Master's thesis

November 22, 2023

On variational autoencoders: theory and applications

Maksym Sevkovych
Registration number: 3330007
In collaboration with: Duckeneers GmbH

Inspector: Univ.-Prof. Dr. Ingo Steinwart

Recently in the realm of machine learning, the power of generative models has revolutionized the way we perceive data representation and creation. This thesis focuses on the captivating domain of Variational Autoencoders (VAEs), a cutting-edge class of machine learning models that seamlessly combine unsupervised learning and data generation. In the course of this thesis we embark on an expedition through the intricate architecture and mathematical elegance that underlie VAEs.

By dissecting the architecture of VAEs, we show their role as both proficient data compressors and imaginative creators. As we navigate the landscapes of latent spaces and probabilistic encodings, we uncover the essential mechanisms driving their flexibility. Applications of VAEs extend from anomaly detection to image generation. However, we will focus on the latter.

Contents

Chapter 1

Preliminary

In order to comprehend the topic of variational autoencoders or even autoencoders in general, we need to consider a couple of preliminary ideas. Those ideas consist mainly of concepts of probability theory and statistics, statistical learning theory, neural networks and their optimization - often referred to as training of neural networks. In this chapter, we will begin with fundamental definitions we need for the statistical learning setting and afterwards tackle the conceptional idea of how to formulate neural networks in a mathematical way and furthermore, we will consider a handful of useful operations that neural networks are capable of doing. Then, we will take a look at some strategies of training neural networks. Lastly, we will consider neural networks operating on images. This discipline of machine learning is usually referred to as computer vision.

1.1 Probability and Statistics

A fundamental topic for this thesis is measure and probability theory and statistics. Especially in Chapter 3 this will be crucial to analyse variational autoencoding neural networks in more detail.
We want to begin by introducing the basic quantities in measure and probability theory, merely to define a notation throughout the thesis. The first quantity we want to introduce is the density function, where we strongly orient ourselves on [12, Chapter 2, Chapter 3]. Furthermore, we want to use the notation the authors used in the same reference. Hence, let $(\Omega, \mathcal{A}, \mu)$ be a measure space. Then we denote M as the set of all measurable numeric functions $f : \Omega \to \bar{\mathbb{R}}$ and moreover, M^+ denotes the set of all non-negative functions in M. With these definitions we can introduce a density function with regard to a measure.

Definition 1.1.1. Let $(\Omega, \mathcal{A}, \mu)$ be a measure space and $f \in M^+$. Then we define

$$f \odot \mu : \mathcal{A} \to \bar{\mathbb{R}},$$
$$(f \odot \mu)(A) \coloneqq \int_A f d\,\mu,$$

as the **measure with density** f **(with regard to** μ**)**. The function $f : \Omega \to \bar{R}_+$ is called **density** of the measure $f \odot \mu$.

A very important question we need to clarify at this point, is when such a density function as we defined in Definition 1.1.1 actually exists. In order to address this, we need to introduce

a property which in a sense compares two measures on their null sets. This property is called absolute continuity, which we introduce in the following.

Definition 1.1.2. Let μ, ν be two measures on $(\Omega, \mathcal{A})$. We call ν **absolute continuous with regard to** μ, if

$$\mu(A) \quad \Longrightarrow \quad \nu(A),$$

holds for all $A \in \mathcal{A}$. We will denote this as $\nu \ll \mu$.

With the Definition 1.1.2 we can introduce a theorem from the measure theory, which is well-known as the Radon-Nikodym Theorem. Since this result is fundamental, we omit the proof at this point and instead kindly refer to e.g. [12, Theorem 2.38].

Theorem 1.1.3. *Let $(\Omega, \mathcal{A}, \mu)$ be a measure space with a σ-finite measure μ. Furthermore, let ν be another measure on $(\Omega, \mathcal{A})$. Then the following assertions are equivalent:*

(i) $\nu \ll \mu$.

(ii) There exists a density $f \in M^+$, such that $\nu = f \odot \mu$.

The next consideration we want to do, is to remember that probability measures are special cases of ordinary measures. These we want to introduce formally now.

Definition 1.1.4. Let $(\Omega, \mathcal{A}, \mathrm{P})$ be a measure space. If $\mathrm{P}(\Omega) = 1$ holds, then we call P a **probability measure** and $(\Omega, \mathcal{A}, \mathrm{P})$ a **probability space**.

With the help of probability measures, which we introduced in Definition 1.1.4, we can introduce the probability distribution. We will introduce it directly for a probability measure on $\mathbb{R}$. We will denote $\mathcal{B}$ as the Borelian σ-algebra on $\mathbb{R}$, see e.g. [12, Definition 1.13].

Definition 1.1.5. Let $(\mathbb{R}, \mathcal{B}, \mathrm{P})$ be a probability space. Then we call the function defined as

$$\begin{aligned} F_{\mathrm{P}} : \mathbb{R} &\to [0,1], \\ x &\mapsto F_{\mathrm{P}}(x) \coloneqq \mathrm{P}\left((-\infty, x)\right), \end{aligned}$$

the **distribution function of P**. We will write shortly **distribution of P**

If we consider the properties of the distribution of a probability measure, we realise that it is a monotonously increasing function. Moreover, we realise that it is continuous on its right side. Lastly, the distribution asymptotically converges to 0 for $x \to -\infty$ and to 1 for $x \to \infty$. These properties allow us to introduce distributions as functions, which fulfil exactly these, which we will see in the following definition.

Definition 1.1.6. Let $F : \mathbb{R} \to \mathbb{R}$ be a monotonously increasing function that is continuous on its right side. If for the function F holds

$$\lim_{x \to -\infty} F(x) = 0 \quad \text{and} \quad \lim_{x \to \infty} F(x) = 1,$$

then we call F a **distribution function**.

The next quantity we need to introduce are random variables. Conceptionally, random variables are functions, which filter the essential information from a random experiment (e.g. rolling dices or drawing cards) and allow us to formulate this condensed problem mathematically correct. Lastly, we can introduce the distribution of such a random variable as the image measure of the probability measure with regard to the random variable, see e.g. [12, Definition 1.42] for more details.

Definition 1.1.7. Let $(\Omega_1, \mathcal{A}_1, \mathrm{P})$ be a probability space and $(\Omega_2, \mathcal{A}_2)$ be a measure space. A function $X : \Omega_1 \to \Omega_2$ that is $\mathcal{A}_1 - \mathcal{A}_2$-measurable is defined as a **random variable**.
Moreover, if we consider $\Omega_2 = \mathbb{R}^n$ with $n = 1, 2, \ldots$, we refer to X as an **n-dimensional real-valued random variable**. If $\Omega_2 = \bar{R}$, we refer to X as a **numerical random variable**.
Lastly, we call the image measure P_X the distribution of X.

Lastly, we need to consider how the distribution of a random variable looks like, if the random variable is not one dimensional. This we want to consider in the following definition.

Definition 1.1.8. Let $(\mathbb{R}^n, \mathcal{B}^n, \mathrm{P})$ be a probability space. Furthermore, let $X = (X_1, \ldots, X_n)$ be a n-dimensional real-valued random variable. Then we call the function

$$\begin{aligned} F : \mathbb{R}^n &\to [0, 1], \\ x = (x_1, \ldots, x_n) &\mapsto \mathrm{P}(X \leq x) \coloneqq \mathrm{P}(X_1, \leq x_1, \ldots X_n \leq x_n), \end{aligned}$$

the **distribution of** X.

We introduced random variables to add a layer of abstraction to a given problem. Therefore, random variables model some kind of process, high-level speaking. One may now pose the question, what happens if we consider more than one random variable at the same time. For example, if we roll two dice we can model each of them as an own random variable. However, we want to capture this entire problem. In order to do so, we can consider the probability distributions of both random variables jointly. Therefore, we introduce the quantity known as the joint probability distribution in the following.

Definition 1.1.9. Let $(\mathbb{R}^n, \mathcal{B}^n, \mathrm{P})$ be a probability space. Furthermore, let $X = (X_1, \ldots, X_n)$ and $Y = (Y_1, \ldots, Y_n)$ be n-dimensional real-valued random variables with distributions P_X and P_Y, respectively. Then we call the function

$$\begin{aligned} F : \mathbb{R}^n \times \mathbb{R}^n &\to [0, 1], \\ (x, y) &\mapsto \mathrm{P}(X \leq x, Y \leq y), \end{aligned}$$

the **joint probability distribution** of $Z \coloneqq (X, Y)$.
Moreover, we note that $\mathrm{P}_Z = \mathrm{P}_X \times \mathrm{P}_Y$ holds.

As well as considering multiple random variables at once, one might do exactly the opposite. What if one does already posses some part of information about the outcome of a probability experiment, e.g. if one rolls two dice but does already know the outcome of one, for whatever reason. This we can consider by marginalizing over one of both random variables, what gives us the marginal distribution. However, in order to formulate this idea of possessing some part of the information, we first need to introduce conditional probabilities. Since if there is partial information on the outcome of a random experiment, the probabilities for the possible events may change. Thus, we introduce this quantity analogous to [8, Chapter 8].

Definition 1.1.10. Let $(\Omega, \mathcal{A}, \mathrm{P})$ be a probability space and $A \in \mathcal{A}$. For any $B \in \mathcal{A}$ we define the **conditional probability of B given A** by

$$\mathrm{P}(B|A) = \begin{cases} \frac{P(A \cap B)}{P(A)}, & \text{if } \mathrm{P}(A) > 0, \\ 0, & \text{else.} \end{cases}$$

At this point we should note, that the conditional probability, introduced in Definition 1.1.10 is indeed a probability measure, see [8, Theorem 8.4]. Lastly, we want to introduce the famous Bayes' formula, see e.g [8, Theorem 8.7]. This will be fundamental for our Bayesian learning setting in Chapter 3.

Theorem 1.1.11. *Let $(\Omega, \mathcal{A}, P)$ be a probability space and I be a countable set. Furthermore, let $(B_i)_{i \in I}$ be a sequence of pairwise disjoint sets with $P(\bigcup_{i \in I} B_i) = 1$.*
Then for any $A \in \mathcal{A}$ with $P(A) > 0$ and any $k \in I$ holds

$$P(B_k|A) = \frac{P(A|B_k)\, P(B_k)}{\sum_{i \in I} P(A|B_i)\, P(B_i)}.$$

Having introduced the fundamental concept of conditional probabilities, we can now return to what we were actually considering and introduce marginal distributions in the following.

Definition 1.1.12. Let X, Y be n-dimensional real-valued random variables with distributions P_X and P_Y, respectively. Furthermore, we define the random variable $Z = (X, Y)$ with probability distribution P_Z. Then we define the **marginal distribution of X given Y** as

$$\mathrm{P}_X(x) \coloneqq \int_y \mathrm{P}_{X|Y}(x|y)\, \mathrm{P}_Y(y) dy = \mathbb{E}\left(\mathrm{P}_{X|Y}(x|y)\right).$$

Since we will be interested in modelling probability distributions to approximate observed data as well as possible, we now want to consider how to construct a family of distributions by tweaking the parameters of the probability distribution. In particular, to generate a family of distributions we alter an underlying base probability density function, hence named standard probability density function. Possible alterations might be shifting or scaling (or both) the standard probability density function. Therefore, we cite a theorem from [1, Theorem 2.1], which proposes exactly such a construction.

Theorem 1.1.13. *Let $p(x)$ be a probability density function and $\mu \in \mathbb{R}$ and $\sigma > 0$ constants. Then the following functions are also probability density functions*

$$g(x; \mu, \sigma) = \frac{1}{\sigma} p\left(\frac{x - \mu}{\sigma}\right).$$

*We refer to the parameters μ as **location parameter** and σ as **scale parameter**. Moreover, we call the family $\mathcal{P}_{\mu,\sigma} = \{g(x; \mu, \sigma) : \mu \text{ and } \sigma > 0\}$ a **location-scale family**.*

Example 1.1.14. The following examples are all location-scale families.

1. Let $\alpha, \beta > 0$ be constants. Then the Gamma distribution $\mathrm{Ga}(\alpha, \beta)$ is a scale family for each value of the shape parameter α

$$p(x; \alpha, \beta) = \frac{\beta^\alpha}{\Gamma(\alpha)} x^{\alpha - 1} e^{-\beta x}.$$

2. Let $\mu \in \mathbb{R}$ and $\sigma > 0$ be constants. Then the Gaussian distribution $\mathcal{N}(\mu, \sigma^2)$ is a location-scale family for both, the location parameter μ and the scale parameter σ, respectively. This leads to

$$p(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\,\sigma^2}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}.$$

Having introduced the basics of probability theory, we now want to consider the actual statistical learning setting that we find ourselves in. We introduce this setting inspired by [19, Chapter 2]. The fundamental idea is that we have some kind of data that represents reality. Our goal is to find a function that approximates the given data as accurately as possible. There are many ways to do so, but we will focus on one certain class of functions. These functions are called neural networks, which we will introduce is Section 1.2. However, before considering neural networks we have to introduce some quantities that we will need later on, first. These quantities are well-known in statistical learning theory and are called loss and risk function. The loss function is a function that measures the point-wise discrepancy between the prediction of a found approximative function to the actually true value. In contrast, the risk function is a function that measures the error with regard to a probability measure or an observed data set.

Definition 1.1.15. Let X, Y be arbitrary input and output spaces. Furthermore, let $t \in \mathbb{R}^n$ be a prediction for $y \in Y$.
Then we define a **supervised loss function** $\mathcal{L} : X \times Y \times \mathbb{R}^n \to [0, \infty)$ as a measurable function that compares a true value $y \in Y \subset \mathbb{R}^n$ to a predicted value $\widehat{y} = t \in \mathbb{R}^n$ at some $x \in X$ in a suitable way.

The recently introduced loss function from Definition 1.1.15 allows us to compare a true label to a prediction. However, since we only require it to be measurable, the function is defined very general. This is solely, because one may be interested in different loss functions in different settings, since the choice of a specific loss functions is a crucial aspect of learning theory, as it directly impacts the behaviour of learning algorithms.
We want to consider a couple of important examples for loss functions, for further reading we kindly refer to [3, Chapter 4.3].

Example 1.1.16. Let $y \in Y \coloneqq \mathbb{R}$ and $t \in \mathbb{R}$, then the following expressions define loss functions.

Squared Error Loss: $\mathcal{L}(y, t) = |y - t|^2$,

Linear Error Loss: $\mathcal{L}(y, t) = |y - t|$.

Now, let $y \in Y \coloneqq \mathbb{R}^n$ and $t \in \mathbb{R}^n$ with $n > 1$, where we denote $y = (y_1, \ldots, y_n)$ and $t = (t_1, \ldots, t_n)$. Then, we consider the squared error loss and the linear error loss pointwise:

Squared Error Loss: $\mathcal{L}(y, t) = \sum_{i=1}^n |y_i - t_i|^2$,

Linear Error Loss: $\mathcal{L}(y, t) = \sum_{i=1}^n |y_i - t_i|$.

However, loss functions only quantify the disparity between one predicted outcome and one actual value. In order to compare the predictions over a wider range of data points, we need to introduce another quantity, the risk function.

Definition 1.1.17. Let X and Y be input and output spaces, $\mathcal{L}$ be a supervised loss function and P be a probability measure on $X \times Y$. Then we define the **risk of the function** f with regard to a loss function $\mathcal{L}$ as

$$\mathcal{R}_{\mathcal{L},\mathrm{P}}(f) \coloneqq \int_{X\times Y} \mathcal{L}\left(x, y, f(x)\right) d\,\mathrm{P}(x, y).$$

In the following, we will denote this as the **$\mathcal{L}$-risk of** f.

Considering applications, one usually wants to compute the risk with regard to observed data instead of a probability measure, since the true probability measure usually is unknown. In this case, the general risk function becomes more tangible, as we see in the following definition.

Definition 1.1.18. Let X, Y be arbitrary input and output spaces, $D = ((x_1, y_1), \ldots, (x_k, y_k))$ be a dataset consisting of $k \in \mathbb{N}$ data points. Furthermore, let $\mathcal{L}$ be a supervised loss function and f be an arbitrary prediction function. Then we define the **empirical $\mathcal{L}$-risk of** f as

$$\mathcal{R}_{\mathcal{L},D}(f) \coloneqq \frac{1}{k}\sum_{i=1}^{k} \mathcal{L}\left(x_i, y_i, f(x_i)\right).$$

We will write $\mathcal{R} \coloneqq \mathcal{R}_{\mathcal{L},D}$, when clear in the given context.

Before continuing to the theory, we want to introduce a short auxiliary lemma that considers the convexity of loss and risk functions

Lemma 1.1.19. *Let $\mathcal{L}$ be a convex loss function and D be an arbitrary dataset of length $k \in \mathbb{N}$. Then the empirical risk function $\mathcal{R}_{\mathcal{L},D}$ is also convex.*

Proof. We begin by considering the Definition 1.1.18 of the empirical risk

$$\mathcal{R}_{\mathcal{L},D}(f) = \frac{1}{k}\sum_{i=1}^{k} \mathcal{L}\left(x_i, y_i, f(x_i)\right).$$

Since $\mathcal{L}$ is assumed to be a convex loss function as is its sum. Therefore, the empirical risk which is a finite sum of convex loss functions, is also a convex function and the assertion is proven. □

1.2 Neural networks

The idea of artificial neural networks originated from analysing mammal's brains. An accumulation of nodes - so called neurons, connected in a very special way that fire an electric impulse to adjacent neurons upon being triggered and transmit information that way. Scientists tried to mimic this natural architecture and replicate this mammal intelligence artificially. This research has been going for almost 80 years and became immensely popular recently through artificial intelligences like OpenAI's ChatGPT or Google's Bard for the broad public. But what actually is a neural network? What happens in a neural network? Those are very interesting and important questions that we will find answers for. We will introduce neural networks inspired by [13, Chapter 3].
As already mentioned, neural networks consist of single neurons that transmit information

upon being „triggered“. Obviously, triggering an artificial neuron can't happen the same way as neurological neurons are being triggered through stimulus. Hence, we need to model the triggering of a neuron in some way. The idea is to filter information that does not exceed a certain stimulus threshold. This filter is usually called activation function. In practice, there are many ways of modelling such activation functions and it mainly depends the use-case what exactly is demanded from the activation function. Therefore, we define activation functions as general as possible.

Definition 1.2.1. A non-constant function $\varphi : \mathbb{R} \to \mathbb{R}$ is called an **activation function** if it is continuous.

Even though there are lots of different activation functions, we want to consider mainly the following ones, see [3, Chapter 6].

Example 1.2.2. The following expressions define activation functions.

Rectified Linear Unit (ReLU): $\varphi(t) = \max\{0, t\}$,

Leaky Rectified Linear Unit (Leaky ReLU): $\varphi(t) = \begin{cases} \alpha t, & t \leq 0, \alpha > 0, \\ t, & t > 0. \end{cases}$

Sigmoid: $\varphi(t) = \dfrac{1}{1 + e^{-t}}$.

Now, having introduced activation functions we can introduce neurons.

Definition 1.2.3. Let $\varphi : \mathbb{R} \to \mathbb{R}$ be an activation function and $w \in \mathbb{R}^k$, $b \in \mathbb{R}$. Then a function $h : \mathbb{R}^k \to \mathbb{R}$ is called **φ-neuron** with weight w and bias b, if

$$h(x) = \varphi\left(\langle w, x\rangle + b\right), \qquad x \in \mathbb{R}^k. \tag{1.1}$$

We call $\theta \coloneqq (w, b)$ the parameters of the neuron h.

If we arrange multiple neurons into a vector, we can define a layer consisting of neurons. This way we can expand the architecture from one to multiple neurons.

Definition 1.2.4. Let $\varphi : \mathbb{R} \to \mathbb{R}$ be an activation function and $W \in \mathbb{R}^{m\times k}$, $b \in \mathbb{R}^m$. Then a function $H : \mathbb{R}^k \to \mathbb{R}^m$ is called **φ-layer** of width m with **weights** W and **biases** b if for all $i = 1, \ldots, m$ the component function h_i of H is a φ-neuron with weight $w_i = W^\top e_i$ and bias $b_i = \langle b, e_i\rangle$, where e_i denotes the standard ONB of $\mathbb{R}^m$.
If we consider $\widehat{\varphi} : \mathbb{R}^k \to \mathbb{R}$ as the component-wise mapping of $\varphi : \mathbb{R} \to \mathbb{R}$, meaning $\widehat{\varphi}(v) = (\varphi(v_1), \ldots, \varphi(v_k))$, we can write the φ-layer $H : \mathbb{R}^k \to \mathbb{R}^m$ as

$$H(x) = \widehat{\varphi}(Wx + b), \qquad x \in \mathbb{R}^k. \tag{1.2}$$

In the following, we will denote the weights W and biases b as **parameters of the neural layer** $\theta \coloneqq (W, b)$.

Finally, we can introduce neural networks as an arrangement of multiple neural layers.

Definition 1.2.5. Let $L \in \mathbb{N}$, $\varphi_1, \ldots, \varphi_L$ be activation functions and $H_1, \ldots, H_L$ be φ_i-layers with parameters $\theta_i = (W_i, b_i)$ for all $i \in \{1, \ldots, L\}$, where the output dimensions of the i-th layer equals the input dimensions of the $(i+1)$-th layer. Furthermore, let $\theta = (\theta_1, \ldots \theta_L)$ describe the **parameters of the neural network**, $\varphi = (\varphi_1, \ldots, \varphi_L)$ and $d_1, \ldots, d_{L+1} \in \mathbb{N}$. Then we define a **$\varphi$-deep neural network** of depth L with parameters θ as

$$\begin{aligned} f_{\varphi,L,\theta} : \mathbb{R}^{d_1} &\to \mathbb{R}^{d_{L+1}} \\ x &\to H_L \circ \ldots \circ H_1(x), \qquad x \in \mathbb{R}^{d_1}, \end{aligned} \tag{1.3}$$

Ultimately, d_1 describes the input dimension and d_{L+1} the output dimension of the neural network.
Lastly, we will write $f \coloneqq f_{\varphi,L,\theta}$, if the activation function φ, the depth L and the parameters θ are clear from the context.

Furthermore, we should consider how the parameters of a neural network actually look like. To do this we introduce the parameter space.

Definition 1.2.6. Let $f_{\varphi,L,\theta}$ be a neural network as in Definition 1.2.5, with parameters $\theta \in \Theta$, where we denote Θ as the **parameter space**, that is a set that contains all feasible parameters.

The parameter space is defined in a very general manner, since we usually do not demand any restrictions other than that the parameters should have fitting dimensions to be compatible with the considered neural networks.
At last, we want to introduce prediction functions. This is a quantity we will refer to oftentimes through the thesis and hence, should mention it at this point. It merely describes a neural network that was trained on a dataset. Hence, the parameters of the neural network are tuned to represent the dataset as good as possible. We will consider training of neural networks in Section 1.3.

Definition 1.2.7. Let $d, n \in \mathbb{N}$ and $X \subseteq \mathbb{R}^d$ and $Y \subseteq \mathbb{R}^n$, that we will refer to as **input** and **output space**. Furthermore, let $D = \{(x_1, y_1), \ldots, (x_k, y_k)\}$ be a **dataset** of length $k \in \mathbb{N}$. Then a neural network that is trained on the dataset D is called **prediction function**. We then denote its parameters as θ_D.

