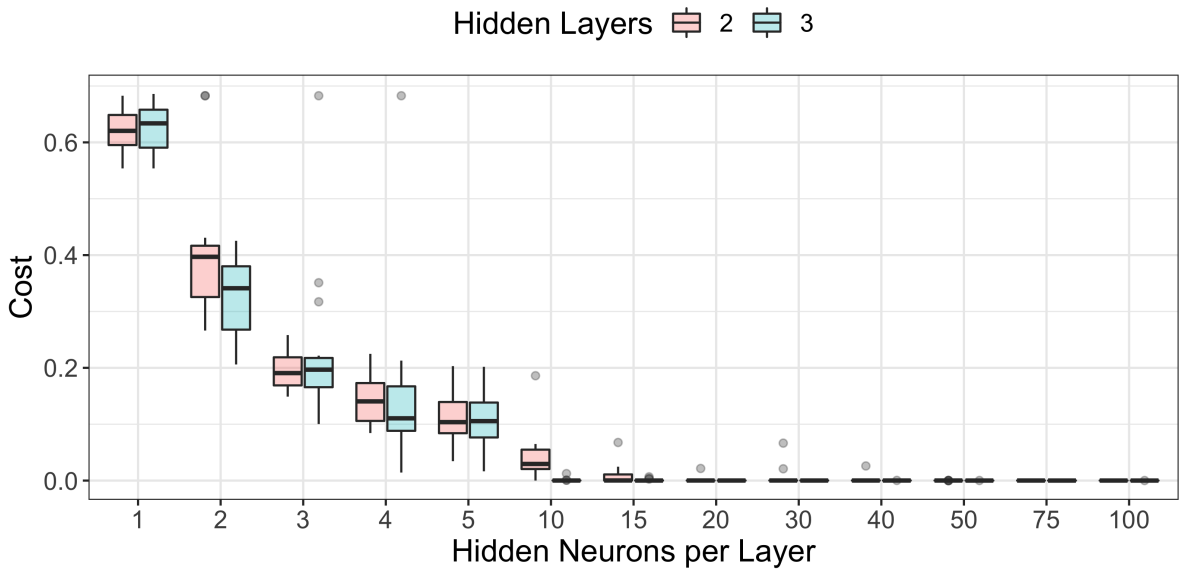


Figure 5.5: The cost of the found local minima for the neural networks with two or three hidden layers.

5.2 Efficiency of Gradient-Based Methods

Our primary motivation to introduce stochastic gradient methods was the fact that gradient descent is not well suited for large training sets. In particular, we argued that gradient descent processes the observations in the training set inefficiently, especially for large training sets. In the following experiment, we want to examine this issue by comparing gradient descent with the presented stochastic gradient methods for training sets of different sizes. Thereby, we let the methods process the same number of observations and investigate which method can make the most progress in terms of cost.