One may realize that neural networks are defined in a way that the activation function may vary in each layer. However, in most applications all hidden layers ($i \in \{1, \ldots, L-1\}$) share the same activation function. The last layer, often referred to as output layer, usually has a different activation function. This activation function may in some use-cases as well be the identity function.
A visual representation of a neural network can be found in Figure 1.1

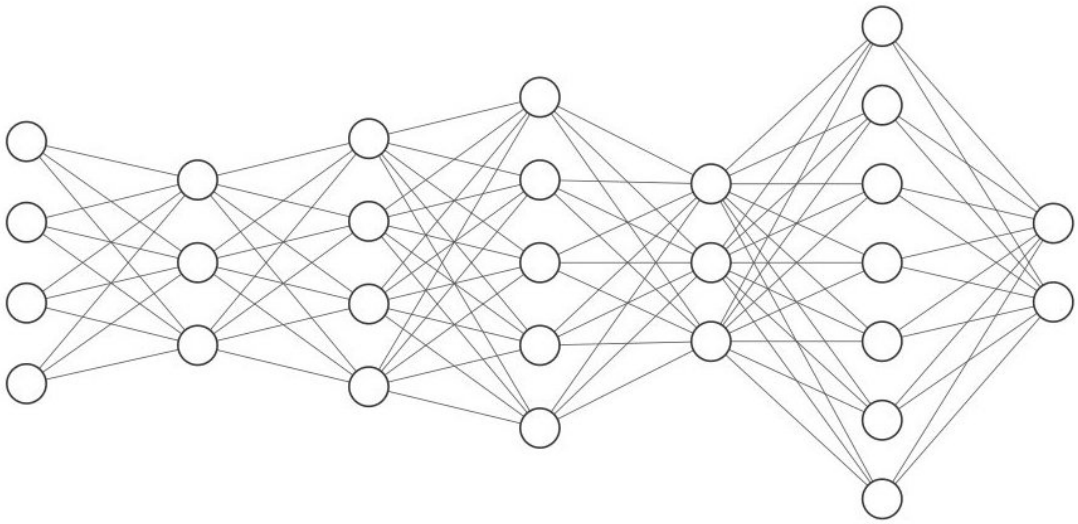


Figure 1.1: A neural network with input $x \in \mathbb{R}^4$ and output $y \in \mathbb{R}^2$. The five hidden layers have dimensions 3, 4, 5, 3 and 7 respectively. The graphic was generated with http://alexlenail.me/NN-SVG/index.html

1.3 Training of neural networks

Since we now know what neural networks are, we want to discuss how to tune them to a specific problem. This procedure is usually referred to as training of a neural network. There are many approaches of how to train a neural network (TODO: cite some approaches then!). However, most of them rely on iteratively finding the gradient - the direction of greatest ascent of the risk function, and afterwards performing an optimization step in the opposite of the found direction. This method of the steepest descent dates way back to the early 19^{th} century and was introduced by Augustin L. Cauchy in the year 1847. For a brief overview we refer to [10]. This method is immensely popular and thus, can be discovered in numerous literature. We will mostly draw inspiration form [5, Chapter XV]. For in-depth explanation, please take a look at this reference.
Let us consider a (non-linear) real-valued function f, which is defined on a real normed space X. We assume that f is bounded below on X and we aim to find an element $x^* \in X$ such that

$$f(x) \geq f(x^*),$$

for all $x \in X$. Hence, x^* minimizes the function f. To solve this problem, one usually constructs a sequence (x_n) that minimizes f, in the sense that

$$\lim_{n\to\infty} f(x_n) = \inf_{x\in X} f(x).$$

In certain cases, one can construct such a sequence such that it converges to an optimum x^*. If the considered function f is assumed to be continuous, then this element will be a solution to the proposed problem.
To construct such a sequence, we assume that the function f is **differentiable**. This means, that the directional derivative

$$\frac{\partial f}{\partial z}(x) := \frac{1}{\|z\|} \lim_{h \to 0^+} \frac{f(x+hz) - f(x)}{h},$$

exists at each point $x \in X$ and for every direction $z \in X$. If the function and its derivative are additionally continuous, we call it **continuously differentiable**.
Furthermore, we assume that there exists a direction for which the derivative takes minimum value, or in other words the direction of steepest descent. This direction is the negative direction of the gradient of f. If we now perform a step towards the direction of steepest descent, we completed an iteration of the so called steepest descent method - or nowadays better known as the gradient descent algorithm. This algorithm we now want to consider in detail.
Let X be a normed space and $f : X \to \mathbb{R}$ be a continuously differentiable function with existing minimum at x^*. Furthermore, let there be a direction of steepest descent in every $x \in X$ and $x^{(0)} \in X$ an arbitrary (initial) element. Assume that we already found the k-th iterate $x^{(k)}$, then we define the $(k+1)$-th iterate by

$$x^{(k+1)} := x^{(k)} - \gamma_{k+1}\, z_{k+1}, \tag{1.4}$$

where z_{k+1} denotes the direction of steepest descent at $x^{(k)}$. The numerical parameter $\gamma^{(k+1)}$ can be found in various ways, e.g. [5, Chapter XV]. However, we want to approach it by finding the optimal step size analytically.
We specify a sequence of positive numbers (γ_k) such that $\gamma_k \to 0$ and $\sum_{k \geq 1} \gamma_k = \infty$. Furthermore, we assume the z_k to be normalized for all $k \in \mathbb{N}$, i.e. $\|z_k\| = 1$. Hence, the step size at the k-th iteration is γ_k. We do this in order to avoid doing to great steps, which would eventually avert the convergence of the algorithm.
Furthermore, we note at this point that we can establish a connection between the problem of minimizing a functional and that of solving a linear functional equation as follows.
Let U be a self-adjoint operator on a Hilbert space $\mathcal{H}$. We assume that the operator U is below bounded by $m > 0$ and above bounded by $M > 0$, i.e. the following equations hold

$$m := \inf_{x \in X, \|x\|=1} \langle Ux, x \rangle, \qquad \text{and} \qquad M := \sup_{x \in X, \|x\|=1} \langle Ux, x \rangle,$$

which implies that U is an injective operator.
Now, lets consider the linear functional equation

$$Ux = y. \tag{1.5}$$

Since U is injective the inverse operator U^{-1} exists. Therefore, there exists one unique solution to (1.5), for each $y \in \mathcal{H}$. Now, we define the functional

$$F(x) = \langle Ux, x \rangle - (\langle x, y \rangle + \langle y, x \rangle) = \langle Ux, x \rangle - 2 \langle x, y \rangle. \tag{1.6}$$

With the help of the functional (1.6) we can formulate the following theorem.

Theorem 1.3.1. *Let $\mathcal{H}$ be a Hilbert space. A solution $x^* \in \mathcal{H}$ of (1.5) yields a minimum of (1.6). Conversely, if (1.6) attains a minimum at $x' \in \mathcal{H}$, then x' is a solution of (1.5), that is, $x' = x^*$.*

Proof. Since we assumed $x^* \in \mathcal{H}$ to be a solution of (1.5), we can express F as

$$F(x) = \langle Ux, x\rangle - \langle x, Ux^*\rangle - \langle Ux^*, x\rangle = \langle U(x - x^*), x - x^*\rangle - \langle Ux^*, x^*\rangle . \tag{1.7}$$

Using the boundedness of U we receive

$$\begin{aligned} F(x) \geq m \langle x - x^*, x - x^*\rangle - \langle Ux^*, x^*\rangle &\geq - \langle Ux^*, x^*\rangle \\ &= \langle Ux^*, x^*\rangle - 2 \langle Ux^*, x^*\rangle = F(x^*). \end{aligned} \tag{1.8}$$

That is, $x^* \in \mathcal{H}$ indeed minimizes the functional F.
To prove the second part of the theorem, we assume that (1.6) attains a minimum at $x' \in \mathcal{H}$. Furthermore, we assume that $x^* \in \mathcal{H}$ is the solution of (1.5). Therefore, we consider the following equation

$$F(x') - F(x^*) = 0.$$

The definition of the functional (1.6) gives us

$$F(x') - F(x^*) = \langle Ux', x'\rangle - 2 \langle x', y\rangle - \langle Ux^*, x^*\rangle - 2 \langle x^*, y\rangle .$$

Now we use the fact, that x^* solves the equation (1.5)

$$\langle Ux', x'\rangle - 2 \langle x', y\rangle - \langle Ux^*, x^*\rangle + 2 \langle x^*, y\rangle = \langle Ux', x'\rangle - 2 \langle x', Ux^*\rangle + \langle Ux^*, x^*\rangle .$$

Lastly, we use the linearity of the inner product as in equation (1.7) and the lower bound of the operator U.

$$\begin{aligned} \langle Ux', x'\rangle - 2 \langle x', Ux^*\rangle + \langle Ux^*, x^*\rangle &= \langle U(x' - x^*), x' - x^*\rangle \\ &\geq m \langle x' - x^*, x' - x^*\rangle , \end{aligned}$$

therefore, $x' = x^*$. □

With the help of Theorem 1.3.1 we realise that finding the solution to a linear functional equation is equivalent to minimizing the corresponding functional. Hence, we want to apply the method of steepest descent to the functional in order to find the solution of the linear functional equation. In other words, applying the method of steepest descent to a given functional gives us a sequence $(x^{(k)})$ that converges to a x^*, which minimizes the functional. This consideration we now want to formulate in detail.

Corollary 1.3.2. *Let $\mathcal{H}$ be a Hilbert space and F be a functional as defined in (1.6). Then the gradient descent algorithm applied to the functional F with arbitrary initial iterate $x^{(0)} \in \mathcal{H}$ looks like*

$$x^{(k)} = x^{(k-1)} - \gamma_{k-1}\, z_{k-1},$$

where $z_k \in \mathcal{H}$ and $\gamma_k > 0$ look like

$$z_k = Ux^{(k-1)} - y \qquad \text{and} \qquad \gamma_k = \frac{\langle z_k, z_k\rangle}{\langle Uz_k, z_k\rangle}.$$

Moreover, we note that we choose an optimal step size γ_k in each iteration.

Proof. Let $x, z \in \mathcal{H}$, then

$$\begin{aligned} F(x+z) &= \langle U(x+z), x+z\rangle - (\langle x+z, y\rangle + \langle y, x+z\rangle) \\ &= \langle Ux, x\rangle - (\langle x, y\rangle + \langle y, x\rangle) + (\langle Ux-y, z\rangle + \langle z, Ux-y\rangle) + \langle Uz, z\rangle \\ &= F(x) + (\langle Ux-y, z\rangle + \langle z, Ux-y\rangle) + \langle Uz, z\rangle . \end{aligned} \tag{1.9}$$

Considering the derivative in direction z gives us

$$\frac{\partial F}{\partial z}(x) = \frac{1}{\|z\|}(\langle Ux-y, z\rangle + \langle z, Ux-y\rangle) = \frac{2\langle Ux-y, z\rangle}{\|z\|},$$

where in the last equation we used the fact that F is a real-values function.
Thus the direction of steepest ascent at $x^{(0)} \in \mathcal{H}$ is given by $z_1 = Ux^{(0)} - y$ and the direction of steepest descent by $-z_1$.
Lastly, to determine the descent value we want to introduce the concept of rays, emanating of a point $x \in \mathcal{H}$ in direction $z \in \mathcal{H}$. This is the set of elements that is generated by $x+\alpha z$, where $\alpha > 0$ is a non-negative real number and $z \in \mathcal{H}$ is the direction of the ray. The restriction of the function F onto this ray is thereby a function of a real variable that we will denote as $f(\alpha; x, z) = F(x+\alpha z)$ for $\alpha \geq 0$.
We consider the ray emanating of our current iterate $x^{(0)}$ in the direction of the steepest descent z_1 and consider it as a real valued function in α. Since we want to find an optimal descent value, we look for the minimum on this ray, what ultimately results in a one dimensional optimization problem. Hence, we consider the equation

$$\frac{\partial f(\alpha; x^{(0)}, -z_1)}{\partial \alpha} = 0, \tag{1.10}$$

where with the use of equation (1.9), we have

$$f(\alpha; x^{(0)}, -z_1) = F(x^{(0)} - \alpha z_1) = F(x^{(0)}) - 2\alpha \langle z_1, z_1\rangle + \alpha^2 \langle Uz_1, z_1\rangle .$$

Thus, the derivative $\partial f(\alpha; x^{(0)}, -z_1)/\partial\alpha$ looks like

$$\frac{\partial f(\alpha; x^{(0)}, -z_1)}{\partial \alpha} = -2\langle z_1, z_1\rangle + 2\alpha \langle Uz_1, z_1\rangle .$$

Considering the condition (1.10) gives us the descent value by solving for α

$$\begin{aligned} &\iff && 2\langle z_1, z_1\rangle = 2\alpha \langle Uz_1, z_1\rangle \\ &\iff && \frac{\langle z_1, z_1\rangle}{\langle Uz_1, z_1\rangle} = \alpha. \end{aligned}$$

Therefore, by defining $\gamma_1 = \alpha$ we receive the optimal descent value in the direction z_1.
In exactly the same way γ_k and z_k can be determined for all $k > 1$ and hence, all assertions from the theorem are proven. □

The gradient descent algorithm with parameters as asserted in Corollary 1.3.2 does converge, as we have seen in the proof. However, one might pose the question how quickly it converges. This question we want to consider in the following theorem.

Theorem 1.3.3. *Let the same assumptions as in Corollary 1.3.2 hold. Then the constructed sequence $(x^{(n)})$ with $(z_n), (\gamma_n)$ as in Corollary 1.3.2, converges to $x^* \in \mathcal{H}$. Its speed of convergence is given by*

$$\left\|x^{(n)} - x^*\right\| \leq \frac{\|z_1\|}{m} \left(\frac{M-m}{M+m}\right)^n,$$

for all $n \in \mathbb{N}_0$.

Proof. Firstly, we rewrite the equation (1.5) with $k > 0$ to

$$\begin{aligned} & Ux = y \\ \iff \quad & 0 = ky - kUx \\ \iff \quad & x = x - kUx + ky. \end{aligned} \tag{1.11}$$

We chose the numerical factor k in a way, that the operator $T = \mathrm{id} - kU$ has the smallest possible norm, where we denote id as the identity operator. Since we assumed U to be lower bounded by m and upper bounded by M, the operator T needs to be lower bounded by $1-km$ and upper bounded by $1 - kM$, where the minimal norm will occur if

$$1 - km = -(1 - kM).$$

This leads to

$$k = \frac{2}{m+M}.$$

Therefore, for $\|T\|$ it holds

$$\|T\| = 1 - km = 1 - \frac{2m}{m+M}, \tag{1.12}$$

as well as

$$\|T\| = kM - 1 = \frac{2M}{m+M} - 1. \tag{1.13}$$

Combining the equations (1.12) and (1.13) leads to

$$\begin{aligned} 2\|T\| &= \left(1 - \frac{2m}{m+M}\right) + \left(\frac{2M}{m+M} - 1\right) = \frac{2M-2m}{m+M} \\ \iff \quad \|T\| &= \frac{M-m}{M+m}. \end{aligned}$$

Moreover, applying (1.11) and rewriting it gives us

$$x_1' = Tx^{(0)} + ky = x^{(0)} - k(Ux^{(0)} - y) = x^{(0)} - kz_1. \tag{1.14}$$

Let us introduce the operator $V = U^{1/2}$, which is the root operator of U, see e.g [5, Theorem V.6.2]. This is the unique operator that fulfils the equation $V^2 = U$. Furthermore, we should not that the operator V indeed exists due to the fact that U is a self-adjoint and hence

positive operator. The same properties then hold for the operator V as well.
This operator V we now plug into (1.7).

$$\begin{aligned} F(x) &= \langle U(x-x^*), x-x^*\rangle - \langle Ux^*, x^*\rangle \\ &= \langle V(x-x^*), V(x-x^*)\rangle - \langle Vx^*, Vx^*\rangle \\ &= \|V(x-x^*)\|^2 - \|Vx^*\|^2 \end{aligned} \tag{1.15}$$

Considering the inequality $F(x_1') \geq F(x^{(1)})$, which holds due to the fact that $x^{(1)}$ is the iterate performed with optimal descent value, leads to

$$F(x^{(1)}) - F(x^*) \leq F(x_1') - F(x^*),$$

which we can rewrite again with the use of (1.15)

$$\left\|V(x^{(1)} - x^*)\right\| \leq \|V(x_1' - x^*)\|. \tag{1.16}$$

Since equation (1.7) was the rewritten equation (1.5), we can write with the definition of T

$$x^* = Tx^* + ky,$$

subtracting this equation from (1.14) gives us

$$x_1' - x^* = T(x^{(0)} - x^*),$$

applying the operator V on both sides gives us

$$V\left(x_1' - x^*\right) = VT(x^{(0)} - x^*). \tag{1.17}$$

With the easy calculation

$$VT = V\left(I - kU\right) = V - kVU = V - kVVV = \left(I - kU\right)V = TV,$$

we realise that the operators T and V commute. Hence, we can rewrite (1.17) to

$$V\left(x_1' - x^*\right) = TV(x^{(0)} - x^*).$$

Considering the norm leads to

$$\begin{aligned} \|V\left(x_1' - x^*\right)\| &= \left\|TV(x^{(0)} - x^*)\right\| \leq \|T\| \left\|V(x^{(0)} - x^*)\right\| \\ &= \frac{M-m}{M+m} \left\|V(x^{(0)} - x^*)\right\| \end{aligned}$$

Therefore, with inequality (1.16) we have

$$\left\|V(x^{(1)} - x^*)\right\| \leq \frac{M-m}{M+m} \left\|V(x^{(0)} - x^*)\right\|.$$

If we apply exactly the same arguments for all $k \in \{1, \dots n\}$, then we have the following bound for the n-th iterate

$$\left\|V(x^{(n)} - x^*)\right\| \leq \frac{M-m}{M+m} \left\|V(x^{(n-1)} - x^*)\right\|,$$

which leads to the bound

$$\left\|V(x^{(n)} - x^*)\right\| \leq \left(\frac{M-m}{M+m}\right)^n \left\|V(x^{(0)} - x^*)\right\|.$$

Furthermore, we realise that the function $t^{-1/2}$ is continuous in $[m, M]$ and therefore especially on the spectrum of U, which leads to the observation that the inverse operator $U^{-1/2} = V^{-1}$ exists.
Moreover, with the help of [5, Theorem V.6.2] (which regards the spectrum S_U of the operator U) the following equations hold

$$\begin{aligned}\left\|V^{-1}\right\| &= \max_{t \in S_U} \frac{1}{\sqrt{t}} = \frac{1}{\sqrt{m}},\\ \|V\| &= \max_{t \in S_U} \sqrt{t} = \sqrt{M}.\end{aligned}$$

Combining the above results, we receive

$$\begin{aligned}\left\|x^{(n)} - x^*\right\| = \left\|V^{-1}V\left(x^{(n)} - x^*\right)\right\| &\leq \left\|V^{-1}\right\| \left\|V\left(x^{(n)} - x^*\right)\right\|\\ &\leq \left\|V^{-1}\right\| \left(\frac{M-m}{M+m}\right)^n \left\|V\left(x^{(0)} - x^*\right)\right\|\\ &\leq \frac{1}{\sqrt{m}} \left(\frac{M-m}{M+m}\right)^n \left\|V\left(x^{(0)} - x^*\right)\right\|. \qquad (1.18)\end{aligned}$$

However, since the unknown element x^* appears on the right-hand side, we want to somehow remove it. In order to do this, we consider

$$\left\|V(x^{(0)} - x^*)\right\| = \left\|V^{-1}U(x^{(0)} - x^*)\right\| \leq \left\|V^{-1}\right\| \left\|U(x^{(0)} - x^*)\right\| = \frac{\|z_1\|}{\sqrt{m}}.$$

Plugging this result into (1.18), gives us

$$\left\|x^{(n)} - x^*\right\| \leq \frac{\|z_1\|}{m} \left(\frac{M-m}{M+m}\right)^n.$$

This is exactly the inequality asserted in the theorem. □

As we saw in Theorem 1.3.3 the gradient descent algorithm does converge, we even determined an upper bound to its convergence speed. However, the algorithm relies on computing the gradient of the function to be optimized. In our setting, this is the gradient of the risk function that has a neural network as prediction function, where the gradient is computed with regard to the parameters of the neural network. Since it is unclear how such a gradient actually looks like, we want to consider some examples.

Example 1.3.4. Let's first consider the easiest neural network possible, two connected neurons. Afterwards, we want to generalize this to a neural network that consists of $L \in \mathbb{N}$ layers, where each layer does only have one neuron. Lastly, we will consider a neural network with two hidden layers, where there are $d_1 > 1$ and $d_2 > 1$ neurons in the first and second layer, respectively. Moreover, let φ be the sigmoid activation function, D be an arbitrary dataset of length $d \in \mathbb{N}$ and $\mathcal{L}$ be the mean squared error loss, see Example 1.1.16.

1. We consider a neural network f, that consists of an input neuron and an output neuron. Hence, we can describe it as

$$f(x) = \varphi(wx + b) \qquad x \in \mathbb{R}, \tag{1.19}$$

where $w \in \mathbb{R}$ is the networks weight and $b \in \mathbb{R}$ is the networks bias.
Since we are interested in computing the gradient of the empirical risk function, we want to formulate it explicitly.

$$\mathcal{R}_{\mathcal{L},D}(f_\theta) = \frac{1}{d}\sum_{i=1}^{d} |y_i - f(x_i)|^2 .$$

Plugging in the explicit definition of the neural network from equation (1.19), gives us

$$\mathcal{R}_{\mathcal{L},D}(f_\theta) = \frac{1}{d}\sum_{i=1}^{d} |y_i - (\varphi(wx_i) + b)|^2 . \tag{1.20}$$

Now, lets consider how the gradient of (1.20) with regard to the parameters $\theta = (w, b)$ of the neural network f_θ looks like. In mathematical notation, this means $\partial\,\mathcal{R}_{\mathcal{L},D}(f_w)/\partial w$. However, since the empirical risk function is the mean of loss functions, the gradient of the empirical risk function is the mean of the gradients of the loss functions as well. Hence, we need to compute $\partial\,\mathcal{L}(f_\theta, y)/\partial\theta$.
At this point, we want to remember the chain rule of differentiation, see e.g. [17, Chapter19.6] which states

$$\frac{\partial f(g(x))}{\partial x} = \frac{\partial g(x)}{\partial x}\frac{\partial f(g(x))}{\partial g(x)}. \tag{1.21}$$

With the help of the chain rule, we can compute the gradient of the loss function by computing it as

$$\frac{\partial\,\mathcal{L}(f_\theta(x), y)}{\partial\theta} = \frac{\partial\,\mathcal{L}(f_\theta(x), y)}{\partial f_\theta(x)}\frac{\partial f_\theta(x)}{\partial\theta}. \tag{1.22}$$

This expression we can compute easily, since $\partial f_\theta(x)/\partial\theta$ is

$$\begin{aligned}\frac{\partial f_\theta(x)}{\partial\theta} &= \left[\frac{\partial(\varphi(wx)+b)}{\partial w}, \frac{\partial(\varphi(wx)+b)}{\partial b}\right]^\intercal \\ &= \left[\frac{\partial(\varphi(wx)+b)}{\partial wx}\frac{\partial wx}{\partial w}, \frac{\partial(\varphi(wx)+b)}{\partial b}\right]^\intercal = \left[x\,\varphi'(x), 1\right]^\intercal ,\end{aligned}$$

where the derivative of the sigmoid function is $\varphi'(x) = \varphi(x)(1 - \varphi(x))$.
Furthermore, the second part $\partial\,\mathcal{L}(f_w(x), y)/\partial f_w(x)$ of the right-hand side (1.22) is

$$\frac{\partial\,\mathcal{L}(f_\theta(x), y)}{\partial f_\theta(x)} = \frac{\partial|f_\theta(x) - y|^2}{\partial f_\theta(x)} = 2\,(f_\theta(x) + b - y) = 2\,(\varphi(wx) + b - y)\,.$$

Therefore, plugging this into equation (1.22), we receive

$$\begin{aligned}\frac{\partial \mathcal{L}(f_\theta(x), y)}{\partial \theta} &= 2\left(\varphi(wx) + b - y\right)\varphi'(x)\left[x\,\varphi'(x), 1\right]^\intercal \\ &= 2\begin{bmatrix} x(\varphi'(x))^2\left(\varphi(wx) + b - y\right) \\ \left(\varphi(wx) + b - y\right)\varphi'(x) \end{bmatrix}.\end{aligned}$$

Lastly, we can insert the data samples $(x, y) \in D$ to compute the gradient of the empirical risk function. This leads to

$$\nabla_\theta \mathcal{R}_{\mathcal{L},D}(f_\theta) = \frac{2}{d}\sum_{(x,y)\in D}\begin{bmatrix} x(\varphi'(x))^2\left(\varphi(wx) + b - y\right) \\ \left(\varphi(wx) + b - y\right)\varphi'(x) \end{bmatrix}.$$

2. Since neural networks usually consist of more than two neurons, we now want to consider how we can generalize the neural network to a more complex architecture.
Let $L \in \mathbb{N}$ and $L+1$ denote the number of layers in the neural network, where each layer consist of one neuron. Hence for each $i = 1, \ldots, L$ holds

$$\begin{aligned} H_i : \mathbb{R} &\to \mathbb{R}, \\ x &\mapsto \varphi\left(wx_i + b_i\right). \end{aligned}$$

For simplicity reasons, we assume that $\varphi = \mathrm{id}$ and $b_i = 0$ for all $i = 1, \ldots L$. Otherwise, the computation would be the same as in the first example. Therefore, we can denote the neural network as

$$f_\theta(x) = H_L \circ \ldots \circ H_1(x) = H_L(H_{L-1}(\ldots(H_1(x))\ldots) \qquad x \in \mathbb{R},$$

where $\theta = (\theta_1, \ldots, \theta_L)$ denotes the parameters of the neural network. Since we assumed $b_i = 0$ for all $i = 1, \ldots L$, we can denote the parameters as $\theta = (w_1, \ldots w_L)$.
Our goal is to compute the gradient of the empirical risk function, what we can do by computing the gradient of the loss function, analogously to the first example. However, instead of considering the whole vector θ at once, we want to compute the gradient for each parameter $\theta_i = w_i$ separately, since for the gradient holds

$$\nabla_\theta \mathcal{L}(f_\theta(x), y) = \begin{bmatrix} \nabla_{w_1}\mathcal{L}(f_\theta(x), y) \\ \vdots \\ \nabla_{w_L}\mathcal{L}(f_\theta(x), y) \end{bmatrix}.$$

To compute the gradients, we start at the last layer and iteratively compute the parameters of the preceding layer. This method is well known as back-propagation, which we will define more general after considering this comparatively easy example.
As already described, we want to start by computing $\partial \mathcal{L}(f_\theta(x), y)/\partial w_L$. Considering the chain rule from equation (1.21), we receive

$$\frac{\partial \mathcal{L}(f_\theta(x), y)}{\partial w_L} = \frac{\partial \mathcal{L}(f_\theta(x), y)}{\partial f_\theta(x)}\frac{\partial f_\theta(x)}{\partial w_L}.$$