Experimental Design

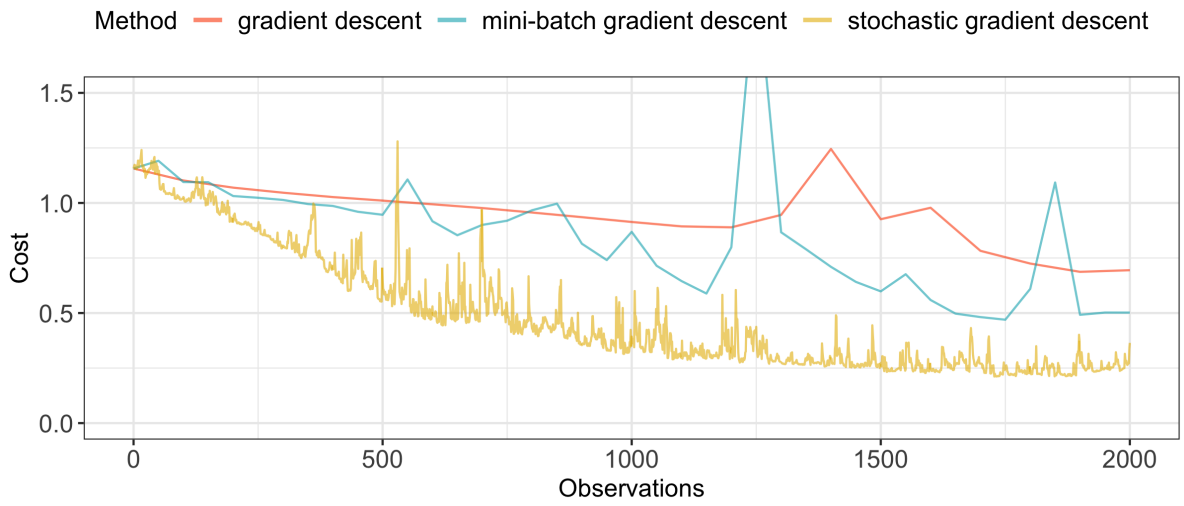
Since we want to observe the impact of the training set size, we have to ensure that the respective training sets are relatively similar. For this purpose, we generated training sets of the sizes 100, 1000, and 10,000 by randomly sampling observations from the well-known California housing dataset. Each observation consists of an input $x \in \mathbb{R}^8$ describing characteristics of a housing block and an output $y \in \mathbb{R}$, which is the median value of the houses in the housing block. We used a neural network with eight input neurons, five hidden layers with twelve neurons per layer, and a single output neuron throughout the experiment. As the activation function for the hidden layers, we employed the ReLU and used the identity as the activation function for the output neuron. Since we dealt with a regression problem, we made use of the square loss.

We compared gradient descent, stochastic gradient descent, and mini-batch gradient descent with a mini-batch size of 50. For each training set, we let these methods process 20 times as many observations as there are in the training set, i.e., let gradient descent make 20 steps. As mentioned, the step size plays a crucial role for gradient-based methods, so we tuned the step size for each method to make the comparison fair. To be more precise, we tried the step sizes $\alpha \in \{1, 0.5, 0.1, 0.05, 0.01, 0.005, 0.001, 0.0005, 0.0001, 0.00005\}$ for each training set and each method and selected the step size that led to the lowest cost. The results of the tuning are summarized in Table 5.1. Finally, we ran the methods with the tuned step sizes and computed the cost after each iteration.

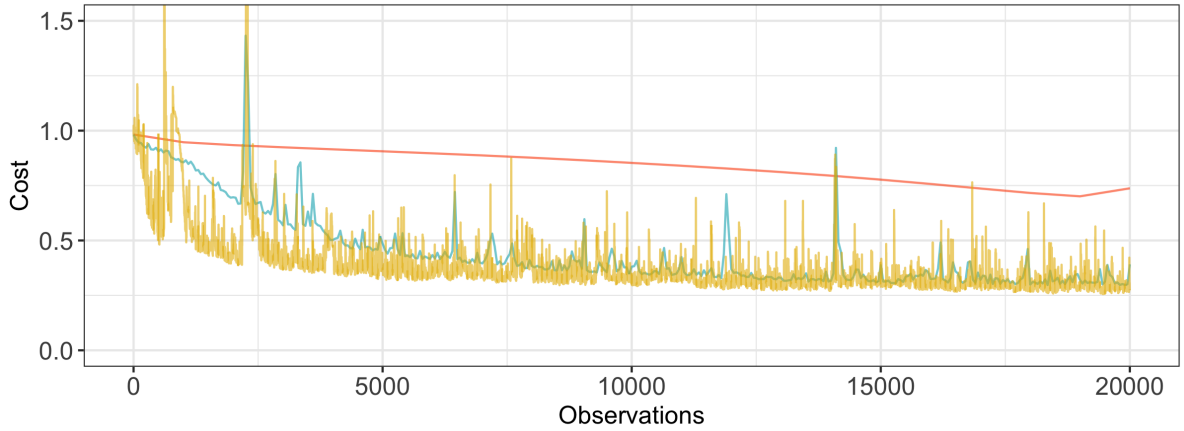
Evaluation of the Results

Figure 5.6 summarizes the outcomes of this experiment as it shows the cost of the iterates for the gradient-based methods. Thereby the results for each training set are plotted separately. The x-axis of the plots shows the number of used observations, and the y-axis the cost of the respective iterates.