At this point, we want to define $x^{(i)} \coloneqq H_i(x)$ for all $i = 1, \dots L$, where $x|in\,\mathbb{R}^{i-1}$. With this definition we can write $f_\theta(x) = H_L(x^{(L-1)})$ and therefore,

$$\frac{\partial\,\mathcal{L}(f_\theta(x), y)}{\partial w_L} = \frac{\partial\,\mathcal{L}(f_\theta(x), y)}{\partial f_\theta(x)} \frac{\partial H_L(x^{(L-1)}}{\partial w_L}. \tag{1.23}$$

As the next step, we want to consider both quantities of the right-hand side of (1.23) separately. The first part $\partial\,\mathcal{L}(f_\theta(x), y)/\partial f_\theta(x)$ results to be

$$\frac{\partial\,\mathcal{L}(f_\theta(x), y)}{\partial f_\theta(x)} = \frac{\partial (f_\theta(x) - y)^2}{\partial f_\theta(x)} = 2(f_\theta(x) - y),$$

and for the second part $\partial H_L(x^{(i-1)})/\partial w_L$ holds

$$\frac{\partial H_L(x^{(i-1)})}{\partial w_L} = \frac{\partial w_L x^{(L-1)}}{\partial w_L} = x^{(L-1)} = x \prod_{i=1}^{L-1} w_i.$$

Therefore, plugging the two results into (1.23) gives us

$$\frac{\partial\,\mathcal{L}(f_\theta(x), y)}{\partial w_L} = 2x(f_\theta(x) - y) \prod_{i=1}^{L-1} w_i.$$

Doing this for all $i = 1, \dots, L-1$ gives us for the parameters w_i

$$\frac{\partial\,\mathcal{L}(f_\theta(x), y)}{\partial w_i} = \frac{\partial\,\mathcal{L}(f_\theta(x), y)}{\partial f_\theta(x)} \frac{\partial H_i(x^{(i-1)}}{\partial w_i},$$

where we define $x^{(0)} \coloneqq x$. Hence, we receive for $i = 2, \dots, L$

$$\frac{\partial\,\mathcal{L}(f_\theta(x), y)}{\partial w_i} = 2x(f_\theta(x) - y) \prod_{k=1}^{i-1} w_k.$$

Considering the case $i = 1$ separately, gives us

$$\begin{aligned}\frac{\partial\,\mathcal{L}(f_\theta(x), y)}{\partial w_1} &= \frac{\partial\,\mathcal{L}(f_\theta(x), y)}{\partial f_\theta(x)} \frac{\partial H_1(x^{(0)}}{\partial w_1} \\ &= \frac{\partial (f_\theta(x) - y)^2}{\partial f_\theta(x)} \frac{\partial w_1 x}{\partial w_1} = 2x(f_\theta(x) - y).\end{aligned}$$

Combining the computed gradients gives us the desired gradient of the loss function

$$\nabla_\theta\,\mathcal{L}(f_\theta(x), y) = \begin{bmatrix} 2x(f_\theta(x) - y) \\ 2x(f_\theta(x) - y)w_1 \\ \vdots \\ 2x(f_\theta(x) - y) \prod_{k=1}^{L-1} w_k \end{bmatrix}.$$

Lastly, we remember that the gradient of the empirical risk function is the mean of gradients of the loss function. Hence, the gradient of the empirical risk function with reard to the parameters of the neural network looks like

$$\nabla_\theta\,\mathcal{R}_{\mathcal{L},D}(f_\theta) = \frac{2}{d} \sum_{(x,y)\in D} \begin{bmatrix} x(f_\theta(x) - y) \\ x(f_\theta(x) - y)w_1 \\ \vdots \\ x(f_\theta(x) - y) \prod_{k=1}^{L-1} w_k \end{bmatrix}.$$

We just saw an example of the gradient of an empirical risk function, where the prediction function is a (deep) neural network in Example 1.3.4. However, we only considered the layers to have one single neuron. Naturally, in practice neural networks rarely have layers with merely one neuron. This generalization we now want to consider. But firstly, we note that the computation of the gradient relied on the chain rule, well known in calculus. Since we now want to generalize the layers to have more than one neuron, we need to extend the chain rule to multiple dimensions. It is then called multi-variable chain rule. This result is fundamental to the immensely popular back-propagation algorithm, which is often used in scenarios where one needs to compute the gradient of a function.

Theorem 1.3.5. *Let $n, m \in \mathbb{N}$ and $f : \mathbb{R}^n \to \mathbb{R}$ and $g : \mathbb{R}^m \to \mathbb{R}^m$ be differentiable functions. Furthermore, we define $y := g(x)$ and $z = f(y)$, then the* ***multi-variable chain rule*** *states that the following holds*

$$\frac{\partial z}{\partial x_i} = \sum_{j \geq 1} \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}.$$

In vector notation this can be equivalently expressed as

$$\nabla_x z = \left(\frac{\partial y}{\partial x} \right)^\intercal \nabla_y z,$$

where $\partial y / \partial x$ is the $n \times m$ Jacobian matrix of g.

At this point, we want to note that the chain rule can even be extended from vectors to tensors. This is very interesting, since in many Machine Learning settings the proposed neural networks operate on tensors. However, we do not want to consider this in depth and rather refer to [3, Chapter 6.5.2].
Returning to the theory we want to consider, with the help of the multi-variable chain rule which we introduced in Theorem 1.3.5 we can compute the gradients of empirical risk functions, where the corresponding neural network has arbitrary size of layers.
The back-propagation algorithm as already described, is in its fundamental idea simple. Since the weights in each layer of the neural network depend on all the consecutive weights, we start computing the gradients for all weights in the last layer and iteratively work our way through the previous layers, until we computed the gradients for every weight. However, formulating this idea formally correct is quite messy. Therefore, we want to introduce some quantities beforehand. We know that each layer H_i looks like

$$\begin{aligned} H_i : \mathbb{R}^{d_{i-1}} &\to \mathbb{R}^{d_i}, \\ x &\mapsto H_i(x) = \varphi(W_i x + b_i), \end{aligned}$$

where d_i describes the output dimension of the i-th layer and d_{i-1} describes the input dimension of the i-th layer, as we defined in Definition 1.2.4. In the following we want to denote the weight matrix as $W^{(i)} := W_i$, since we will need the index otherwise. Furthermore, we want to denote the weight matrix as $W^{(i)} = (w^{(i)}_{jk})$, where each of the entries w^i_{jk} describes the weight between the k-th neuron in the $(i-1)$-th layer and the j-th neuron in the i-th layer. With these considerations we can write the input to the j-th neuron of the i-th layer, which we want to

denote as $z_j^{(i)}$, as

$$z_j^{(i)} = \sum_{k=1}^{d_{i-1}} w_{jk}^{(i)} \, \varphi(z_k^{(i-1)}) + b_j^{(i)},$$

where $b_j^{(i)}$ denotes the j-th entry of the bias in the i-th layer.
To simplify the notation even more, we will denote $a_j^{(i)} = \varphi(z_j^{(i)})$. We note that the $z_j^{(i)}$ are defined in such a way that they describe the output of a neuron before applying the activation function and the $a_j^{(i)}$ are defined so that they describe the output of a neuron after applying the activation function. Hence, we will call this quantity the **activation** of the j-th neuron in the i-th layer.
Moreover, we note that the gradient of the risk function, which is the mean of gradients of the loss function with regard to the parameters of the neural network θ, can be denoted as follows

$$\nabla_\theta \, \mathcal{R}_{\mathcal{L},D}(f_\theta) = \begin{bmatrix} \nabla_{\theta_1} \, \mathcal{R}_{\mathcal{L},D}(f_\theta) \\ \vdots \\ \nabla_{\theta_L} \, \mathcal{R}_{\mathcal{L},D}(f_\theta) \end{bmatrix},$$

where again the gradients $\nabla_{\theta_i} \, \mathcal{R}_{\mathcal{L},D}(f_\theta)$ can be expressed as a vector of gradients with regard to the weights and biases of the i-th layer.
With the above considerations we can compute the gradients of the risk function in each parameter. However, the computation is slightly different in the output layer and in the hidden layers. First, we want to consider the gradients for the output layer. We know that if we consider the mean squared error loss $\mathcal{L}$ for an arbitrary data sample (x, y) the loss function looks like

$$\mathcal{L}(y, f(x)) = \sum_{j=0}^{d_L - 1} \left(a_j^{(L)} - y_j \right)^2, \tag{1.24}$$

such that it is the squared sum over all entries of the output vector, which is the activation vector of the last layer $a^{(L)} \coloneqq (a_1^{(L)}, \ldots, a_{d_L}^{(L)})$. Moreover, we consider $\nabla_{\theta_L} \, \mathcal{R}_{\mathcal{L},D}(f_\theta)$ as the vector of gradients

$$\nabla_{\theta_L} \, \mathcal{R}_{\mathcal{L},D}(f_\theta) = \begin{bmatrix} \nabla_{w_{11}^{(L)}} \, \mathcal{R}_{\mathcal{L},D}(f_\theta) \\ \vdots \\ \nabla_{w_{jk}^{(L)}} \, \mathcal{R}_{\mathcal{L},D}(f_\theta) \\ \nabla_{b_1^{(L)}} \, \mathcal{R}_{\mathcal{L},D}(f_\theta) \\ \vdots \\ \nabla_{b_j^{(L)}} \, \mathcal{R}_{\mathcal{L},D}(f_\theta) \end{bmatrix}.$$

These gradients we can compute easily now. We already mentioned that the gradient of the risk function is the mean of gradients of the loss functions. Hence, we can consider the gradients of

the loss functions analogously to the gradient of the risk function as

$$\nabla_{\theta_L} \mathcal{L}(y, f_\theta(x)) = \begin{bmatrix} \nabla_{w_{11}^{(L)}} \mathcal{L}(y, f_\theta(x)) \\ \vdots \\ \nabla_{w_{jk}^{(L)}} \mathcal{L}(y, f_\theta(x)) \\ \nabla_{b_1^{(L)}} \mathcal{L}(y, f_\theta(x)) \\ \vdots \\ \nabla_{b_j^{(L)}} \mathcal{L}(y, f_\theta(x)) \end{bmatrix}.$$

Explicitly, the computation of these gradients looks like

$$\nabla_{w_{jk}^{(L)}} \mathcal{L}(y, a^{(L)}) = \frac{\partial \mathcal{L}(y, a^{(L)})}{\partial w_{jk}^{(L)}} = \frac{\partial \mathcal{L}(y, f_\theta(x))}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial w_{jk}^{(L)}} = \frac{\partial \mathcal{L}(y, f_\theta(x))}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}}.$$

Now, remembering the definition of these quantities, we can compute each of the three factors on the right-hand side. It holds then with equation (1.24)

$$\frac{\partial \mathcal{L}(y, a^{(L)})}{\partial a_j^{(L)}} = \frac{\partial \sum_{j=0}^{d_L - 1} \left(a_j^{(L)} - y_j \right)^2}{\partial a_j^{(L)}} = 2(a_j^{(L)} - y_j). \tag{1.25}$$

For the second term holds

$$\frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} = \frac{\partial \varphi(z_j^{(L)})}{\partial z_j^{(L)}} = \varphi'(z_j^{(L)}), \tag{1.26}$$

and for the third term holds

$$\frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}} = \frac{\partial \left(\sum_{l=1}^{d_{L-1}} w_{jl}^{(L)} a_l^{(L-1)} + b_j^{(L)} \right)}{\partial w_{jk}^{(L)}} = a_k^{(L-1)}. \tag{1.27}$$

Therefore, combining the three partial derivatives gives us the gradient. This leads to

$$\nabla_{w_{jk}^{(L)}} \mathcal{L}(y, f_\theta(x)) = 2(a_j^{(L)} - y_j)\, \varphi'(z_j^{(L)}) a_k^{(L-1)}.$$

At this point we want to note quickly, that if we consider the partial derivative of the risk function with respect to the biases $b_j^{(L)}$, only the third term of the right-hand side changes. This term we considered in equation (1.27). The new partial derivative is then

$$\frac{\partial z_j^{(L)}}{\partial b_j^{(L)}} = \frac{\partial \left(\sum_{k=1}^{d_{L-1}} w_{jk}^{(L)} a_k^{(L-1)} + b_j^{(L)} \right)}{\partial b_j^{(L)}} = 1.$$

With these considerations we now know how to compute the gradient of the risk function with respect to θ_L, the parameters of the last layer. Before proceeding to compute the gradients with respect to the remaining weights, we want to first consider what we mentioned earlier. We said that computing the gradient is slightly different in the output layer than in the previous, the

hidden layers. This is due to the fact that a weight that is located in the output layer affects only the activation of one output neuron. E.g. if we consider the weight $w_{jk}^{(L)}$ which connects the k-th neuron of the $(L-1)$-th layer with the j-th neuron of the L-th layer, then it only affects this single output neuron. In contrast, the weights in all other previous layers, the hidden layers, affect the activations of all output neurons. Since we consider neural networks where each neuron affects all activations of the subsequent layer, it is sufficient to alter any weight in the hidden layers to affect the activation of the corresponding neuron which is attached to this weight and afterwards, this one altered activation has effect on all activations in all subsequent layers. Taking this into consideration, we can compute the gradient of the risk function with respect to the weight $w_{jk}^{(L-1)}$, which connects the k-th neuron of the $(L-2)$-th layer with the j-th neuron of the $(L-1)$-th layer. We receive

$$\nabla_{w_{jk}^{(L-1)}} \mathcal{L}(y, a^{(L)}) = \frac{\partial \mathcal{L}(y, a^{(L)})}{\partial w_{jk}^{(L-1)}}.$$

As we described the weight $w_{jk}^{(L-1)}$ affects all activations of the output layer. Thus, applying the chain rules gives us

$$\frac{\partial \mathcal{L}(y, a^{(L)})}{\partial w_{jk}^{(L-1)}} = \sum_{h=1}^{d_L} \frac{\partial \mathcal{L}(y, a^{(L)})}{\partial a_h^{(L)}} \frac{\partial a_h^{(L)}}{\partial w_{jk}^{(L-1)}},$$

where we sum the partial derivatives with respect to all neurons $a_h^{(L)}$ in the output layer. Applying the chain rule three more times gives us

$$\begin{aligned}
\frac{\partial \mathcal{L}(y, a^{(L)})}{\partial w_{jk}^{(L-1)}} &= \sum_{h=1}^{d_L} \frac{\partial \mathcal{L}(y, a^{(L)})}{\partial a_h^{(L)}} \frac{\partial a_h^{(L)}}{\partial z_h^{(L)}} \frac{\partial z_h^{(L)}}{\partial w_{jk}^{(L-1)}}, \\
&= \sum_{h=1}^{d_L} \frac{\partial \mathcal{L}(y, a^{(L)})}{\partial a_h^{(L)}} \frac{\partial a_h^{(L)}}{\partial z_h^{(L)}} \frac{\partial z_h^{(L)}}{\partial a_j^{(L-1)}} \frac{\partial a_j^{(L-1)}}{\partial w_{jk}^{(L-1)}}, \\
&= \sum_{h=1}^{d_L} \frac{\partial \mathcal{L}(y, a^{(L)})}{\partial a_h^{(L)}} \frac{\partial a_h^{(L)}}{\partial z_h^{(L)}} \frac{\partial z_h^{(L)}}{\partial a_j^{(L-1)}} \frac{\partial a_j^{(L-1)}}{\partial z_j^{(L-1)}} \frac{\partial z_j^{(L-1)}}{\partial w_{jk}^{(L-1)}}.
\end{aligned}$$

We realise, that the first three partial derivatives of the right-hand side are exactly the same computation as when we computed the gradient with regard to a weight that is located in the output layer. Furthermore, the last two expressions consider the effect of the altered activation of the $(L-1)$-th layer on the L-th, the output layer. Lastly, we want to explicitly consider the last partial derivative of the right-hand side. It holds that

$$\frac{\partial z_j^{(L-1)}}{\partial w_{jk}^{(L-1)}} = \frac{\partial \left(\sum_{l=1}^{d_{L-2}} w_{jl}^{(L-1)} a_l^{(L-2)} + b_j^{(L-1)} \right)}{\partial w_{jk}^{(L-1)}} = a_k^{(L-2)},$$

which is the same result as for the output layer, see (1.27). Therefore, we can apply exactly the same computation until we reach the input layer. This procedure allows us to component-wise compute the gradient $\nabla_\theta \mathcal{L}(a^{(L)}, y)$, at last. Finally, we can compute the gradient of the risk function with respect to the parameters θ by averaging the gradients of the loss function over

the given dataset D.
Having tackled the back-propagation algorithm we now know how to compute the gradient of the risk function with regard to the parameters of a given prediction function. However, as the back-propagation algorithm clearly shows, the computation of this gradient is highly expensive, since one has to compute the gradient for every single weight in the entire neural network, separately. A popular approach is to relax this computation by only considering the gradient in one sample (or multiple samples, we speak of a mini-batch then). We want to take a closer look at this algorithm as well. Since it is quite popular, there are many good references in literature, e.g see [18, Chapter 13.3.2], [16, Chapter 4.2] and [20]. We will focus on the latter reference.
Beforehand, we need to consider the empirical risk, which we defined in Definition 1.1.18. Since we take the average of the loss functions with regard to the whole data set D, we can denote it as an integral over the dataset D

$$\mathcal{R}_{\mathcal{L},D}(f) = \frac{1}{k}\sum_{i=1}^{k} \mathcal{L}\left(x_i, y_i, f(x_i)\right) = \int_D \mathcal{L}(x, y, f(x))d(x, y).$$

If we now randomly choose a subset $B \subseteq D$ of length $b \in \{1, \dots, n\}$ which we will call **mini-batch of size** b, we can approximate the empirical risk $\mathcal{R}_{\mathcal{L},D}(f)$ through $\mathcal{R}_{\mathcal{L},B}(f)$, where

$$\mathcal{R}_{\mathcal{L},B}(f) \coloneqq \int_B \mathcal{L}(x, y, f(x))d(x, y).$$

One may quickly realise, that if $b = n$, we are in the regular gradient descent setting. If one chooses to be $b = 1$, i.e. the subset B consisting of one single sample, then we speak of **stochastic gradient descent**, otherwise of **stochastic gradient descent with batch-size** b.
In the following, we want to summarize the data samples $\omega_i = (x_i, y_i)$. Hence, we can write

$$\mathcal{R}_{\mathcal{L},B}(f) = \int_B \mathcal{L}(\omega, f(x))d\omega.$$

Furthermore, we remember that we essentially are interested in the setting, where f is a neural network. Since the neural network f has parameters θ, which we denoted as f_θ, we will now simplify the notation as follows

$$\mathcal{R}_{\mathcal{L},B}(f_\theta) = \int_B \mathcal{L}(\omega, f_\theta(x))d\omega =: \int_B \mathcal{L}(\omega, \theta)d\omega =: \mathcal{R}_{\mathcal{L},B}(\theta).$$

Our goal is to find a minimum of $\theta \mapsto \mathcal{R}_{\mathcal{L},B}(\theta)$. We approach this problem the same way as we approached the gradient descent algorithm, where we iteratively update the parameters θ to minimize the risk. Therefore, we define the update rule as follows. Lets assume we already found the n-th iterate $\theta^{(n)}$. Now we randomly choose a sample ω_n and compute its loss with regard to the current parameters $\theta^{(n)}$. Afterwards, we compute the gradient of the empirical risk in the sample ω_n and update the parameters. This looks like

$$\theta^{(n+1)} = \theta^{(n)} - \gamma_n \nabla_\theta \mathcal{L}(\omega_n, \theta^{(n)}). \tag{1.28}$$

This way we define a sequence of parameters $(\theta^{(n)})$, which we will show to converge to some θ^* that minimizes the empirical risk $\mathcal{R}_{\mathcal{L},B}(\theta)$, at least under some assumptions we want to consider first.

Assumption 1.3.6. We assume that the risk function $\mathcal{R}_{\mathcal{L},\omega}$ meets the following conditions.

1. The gradient of $\mathcal{R}_{\mathcal{L},\omega}(\theta)$ is bounded, i.e.

$$\exists G > 0 : \sup_{\omega \in D} \left\| \nabla_\theta \mathcal{R}_{\mathcal{L},\omega}(\theta) \right\|^2 \leq G, \tag{1.29}$$

   for all $\theta \in \Theta$.

2. $\mathcal{R}_{\mathcal{L},\omega}(\theta)$ is strongly convex, i.e.

$$\exists\, \mu > 0 : \mathcal{R}_{\mathcal{L},\omega}(\theta_2) \geq \mathcal{R}_{\mathcal{L},\omega}(\theta_1) + \langle \nabla_\theta \mathcal{R}_{\mathcal{L},\omega}(\theta_2), \theta_2 - \theta_1 \rangle + \frac{\mu}{2} \left\| \theta_1 - \theta_2 \right\|^2, \tag{1.30}$$

   for all $\theta_1, \theta_2 \in \Theta$.

At this point we should note that the second assumption is quite restrictive and can indeed be relaxed. Since neural networks rarely are convex and especially not strictly convex functions in their parameters, the corresponding empirical risk function is neither. However, relaxing this assumption makes the proof far more challenging, what we do not want to tackle in this thesis. Instead we want to reference e.g. [9], where the authors prove the stochastic gradient descent algorithm to converge even for non-convex functions, which still have to fulfil some constraints, such as Hölder continuity and the Polyak-Łojasiewicz condition. Nonetheless, these conditions are far less restrictive and neural networks are proposed to meet those.
Having explored nuances of the made assumptions, we now return to the central focus, which is the convergence of the stochastic gradient descent algorithm.

Theorem 1.3.7. *Let $(\Omega, \mathcal{A}, P)$ be a probability space, X be an input space and $Y \subseteq \mathbb{R}$ be a label space and let Θ be a parameter space. Furthermore, let $\mathcal{L}$ denote a supervised loss function and let the corresponding empirical risk $\mathcal{R}_{\mathcal{L},\omega}$ be continuously differentiable in every $\omega \in X \times Y$. Then the stochastic gradient descent algorithm with update rule* (1.28) *and posed Assumption 1.3.6 satisfies the following assertions:*

1. *The empirical risk function $\mathcal{R}_{\mathcal{L},\omega}$ has a unique minimum at $\theta^* \in \Theta$.*

2. *For any $n \geq 0$ denote $\theta^{(n)} := \theta^{(n)}(\omega_1, \ldots, \omega_{n-1})$, since it depends on the choices of $\omega_1, \ldots, \omega_{n-1}$. Furthermore, we denote d_n as*

$$d_n = \mathbb{E}\left(\left\| \theta^{(n)} - \theta^* \right\|^2 \right) := \int_{(X \times Y)^{n-1}} \left\| \theta^{(n)} - \theta^* \right\|^2 d\omega_1 \ldots d\omega_{n-1}. \tag{1.31}$$

   Then for d_{n+1} holds

$$d_{n+1} \leq (1 - \gamma_n \mu) d_n + \gamma_n^2 B. \tag{1.32}$$

3. *For any $\epsilon > 0$ there exists a $\gamma > 0$ such that if $\gamma_n = \gamma$, then*

$$\limsup_{n \to \infty} \left(\left\| \theta^{(n)} - \theta^* \right\|^2 \right) \leq \epsilon\,. \tag{1.33}$$

4. If the sequence (γ_n) meets the conditions

$$\gamma_n \to 0 \qquad \textit{and} \qquad \sum_{n\geq 0} \gamma_n = \infty, \tag{1.34}$$

then $d_n \to 0$, that is $\lim_{n\to\infty} \theta^{(n)} = \theta^*$, where the convergence is the L^2-convergence of random variables, see e.g. [8, Chapter 7].