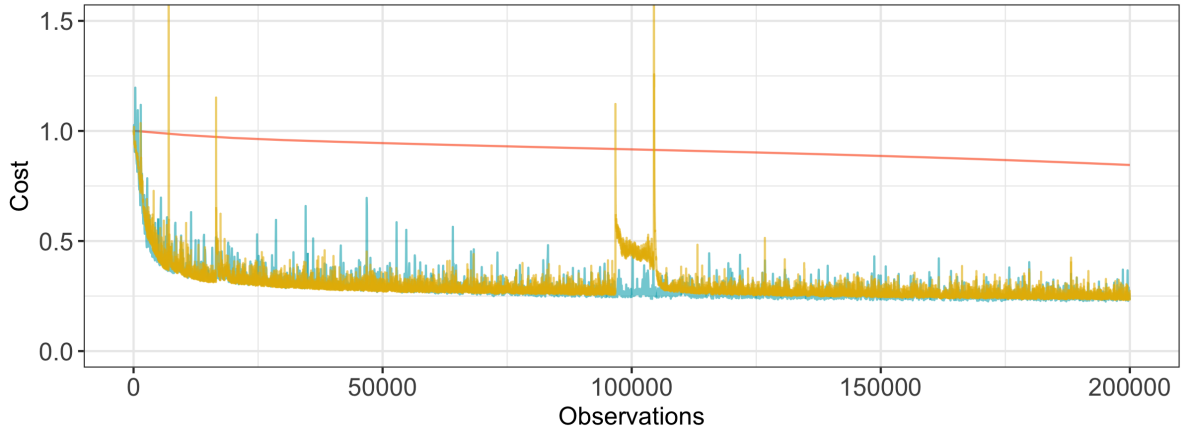
Since stochastic gradient descent requires only one observation per iteration, it can make significantly more steps with the same number of observations than the other methods, as we can observe in Figure 5.6. Further, it is apparent that the iterates of stochastic gradient descent exhibit fluctuations, as we would expect from a stochastic



(a) training set with 100 observations



(b) training set with 1000 observations



(c) training set with 10,000 observations

Figure 5.6: Costs of the iterates of gradient-based methods as a function of the number of observations used.

training set size	method	tuned step size
100	stochastic gradient descent	0.005
100	mini-batch gradient descent	0.1
100	gradient descent	0.1
1000	stochastic gradient descent	0.01
1000	mini-batch gradient descent	0.05
1000	gradient descent	0.1
10,000	stochastic gradient descent	0.001
10,000	mini-batch gradient descent	0.1
10,000	gradient descent	0.05

Table 5.1: Results of the step size tuning.

gradient method. Nonetheless, stochastic gradient descent has reduced costs the fastest across all training sets. Overall, it seems as if stochastic gradient descent processed the observations the most effectively, followed by mini-batch gradient descent.

The situation is different for gradient descent. Compared to the stochastic gradient methods, which seem to be relatively invariant of the training set size, gradient descent makes less and less progress with the same amount of observations as the training set size increases. This observation is consistent with our assumption that gradient descent processes observations inefficiently for large training sets and exemplifies why stochastic gradient methods are often preferred in such a setting.

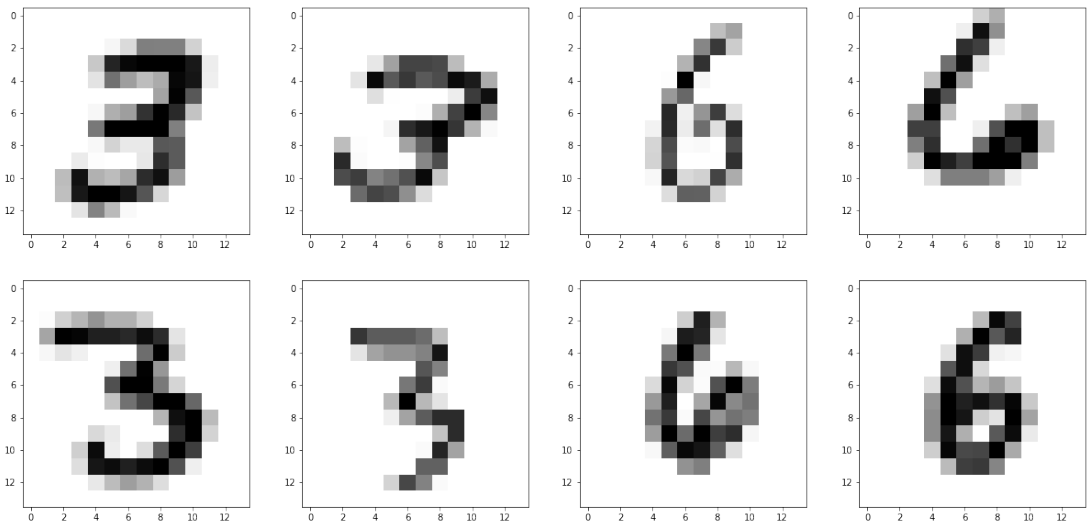
5.3 Asymptotic Properties of Stochastic Gradient Methods

Theorem 4.8 is a central result for stochastic gradient methods. It states that for a sufficiently small step size, the expected average squared norm of the gradients corresponding to the first K iterates of a stochastic gradient method can be bounded as follows:

$$\mathbb{E}\left[\frac{1}{K}\sum_{k=1}^{K}\|\nabla f(\theta_k)\|_2^2\right] \leq \alpha LM + \frac{2(f(\theta_1) - f_{inf})}{K\alpha} \tag{5.1}$$

$$\xrightarrow{K\to\infty} \alpha LM. \tag{5.2}$$

For large K, the upper bound is proportional to the step size α, and the noise of the stochastic gradients embodied by the term M. This relationship suggests that we can lower the expected average squared norm of the gradients by decreasing the step size



(a) input images of the class $y = 0$ (b) input images of the class $y = 1$

Figure 5.7: Sample observations from the modified MNIST training set.

and the noise term. In this experiment, we examine this relationship empirically for the training of neural networks.

Experimental Design

For the training set in this experiment, we used a downscaled and downsampled version of the famous MNIST dataset [12]. Each input $x \in \mathbb{R}^{14\times 14}$ is a 14×14 greyscale image that shows either the digit three or six. Respectively the output $y \in \{0, 1\}$ takes on the value 0 for the digit three and 1 for the digit six. Figure 5.7 exemplarily depicts input images of both classes. We used a neural network with 14×14 input neurons, three hidden layers with width ten, and a single output neuron for this binary classification problem. As the activation function for the hidden layers, we used the ReLU and employed the logistic function for the output neuron.

Since we are mainly interested in the asymptotic behavior, we want the second term in boundary (5.1) to vanish. One way to achieve this is by choosing an initial value θ_1 for which the difference $f(\theta_1) - f_{\text{inf}}$ is small. We determined such an initial value by applying gradient descent for 25,000 iterations. Another way is to choose a large K, i.e., let the stochastic gradient method make many steps. Therefore we have chosen $K = 30,000$.