Proof. Assertion 1: The existence of a minimum follows from the continuity and lower boundedness of $\mathcal{R}_{\mathcal{L},\omega}$. The uniqueness follows from the strong convexity of $\mathcal{R}_{\mathcal{L},\omega}$.
Assertion 2: It holds that

$$\begin{aligned} d_{n+1} &= \mathbb{E}\left(\left\|\theta^{(n+1)} - \theta^*\right\|^2\right) = \mathbb{E}\left(\left\|\theta^{(n)} - \theta^* - \gamma_n \nabla_\theta \mathcal{L}(\omega_n, \theta^{(n)})\right\|^2\right) \\ &= \mathbb{E}\left(\left\|\theta^{(n)} - \theta^*\right\|^2\right) - 2\,\gamma_n\,\mathbb{E}\left(\left\langle \theta^{(n)} - \theta^*, \nabla_\theta \mathcal{L}(\omega_n, \theta^{(n)})\right\rangle\right) + \mathbb{E}\left(\gamma_n^2 \left\|\nabla_\theta \mathcal{L}(\omega_n, \theta^{(n)})\right\|^2\right). \end{aligned} \tag{1.35}$$

First, we consider that

$$\mathbb{E}\left(\left\langle \theta^{(n)} - \theta^*, \nabla_\theta \mathcal{L}(\omega_n, \theta^{(n)})\right\rangle\right) = \mathbb{E}\left(\theta^{(n)} - \theta^*, \nabla_\theta \mathcal{R}_{\mathcal{L},\omega_n}\right),$$

what we can bound with the help of the second assumption (1.30) by

$$\mathbb{E}\left(\theta^{(n)} - \theta^*, \nabla_\theta \mathcal{R}_{\mathcal{L},\omega_n}\right) \geq \mathbb{E}\left(\mathcal{R}_{\mathcal{L},\omega_n}(\theta^{(n)}) - \mathcal{R}_{\mathcal{L},\omega_n}(\theta^*) + \frac{\mu}{2}\left\|\theta^{(n)} - \theta^*\right\|^2\right).$$

Since θ^* minimizes $\mathcal{R}_{\mathcal{L},\omega}$, the difference $\mathbb{E}(\mathcal{R}_{\mathcal{L},\omega_n}(\theta^{(n)})) - \mathbb{E}(\mathcal{R}_{\mathcal{L},\omega_n}(\theta^*))$ is positive and we can bound this even further to

$$\mathbb{E}\left(\mathcal{R}_{\mathcal{L},\omega_n}(\theta^{(n)}) - \mathcal{R}_{\mathcal{L},\omega_n}(\theta^*) + \frac{\mu}{2}\left\|\theta^{(n)} - \theta^*\right\|^2\right) \geq \frac{\mu}{2}\,\mathbb{E}\left(\left\|\theta^{(n)} - \theta^*\right\|^2\right).$$

Lastly, we consider (1.35) and use the first assumption (1.29). This gives us

$$\begin{aligned} &\mathbb{E}\left(\left\|\theta^{(n)} - \theta^*\right\|^2\right) - 2\,\gamma_n\,\mathbb{E}\left(\left\langle \theta^{(n)} - \theta^*, \nabla_\theta \mathcal{L}(\omega_n, \theta^{(n)})\right\rangle\right) + \mathbb{E}\left(\left\|\nabla_\theta \mathcal{L}(\omega_n, \theta^{(n)})\right\|^2\right) \\ &\quad \leq d_n - \gamma_n\,\mu\,d_n + \gamma_n^2\,B = (1 - \gamma_n\,\mu)\,d_n + \gamma_n^2\,B. \end{aligned}$$

Assertion 3: Assume that (γ_n) is a constant sequence with value $\gamma > 0$. Then the inequality (1.32) is equal to

$$\begin{aligned} & & d_{n+1} &\leq (1 - \gamma\,\mu) d_n + \gamma^2\,B \\ \iff & & d_{n+1} - \gamma\,\frac{B}{\mu} &= (1 - \gamma\,\mu)\,d_n + \gamma^2\,\frac{\mu}{\mu}B - \gamma\,\frac{B}{\mu} \\ \iff & & d_{n+1} - \gamma\,\frac{B}{\mu} &= (1 - \gamma\,\mu)\left(d_n - \gamma\,\frac{B}{\mu}\right). \end{aligned} \tag{1.36}$$

Furthermore, one may quickly realise that by applying inequality (1.36) twice holds

$$d_{n+2} - \gamma\,\frac{B}{\mu} \leq (1 - \gamma\,\mu)\,(1 - \gamma\,\mu)\left(d_{n+1} - \gamma\,\frac{B}{\mu}\right) \leq (1 - \gamma\,\mu)^2\left(d_n - \gamma\,\frac{B}{\mu}\right).$$

Therefore, by applying inequality (1.36) k times we receive

$$d_{n+k} - \gamma \frac{B}{\mu} \leq (1 - \gamma \mu)^k \left(d_n - \gamma \frac{B}{\mu} \right).$$

Since $d_n > 0$ for all $n \geq 0$ by construction, taking $k \to \infty$ we obtain

$$\limsup_{k \to \infty} \left(d_k - \gamma \frac{B}{\mu} \right) = 0,$$

or equally

$$\limsup_{k \to \infty} (d_k) = \gamma \frac{B}{\mu}.$$

Lastly, since we are free to choose the constant γ, we define it as $\gamma \coloneqq \epsilon \mu / B$. This gives us

$$\limsup_{k \to \infty} (d_k) = \limsup_{k \to \infty} \left(\left\| \theta^{(k)} - \theta^* \right\|^2 \right) \leq \epsilon .$$

Assertion 4: Assume that the sequence (γ_n) is non-constant and assume that $\epsilon > 0$. Furthermore, with the same idea as in inequality (1.36) we receive

$$d_{n+1} - \gamma_n \frac{B}{\mu} \leq (1 - \gamma_n \mu) \left(d_n - \gamma_n \frac{B}{\mu} \right).$$

If we now define $\epsilon_n \coloneqq \gamma_n B / \mu$, this gives us

$$d_{n+1} - \epsilon_n \leq (1 - \gamma_n \mu) (d_n - \epsilon_n) . \tag{1.37}$$

Therefore, applying inequality (1.37) twice gives us

$$d_{n+2} - \epsilon_n \leq (1 - \gamma_n \mu) (d_{n+1} - \epsilon_n) \leq \left(1 - \gamma_{n+1} \mu\right) (1 - \gamma_n \mu) (d_n - \epsilon_n) .$$

Applying inequality (1.37) k times iteratively leads to

$$d_{n+k} - \epsilon_n \leq \prod_{l=n}^{n+k-1} (1 - \gamma_l \mu) (d_n - \epsilon_n) . \tag{1.38}$$

Now, we consider the product on the right-hand side to fulfil

$$0 \leq \prod_{l=n}^{n+k-1} (1 - \gamma_l \mu) = \exp \left(\sum_{l=n}^{n+k-1} \log (1 - \gamma_l \mu) \right).$$

since for $x \in (0, 1)$ holds $\log(1 - x) \leq -x$, we have

$$\exp \left(\sum_{l=n}^{n+k-1} \log (1 - \gamma_l \mu) \right) \leq \exp \left(\sum_{l=n}^{n+k-1} (-\gamma_l \mu) \right) \overset{k \to \infty}{\longrightarrow} \exp(-\infty) = 0,$$

due to the assumption, that $\sum_{n\geq 0}\gamma_n = \infty$ and therefore, $\sum_{n\geq l}\gamma_n = \infty$ for $l \in \mathbb{N}$ as well. Plugging this result into inequality (1.38), we receive

$$\lim_{k\to\infty} d_{n+k} - \epsilon_n \, 0.$$

Lastly, we note that ϵ_n was defined as $\epsilon_n = \gamma_n \, B/\,\mu$, where B, m are constants. Thus, since γ_n is assumed to converge towards 0, the same holds for ϵ_n. Therefore, it holds that

$$\lim_{n\to\infty} d_n = 0,$$

which is exactly Assertion 4. □

Lastly, we want to consider another powerful optimization algorithm that adapts learning rates based on past gradient magnitudes and momenta - it is called „Adaptive Moment Estimation (Adam)".
However, in order to formulate the algorithm formally we need to introduce stochastic moments first. We do so analogously to [14, Chapter 5]

Definition 1.3.8. Let $(\Omega, \mathcal{A}, \mathrm{P})$ be a probability space, $X : \Omega \to \mathbb{R}$ be a random variable and $k \in \mathbb{N}$. Then we define the **k-th moment** of X as

$$m_k \coloneqq \mathbb{E}\left[X^k\right] = \int_\Omega X d\,\mathrm{P}\,.$$

Moreover, we define the **k-th central moment** of X as

$$\mu_k \coloneqq \mathbb{E}\left[(X-\mu)^k\right] = \int_\Omega (X-\mu)^k \, d\,\mathrm{P},$$

where we denote $\mu \coloneqq \mathbb{E}[X] = m_1$.
Lastly, we say that the k-th moment exists, if $|\,\mu_k\,| < \infty$ holds.

Usually, the first moment is referred to as mean and the second central moment as variance of a random variable X. Another important quantity that considers moments of random variables is the moment-generating function. As the name already suggests, this function will allow us to explicitly compute the moments.

Definition 1.3.9. Let $(\Omega, \mathcal{A}, \mathrm{P})$ be a probability space, $X : \Omega \to \mathbb{R}$ be a random variable and let $D \coloneqq \{s \in \mathbb{R} : \mathbb{E}(\exp(sX)) < \infty\}$. Then we call the function

$$\begin{aligned} M : D &\to \mathbb{R}, \\ s &\mapsto \mathbb{E}(\exp(sX)) = \int_\Omega \exp(sX) d\,\mathrm{P}(x), \end{aligned}$$

the **moment-generating function of** X.

At this point, we want to mention a theorem which considers the existence of moments. However, we only take a quick glance at the result and will not prove it. Instead we refer to the original literature [12, Theorem 4.21]

Theorem 1.3.10. *Let X be a random variable with moment-generating function $M : d \to \mathbb{R}$. If there exists an $a > 0$ such that $(-a, a) \subset D$, then all moments of X exist and it holds*

$$M(s) = \sum_{n=0}^{\infty} \frac{s^n}{n!} \mathbb{E}(X^n), \qquad s \in (-a, a).$$

Particularly, M is infinitely differentiable on $(-a, a)$ with n-th derivative

$$M^{(n)}(0) = \mathbb{E}(X^n).$$

With the help of the Definition 1.3.8 of stochastic moments, we can formulate the proposed optimization algorithm. For further reading we kindly refer to [6] or [3, Chapter 8].

Algorithm 1 Adam optimizer

Let D be an arbitrary dataset and $g_t \coloneqq \nabla_\theta \mathcal{R}_{\mathcal{L},D}(f_t(\theta))$ denote the gradient, i.e. the vector of partial derivatives of $\mathcal{R}_{\mathcal{L},D}(f_t(\theta))$ with regard to θ evaluated at time step t. Furthermore, let $g_t^2 \coloneqq g_t \odot g_t$ denote the element-wise square of g_t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Prediction function with parameters θ
Require: θ_0: Initial parameter vector

$m_0, v_0 \leftarrow 0$ (Initialize 1^{st} and 2^{nd} moment vector)
$t \leftarrow 0$ (Initialize time step)
while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta \mathcal{R}_{\mathcal{L},D}(f_t(\theta_{t-1}))$ ▷ Get gradients w.r.t. prediction function at time step t,
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ ▷ Update biased first moment estimate,
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ ▷ Update biased second moment estimate,
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ ▷ Compute bias-corrected first moment estimate,
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ ▷ Compute bias-corrected second moment estimate,
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ ▷ Update parameters with bias- corrected moments,
end while
return θ_t ▷ Return Resulting parameters.

However, Algorithm 1 was later proven to be not converging in certain settings, see e.g. [15]. The authors proposed another approach to the optimization problem, where they first introduced a general formulation of the algorithm, see Algorithm 2. Afterwards, they proposed an alternative approach called the AMSGrad algorithm. In contrast to Adam, AMSGrad uses the maximum of past squared gradients rather than the exponential average to update the parameters. This way the authors were able to fix the issues of the original algorithm Adam. But in order to introduce AMSGrad formally, we need to define some quantities first.

Definition 1.3.11. Let $\mathcal{F} \subset \mathbb{R}^d$ be a set of points. We say that $\mathcal{F}$ has **bounded diameter** $D_\infty < \infty$ if for all $x, y \in \mathcal{F}$ holds

$$\|x - y\|_\infty < D_\infty.$$

Definition 1.3.12. Let $y \in \mathbb{R}^d$, $\mathcal{F} \subset \mathbb{R}^d$ with bounded diameter D_∞ and $X : \mathbb{R}^d \to \mathbb{R}^d$ be a self-adjoint operator. Then we define the **X-projection of y onto $\mathcal{F}$** as

$$\Pi_{\mathcal{F},X}(y) = \min_{x \in \mathcal{F}} \left\| X^{1/2}(x - y) \right\|.$$

If the operator X is the identity $\mathbb{1}$, we reduce the notation to $\Pi_{\mathcal{F}} \coloneqq \Pi_{\mathcal{F},\mathbb{1}}$.

Now, we introduce a short technical assumption concerning the recently introduced projection.

Lemma 1.3.13. *Let $\mathcal{F} \subset \mathbb{R}^d$ be a set of points and $\Pi_{\mathcal{F},X}$ be an X-projection with operator X. Then the following assertion holds for all $x^* \in \mathcal{F}$.*

$$\Pi_{\mathcal{F},X}\left(x^*\right) = x^*.$$

Proof. Let's first consider the definition of the projection $\Pi_{\mathcal{F}}$.

$$\Pi_{\mathcal{F},X}\left(x^*\right) = \min_{x \in \mathcal{F}} \left\| X^{1/2}(x - x^*) \right\|.$$

Hence, for any point x' that minimizes the right hand side holds

$$x' = \arg\min_{x \in \mathcal{F}} \left\| X^{1/2}(x - x^*) \right\|,$$

what can be considered equally as

$$\iff \qquad x' = \arg\min_{x \in \mathcal{F}} \left(x - x^*\right).$$

But since we assumed that $x^* \in \mathcal{F}$, it follows that $x' = x^*$ and the assertion holds. □

Another quantity we need to introduce is the so called regret of an algorithm. It essentially quantifies how much an algorithm would have performed better if it had known the best action in advance. In other words, the regret quantifies the cost of not making the optimal decisions at each step. This is a common approach in on-line learning settings.

Definition 1.3.14. Let $T \in \mathbb{N}$, D be an arbitrary dataset, $\Theta \subset \mathbb{R}^d$ be a parameter space and $\mathcal{L}$ be a loss function. Furthermore, let $f_t(\theta_t)$ be a prediction function with parameters $\theta_t \in \Theta$ at time step t. Then we define the **regret** as the function

$$R_T = \sum_{t=1}^{T} \mathcal{R}_{\mathcal{L},D}\left(f_t(\theta_t)\right) - \min_{\theta \in \Theta} \sum_{t=1}^{T} \mathcal{R}_{\mathcal{L},D}\left(f_t(\theta)\right).$$

Algorithm 2 Generic Adaptive Method Setup

Let g_t be as in Algorithm 1, D be an arbitrary dataset and let $T \in \mathbb{N}$. Furthermore, let Θ be a parameter space.

Require: $\{\alpha_t\}_{t=1}^T$: Stepsizes
Require: $\{\phi_t, \psi_t\}_{t=1}^T$: Sequence of functions
Require: $f(\theta)$: Prediction function with parameters θ
Require: $\theta_1 \in \Theta$: Initial parameter vector

1: **for** $t = 1, \ldots, T$ **do**
2: $\quad g_t \leftarrow \nabla_\theta \mathcal{R}_{\mathcal{L},D}(f_t(\theta_t))$ ▷ Get gradients w.r.t. prediction function at time step t,
3: $\quad m_t \leftarrow \phi_t(g_1, \ldots, g_t)$ ▷ Update biased first moment estimate,
4: $\quad V_t \leftarrow \psi_t(g_1, \ldots, g_t)$ ▷ Update biased second moment estimate,
5: $\quad \widehat{\theta}_{t+1} \leftarrow \theta_t - \alpha_t m_t / \sqrt{V_t}$ ▷ Compute biased updated parameters,
6: $\quad \theta_{t+1} \leftarrow \Pi_{\Theta, \sqrt{V_t}}\left(\widehat{\theta}_{t+1}\right)$ ▷ Unbias updated parameters,
7: **end for**
8: **return** θ_t ▷ Return resulting parameters.

We realize that upon defining ϕ_t and ψ_t in Algorithm 2 in a suitable way, we can obtain various familiar algorithms. We want to consider them in the following example.

Example 1.3.15. Let the same assumptions as in Algorithm 2 hold. Then the following quantities define iterative optimization algorithms.

1. Let $(\phi_t)_t$ and $(\psi_t)_t$ be defined as

$$\phi_t(g_1, \ldots, g_t) = g_t,$$
$$\psi_t(g_1, \ldots, g_t) = \mathbb{1}.$$

Then the resulting update rule looks like

$$\theta_{t+1} \leftarrow \theta_t - \alpha_t g_t,$$

which is exactly the SGD algorithm as proposed in Theorem 1.3.7.

2. Let ϕ_t and ψ_t be defined as

$$\phi_t(g_1, \ldots, g_t) = (1 - \beta_1) \sum_{i=1}^t \beta_1^{t-i} g_i,$$
$$\psi_t(g_1, \ldots, g_t) = (1 - \beta_2) \operatorname{diag}\left(\sum_{i=1}^t \beta_2^{t-i} g_t^2\right).$$

Then the resulting update rule looks like

$$m_t \leftarrow (1 - \beta_1) \sum_{i=1}^t \beta_1^{t-i} g_i,$$
$$v_t \leftarrow (1 - \beta_2) \operatorname{diag}\left(\sum_{i=1}^t \beta_2^{t-i} g_t^2\right),$$

which is the same as in Algorithm 1 (without the bias-correction, but the argument still holds, see [15]). Furthermore, the X-projection is obsolete for the choice of $X = \mathbb{1}$ and $\mathcal{F} = \mathbb{R}^d$. This follows directly from Lemma 1.3.13.

With the help of the general algorithm proposed in Algorithm 2, we can introduce an algorithm, which can be proven to converge in a non-convex setting. Since neural networks ultimately are non-convex functions, this is exactly what we are interested in.

Algorithm 3 AMSGrad Optimizer

Let the same assumptions as in Algorithm 2 hold.

Require: $\{\alpha_t\}_{t=1}^T$: Stepsizes
Require: $\{\beta_{1t}\}_{t=1}^T, \beta_2$, with $\beta_{1t}, \beta_2 \in [0, 1)$: Decay rates for the moment estimates
Require: $f(\theta)$: Prediction function with parameters θ
Require: $\theta_1 \in \Theta$: Initial parameter vector

$m_0, v_0 \leftarrow 0$ (Initialise 1^{st} and 2^{nd} moment vector)
for $t = 1, \ldots, T$ **do**
  $g_t \leftarrow \nabla_\theta \mathcal{R}_{\mathcal{L},D}(f_t(\theta_t))$ ▷ Get gradients w.r.t. prediction function at time step t,
  $m_t \leftarrow \beta_{1t} m_{t-1} + (1 - \beta_{1t}) g_t$ ▷ Update biased first moment estimate,
  $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$ ▷ Update biased second moment estimate,
  $\widehat{v}_t \leftarrow \max\{\widehat{v}_{t-1}, v_t\}$ ▷ Compute bias-corrected first moment estimate,
  $\widehat{V}_t \leftarrow \operatorname{diag}(\widehat{v}_t)$ ▷ Compute bias-corrected second moment estimate,
  $\theta_{t+1} \leftarrow \Pi_{\Theta, \sqrt{\widehat{V}_t}} \left(\theta_t - \alpha_t m_t / \sqrt{\widehat{v}_t}\right)$ ▷ Update parameters,
end for
return θ_{t+1} ▷ Return resulting parameters.

The AMSGrad optimizer, defined in Algorithm 3 does indeed converge, as proven in [15, Theorem 4]. We take a quick look at the theorem and its proof. However, for a deeper understanding of the alternative algorithms, please refer to [15], since this would go beyond the scope of this thesis' topic.
First, we need an auxiliary lemma ans cite it from [15, Lemma 4], but since their proof does not align with the proof of the original paper [11, Lemma 3] we will adjust the proof to the original one.

Lemma 1.3.16. *Let $\mathcal{S}_+^{\mathrm{d}}$ denote the set of all positive definite $d \times d$-matrices and let $Q \in \mathcal{S}_+^{\mathrm{d}}$. Furthermore, let $\mathcal{F} \subset \mathbb{R}^d$ be a convex set.*
Suppose $z_1, z_2 \in \mathbb{R}^d$ and $u_1 = \min_{x \in \mathcal{F}} \|Q^{1/2}(x - z_1)\|$ as well as $u_2 = \min_{x \in \mathcal{F}} \|Q^{1/2}(x - z_2)\|$. Then the following inequality holds

$$\left\|Q^{1/2}(u_1 - u_2)\right\| \leq \left\|Q^{1/2}(z_1 - z_2)\right\|.$$

Proof. We begin with defining

$$B(u, z) \coloneqq \frac{1}{2}\left\|Q^{1/2}(u - z)\right\|^2 = \frac{1}{2}\langle u - z, Q(u - z)\rangle. \tag{1.39}$$

Hence, we can write

$$u_1 = \arg\min_{x \in \mathcal{F}} B(x, z_1). \tag{1.40}$$

If we now consider the gradient of (1.39), we receive

$$\nabla_x B(x, z_1) = \nabla_x \frac{1}{2} \langle x - z_1, Q(x - z_1) \rangle = Q(x - z_1). \tag{1.41}$$

Therefore, it holds that

$$\langle Q(u_1 - z_1), u_2 - u_1 \rangle \geq 0, \tag{1.42}$$

otherwise for $\delta > 0$ sufficiently small it would mean that $u_1 + \delta(u_2 - u_1) \in \mathcal{F}$, due to the convexity of $\mathcal{F}$, and therefore would be closer to z_1 than u_1. This contradicts the assumption, that u_1 is the best approximation of z_1 in $\mathcal{F}$, i.e. fulfils equation (1.40).
With the exact same argument it holds that

$$\langle Q(u_2 - z_2), u_1 - u_2 \rangle \geq 0,$$

If we combine the two inequalities (1.41) and (1.42), we receive

$$\begin{aligned} & \langle Q(u_1 - z_1), u_2 - u_1 \rangle + \langle Q(u_2 - z_2), u_1 - u_2 \rangle \geq 0, \\ \iff \quad & \langle Q(u_1 - z_1), u_2 - u_1 \rangle - \langle Q(u_2 - z_2), u_2 - u_1 \rangle \geq 0. \end{aligned}$$

These inequalities are due to $Q \in \mathcal{S}_+^{\mathrm{d}}$ equivalent to

$$\begin{aligned} & \langle u_1 - z_1, Q(u_2 - u_1) \rangle - \langle u_2 - z_2, Q(u_2 - u_1) \rangle \geq 0, \\ \iff \quad & \langle z_2 - z_1, Q(u_2 - u_1) \rangle - \langle u_2 - u_1, Q(u_2 - u_1) \rangle \geq 0, \\ \iff \quad & \langle z_2 - z_1, Q(u_2 - u_1) \rangle \geq \langle u_2 - u_1, Q(u_2 - u_1) \rangle. \end{aligned}$$

For readability reasons we now define $\widehat{u} \coloneqq (u_2 - u_1)$ and $\widehat{z} \coloneqq (z_2 - z_1)$. With these notations we receive

$$\langle \widehat{z}, Q\widehat{u} \rangle \geq \langle \widehat{u} Q \widehat{u} \rangle.$$

Now, lets consider the inequality

$$\langle \widehat{z} - \widehat{u}, Q(\widehat{z} - \widehat{u}) \rangle \geq 0.$$

Considering the fact that $Q \in \mathcal{S}_+^{\mathrm{d}}$ it holds that

$$\begin{aligned} & \langle \widehat{z} - \widehat{u}, Q(\widehat{z} - \widehat{u}) \rangle \geq 0, \\ \iff \quad & \langle \widehat{z}, Q\widehat{z} \rangle - 2 \langle \widehat{u}, Q\widehat{z} \rangle + \langle \widehat{u}, Q\widehat{u} \rangle \geq 0. \end{aligned}$$

Thus,

$$\langle \widehat{z}, Q\widehat{z} \rangle \geq 2 \langle \widehat{u}, Q\widehat{z} \rangle - \langle \widehat{u}, Q\widehat{u} \rangle \geq 2 \langle \widehat{u}, Q\widehat{u} \rangle - \langle \widehat{u}, Q\widehat{u} \rangle = \langle \widehat{u}, Q\widehat{u} \rangle,$$

where the second inequality is due to the fact that $u_1 = \min_{x \in \mathcal{F}} \|Q^{1/2}(x - z_1)\|$ as well as $u_2 = \min_{x \in \mathcal{F}} \|Q^{1/2}(x - z_2)\|$.
Lastly, considering the definitions of $\widehat{u}$ and $\widehat{w}$ we achieved

$$\left\| Q^{1/2}(u_1 - u_2) \right\|^2 = \langle \widehat{u}, Q\widehat{u} \rangle \leq \langle \widehat{z}, Q\widehat{z} \rangle = \left\| Q^{1/2}(z_1 - z_2) \right\|^2.$$

Taking the square root of both sides leads to the assertion. □

Theorem 1.3.17. *Let $(\theta_t)_{t\in\mathbb{N}}$ and $(v_t)_{t\in\mathbb{N}}$ be sequences as in Algorithm 3, D be an arbitrary dataset of length $d \in \mathbb{N}$ and $\alpha_t = \alpha/\sqrt{t}$ with $\alpha > 0$ and $\beta_1 = \beta_{11}$, $\beta_{1t} \leq \beta_1$ for all $t \in \{1,\dots,T\}$ and $\gamma = \beta_1/\sqrt{\beta_2} < 1$. Assume that Θ has bounded diameter $D_\infty < \infty$ and $\max_{\theta\in\Theta} \|\nabla_\theta \mathcal{R}_{\mathcal{L},D}(f_t(\theta_t))\|_\infty = G_\infty < \infty$ for all $t \in \{1,\dots,T\}$. Furthermore, let $\mathcal{L}$ be a convex loss function. Then for θ_t the following bound for the regret holds*

$$\sum_{t=1}^{T} \mathcal{R}_{\mathcal{L},D}(f_t(\theta_t)) - \mathcal{R}_{\mathcal{L},D}(f_t(\theta^*))$$
$$\leq \frac{D_\infty^2\sqrt{T}}{2\alpha(1-\beta_1)} \sum_{i=1}^{d} \widehat{v}_{T,i}^{1/2} + \frac{D_\infty^2}{(1-\beta_1)^2} \sum_{t=1}^{T}\sum_{i=1}^{d} \frac{\beta_{1t}\widehat{v}_{t,i}^{1/2}}{\alpha_t} + \frac{\alpha\sqrt{1+\log T}}{(1-\beta_1)^2(1-\gamma)\sqrt{(1-\beta_2)}} \sum_{i=1}^{d} \|g_{1:T,i}\|_2,$$

where we denote for readability reasons $g_{1:t} \coloneqq (g_1,\dots,g_t)$ and with $g_{j,i}$ and with $v_{j,i}$ we denote the i-th component of g_j and v_j, respectively.