Because we can not measure the expected value of the average squared norm of the gradients, we estimated it with the mean of 20 independent runs starting from the initial point we determined with gradient descent:

$$\mathbb{E}\left[\frac{1}{30000}\sum_{k=1}^{30000}\|\nabla f(\theta_k)\|_2^2\right] \approx \frac{1}{20}\sum_{i=1}^{20}\left[\frac{1}{30000}\sum_{k=1}^{30000}\|\nabla f(\theta_k^{(i)})\|_2^2\right]. \tag{5.3}$$

To observe the impact of the step size, we computed the right-hand term in (5.3) multiple times for stochastic gradient descent. Each time we used a different step size $\alpha \in \{0.01, 0.009, 0.008, 0.007, 0.006, 0.005, 0.004, 0.003, 0.002, 0.001\}$. In contrast to the step size, we can not adjust the noise term M directly. Nevertheless, as we argued in Section 4.4.2, mini-batch gradient descent with mini-batch size $|B|$ reduces M by a factor of $\frac{1}{|B|}$ compared to stochastic gradient descent. Therefore we tried to measure the impact of the noise term M by computing the right-hand term in (5.3) for mini-batch gradient descent with different mini-batch sizes. To be more precise, we fixed the step size to $\alpha = 0.001$ and used the mini-batch sizes 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10.

Evaluation of the Results

Let us begin by analyzing the influence of the different step sizes on the average squared norm of the gradients. Figure 5.8 summarizes the results of the experiment. The x-axis shows the employed step size, and the y-axis the average squared norm of the gradients corresponding to the first 30,000 iterates of stochastic gradient descent. Each run of stochastic gradient descent is represented as a black dot, whereas the respective mean for each step size is plotted as a red square. To highlight the general trend, we connected the means with a red line.

As can be seen from this figure, there are significant differences among the runs with the same step size. This reflects the nature of stochastic gradient descent and exemplifies why we have always considered the expected value in the theoretical results of stochastic gradient methods. By looking at the mean values, it can be observed that the average squared norm of the gradients decreases for smaller step sizes, as we assumed. Furthermore, the mean values almost lie on a straight line.

Now let us analyze how the noise term M impacts the average squared norm of the gradients. Recall that we adjusted the noise term indirectly through the mini-batch size, as M is proportional to $\frac{1}{|B|}$. Figure 5.9 summarizes the respective results. The x-axis shows the inverse of the employed mini-batch size, and the y-axis shows the average squared norm of the gradients corresponding to the first 30,000 iterates of mini-batch gradient descent. In order to display the results more concisely, we scaled the x-axis and the y-axis logarithmically. Each run of mini-batch gradient descent is represented as a black dot, while the mean for each mini-batch size is plotted as a red square. Again, we connected the means with a red line.

From Figure 5.9, we can note similar tendencies as for the step sizes. There are noticeable differences among the individual runs, and a clear trend for the mean values can be observed. For smaller values of $\frac{1}{|B|}$ and thus less noise M, the average squared