Proof. First, we observe with the Definition 1.3.12 of the projection

$$\theta_{t+1} = \Pi_{\Theta,\sqrt{\widehat{V}_t}}\left(\theta_t - \alpha_t \widehat{V}_t^{-1/2} m_t\right) = \min_{\theta\in\Theta} \left\|\widehat{V}_t^{1/4}\left(\theta - \left(\theta_t - \alpha_t \widehat{V}_t^{-1/2} m_t\right)\right)\right\|.$$

Furthermore, with Lemma 1.3.13 it follows, that $\Pi_{\Theta,\sqrt{\widehat{V}_t}}(\theta^*) = \theta^*$ for all $\theta^* \in \Theta$.

Now, using Lemma 1.3.16 with $u_1 = \theta_{t+1}$, $u_2 = \theta^*$ and $Q = \widehat{V}_t^{1/4}$ gives us

$$\begin{aligned}
\left\|\widehat{V}_t^{1/4}(\theta_{t+1}-\theta^*)\right\|^2 &\leq \left\|\widehat{V}_t^{1/4}\left(\theta_t - \alpha_t\widehat{V}_t^{-1/2}m_t - \theta^*\right)\right\|^2, \\
&= \left\|\widehat{V}_t^{1/4}(\theta_t-\theta^*)\right\|^2 + \alpha_t^2\left\|\widehat{V}_t^{-1/4}m_t\right\|^2 - 2\alpha_t\left\langle m_t, \theta_t-\theta^*\right\rangle, \\
&= \left\|\widehat{V}_t^{1/4}(\theta_t-\theta^*)\right\|^2 + \alpha_t^2\left\|\widehat{V}_t^{-1/4}m_t\right\|^2 - 2\alpha_t\left\langle \beta_{1t}m_{t-1} + (1-\beta_{1t})\,g_t, \theta_t-\theta^*\right\rangle.
\end{aligned}$$

If we now rearrange the last inequality, we receive

$$\begin{aligned}
&\left\|\widehat{V}_t^{1/4}(\theta_{t+1}-\theta^*)\right\|^2 - \left\|\widehat{V}_t^{1/4}(\theta_t-\theta^*)\right\|^2 - \alpha_t^2\left\|\widehat{V}_t^{-1/4}m_t\right\|^2 \\
&\quad\leq -2\alpha_t\left\langle \beta_{1t}m_{t-1} + (1-\beta_{1t})\,g_t, \theta_t-\theta^*\right\rangle, \\
\iff\quad &\left\|\widehat{V}_t^{1/4}(\theta_{t+1}-\theta^*)\right\|^2 - \left\|\widehat{V}_t^{1/4}(\theta_t-\theta^*)\right\|^2 - \alpha_t^2\left\|\widehat{V}_t^{-1/4}m_t\right\|^2 \\
&\quad\leq -2\alpha_t\beta_{1t}\left\langle m_{t-1}, \theta_t-\theta^*\right\rangle - 2\alpha_t(1-\beta_{1t})\left\langle g_t, \theta_t-\theta^*\right\rangle, \\
\iff\quad &\left\|\widehat{V}_t^{1/4}(\theta_{t+1}-\theta^*)\right\|^2 - \left\|\widehat{V}_t^{1/4}(\theta_t-\theta^*)\right\|^2 - \alpha_t^2\left\|\widehat{V}_t^{-1/4}m_t\right\|^2 \\
&\quad + 2\alpha_t\beta_{1t}\left\langle m_{t-1}, \theta_t-\theta^*\right\rangle \leq -2\alpha_t(1-\beta_{1t})\left\langle g_t, \theta_t-\theta^*\right\rangle, \\
\iff\quad &-\frac{1}{2\alpha_t(1-\beta_{1t})}\left[\left\|\widehat{V}_t^{1/4}(\theta_{t+1}-\theta^*)\right\|^2 - \left\|\widehat{V}_t^{1/4}(\theta_t-\theta^*)\right\|^2 - \alpha_t^2\left\|\widehat{V}_t^{-1/4}m_t\right\|^2\right. \\
&\quad\left. + 2\alpha_t\beta_{1t}\left\langle m_{t-1}, \theta_t-\theta^*\right\rangle\right] \geq \left\langle g_t, \theta_t-\theta^*\right\rangle, \\
\iff\quad &\frac{1}{2\alpha_t(1-\beta_{1t})}\left[\left\|\widehat{V}_t^{1/4}(\theta_t-\theta^*)\right\|^2 - \left\|\widehat{V}_t^{1/4}(\theta_{t+1}-\theta^*)\right\|^2\right] + \frac{\alpha_t}{2(1-\beta_{1t})}\left\|\widehat{V}_t^{-1/4}m_t\right\|^2 \\
&\quad - \frac{\beta_{1t}}{1-\beta_{1t}}\left\langle m_{t-1}, \theta_t-\theta^*\right\rangle \geq \left\langle g_t, \theta_t-\theta^*\right\rangle,
\end{aligned}$$

And if we now apply the Cauchy-Schwarz inequality and the Young's inequality, this leads to

$$\begin{aligned}\iff \quad \langle g_t, \theta_t - \theta^* \rangle \leq \frac{1}{2\alpha_t (1-\beta_{1t})} \left[\left\| \widehat{V}_t^{1/4} (\theta_t - \theta^*) \right\|^2 - \left\| \widehat{V}_t^{1/4} (\theta_{t+1} - \theta^*) \right\|^2 \right] \\ + \frac{\alpha_t}{2(1-\beta_{1t})} \left\| \widehat{V}_t^{-1/4} m_t \right\|^2 + \frac{\beta_{1t}}{2(1-\beta_{1t})} \alpha_t \left\| \widehat{V}_t^{-1/4} m_{t-1} \right\|^2 \\ + \frac{\beta_{1t}}{2\alpha_t (1-\beta_{1t})} \left\| \widehat{V}_t^{1/4} (\theta_t - \theta^*) \right\|^2 . \end{aligned} \tag{1.43}$$

The next step is a common approach in bounding the regret, where we will use the convexity of the risk function $\mathcal{R}_{\mathcal{L},D}(f_t(\theta_t))$ in each step, which follows from the fact that $\mathcal{L}$ is a convex loss function and Lemma 1.1.19. With these facts holds

$$\sum_{t=1}^{T} \mathcal{R}_{\mathcal{L},D}(f_t(\theta_t)) - \mathcal{R}_{\mathcal{L},D}(f_t(\theta^*)) \leq \sum_{t=1}^{T} \langle g_t, \theta_t - \theta^* \rangle ,$$

which we can do, since we consider the gradient of the risk $\mathcal{R}_{\mathcal{L},D}(f_t)$ function with regard to the parameters θ_t. The inequality ultimately follows from the definition of the weak subgradient, see e.g. [4, Remark 16.11.].
This we can further bound by the previously considered inequality (1.43).

$$\begin{aligned}\sum_{t=1}^{T} \langle g_t, \theta_t - \theta^* \rangle \leq \sum_{t=1}^{T} \Bigg[\frac{1}{2\alpha_t (1-\beta_{1t})} \left[\left\| \widehat{V}_t^{1/4} (\theta_t - \theta^*) \right\|^2 - \left\| \widehat{V}_t^{1/4} (\theta_{t+1} - \theta^*) \right\|^2 \right] \\ + \frac{\alpha_t}{2(1-\beta_{1t})} \left\| \widehat{V}_t^{-1/4} m_t \right\|^2 + \frac{\beta_{1t}}{2(1-\beta_{1t})} \alpha_t \left\| \widehat{V}_t^{-1/4} m_{t-1} \right\|^2 \\ + \frac{\beta_{1t}}{2\alpha_t (1-\beta_{1t})} \left\| \widehat{V}_t^{1/4} (\theta_t - \theta^*) \right\|^2 \Bigg].\end{aligned}$$

If we now consider the following inequality, where we use the fact that $\beta_1 \geq \beta_{1t}$ for all $t = 1, \ldots, T$

$$\begin{aligned}&\frac{\alpha_t}{2(1-\beta_{1t})} \left\| \widehat{V}_t^{-1/4} m_t \right\|^2 + \frac{\beta_{1t}}{2(1-\beta_{1t})} \alpha_t \left\| \widehat{V}_t^{-1/4} m_{t-1} \right\|^2 \\ &= \frac{\alpha_t}{2(1-\beta_{1t})} \left\| \widehat{V}_t^{-1/4} m_t \right\|^2 + \frac{\beta_{1t}}{2(1-\beta_{1t})} \alpha_t \left\| \widehat{V}_t^{-1/4} \left(\frac{m_t - (1-\beta_{1t}) g_t}{\beta_{1t}} \right) \right\|^2 , \\ &\leq \frac{\alpha_t}{2(1-\beta_{1t})} \left\| \widehat{V}_t^{-1/4} m_t \right\|^2 + \frac{1}{2(1-\beta_{1t})} \alpha_t \left\| \widehat{V}_t^{-1/4} m_t \right\|^2 , \\ &\leq \frac{\alpha_t}{1-\beta_1} \left\| \widehat{V}_t^{-1/4} m_t \right\|^2 ,\end{aligned}$$

then this leads ultimately to

$$\begin{aligned}\sum_{t=1}^{T} \mathcal{R}_{\mathcal{L},D}(f_t(\theta_t)) - \mathcal{R}_{\mathcal{L},D}(f_t(\theta^*)) \leq \sum_{t=1}^{T} \Bigg[\frac{1}{2\alpha_t (1-\beta_{1t})} \left[\left\| \widehat{V}_t^{1/4} (\theta_t - \theta^*) \right\|^2 - \left\| \widehat{V}_t^{1/4} (\theta_{t+1} - \theta^*) \right\|^2 \right] \\ + \frac{\alpha_t}{1-\beta_1} \left\| \widehat{V}_t^{-1/4} m_t \right\|^2 + \frac{\beta_{1t}}{2\alpha_t (1-\beta_{1t})} \left\| \widehat{V}_t^{1/4} (\theta_t - \theta^*) \right\|^2 \Bigg].\end{aligned} \tag{1.44}$$

We now proceed by bounding the second term of the right-hand side, separately.
In order to do so, we first use the definition of $\widehat{v}_T$, which is the maximum of all v_t until the current time step T. This gives us

$$\begin{aligned}
\sum_{t=1}^{T} \alpha_t \left\| \widehat{V}_t^{-1/4} m_t \right\|^2 &= \sum_{t=1}^{T-1} \alpha_t \left\| \widehat{V}_t^{-1/4} m_t \right\|^2 + \alpha_T \sum_{i=1}^{d} \frac{m_{T,i}^2}{\sqrt{\widehat{v}_{T,i}}}, \\
&\leq \sum_{t=1}^{T-1} \alpha_t \left\| \widehat{V}_t^{-1/4} m_t \right\|^2 + \alpha_T \sum_{i=1}^{d} \frac{m_{T,i}^2}{\sqrt{v_{T,i}}}, \\
&\leq \sum_{t=1}^{T-1} \alpha_t \left\| \widehat{V}_t^{-1/4} m_t \right\|^2 + \alpha \sum_{i=1}^{d} \frac{(\sum_{t=1}^{T} (1-\beta_{1t}) \prod_{k=1}^{T-t} \beta_{1(T-k+1)} g_{t,i})^2}{\sqrt{T((1-\beta_2)\sum_{t=1}^{T} \beta_2^{T-t} g_{t,i}^2)}},
\end{aligned}$$

where the last inequality follows from the update rules of m_t and v_t in Algorithm 3, respectively.
If we now apply the Cauchy-Schwarz inequality, we receive

$$\begin{aligned}
\sum_{t=1}^{T} \alpha_t \left\| \widehat{V}_t^{-1/4} m_t \right\|^2 \leq \sum_{t=1}^{T-1} \alpha_t \left\| \widehat{V}_t^{-1/4} m_t \right\|^2 \\
+ \alpha \sum_{i=1}^{d} \frac{(\sum_{t=1}^{T} \prod_{k=1}^{T-t} \beta_{1(T-k+1)})(\sum_{t=1}^{T} \prod_{k=1}^{T-t} \beta_{1(T-k+1)} g_{t,i}^2)}{\sqrt{T((1-\beta_2)\sum_{t=1}^{T} \beta_2^{T-t} g_{t,i}^2)}}.
\end{aligned}$$

Now, considering the fact that $\beta_{1t} \leq \beta_1$ for all $t = 1, \ldots, T$ again, gives us

$$\sum_{t=1}^{T} \alpha_t \left\| \widehat{V}_t^{-1/4} m_t \right\|^2 \leq \sum_{t=1}^{T-1} \alpha_t \left\| \widehat{V}_t^{-1/4} m_t \right\|^2 + \alpha \sum_{i=1}^{d} \frac{(\sum_{t=1}^{T} \beta_1^{T-t})(\sum_{t=1}^{T} \beta_1^{T-t} g_{t,i}^2)}{\sqrt{T((1-\beta_2)\sum_{t=1}^{T} \beta_2^{T-t} g_{t,i}^2)}}.$$

The next two steps are considering the inequality $\sum_{t=1}^{T} \beta_1^{T-t} \leq 1/(1-\beta_1)$ and afterwards using the linearity of the square root. This gives us

$$\begin{aligned}
\sum_{t=1}^{T} \alpha_t \left\| \widehat{V}_t^{-1/4} m_t \right\|^2 &\leq \sum_{t=1}^{T-1} \alpha_t \left\| \widehat{V}_t^{-1/4} m_t \right\|^2 + \frac{\alpha}{1-\beta_1} \sum_{i=1}^{d} \frac{\sum_{t=1}^{T} \beta_1^{T-t} g_{t,i}^2}{\sqrt{T((1-\beta_2)\sum_{t=1}^{T} \beta_2^{T-t} g_{t,i}^2)}}, \\
&\leq \sum_{t=1}^{T-1} \alpha_t \left\| \widehat{V}_t^{-1/4} m_t \right\|^2 + \frac{\alpha}{(1-\beta_1)\sqrt{T(1-\beta_2)}} \sum_{i=1}^{d} \sum_{t=1}^{T} \frac{\beta_1^{T-t} g_{t,i}^2}{\sqrt{\beta_2^{T-t} g_{t,i}^2)}}, \\
&\leq \sum_{t=1}^{T-1} \alpha_t \left\| \widehat{V}_t^{-1/4} m_t \right\|^2 + \frac{\alpha}{(1-\beta_1)\sqrt{T(1-\beta_2)}} \sum_{i=1}^{d} \sum_{t=1}^{T} \gamma^{T-t} \left| g_{t,i} \right|,
\end{aligned}$$

where we used the definition of $\gamma = \beta_1/\sqrt{\beta_2}$ in the last step.
If we consider similar upper bounds for all time steps $t = 1, \ldots, T-1$ as we did for the last

time step, we receive

$$\begin{aligned}\sum_{t=1}^{T}\alpha_t\left\|\widehat{V}_t^{-1/4}m_t\right\|^2 &\leq \sum_{t=1}^{T}\frac{\alpha}{(1-\beta_1)\sqrt{t(1-\beta_2)}}\sum_{i=1}^{d}\sum_{j=1}^{t}\gamma^{t-j}\left|g_{j,i}\right|,\\ &= \frac{\alpha}{(1-\beta_1)\sqrt{(1-\beta_2)}}\sum_{t=1}^{T}\sum_{i=1}^{d}\frac{1}{\sqrt{t}}\sum_{j=1}^{t}\gamma^{t-j}\left|g_{j,i}\right|,\\ &= \frac{\alpha}{(1-\beta_1)\sqrt{(1-\beta_2)}}\sum_{t=1}^{T}\sum_{i=1}^{d}\left|g_{t,i}\right|\sum_{j=t}^{T}\frac{\gamma^{j-t}}{\sqrt{j}},\\ &\leq \frac{\alpha}{(1-\beta_1)\sqrt{(1-\beta_2)}}\sum_{t=1}^{T}\sum_{i=1}^{d}\left|g_{t,i}\right|\sum_{j=t}^{T}\frac{\gamma^{j-t}}{\sqrt{t}}.\end{aligned}$$

Since $\gamma < 1$, we can apply the fact that for geometric series $\sum_{k\geq 1} q^k$, where $q \in (0,1)$ holds $\sum_{k\geq 1} q^k = 1/(1-q)$. This gives us

$$\begin{aligned}\sum_{t=1}^{T}\alpha_t\left\|\widehat{V}_t^{-1/4}m_t\right\|^2 &\leq \frac{\alpha}{(1-\beta_1)\sqrt{(1-\beta_2)}}\sum_{t=1}^{T}\sum_{i=1}^{d}\left|g_{t,i}\right|\frac{1}{(1-\gamma)\sqrt{t}},\\ &\leq \frac{\alpha}{(1-\beta_1)(1-\gamma)\sqrt{(1-\beta_2)}}\sum_{i=1}^{d}\left\|g_{1:T,i}\right\|_2\sqrt{\sum_{t=1}^{T}\frac{1}{t}},\end{aligned}$$

where we used the definition of the Euclidean norm $\|\cdot\|_2$ and the fact that $\sum_{t\geq 1} 1/\sqrt{t} \leq \sqrt{\sum_{t\geq 1} 1/t}$ in the last step.
If we now apply the bound on harmonic sums $\sum_{t=1}^{T} 1/t \leq (1+\log T)$, we receive

$$\begin{aligned}\sum_{t=1}^{T}\alpha_t\left\|\widehat{V}_t^{-1/4}m_t\right\|^2 &\leq \frac{\alpha}{(1-\beta_1)(1-\gamma)\sqrt{(1-\beta_2)}}\sum_{i=1}^{d}\left\|g_{1:T,i}\right\|_2\sqrt{1+\log T},\\ &= \frac{\alpha\sqrt{1+\log T}}{(1-\beta_1)(1-\gamma)\sqrt{(1-\beta_2)}}\sum_{i=1}^{d}\left\|g_{1:T,i}\right\|_2. \qquad (1.45)\end{aligned}$$

Now, plugging inequality (1.45) into inequality (1.44) which we were interested in originally, gives us

$$\begin{aligned}&\sum_{t=1}^{T}\mathcal{R}_{\mathcal{L},D}(f_t(\theta_t)) - \mathcal{R}_{\mathcal{L},D}(f_t(\theta^*))\\ &\leq \sum_{t=1}^{T}\left[\frac{1}{2\alpha_t(1-\beta_{1t})}\left[\left\|\widehat{V}_t^{1/4}(\theta_t-\theta^*)\right\|^2 - \left\|\widehat{V}_t^{1/4}(\theta_{t+1}-\theta^*)\right\|^2\right]\right.\\ &\quad\left.+\frac{\beta_{1t}}{2\alpha_t(1-\beta_{1t})}\left\|\widehat{V}_t^{1/4}(\theta_t-\theta^*)\right\|^2\right] + \frac{\alpha\sqrt{1+\log T}}{(1-\beta_1)^2(1-\gamma)\sqrt{(1-\beta_2)}}\sum_{i=1}^{d}\left\|g_{1:T,i}\right\|_2.\end{aligned}$$

Pulling out the first summand of the sum and shifting the index leads to

$$\begin{aligned}
&\sum_{t=1}^{T} \mathcal{R}_{\mathcal{L},D}(f_t(\theta_t)) - \mathcal{R}_{\mathcal{L},D}(f_t(\theta^*)) \\
&\leq \frac{1}{2\alpha_1(1-\beta_1)} \left\| \widehat{V}_1^{1/4} (\theta_1 - \theta^*) \right\|^2 + \sum_{t=2}^{T} \left[\frac{\left\| \widehat{V}_t^{1/4} (\theta_t - \theta^*) \right\|^2}{2\alpha_t(1-\beta_{1t})} - \frac{\left\| \widehat{V}_{t-1}^{1/4} (\theta_t - \theta^*) \right\|^2}{2\alpha_{t-1}(1-\beta_{1(t-1)})} \right] \\
&\quad + \sum_{t=1}^{T} \left[\frac{\beta_{1t}}{2\alpha_t(1-\beta_{1t})} \left\| \widehat{V}_t^{1/4} (\theta_t - \theta^*) \right\|^2 \right] + \frac{\alpha\sqrt{1+\log T}}{(1-\beta_1)^2(1-\gamma)\sqrt{(1-\beta_2)}} \sum_{i=1}^{d} \left\| g_{1:T,i} \right\|_2 .
\end{aligned}$$

Now we expand the first sum and pull out the factor $1/2$

$$\begin{aligned}
&\sum_{t=1}^{T} \mathcal{R}_{\mathcal{L},D}(f_t(\theta_t)) - \mathcal{R}_{\mathcal{L},D}(f_t(\theta^*)) \\
&\leq \frac{1}{2\alpha_1(1-\beta_1)} \left\| \widehat{V}_1^{1/4} (\theta_1 - \theta^*) \right\|^2 \\
&\quad + \frac{1}{2} \sum_{t=2}^{T} \left[\frac{\left\| \widehat{V}_t^{1/4} (\theta_t - \theta^*) \right\|^2}{\alpha_t(1-\beta_{1(t-1)})} - \frac{\left\| \widehat{V}_t^{1/4} (\theta_t - \theta^*) \right\|^2}{\alpha_t(1-\beta_{1(t-1)})} + \frac{\left\| \widehat{V}_t^{1/4} (\theta_t - \theta^*) \right\|^2}{\alpha_t(1-\beta_{1t})} \right. \\
&\quad \left. - \frac{\left\| \widehat{V}_{t-1}^{1/4} (\theta_t - \theta^*) \right\|^2}{\alpha_{t-1}(1-\beta_{1(t-1)})} \right] + \sum_{t=1}^{T} \left[\frac{\beta_{1t}}{2\alpha_t(1-\beta_{1t})} \left\| \widehat{V}_t^{1/4} (\theta_t - \theta^*) \right\|^2 \right] \\
&\quad + \frac{\alpha\sqrt{1+\log T}}{(1-\beta_1)^2(1-\gamma)\sqrt{(1-\beta_2)}} \sum_{i=1}^{d} \left\| g_{1:T,i} \right\|_2 .
\end{aligned}$$

If we now consider the fact, that $\beta_{1t} \leq \beta_1$ for all $t = 1, \ldots, T$ and the following observation

$$\frac{\left\| \widehat{V}_t^{1/4} (\theta_t - \theta^*) \right\|^2}{\alpha_t(1-\beta_{1t})} - \frac{\left\| \widehat{V}_t^{1/4} (\theta_t - \theta^*) \right\|^2}{\alpha_t(1-\beta_{1(t-1)})} \leq \frac{\beta_{1t}}{\alpha_t (1-\beta_1)^2} \left\| \widehat{V}_t^{1/4} (\theta_t - \theta^*) \right\|^2 ,$$

then, this leads to

$$
\begin{aligned}
&\sum_{t=1}^{T} \mathcal{R}_{\mathcal{L},D}(f_t(\theta_t)) - \mathcal{R}_{\mathcal{L},D}(f_t(\theta^*)) \\
&\leq \frac{1}{2\alpha_1(1-\beta_1)} \left\| \widehat{V}_1^{1/4} (\theta_1 - \theta^*) \right\|^2 \\
&\quad + \frac{1}{2} \sum_{t=2}^{T} \left[\frac{\left\| \widehat{V}_t^{1/4} (\theta_t - \theta^*) \right\|^2}{\alpha_t (1-\beta_{1(t-1)})} - \frac{\left\| \widehat{V}_{t-1}^{1/4} (\theta_t - \theta^*) \right\|^2}{\alpha_{t-1}(1-\beta_{1(t-1)})} \right] \\
&\quad + \sum_{t=1}^{T} \left[\frac{\beta_{1t}}{2\alpha_t(1-\beta_{1t})} \left\| \widehat{V}_t^{1/4} (\theta_t - \theta^*) \right\|^2 \right] + \sum_{t=2}^{T} \left[\frac{\beta_{1t}}{\alpha_t (1-\beta_1)^2} \left\| \widehat{V}_t^{1/4} (\theta_t - \theta^*) \right\|^2 \right] \\
&\quad + \frac{\alpha\sqrt{1+\log T}}{(1-\beta_1)^2(1-\gamma)\sqrt{(1-\beta_2)}} \sum_{i=1}^{d} \left\| g_{1:T,i} \right\|_2 , \\
&\leq \frac{1}{2\alpha_1(1-\beta_1)} \left\| \widehat{V}_1^{1/4} (\theta_1 - \theta^*) \right\|^2 \\
&\quad + \frac{1}{2(1-\beta_1)} \sum_{t=2}^{T} \left[\frac{\left\| \widehat{V}_t^{1/4} (\theta_t - \theta^*) \right\|^2}{\alpha_t} - \frac{\left\| \widehat{V}_{t-1}^{1/4} (\theta_t - \theta^*) \right\|^2}{\alpha_{t-1}} \right] \\
&\quad + \sum_{t=1}^{T} \left[\frac{\beta_{1t}}{\alpha_t (1-\beta_1)^2} \left\| \widehat{V}_t^{1/4} (\theta_t - \theta^*) \right\|^2 \right] + \frac{\alpha\sqrt{1+\log T}}{(1-\beta_1)^2(1-\gamma)\sqrt{(1-\beta_2)}} \sum_{i=1}^{d} \left\| g_{1:T,i} \right\|_2 ,
\end{aligned}
$$

where we simply extended the third sum with the non-negative summand for $t = 1$, in order to join it with the second sum.
Furthermore, using the definition of the operator $\widehat{V}_t$ gives us

$$
\begin{aligned}
&\sum_{t=1}^{T} \mathcal{R}_{\mathcal{L},D}(f_t(\theta_t)) - \mathcal{R}_{\mathcal{L},D}(f_t(\theta^*)) \\
&\leq \frac{1}{2\alpha_1(1-\beta_1)} \sum_{i=1}^{d} \widehat{v}_{1,i}^{1/2} (\theta_{1,i} - \theta_i^*)^2 + \frac{1}{2(1-\beta_1)} \sum_{t=2}^{T} \sum_{i=1}^{d} \left[\left(\frac{\widehat{v}_{t,i}^{1/2}}{\alpha_t} - \frac{\widehat{v}_{t-1,i}^{1/2}}{\alpha_{t-1}} \right) (\theta_{t,i} - \theta_i^*)^2 \right] \\
&\quad + \frac{1}{(1-\beta_1)^2} \sum_{t=1}^{T} \sum_{i=1}^{d} \frac{\beta_{1t} \widehat{v}_{t,i}^{1/2} (\theta_{t,i} - \theta_i^*)^2}{\alpha_t} + \frac{\alpha\sqrt{1+\log T}}{(1-\beta_1)^2(1-\gamma)\sqrt{(1-\beta_2)}} \sum_{i=1}^{d} \left\| g_{1:T,i} \right\|_2 ,
\end{aligned}
$$

Since we assumed that the parameter space Θ has bounded diameter D_∞, we can write as well

$$\begin{aligned}
&\sum_{t=1}^{T} \mathcal{R}_{\mathcal{L},D}(f_t(\theta_t)) - \mathcal{R}_{\mathcal{L},D}(f_t(\theta^*)) \\
&\leq \frac{1}{2\alpha_1(1-\beta_1)} \sum_{i=1}^{d} \widehat{v}_{1,i}^{1/2} D_\infty^2 + \frac{1}{2(1-\beta_1)} \sum_{t=2}^{T} \sum_{i=1}^{d} \left[\left(\frac{\widehat{v}_{t,i}^{1/2}}{\alpha_t} - \frac{\widehat{v}_{t-1,i}^{1/2}}{\alpha_{t-1}} \right) D_\infty^2 \right] \\
&\quad + \frac{1}{(1-\beta_1)^2} \sum_{t=1}^{T} \sum_{i=1}^{d} \frac{\beta_{1t} \widehat{v}_{t,i}^{1/2} D_\infty^2}{\alpha_t} + \frac{\alpha\sqrt{1+\log T}}{(1-\beta_1)^2(1-\gamma)\sqrt{(1-\beta_2)}} \sum_{i=1}^{d} \left\| g_{1:T,i} \right\|_2 ,
\end{aligned}$$

Lastly, we realize that the first double sum is of telescopic nature. Hence, we can reduce it to

$$\begin{aligned}
&\sum_{t=1}^{T} \mathcal{R}_{\mathcal{L},D}(f_t(\theta_t)) - \mathcal{R}_{\mathcal{L},D}(f_t(\theta^*)) \\
&\leq \frac{D_\infty^2 \sqrt{T}}{2\alpha(1-\beta_1)} \sum_{i=1}^{d} \widehat{v}_{T,i}^{1/2} + \frac{D_\infty^2}{(1-\beta_1)^2} \sum_{t=1}^{T} \sum_{i=1}^{d} \frac{\beta_{1t} \widehat{v}_{t,i}^{1/2}}{\alpha_t} \\
&\quad + \frac{\alpha\sqrt{1+\log T}}{(1-\beta_1)^2(1-\gamma)\sqrt{(1-\beta_2)}} \sum_{i=1}^{d} \left\| g_{1:T,i} \right\|_2 ,
\end{aligned}$$

where we additionally included the definition of α_T. □

1.4 Neural networks in computer vision

Lastly in this chapter, we want to apply the theory of neural networks to the setting we actually are interested in. This setting is usually called computer vision - basically, machine learning that is applied to images and videos. Since we want to apply neural networks to a problem that deals with images, we need to know how to view images from a mathematical perspective. In order to do so, we need to introduce some quantities first. The first important quantity is a pixel. Basically, this is a single point in the image.

Definition 1.4.1. Let $d \in \mathbb{N}$ and $\Psi = \{0, \ldots, 255\}$. Then we define a **pixel with d channels** as

$$\psi = (\psi_1, \psi_2, \ldots, \psi_d) ,$$

where for all $i = 1, \ldots, d$ holds $\psi_i \in \Psi$. Hence, for each pixel holds $\psi \in \Psi^d$

Remark 1.4.2. We want to distinguish mainly two kinds of pixels. If $d = 1$, we speak of a **black and white pixel**.
If $d = 3$, we speak of an **RGB pixel**. Here, RGB stands for the red, green and blue color channels.

As one may already know, images consist of multiple pixels that are aligned in a grid. We will refer to this grid as a pixel domain as formulated in the following definition.

Definition 1.4.3. Let $M \in \mathbb{N}$ be the **horizontal amount of pixels** and $N \in \mathbb{N}$ be the **vertical amount of pixels**. Then we call the grid Ω defined by

$$\Omega \coloneqq \{1, \dots, M\} \times \{1, \dots, N\} \subset \mathbb{N}^2,$$

the **pixel domain** with the tuple (M, N) being called the **resolution**.

If we now combine the definitions of a pixel and a pixel domain, we can define a mathematical representation of an image - a so called digital image.

Definition 1.4.4. Let $d \in \mathbb{N}$ be the number of channels and Ω a pixel domain with the resolution (M, N). Then we define a **digital image** by

$$\psi = \left(\psi_{ij}\right)_{i,j} = \left(\psi_{ij,1}, \dots, \psi_{ij,d}\right), \qquad (i,j) \in \Omega,$$

where each ψ_{ij} is a pixel with d channels.
Since for each pixel ψ_{ij} holds $\psi_{ij} \in \Psi^d$, we define the **image domain** as $\Psi^{d \times M \times N}$. We will write $\psi \in \Psi_{d,\Omega} \coloneqq \Psi^{d \times M \times N}$ from now on. If the context is clear, we will reduce the notation up to Ψ.

However, since we usually consider floats and not integers in numerical mathematics, we need to consider a way to represent pixels as floats. In order to do that, we introduce the normed image domain.

Definition 1.4.5. Let Ω be a pixel domain and $d \in \mathbb{N}$ a number of channels. Then we call the Ψ', defined as

$$\Psi' \coloneqq \frac{1}{255}\Psi = \left\{\frac{0}{255}, \frac{1}{255}, \dots, \frac{255}{255}\right\},$$

the **normed image domain**. Obviously, it holds $\Psi' \subset [0,1]$.

We realize that we can easily transform images from an ordinary image domain to a normed image domain by dividing each pixel value by 255. Equally, we can transform images the other way around by multiplying each pixel value by 255.
Since neural networks rarely produce a value that is a fraction with denominator 255, we need to find a way to process such values. This we will do in the following lemma.

Lemma 1.4.6. *Let $d \in \mathbb{N}$ be a number of channels and let $p \in [0,1]^d$. Then we can consider p as a pixel by transforming it through*

$$\psi_i = \lceil 255 \cdot p_i - 0.5 \rceil, \qquad i = 1, \dots, d.$$

Proof. Since $p \in [0,1]^d$, we can denote p as

$$p = (p_1, \dots, p_d),$$

where for all $i = 1, \dots, d$ holds $p_i \in [0,1]$.
Hence, if we multiply each p_i with the scalar 255, it follows that

$$255 \cdot p = (255p_1, \dots, 255p_d) \in [0, 255]^d.$$

Considering the Gauss brackets gives us the representation defined in Definition 1.4.1. □

Now having formally defined what an image is, we can consider how a neural network operating on images looks like. In order to do this, it is sufficient to consider a single neural layer, since neural networks consist of multiple layers. But first, we need some technical assertions to understand how we can actually feed images into neural networks, since images are interpretable as matrices and neural networks often operate on arrays.

Lemma 1.4.7. *Let Ω be a pixel domain with resolution (M, N) and $d \in \mathbb{N}$ denote the number of channels. Then the corresponding image domain $\Psi_{d,\Omega}$ can be represented as $\Psi_{d,\Omega} = \Psi^{d \times M \cdot N \times 1}$, i.e. by arranging all pixels in one vector we can „flatten“ an image.*

Proof. Let ψ be a matrix with entries $\psi_{ij} \in \Psi_d$, what follows from the Definition 1.4.4 of an image. Hence, ψ is an $M \times N$- matrix.
Define the rows of ψ by $\widehat{\psi}_i \coloneqq (\psi_{i1}, \dots, \psi_{iN})$ for all $i \in \{1, \dots, M\}$. Then the matrix representation of the picture ψ can be written as

$$\begin{pmatrix} \psi_{11} & \cdots & \psi_{1N} \\ \vdots & \ddots & \vdots \\ \psi_{M1} & \cdots & \psi_{MN} \end{pmatrix} = \begin{pmatrix} \widehat{\psi}_1 \\ \vdots \\ \widehat{\psi}_M \end{pmatrix}.$$

If we now transpose each $\widehat{\psi}_i$ and keep the same representation, we transform the $M \times N$ matrix into an $M \cdot N$-dimensional array

$$\begin{pmatrix} \widehat{\psi}_1^\mathsf{T} \\ \vdots \\ \widehat{\psi}_M^\mathsf{T} \end{pmatrix} \in \Psi^{MN}.$$

□

Lemma 1.4.7 allows us to feed images into a neural network by considering them as one large array. However, it still is unclear how the images are processed throughout the neural network. In order to formulate this, we need to define a function operating on images.

Definition 1.4.8. Let Ω_0 and Ω_1 be pixel domains with resolutions (M_0, N_0) and (M_1, N_1) respectively. Furthermore, let $d \in \mathbb{N}$ be an arbitrary number of channels.
Then we define an **image operator** T as the continuous mapping

$$\begin{aligned} T : \Psi_{d,\Omega_0} &\to \Psi_{d,\Omega_1} \\ \psi_0 &\mapsto \psi_1 \coloneqq T\,\psi_0, \end{aligned}$$

where T does not change the number of channels d. Thus, we shorten Ψ_{d,Ω_0} to Ψ_{Ω_0} in the following.

The Definition 1.4.8 is quite general, since we do not demand any specific properties from the image operator T. There are some image operators that are commonly used in computer vision, we will take a look at those in the course of this section.
These technicalities help us introduce neural networks operating on images.

Definition 1.4.9. Let φ be an arbitrary activation function and $\widehat{\varphi}$ the component-wise mapping of φ as in Definition 1.2.4 and Ω_0 be an arbitrary pixel domain with resolution $(M_0, N_0) \in \mathbb{N}^2$. Let ψ be an image with number of channels $d \in \mathbb{N}$.
Then a neural layer that operates on images looks as follows

$$\begin{aligned} H : \Psi_{\Omega_0} &\to \Psi_{\Omega_1} \\ \psi_0 &\mapsto \widehat{\varphi}\left(T\,\psi_0 + b\right), \end{aligned}$$

where T is an image operator as in Definition 1.4.8, $b \in \Psi_{\Omega_0}$ is a bias and Ω_1 a pixel domain with resolution (M_1, N_1).

With the help of Definition 1.4.9 we realise, that each layer H_i of a neural network in a computer vision setting represents an own image space, denoted by Ψ_{Ω_i}. Meaning, that all elements fed to the corresponding layer are images with resolution (M_i, N_i).
Now we want to consider some useful examples of image operators. First, we will take a look at multidimensional convolutions, hence convolutions on images. For more details please look at [3, Chapter 9].

Definition 1.4.10. Let Ω_0, Ω_1 be pixel spaces with arbitrary but fixed number of channels and resolutions (M_0, N_0) and (M_1, N_1) respectively. Then the **image convolution operator** $T_{\text{conv}} : \Psi_{\Omega_0} \to \Psi_{\Omega_1}$ is defined by

$$\begin{aligned} T_{\text{conv}} : \Psi_{\Omega_0} &\to \Psi_{\Omega_1} \\ \psi_0 &\mapsto \psi_1 \coloneqq T_{\text{conv}}\,\psi_0\,. \end{aligned}$$

This operator is defined component-wise by

$$(\psi_1)_{ij} \coloneqq (\psi_0 * k)_{ij} = \sum_{m,n=1}^{s} (\psi_0)_{m+i,n+j}\, k_{mn}, \tag{1.46}$$

where $(i, j) \in \Omega_1 \coloneqq \{1, \ldots, M_0 - s + 1\} \times \{1, \ldots, N_0 - s + 1\}$. Hence, $M_1 = M_0 - s + 1$ and $N_1 = N_0 - s + 1$.
Furthermore, k is called an s-**convolution kernel**, an image with resolution (s, s) with $s \in \mathbb{N}$ and $s \leq \min\{M_0, N_0\}$.

Another very useful example is the pooling operator. We will distinguish between average and min- and max-pooling.

Definition 1.4.11. Let Ω_0, Ω_1 be pixel spaces with arbitrary but fixed number of channels and resolutions (M_0, N_0) and (M_1, N_1) respectively. Furthermore, $s \in \mathbb{N}$ with $s \leq \min\{M_0, N_0\}$ is called **stride** and $I \coloneqq \{1, \ldots, p_1\} \times \{1, \ldots, p_2\} \subset \mathbb{N}^2$ with $p_1, p_2 \in \mathbb{N}$ is called **pooling**.
Then the **average-pooling operator** $T_{\text{avg}} : \Psi_{\Omega_0} \to \Psi_{\Omega_1}$ with stride s and pooling I is defined by

$$\begin{aligned} T_{\text{avg}} : \Psi_{\Omega_0} &\to \Psi_{\Omega_1} \\ \psi_0 &\mapsto \psi_1 \coloneqq T_{\text{avg}}\,\psi_0\,. \end{aligned}$$

This operator is defined component-wise by

$$(\psi_1)_{ij} \coloneqq \frac{1}{p_1 p_2} \sum_{(m,n)\in I} (\psi_0)_{m+i,n+j}\,, \tag{1.47}$$

where $(i, j) \in \Omega_1 \coloneqq \{1, \ldots, M_0 - p_1 + 1\} \times \{1, \ldots, N_0 - p_2 + 1\}$. Hence, $M_1 = M_0 - p_1 + 1$ and $N_1 = N_0 - p_2 + 1$.

Definition 1.4.12. Let Ω_0, Ω_1 be pixel spaces with arbitrary but fixed number of channels and resolutions (M_0, N_0) and (M_1, N_1) respectively.
Then the **min-pooling operator** $T_{\min} : \Psi_{\Omega_0} \to \Psi_{\Omega_1}$ with stride s and pooling I is defined by

$$\begin{aligned} T_{\min} : \Psi_{\Omega_0} &\to \Psi_{\Omega_1} \\ \psi_0 &\mapsto \psi_1 \coloneqq T_{\min}\, \psi_0 \,. \end{aligned}$$

This operator is defined component-wise by

$$(\psi_1)_{ij} \coloneqq \min_{(k,l)\in I} (\psi_0)_{kl} \,, \tag{1.48}$$

where $(i,j) \in \Omega_1 \coloneqq \{1, \ldots, M_0 - p_1 + 1\} \times \{1, \ldots, N_0 - p_2 + 1\}$. Hence, $M_1 = M_0 - p_1 + 1$ and $N_1 = N_0 - p_2 + 1$.

Definition 1.4.13. Let Ω_0, Ω_1 be pixel spaces with arbitrary but fixed number of channels and resolutions (M_0, N_0) and (M_1, N_1) respectively.
Then the **max-pooling operator** $T_{\max} : \Psi_{\Omega_0} \to \Psi_{\Omega_1}$ with stride s and pooling I is defined by

$$\begin{aligned} T_{\max} : \Psi_{\Omega_0} &\to \Psi_{\Omega_1} \\ \psi_0 &\mapsto \psi_1 \coloneqq T_{\max}\, \psi_0 \,. \end{aligned}$$

This operator is defined component-wise by

$$(\psi_1)_{ij} \coloneqq \max_{(k,l)\in I} (\psi_0)_{kl} \,, \tag{1.49}$$

where $(i,j) \in \Omega_1 \coloneqq \{1, \ldots, M_0 - p_1 + 1\} \times \{1, \ldots, N_0 - p_2 + 1\}$. Hence, $M_1 = M_0 - p_1 + 1$ and $N_1 = N_0 - p_2 + 1$.

At this point, we should mention that each of the recently introduced operators can be used to connect two neural layers. This we will put down in writing in the following proposition.

Definition 1.4.14. Let T_{conv} be an image convolution operator with s-convolution kernel k. Furthermore, let Ψ_0 and Ψ_1 be arbitrary image domains and φ be an arbitrary activation function.
Then

$$\begin{aligned} H_{\text{conv}} : \Psi_0 &\to \Psi_1, \\ \psi_0 &\mapsto H_{\text{conv}}(\psi_0) = \widehat{\varphi}\,(T_{\text{conv}}\, \psi_0 + b) \,, \end{aligned}$$

defines a neural layer, where the parameters θ are the s-convolution kernel k. We will call such a layer a **convolutional neural layer** and denote the layers' parameters $\theta = T_{\text{conv}}$, since the kernel defines the convolution operator uniquely.

Chapter 2

Autoencoders

Having introduced the basics of neural networks in Chapter 1, we can consider a specific architecture of a neural network, a so called autoencoding neural network, or simply autoencoder in short. The idea of autoencoders is to take a given input, compress the input to a smaller dimension, which we will refer to as encoding and afterwards, expand it as accurately as possible to the original dimension again, which we will refer to as decoding. Note that this compression of the data is usually referred to as dimensionality reduction by theoreticians and as feature extraction by software engineers in literature. Those are simply different terms for the same idea. Such an architecture is often describes as a discriminative model, i.e. a model that tries to learn the mapping between input data and output labels directly. This is the counterpart to generative models, which we will take a look at in Chapter 3. The goal of such an autoencoding neural network is as already described, to learn how to compress data to a lower dimensional representation and afterwards, reconstruct the original representation as precisely as possible.
A very important application of autoencoders, which is commonly used in the realm of Machine Learning. Most state of the art Machine Learning models use autoencoding structures, since it is way more efficient to first encode the data and then run a model on the encoded data. This is due to the fact, that if we succeed in encoding and decoding the data without loss of information, we then can perform operations, e.g. classification, on the lower dimensional data. Since we can easily reduce the dimensionality by magnitudes, which we will see in Section 2.3. This way processing the samples can happen much faster compared to the non-encoded data samples and secondly, it makes storing data (on the drive and in memory) much more efficient. In this chapter we want to consider how to formulate autoencoding neural networks from a mathematical point of view, take a look at some important results and lastly, analyse the theory in multiple applications using Python.

2.1 Conceptional ideas

As already mentioned, an autoencoding neural network first encodes the input data to a smaller representation. The dimension of this smaller representation is usually referred to as bottleneck of the autoencoder. Afterwards, the autoencoding neural network decodes the data to its original dimension. Hence, we can divide these two steps into separate architectures - the encoding and the decoding part of the neural network, which we will formulate separately.
In Figure 2.1 we can take a look at a visual example of an autoencoding architecture. Each circle represents a neuron and each line represents a connection between neurons.

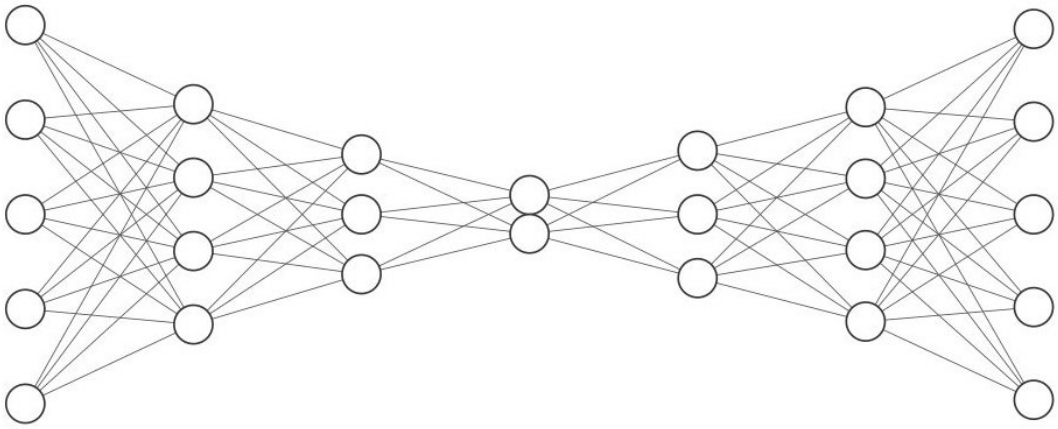


Figure 2.1: An autoencoding neural network with input and output $x, y \in \mathbb{R}^5$. The five hidden layers have dimensions 4, 3, 2, 3 and 4 respectively. Hence, the bottleneck dimension is 2 in this example. The graphic was generated with http://alexlenail.me/NN-SVG/index.html

If we divide the autoencoding structure as described above, we firstly obtain the encoder as we can see in Figure 2.2 or formally defined as follows.

Definition 2.1.1. Let Θ be a parameter space and $\theta \in \Theta$ a set of parameters, $L \in \mathbb{N}$ and $d_1, \ldots, d_L \in \mathbb{N}$. Let further φ be an activation function and $f_{\varphi,L,\theta}$ a neural network.
If the neural network $f_{\varphi,L,\theta}$ fulfils the condition $n_i = d_1 \geq \ldots \geq d_L = n_o$ with $n_i, n_o \in \mathbb{N}$ being the input and output dimensions respectively, then we speak of an **encoding neural network** (or short: **encoder**) and denote it as f_e.

For the second part of the divided autoencoding structure, we obtain the decoder as we can see in Figure 2.3. We can define this architecture analogously to the encoder in Definition 2.1.1.

Definition 2.1.2. Let Θ be a parameter space and $\theta \in \Theta$ a parameter, $L \in \mathbb{N}$ and $d_1, \ldots, d_L \in \mathbb{N}$. Let further φ be an activation function and $f_{\varphi,L,\theta}$ a neural network.
If the neural network $f_{\varphi,L,\theta}$ fulfils the condition $n_i = d_1 \leq \ldots \leq d_L = n_o$ with $n_i, n_o \in \mathbb{N}$ being the input and output dimensions respectively, then we speak of an **decoding neural network** (or short: **decoder**).

Before combining the encoding and the decoding structure to obtain the autoencoding neural network, we need to consider the following technicality first.

Lemma 2.1.3. *Let f_1, f_2 be two neural networks of depths $L_1, L_2 \in \mathbb{N}$ with parameters $\theta_1, \theta_2 \in \Theta$, where Θ is an arbitrary parameter space. Furthermore, let the dimensions of each layer be $d_1, \ldots, d_{L_1} \in \mathbb{N}$ of f_1 and $\widetilde{d}_1, \ldots, \widetilde{d}_{L_2} \in \mathbb{N}$ of f_2. Additionally, let $d_{L_1} = \widetilde{d}_1$.*
Then their composition $f_2 \circ f_1$ is a neural network of depth $L_1 + L_2$ with parameters (θ_1, θ_2).