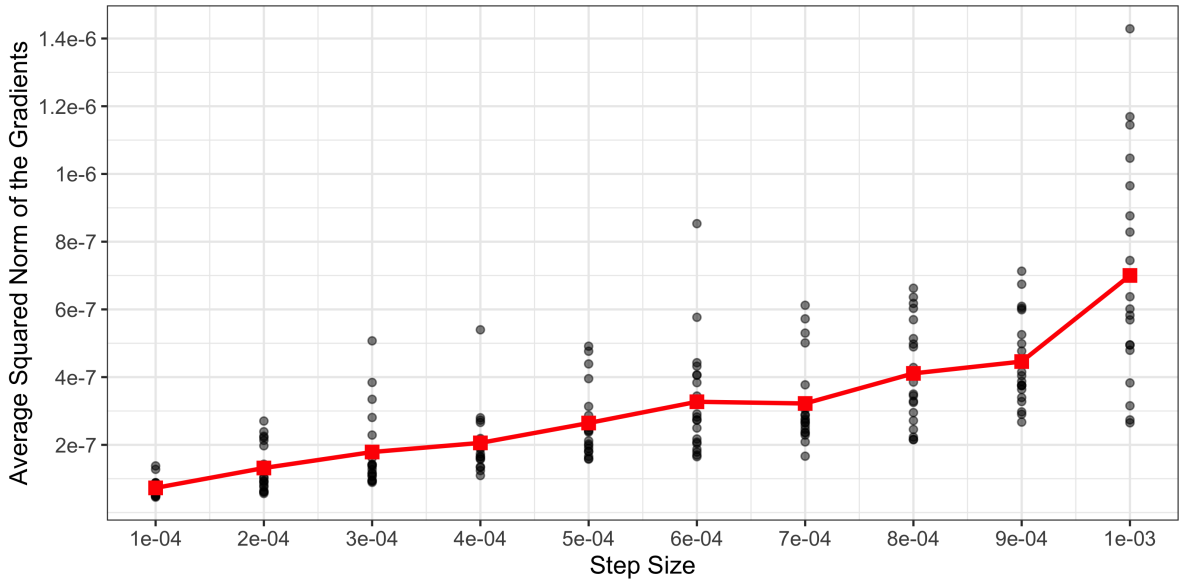


Figure 5.8: The average squared norm of the gradients corresponding to the first 30,000 iterates of stochastic gradient descent as a function of the step size.

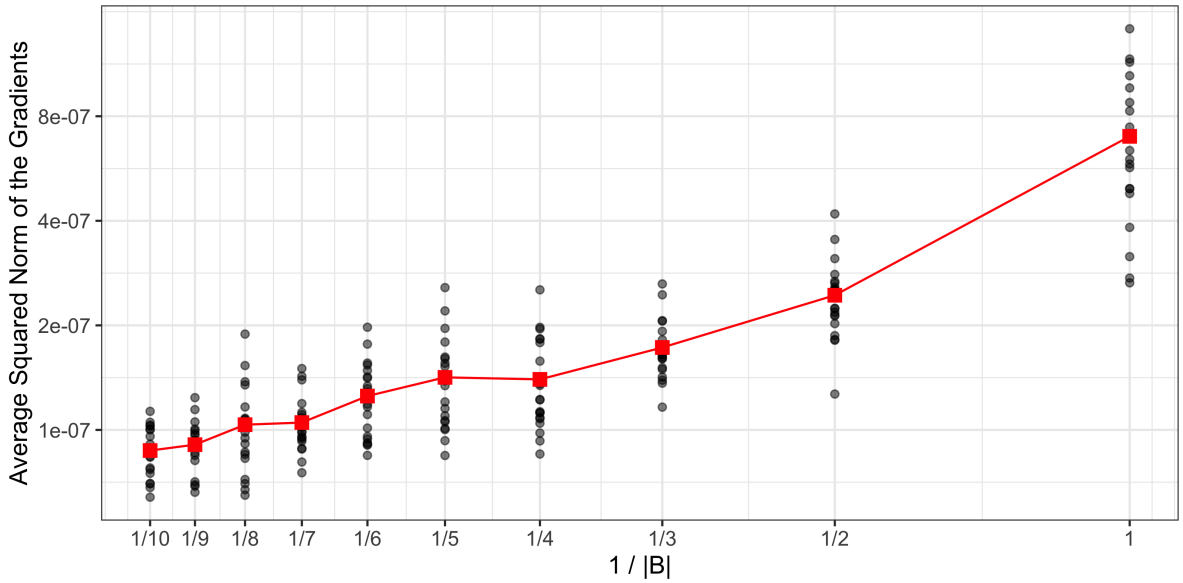


Figure 5.9: The average squared norm of the gradients corresponding to the first 30,000 iterates of mini-batch gradient descent as a function of the inverse mini-batch size.

gradient norm decreases. The mean values lie approximately on a line, indicating that the mean of the average squared gradient norm is almost proportional to the noise term M.