Proof. Since f_1 is a neural network of depth L_1 with parameters θ_1, its architecture looks like

$$f_1(x) = H_{L_1} \circ H_{L_1-1} \circ \ldots H_2 \circ H_1(x), \qquad x \in \mathbb{R}^{d_1}\,. \tag{2.1}$$

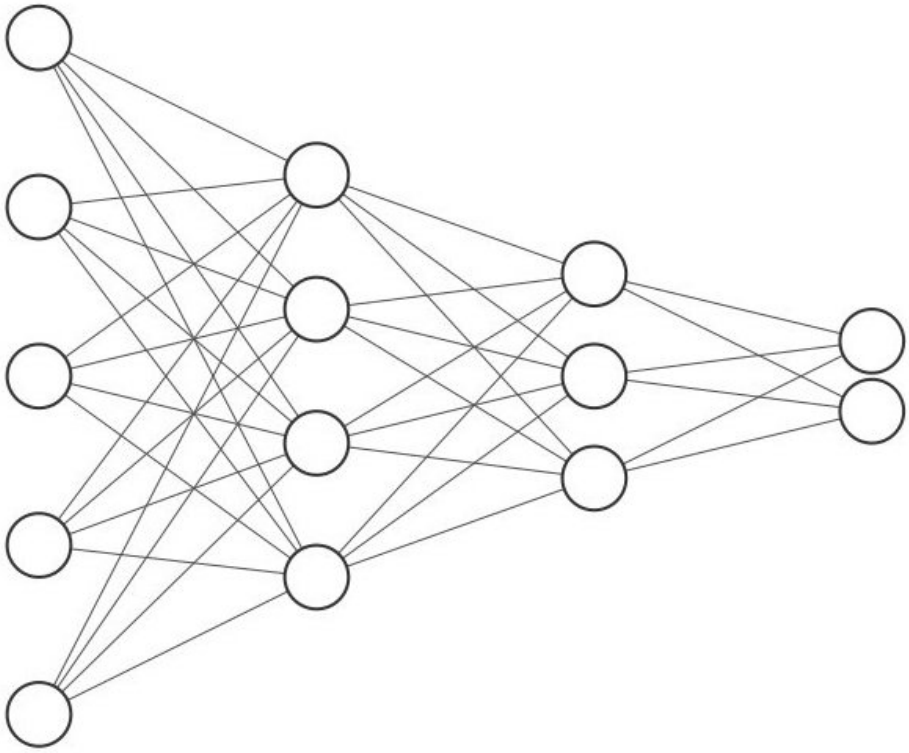


Figure 2.2: An encoding neural network with input $x \in \mathbb{R}^5$ and output $y \in \mathbb{R}^2$. The two hidden layers have dimensions 4 and 3. Hence, the encoder reduces the data dimensionality from 5 to 2 dimension. The graphic was generated with http://alexlenail.me/NN-SVG/index.html

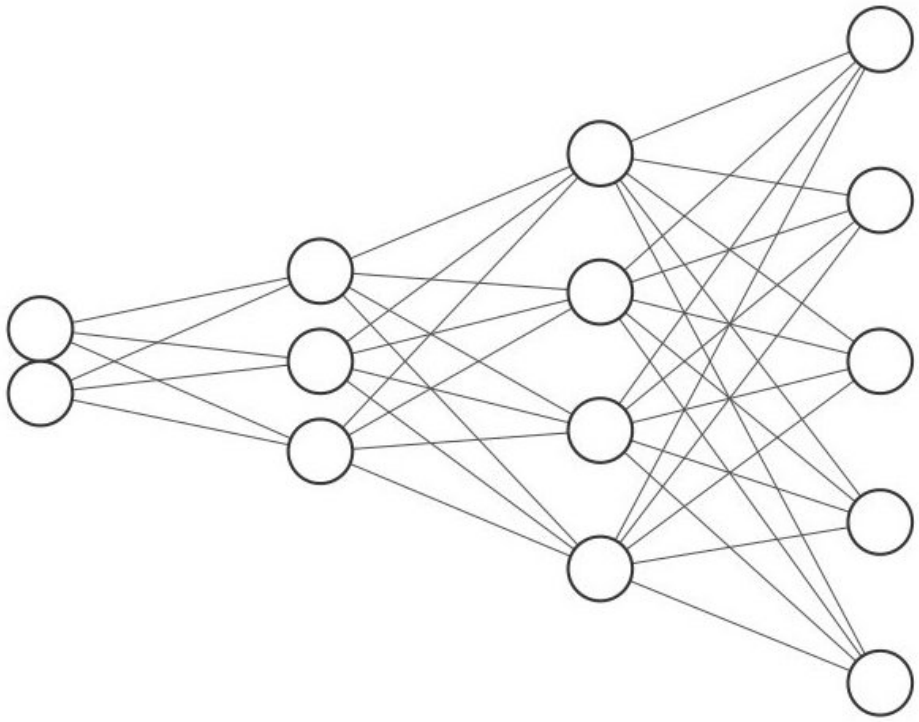


Figure 2.3: A decoding neural network with input $x \in \mathbb{R}^2$ and output $y \in \mathbb{R}^5$. The two hidden layers have dimensions 3 and 4. Hence, the decoder expands the data dimensionality from 2 to 5 dimensions. The graphic was generated with http://alexlenail.me/NN-SVG/index.html

Analogously, we can write f_2 as

$$f_2(y) = \widetilde{H}_{L_2} \circ \widetilde{H}_{L_2-1} \circ \dots \widetilde{H}_2 \circ \widetilde{H}_1(y), \qquad y \in \mathbb{R}^{\widetilde{d}_1}. \tag{2.2}$$

Since we assumed that the output dimension d_{L_1} of the neural network f_1 is equal to the input dimension $\widetilde{d}_1$ of the neural network f_2, we can consider the result of (2.1) as input for (2.2)

$$y \coloneqq H_{L_1} \circ H_{L_1-1} \circ \dots H_2 \circ H_1(x), \qquad x \in \mathbb{R}^{d_1}.$$

Hence, we obtain

$$\begin{aligned} f_2(y) &= \widetilde{H}_{L_2} \circ \widetilde{H}_{L_2-1} \circ \dots \widetilde{H}_2 \circ \widetilde{H}_1(y), \\ f_2\left(f_1\left(x\right)\right) &= \widetilde{H}_{L_2} \circ \widetilde{H}_{L_2-1} \circ \dots \widetilde{H}_2 \circ \widetilde{H}_1 \circ H_{L_1} \circ H_{L_1-1} \circ \dots H_2 \circ H_1(x), \qquad x \in \mathbb{R}^{d_1}. \end{aligned} \tag{2.3}$$

Therefore, from (2.3) follows that the composition $f_2 \circ f_1$ is a neural network of depth $L_1 + L_2$. Lastly, we consider the parameters θ of the neural network $f_2 \circ f_1$. Since the parameters of a neural network were defined as $\theta = (\theta_1, \dots, \theta_L)$, where each entry is defined as $\theta_i = (W_i, b_i)$ and denotes the weight and bias of each layer H_i or $\widetilde{H}_i$, respectively, we can write the parameters of both neural networks as

$$\begin{aligned} \theta_1 &\coloneqq (\theta_1, \dots, \theta_{L_1}) = ((W_1, b_1), \dots, (W_{L_1}, b_{L_1})), \\ \theta_2 &\coloneqq (\widetilde{\theta}_1, \dots, \widetilde{\theta}_{L_2}) = \left((\widetilde{W}_1, \widetilde{b}_1), \dots, (\widetilde{W}_{L_2}, \widetilde{b}_{L_2})\right). \end{aligned}$$

Hence, the composition $f_2 \circ f_1$ has the parameters

$$\theta = \left((W_1, b_1), \dots, (W_{L_1}, b_{L_1}), (\widetilde{W}_1, \widetilde{b}_1), \dots, (\widetilde{W}_{L_2}, \widetilde{b}_{L_2})\right) = \left(\theta_1, \dots \theta_{L_1}, \widetilde{\theta}_1, \dots, \widetilde{\theta}_{L_2}\right) =: (\theta_1, \theta_2).$$

□

Lemma 2.1.3 allows us to consider a modern approach to neural networks, a so called modular approach. Essentially, we consider entire structures like the encoding and the decoding neural network as a self-contained module. These modules can now easily be put together by considering them as a composition. This is very useful in practice, since modern neural networks consist of thousands of layers and billions of parameters. Considering a modular approach one can therefore divide the whole neural network and tune each module separately.

Theorem 2.1.4. *Let Θ be a parameter space, $N \in \mathbb{N}$ and $L_1, \dots, L_N \in \mathbb{N}$. Furthermore, let $f_1, \dots, f_N$ be neural networks with parameters $\theta_1, \dots, \theta_N \in \Theta$ and depths $L_1, \dots, L_N$, respectively. Lastly, let the output dimension of f_i match the input dimension of f_{i+1} for all $i \in \{1, \dots, N-1\}$.*
Then the composition

$$f \coloneqq f_N \circ f_{N-1} \circ \dots \circ f_1,$$

is a neural network with parameters $\theta = (\theta_1, \dots \theta_N)$ of depth $L = L_1 + \dots + L_N$.

Proof. Applying Lemma 2.1.3 to f_1 and f_2 yields the composed neural network $f^{(1)} \coloneqq f_2 \circ f_1$ with parameters $\theta^{(1)} \coloneqq (\theta_1, \theta_2)$ and depth $L^{(1)} \coloneqq L_1 + L_2$.
If we now apply Lemma 2.1.3 once again to $f^{(1)}$ and f_3, we receive $f^{(2)} \coloneqq f_3 \circ f^{(1)}$ with

parameters $\theta^{(2)} := (\theta^{(1)}, \theta_3) = (\theta_1, \theta_2, \theta_3)$ and depth $L^{(2)} := L^{(1)} + L_3 = L_1 + L_2 + L_3$.
We realize, that iteratively applying Lemma 2.1.3 yields after $N-1$ applications

$$\begin{aligned} f^{(N-1)} &= f_N \circ f_{N-1} \circ \ldots \circ f_1, \\ \theta^{(N-1)} &= (\theta_1, \ldots, \theta_N), \\ L^{(N-1)} &= \sum_{i=1}^{N-1} L_i. \end{aligned}$$

Therefore the assertion is proven. □

Definition 2.1.5. Let f_e and f_d be an encoding and a decoding neural network with input dimension n_i in $\mathbb{N}$ and output of the encoding neural network $n_b \in \mathbb{N}$, that we will refer to as **bottleneck** of the autoencoding neural network. Then we define an **autoencoding neural network** f_a as the composition

$$\begin{aligned} f_a : \mathbb{R}^{n_i} &\to \mathbb{R}^{n_i}, \\ x &\mapsto (f_d \circ f_e)(x). \end{aligned}$$

Remark 2.1.6. Let f_e and f_d be an encoding and a decoding neural network as in Definition 2.1.5. Then the composition $f_d \circ f_e$ is indeed a neural network, following from Lemma 2.1.3.

2.2 Training of autoencoders

When tackling the question of how to train an autoencoding neural network, we realize that in contrast to regular neural networks, where we compare the output of the neural network to a label, in the current setting we can compare the input data to the computed output, since the goal of an autoencoding neural network ultimately is to alter and reconstruct images. In other words, we approach this optimization problem in an unsupervised learning setting. This forces us to consider slightly different loss functions than in the supervised learning setting, since we now want to compare the predicted value to the input.

Definition 2.2.1. Let $X \subseteq \mathbb{R}^d$ be an input space and let $p : X \to \mathbb{R}^n$ be a prediction function. Furthermore, let $\widehat{x} := p(x)$ be the predicted value of $x \in X$. Then a measurable function defined as

$$\begin{aligned} \mathcal{L} : X \times \mathbb{R}^n &\to [0, \infty), \\ (x, \widehat{x}) &\mapsto \mathcal{L}(x, \widehat{x}), \end{aligned}$$

is called **unsupervised loss function**.

There are multiple important loss functions in computer vision. We will consider a couple of those in the following example. For further details we refer to [2]

Example 2.2.2. Let Ω be a pixel domain with resolution (M, N) and d the number of channels. Furthermore, let f be a neural network with arbitrary but fixed architecture. Then the following functions are loss functions operating on images.

Mean Squared Error (MSE):

$$\mathcal{L}_{\text{MSE}}(\psi, f(\psi)) = \left(\sum_{i=1}^{M} \sum_{j=1}^{N} \left| \psi_{ij} - f(\psi)_{ij} \right|^2 \right)^{1/2},$$

Binary Cross-Entropy (BCE):

$$\mathcal{L}_{\text{BCE}}(\psi, f(\psi)) = -\frac{1}{MN} \sum_{i=1}^{M} \sum_{j=1}^{N} \Big(\psi_{ij} \log \Big(f(\psi)_{ij} \Big) + \big(1 - \psi_{ij}\big) \log \Big(1 - f(\psi)_{ij} \Big) \Big),$$

where ψ denotes an image defined on Ψ_Ω.

Remark 2.2.3. The Binary Cross-Entropy loss function is usually used for binary classification problems. However, it still works in computer vision.

2.3 Applications

In this section we want to introduce and train a couple of specific autoencoding neural networks on the MNIST dataset - a dataset consisting of handwritten digits. We will consider various architectures of neural networks and visualise the results in a comprehensible manner.
First, we want to take a look at the most simple architecture, a fully connected linear neural network, where we want to introduce the encoder and the decoder separately.

Definition 2.3.1. Let Θ be an arbitrary parameter space, $L \in \mathbb{N}$ and $d_1, \ldots, d_L \in \mathbb{N}$, where $d_1 \geq \ldots \geq d_L$. Furthermore, let $\widehat{\varphi}$ be an arbitrary activation function.
Then an encoding neural network, where each layer $H_1, \ldots, H_L$ is defined as

$$H_i(x) = \widehat{\varphi}(W_i x + b_i), \qquad x \in \mathbb{R}^{d_i}, i \in \{1, \ldots, L\},$$

where $\theta_i = (W_i, b_i) \in \Theta$ are the parameters of the i-th layer, see (1.2). Such an encoding neural network is called a **linear encoder**, since the operations considered in the layers are linear operations.

Analogously, we define a linear decoding neural network as follows.

Definition 2.3.2. Let Θ be an arbitrary parameter space, $L \in \mathbb{N}$ and $d_1, \ldots, d_L \in \mathbb{N}$, where $d_1 \leq \ldots \leq d_L$. Furthermore, let $\widehat{f}$ be an arbitrary activation function.
Then a decoding neural network, where each layer $H_1, \ldots, H_L$ is defined as

$$H_i(x) = \widehat{\varphi}(W_i x + b_i), \qquad x \in \mathbb{R}^{d_i}, i \in \{1, \ldots, L\},$$

where $\theta_i = (W_i, b_i) \in \Theta$ are the parameters of the i-th layer H_i, see (1.2). Such a decoding neural network is called a **linear decoder**, since the operations considered in the layers are linear operations.

With the Definition 2.3.1 of the linear encoder and the Definition 2.3.2 of the linear decoder, we now can define a linear autoencoder as their composition.

Definition 2.3.3. Let f_e and f_d be a linear encoder and a linear decoder. Then a **linear autoencoder** f_{lin} is defined as the composition

$$f_{\text{lin}} \coloneqq f_d \circ f_e.$$

The output dimension of the linear encoder f_e is called **bottleneck** of the autoencoder.

Before considering specific examples on the MNIST dataset, we need to consider some properties of the said dataset first.

Remark 2.3.4. The MNIST dataset D consists of greyscale images with a resolution of $(28, 28)$. Hence, the images are defined on the pixel domain Ω of D with $\Omega = \{1, \dots, 28\} \times \{1, \dots, 28\}$ with only one channel.

Now, let us take a look at a specific example of a linear autoencoder defined on the MNIST dataset.

Algorithm 4 Linear Autoencoder

Let the input and output dimensions be $n_i, n_o \in \mathbb{N}$. Furthermore, let the linear encoder and the linear decoder have k hidden linear layers with dimensions $n_1, n_2, \dots, n_k \in \mathbb{N}$ with bottleneck $n_b \in \mathbb{N}$.
Furthermore, let the chosen optimizer be Adam or AMSGrad with a learning rate $\gamma > 0$ and the chosen loss function be the MSE loss function. Then the training of a linear autoencoder looks as follows.

Require: γ: Step size

```
for epoch in epochs do
    for image in batch do
        image = image.reshape(784)          ▷ Convert the image from matrix to array.
        encoded = encoder(image)            ▷ Encode the image onto latent space.
        reconstructed = decoder(encoded)    ▷ Decode the encoded image.
        loss = MSE(reconstructed, image)    ▷ Compare the output to the input.
        optimization(loss, γ)               ▷ Perform an optimization step.
    end for
end for
```

Let's take a look at the trained autoencoders using the Adam and the AMSGrad optimizer. The first thing we want to do is to consider how the latent space looks like. The latent space is the space, where the encoding part of the neural network maps the input. That is, since we process samples from the MNIST dataset, whose properties we considered in Remark 2.3.4, we know that the input dimension is 784. The first autoencoder we want to train will have a bottleneck dimension of 2, so it reduces the dimensions of the input from 784 to just 2. This resulting 2-dimensional vector we can easily visualise in a coordinate system, which we do in the left charts of Figure 2.4, where we used Adam for optimization and Figure 2.5, where we used AMSGrad for optimization, respectively. In these charts we see point clouds of ten different colours, where each point represents an encoding of a sample. The colours of the encoding

represents what digit the sample had, what we can see in the color map on the right-hand sides of the said charts. Furthermore, since we know that the MNIST dataset has samples with ten different possible labels, the digits from 0 to 9, we hope to see ten different clusters in the visualisation of the latent space. This would mean, that the encoder somehow „groups“ samples with the same label to the same location in the latent space. Taking a look at the Figure 2.4 and Figure 2.5 we see, that some digits are separated very well, e.g. the digits 0 and 1 are clearly separated from the rest. On the other hand, clusters of the digits that look similar, e.g. 3 and 8 or 4 and 9, are not separated at all. Hence, when feeding the encodings into the decoding part of the autoencoder, i.e. reconstructing the image of the digit, it will be hard to see a difference between those digits. This we can see clearly in Figure 2.6 and Figure 2.7, where in each figure the left chart shows 100 samples from the MNIST dataset, with ten samples for each of the ten digits and on the right side of the figures we can see the corresponding digit fed into the autoencoder. We see, that the samples, which are encoded in a way that they are clearly separated from the other digits are reconstructed in a way that we can recognize the original digit. However, samples that are not encoded as nicely, are not recognizable when reconstructed. Another thing we want to take a look at is the reconstruction of the latent space, explicitly. We already described that the encoder maps the samples onto a 2-dimensional vector. Hence, we can take a look at the entire 2-dimensional plane and see what the decoder does. This we can see on the right-hand side of Figure 2.4 and Figure 2.5, respectively. Basically, we consider a mesh of 10×10 nodes in the coordinate system, where each node has the same distance to its neighbour. These nodes then are being fed into the decoder. Conceptionally, we see how each vector in this plane corresponds to a reconstructed image.
Lastly, we want to take a quick look at the training progress of the two autoencoders. These we can see in Figure 2.8 for the Adam optimizer and in Figure 2.9 for the AMSGrad optimizer. We see that in roughly the first 100 epochs the training loss falls dramatically. From around epoch 1000 on, the training loss is decreased only slowly. This is due to the fact, that we chose the step size so small, that convergence takes a lot of time. Furthermore, we want to mention the fact that the training loss does not decrease monotonous. Since, we do not consider the entire dataset, we do not chose the exact gradient of the risk function in each training step. Therefore, it happens that the optimizer chooses a direction, which in the end results to increase the training loss. However, we can clearly see that on average the loss is being minimized.

Now, let's take a look at what happens if we increase the bottleneck dimension. The next experiment we want to conduct is an autoencoder that has the same architecture as the previous one, but has bottleneck dimension 3 instead. This allows the neural network to save more information upon encoding the data and hence, it should be able to produce better reconstructions. However, it makes visualising the latent space a bit more challenging. In the first experiment we were able to visualize the latent space in a plane, now we have to consider it in a 3-d space. In order to do so, we created an interactive visualisation, which can be found in the attached Python code. For the sake of completeness, we want to show the visualization here as well. In Figure 2.10 and Figure Figure 2.11 we can see two different perspectives of the 3-dimensional latent space, respectively. The first figure depicts the latent space of an autoencoder trained with an Adam optimizer and the second figure shows the latent space of an autoencoder trained with an AMSgrad optimizer. We can see in these figures clearly that the encoder behaves the same way as it did in the 2-dimensional setting, that is for each digit we can see that the encodings for one digit are all aligned on a ray. The more the digits differ,

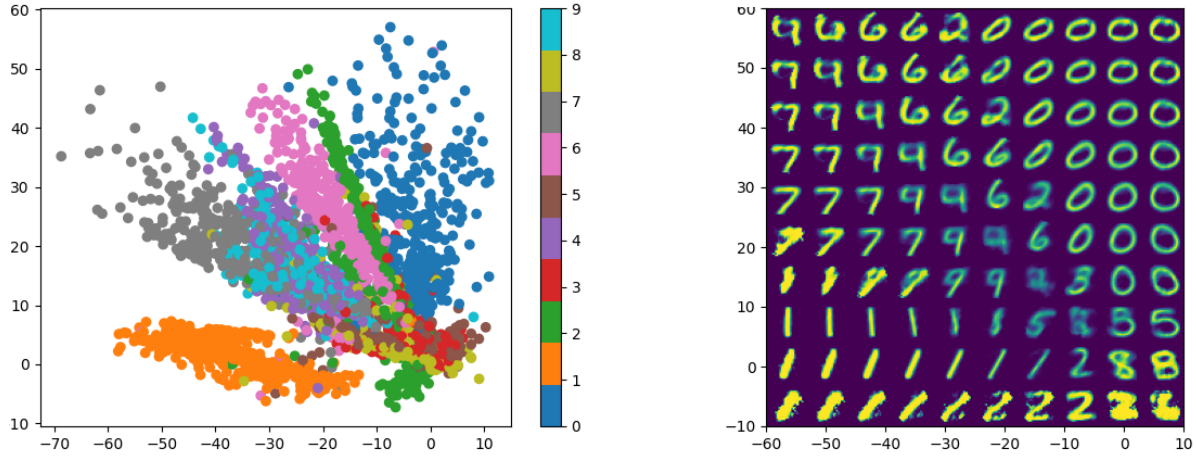


Figure 2.4: On the left side, the figure illustrates the latent space of the linear autoencoder with bottleneck $n_b = 2$ optimized with an Adam optimizer, where each dot is one encoded image of a digit. The color and the corresponding color map represent the digit that was encoded. On the right side the figure illustrates the corresponding reconstruction through the autoencoder. Each coordinate tuple is fed into the decoding architecture of the autoencoder to generate an image.

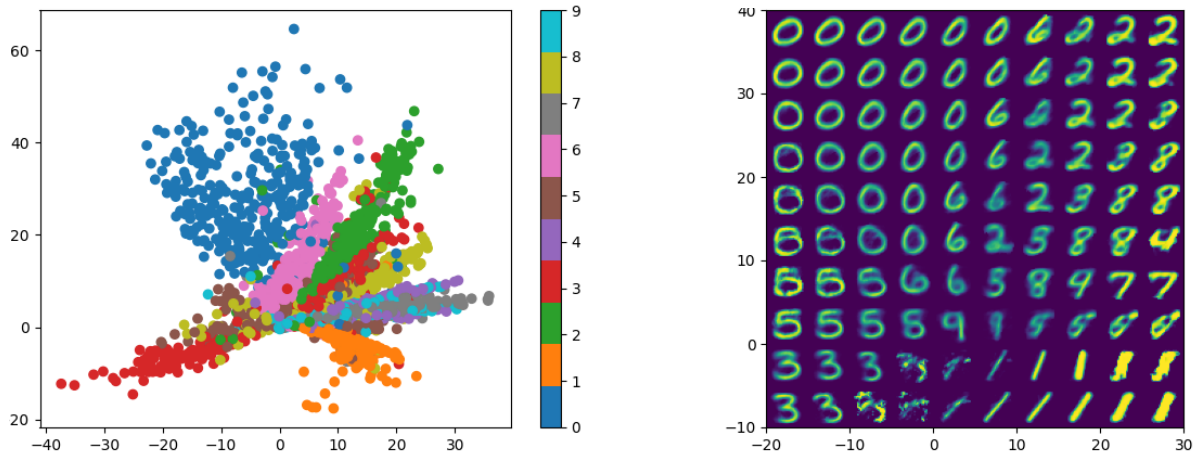


Figure 2.5: On the left side, the figure illustrates the latent space of the linear autoencoder with bottleneck $n_b = 2$ optimized with an AMSGrad optimizer, where each dot is one encoded image of a digit. The color and the corresponding color map represent the digit that was encoded. On the right side the figure illustrates the corresponding reconstruction through the autoencoder. Each coordinate tuple is fed into the decoding architecture of the autoencoder to generate an image.