We want to stress that we only considered a single example; the results could be different for other problems, network architectures, or other initial points. Nevertheless, for this example, we observed that the mean of the average squared norm of the gradients, which we used to estimate the expected value in (5.1), is approximately proportional to the step size and the noise term M. This observation is indicative of the goodness of the asymptotic bound (5.2), which is also proportional to these two quantities.

6 Conclusion

6.1 Summary

From a mathematical perspective, a neural network is a parameterized prediction function. In order to make accurate predictions for a given task, the parameters of the neural network must be chosen appropriately. This is done during the training phase, which involves solving an optimization problem that follows the principle of empirical risk minimization. Thereby, the optimization problem is constructed around the training set and a suitable loss function.

However, the objective function of this optimization problem is non-convex and thus could exhibit local minima with high costs compared to global minima. Since gradient-based methods are attracted to local minima, their application for this optimization problem seems questionable at first glance. Nevertheless, theoretical results for specific neural networks show that such bad local minima do not necessarily occur. The findings of the numerical experiment further indicate that the local minima become more similar in terms of cost as the number of hidden neurons in the neural network increases. Thus, if the neural network has a decent size, it usually suffices to search for a local minimum of the objective function.

Before gradient-based methods can be applied, a way to compute the respective gradients is required. For this purpose, the backpropagation algorithm is employed, which is essentially a repeated application of the chain rule. From a theoretical point of view, gradient descent is a suitable method for training neural networks because it commonly converges against a local minimum of the objective function. However, the effort for computing the gradient of the objective function increases with the training set size making gradient descent infeasible for large training sets. Furthermore, gradient descent processes the observations in the training set inefficiently, as exemplified by the second numerical experiment.

In this regard, stochastic gradient methods represent a promising alternative as they work with cheap gradient approximations independent of the training set size. On the downside, the iterations of stochastic gradient methods are subject to noise. This noise prevents stochastic gradient methods with fixed step sizes from converging against stationary points of the objective function. In order to deal with the noise, one can lower the step size or employ a larger mini-batch, as the third experiment illustrated. Furthermore, convergence to stationary points can be achieved by using a diminishing step size sequence. Despite their flaws, stochastic gradient methods are commonly used

for the training of neural networks, partly because they lack viable alternatives.

6.2 Future Work

In this section, we present several topics that exceed the scope of this bachelor thesis and could be the subject of future work.

6.2.1 Different Architectures

In this work, we confined ourselves to fully connected feed-forward neural networks, as they are relatively well understood from a mathematical standpoint. While such architectures are sufficient for many problems, highly complex tasks usually demand alternative approaches. For example, so-called *convolutional neural networks* are frequently used for computer vision tasks. The building blocks of convolutional neural networks are *convolutional layers* in which each neuron receives only a fraction of the outputs of the previous layer. Furthermore, such neural networks often use *pooling layers*, which aggregate the outputs of the previous layer.

It would be interesting to investigate how such layers affect the properties of the objective function and, particularly, its local minima. For instance, extending the numerical experiment in Section 5.1 to other types of neural networks could be insightful.

6.2.2 Other Stochastic Gradient Methods

Based on stochastic and mini-batch gradient descent, several variants have been developed. These variants usually attempt to improve the classical versions by incorporating information from previous iterations into the update rule. For example, stochastic gradient methods with *momentum* remember the previous update vector and add a fraction of it to the current update vector. By the addition of momentum, one hopes to accelerate the stochastic gradient methods in relevant directions while also dampening oscillations.

Furthermore, so-called *adaptive methods* emerged in the last decade. Based on the previous iterates, these methods rescale the components of the update vector. Typically, components associated with large and frequent updates are scaled down, whereas components with smaller and less frequent updates are scaled up. By doing so, adaptive methods try to enforce equal progress along each axis.

These variants empirically outperform the classical stochastic gradient methods for many problems and are therefore frequently used in practice. However, their success for non-convex problems is still poorly understood. Therefore it would be interesting to examine their behavior in further detail.

Bibliography

[1] M. Anthony and P. L. Bartlett. *Neural Network Learning: Theoretical Foundations*. Cambridge University Press, 1999. DOI: 10.1017/CBO9780511624216.

[2] P. Baldi and K. Hornik. Neural networks and principal component analysis: learning from examples without local minima. *Neural Networks*, 2(1):53–58, 1989. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(89)90014-2.

[3] C. M. Bishop. *Pattern recognition and machine learning*. Information science and statistics. Springer, New York, 2006. ISBN: 978-0-387-31073-2.

[4] L. Bottou, F. E. Curtis, and J. Nocedal. Optimization Methods for Large-Scale Machine Learning. *SIAM Review*, 60(2):223–311, Jan. 2018. ISSN: 0036-1445, 1095-7200. DOI: 10.1137/16M1080173. (Visited on 07/20/2021).

[5] H. Bruhn-Fujimoto. Introduction to ML; Lecture Notes. Ulm University. 2020.

[6] Y. N. Dauphin, R. Pascanu, C. Gulcehre, K. Cho, S. Ganguli, and Y. Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, pages 2933–2941, Montreal, Canada. MIT Press, 2014.

[7] M. P. Deisenroth, A. A. Faisal, and C. S. Ong. *Mathematics for machine learning*. Cambridge University Press, Cambridge ; New York, NY, 2020. ISBN: 978-1-108-67993-0.

[8] R. Ge, F. Huang, C. Jin, and Y. Yuan. Escaping from saddle points - online stochastic gradient for tensor decomposition. *CoRR*, abs/1503.02101, 2015. arXiv: 1503.02101. URL: http://arxiv.org/abs/1503.02101.

[9] A. Geiger. Deep Learning; Lecture Notes. University of Tübingen. 2021. URL: https://drive.google.com/file/d/1JCWTApGieKZmFW4RjCANZM8J6igcsdYg/view.

[10] A. Géron. *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, first edition, 2017. ISBN: 978-1-4919-6229-9.

[11] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[12] Y. LeCun, C. Cortes, and C. Burges. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010.

[13] J. Lee, M. Simchowitz, M. I. Jordan, and B. Recht. Gradient descent converges to minimizers. *ArXiv*, abs/1602.04915, 2016.

[14] U. Luxburg and B. Schölkopf. Statistical learning theory: models, concepts, and results. *Handbook of the History of Logic*, 10, Nov. 2008. DOI: 10.1016/B978-0-444-52936-7.50016-1.

[15] Martın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: large-scale machine learning on heterogeneous systems, 2015. URL: https://www.tensorflow.org/. Software available from tensorflow.org.

[16] K. P. Murphy. *Probabilistic Machine Learning: An introduction*. MIT Press, 2022. URL: probml.ai.

[17] M. A. Nielsen. *Neural networks and deep learning*, volume 25. Determination press San Francisco, CA, 2015.

[18] B. T. Poliak. *Introduction to optimization*. Translations series in mathematics and engineering. Optimization Software, Publications Division, New York, 1987. ISBN: 978-0-911575-14-9.

[19] S. Ruder. An overview of gradient descent optimization algorithms. *ArXiv*, abs/1609.04747, 2016.

[20] D. Soudry and Y. Carmon. No bad local minima: data independent training error guarantees for multilayer neural networks. *ArXiv*, abs/1605.08361, 2016.

[21] M. Ulbrich and S. Ulbrich. *Nonlinear optimization. (Nichtlineare Optimierung.)* German. Basel: Birkhäuser, 2012. ISBN: 978-3-0346-0142-9/pbk.

[22] K. Urban. Numerical Methods for Data Science; Lecture Notes. Ulm University. 2021.

Name: Nils Daniel Fleischmann Matrikelnummer: 981781

Erklärung
Ich erkläre, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den 15.10.2021 Nils Fleischmann

Nils Daniel Fleischmann