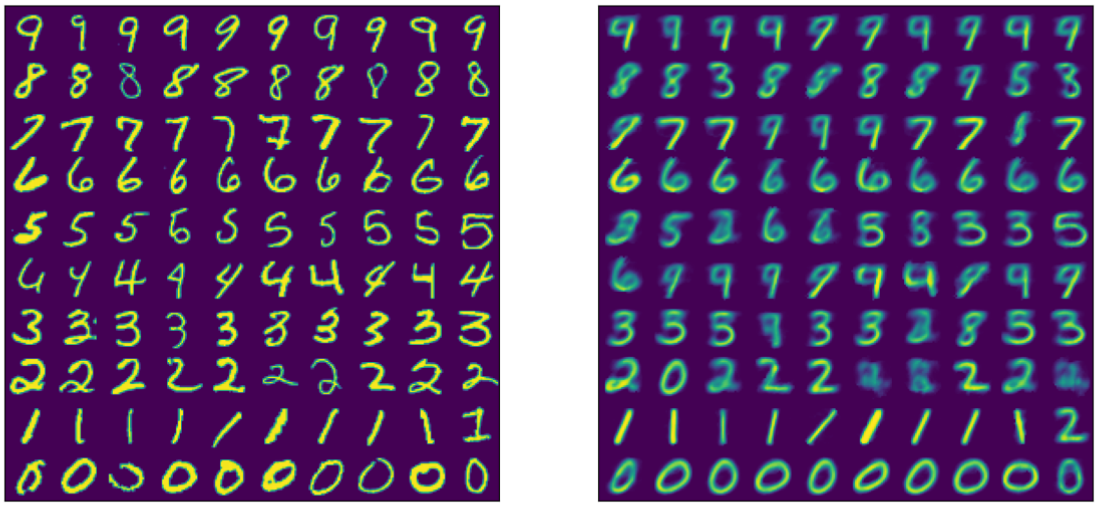


Figure 2.6: On the left side, the figure illustrates 100 original digits from the MNIST dataset. On the right side, the figure illustrates the same digits after feeding them through the linear autoencoder with bottleneck $n_b = 2$ optimized with an Adam optimizer.

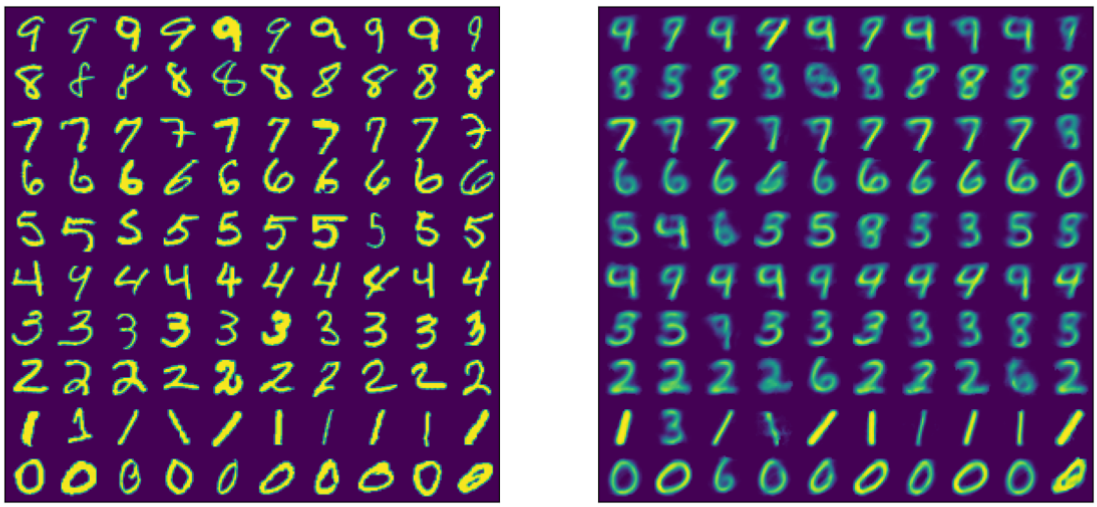


Figure 2.7: On the left side, the figure illustrates 100 original digits from the MNIST dataset. On the right side, the figure illustrates the same digits after feeding them through the linear autoencoder with bottleneck $n_b = 2$ optimized with an AMSGrad optimizer.

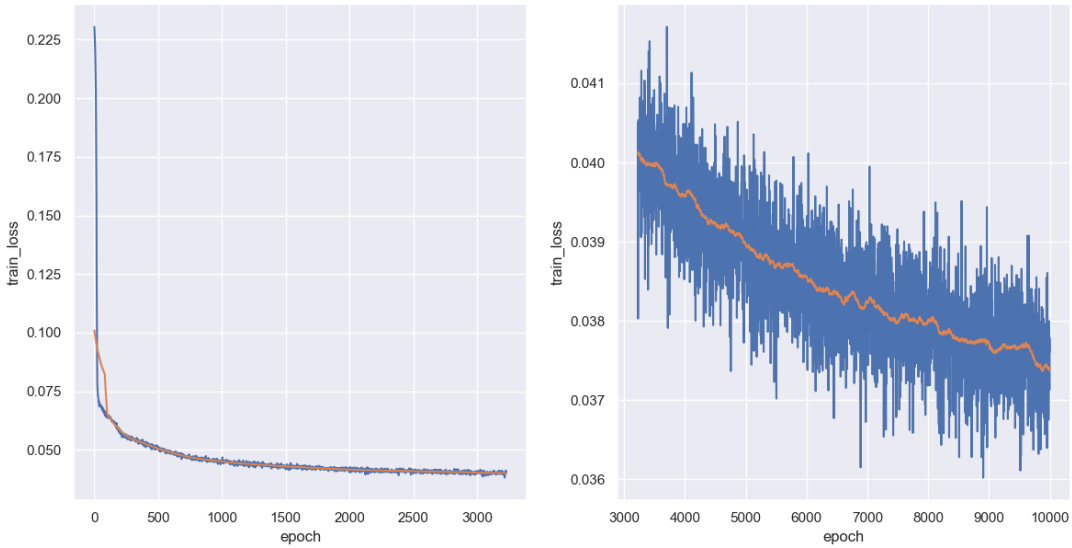


Figure 2.8: The figure illustrates the training progresses of the linear autoencoder with bottleneck $n_b = 2$ optimized with an Adam optimizer with epochs on one axis and corresponding training loss on the other axis.On the left side we see the first 3.500 epochs and on the right side the following epochs until 10.000. The blue line represents the loss in each epoch and the orange line represents the moving average over 100 epochs to point out the trend of the training progress.

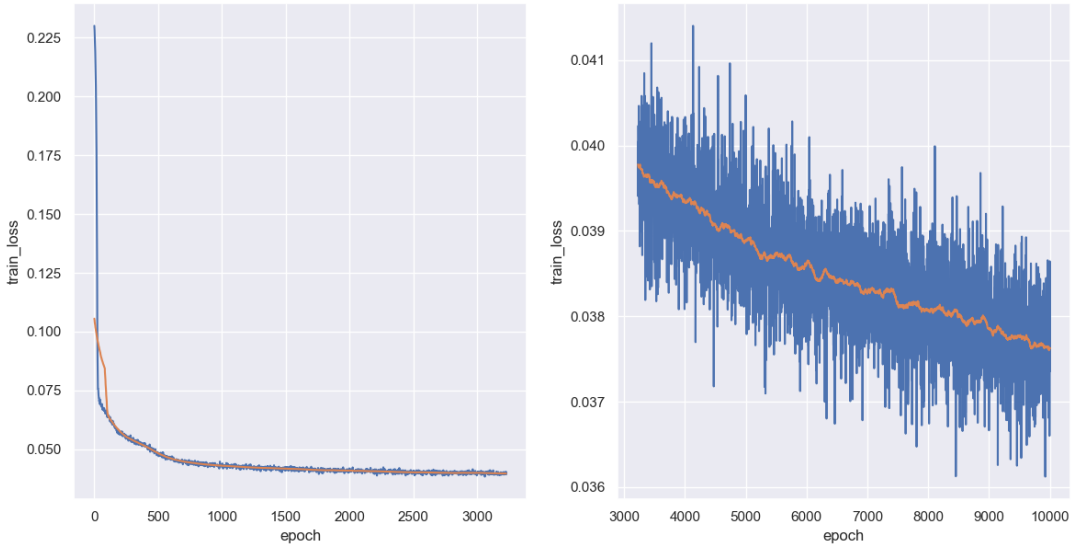


Figure 2.9: The figure illustrates the training progresses of the linear autoencoder with bottleneck $n_b = 2$ optimized with an AMSGrad optimizer with epochs on one axis and corresponding training loss on the other axis.On the left side we see the first 3.500 epochs and on the right side the following epochs until 10.000. The blue line represents the loss in each epoch and the orange line represents the moving average over 100 epochs to point out the trend of the training progress.

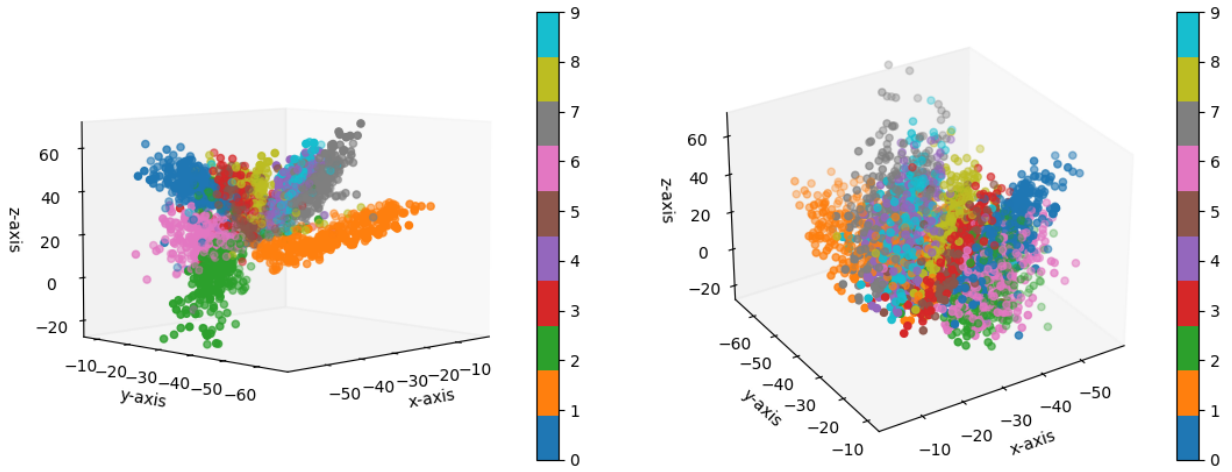


Figure 2.10: The figure illustrates the latent space of the linear autoencoder with bottleneck $n_b = 3$ optimized with an Adam optimizer from two different perspectives. Each dot is one encoded image of a digit. The color and the corresponding color map represent the digit that was encoded.

the farther apart are the rays. For example, we can see clearly in Figure 2.10 that the cluster of encoded 2's is isolated well, as well as the cluster of encoded 1's. In contrast, the clusters of encoded 4's, 7's and 9's is heavily intertwined, which results in bad reconstructions, as we can see in Figure 2.12. The same behaviour we can see in Figure 2.11, which depicts the latent space of the autoencoder, where we used the AMSgrad optimizer. The clusters are indeed slightly better separated, but still not good. This we can see in the reconstructions in Figure 2.13. Lastly, we take a look at the training progress of the two described autoencoders, which we can see in Figure 2.14 with the Adam optimizer and in Figure 2.15 with the AMSGrad optimizer, respectively. We can see that the moving average of the training loss is slightly less smooth than it was in the two dimensional setting, but the overall training progress is very similar though.

As we saw in Figure 2.12 and in Figure 2.13, the linear autoencoder does produce recognisable digits in its reconstructions, but it still performs quite poorly. To address this issue, we propose another architecture of an autoencoding neural network. In this setting, we now consider convolutional layers instead of linear layers, this means that the connections between each layer are no longer matrix multiplications, but convolutions instead. Hence, we introduce the convolutional encoder and the convolutional decoder as follows.

Definition 2.3.5. Let Θ be a parameter space, $L \in \mathbb{N}$ and $d_1, \dots, d_L \in \mathbb{N}$ be the number of channels as well as $(M_1, N_1), \dots, (M_L, N_L) \in \mathbb{N}^2$ be the resolution of the pixel domain in the i-th layer, where $M_1 \geq \dots \geq M_L$ and $N_1 \geq \dots \geq N_L$. Furthermore, let φ be an arbitrary activation function. Then an encoding neural network, where each layer $H_1, \dots, H_L$ is defined as

$$H_i(\psi) = \varphi(T_{\text{conv},i}\, \psi + b_i), \qquad \psi \in \Psi_{d_i,\Omega_i}, i \in \{1, \dots, L\},$$

where $\theta_i = (T_{\text{conv},i}, b_i) \in \Theta$ are the parameters and Ψ_{d_i,Ω_i} the image domain of the i-th layer, see Definition 1.4.14. Such an encoding neural network is called a **convolutional encoder**.

Analogously, we define a convolutional decoding neural network as follows.

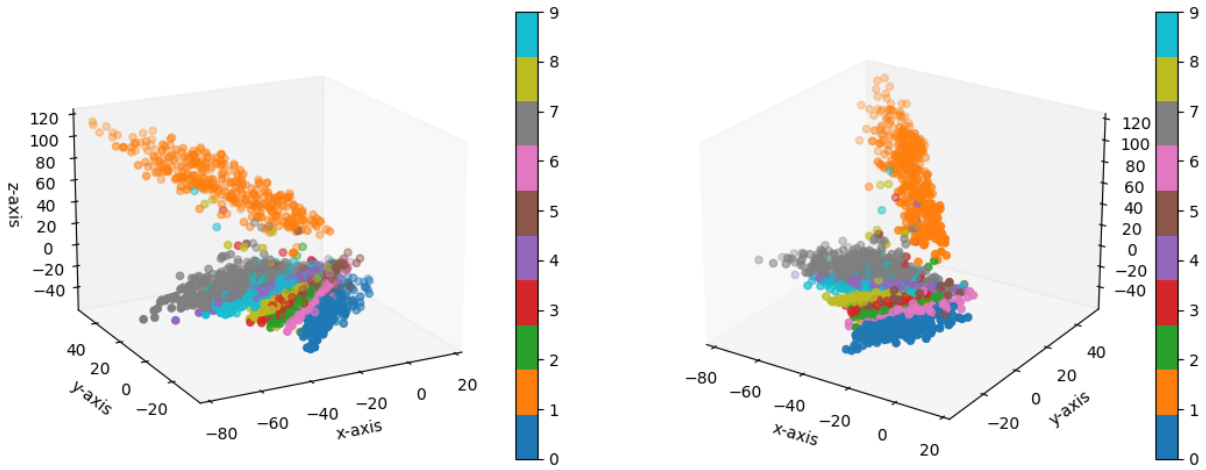


Figure 2.11: The figure illustrates the latent space of the linear autoencoder with bottleneck $n_b = 3$ optimized with an AMSGrad optimizer from two different perspectives. Each dot is one encoded image of a digit. The color and the corresponding color map represent the digit that was encoded.

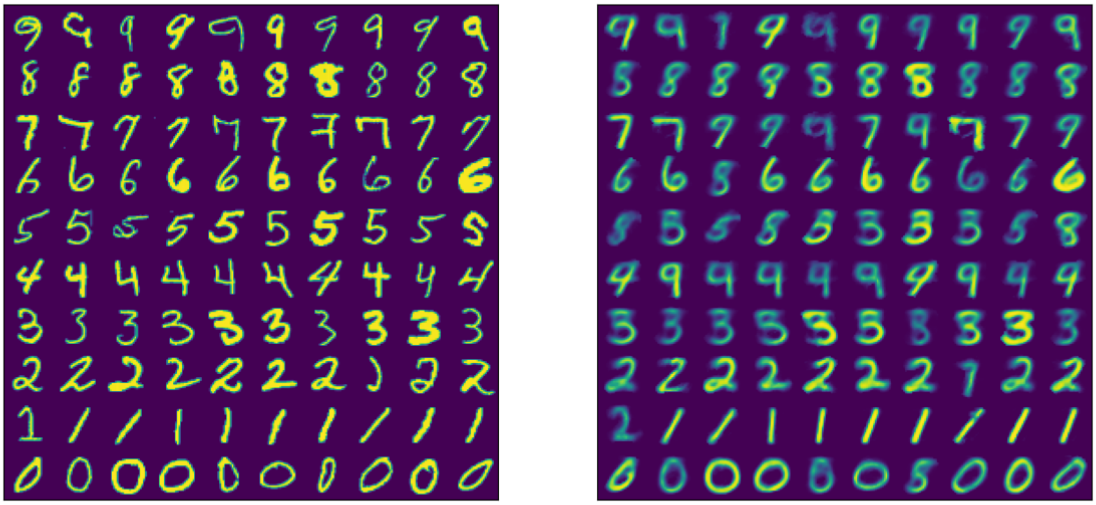


Figure 2.12: On the left side, the figure illustrates 100 original digits from the MNIST dataset. On the right side, the figure illustrates the same digits after feeding them through the linear autoencoder with bottleneck $n_b = 3$ optimized with an Adam optimizer.

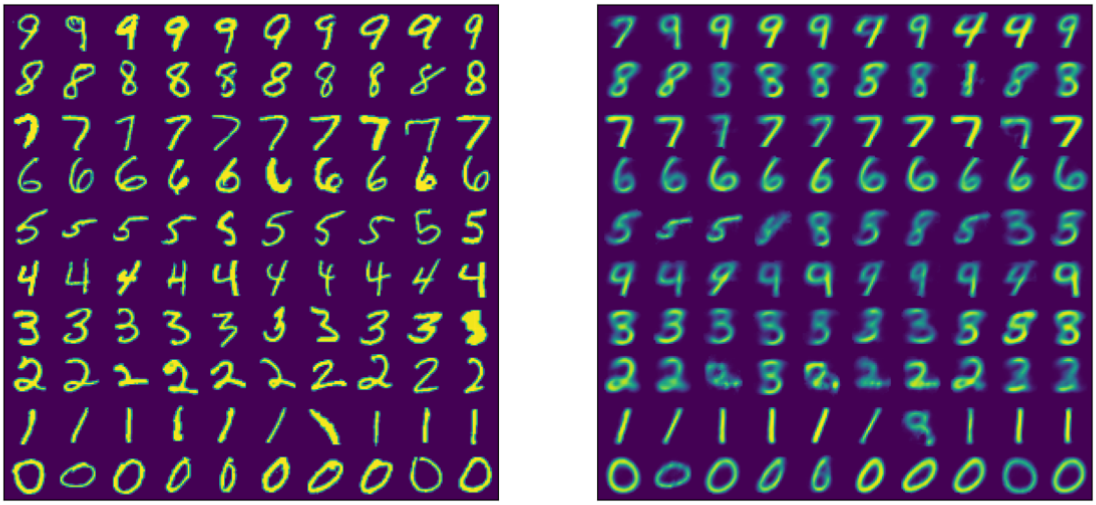


Figure 2.13: On the left side, the figure illustrates 100 original digits from the MNIST dataset. On the right side, the figure illustrates the same digits after feeding them through the linear autoencoder with bottleneck $n_b = 3$ optimized with an AMSGrad optimizer.

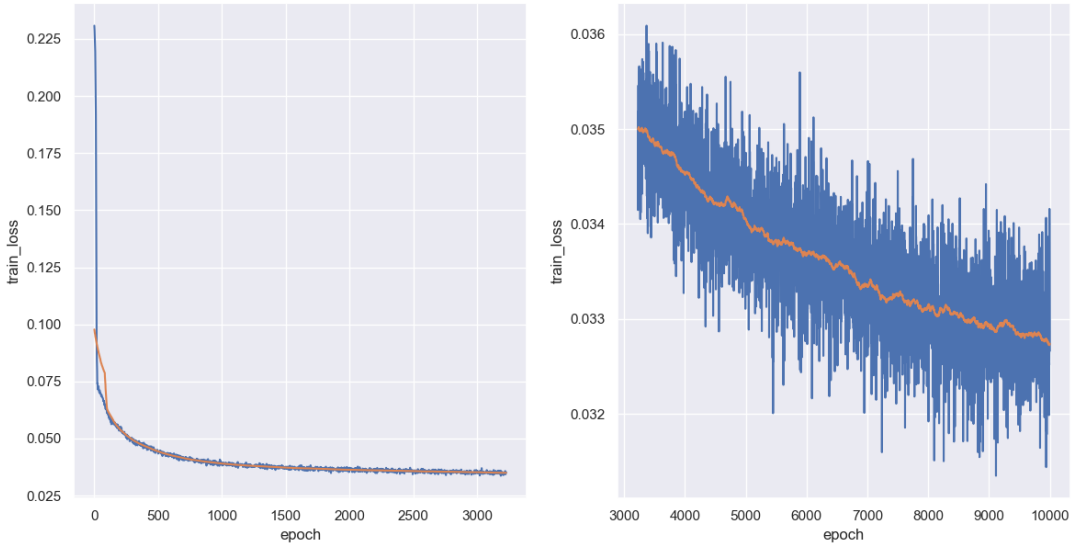


Figure 2.14: The figure illustrates the training progresses of the linear autoencoder with bottleneck $n_b = 3$ optimized with an Adam optimizer with epochs on one axis and corresponding training loss on the other axis.On the left side we see the first 3.500 epochs and on the right side the following epochs until 10.000. The blue line represents the loss in each epoch and the orange line represents the moving average over 100 epochs to point out the trend of the training progress.

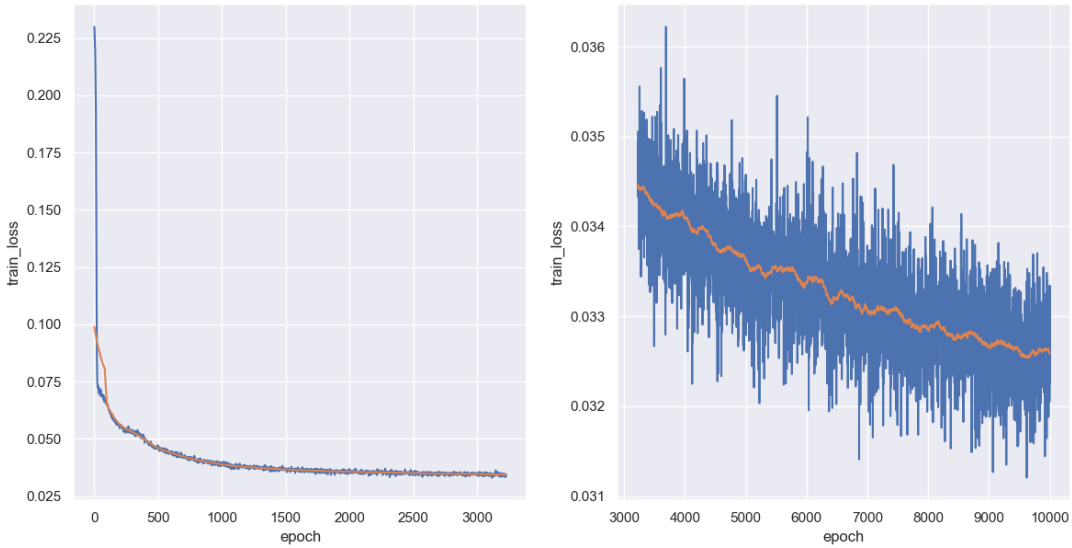


Figure 2.15: The figure illustrates the training progresses of the linear autoencoder with bottleneck $n_b = 3$ optimized with an AMSGrad optimizer with epochs on one axis and corresponding training loss on the other axis.On the left side we see the first 3.500 epochs and on the right side the following epochs until 10.000. The blue line represents the loss in each epoch and the orange line represents the moving average over 100 epochs to point out the trend of the training progress.

Definition 2.3.6. Let Θ be a parameter space, $L \in \mathbb{N}$ and $d_1, \ldots, d_L \in \mathbb{N}$ be the number of channels as well as $(M_1, N_1), \ldots, (M_L, N_L) \in \mathbb{N}^2$ be the resolution of the pixel domain in the i-th layer, where $M_1 \leq \ldots \leq M_L$ and $N_1 \geq \ldots \geq N_L$. Furthermore, let $\widehat{\varphi}$ be an arbitrary activation function. Then a decoding neural network, where each layer $H_1, \ldots, H_L$ is defined as

$$H_i(\psi) = \widehat{\varphi}(T_{\text{conv},i}\, \psi + b_i), \qquad \psi \in \Psi_{d_i,\Omega_i}, i \in \{1, \ldots, L\},$$

where $\theta_i = (T_{\text{conv},i}, b_i) \in \Theta$ are the parameters and Ψ_{d_i,Ω_i} the image domain of the i-th layer, see Definition 1.4.14. Such a decoding neural network is called a **convolutional decoder**.

As we have seen in Lemma 2.1.3, we can plug the convolutional encoder and the convlutional decoder together, as long as the output dimension of the first matches the input dimension of the latter. This gives us the following architecture.

Definition 2.3.7. Let f_e and f_d be a convolutional encoder and a convolutional decoder. Then a **convolutional autoencoder** f_{conv} is defined as the composition

$$f_{\text{conv}} \coloneqq f_d \circ f_e.$$

Now, lets define a specific example of a convolutional autoencoder and afterwards take a look at its performance with some tangible visualisations.

In contrast to the linear autoencoder, the convolutional autoencoder architecture does not allow us to visualise the latent space as easily. The reason is that we designed our two linear autoencoders, each with distinct bottleneck dimensions of 2 and 3, respectively. This choice

Algorithm 5 Convolutional Autoencoder

Let the input and output dimensions be $(M_i, N_i, d_i), (M_o, N_o, d_o) \in \mathbb{N}^{2\times 1}$, where (M_j, N_j) denotes the resolution of the image domain and d_j the amount of channels in the j-th layer. Furthermore, let the convolutional encoder and the convolutional decoder have k hidden linear layers.
Let the chosen optimizer be AMSGrad with a learning rate $\gamma > 0$ and the chosen loss function be the MSE loss function. Then the training of a convolutional autoencoder looks as follows.

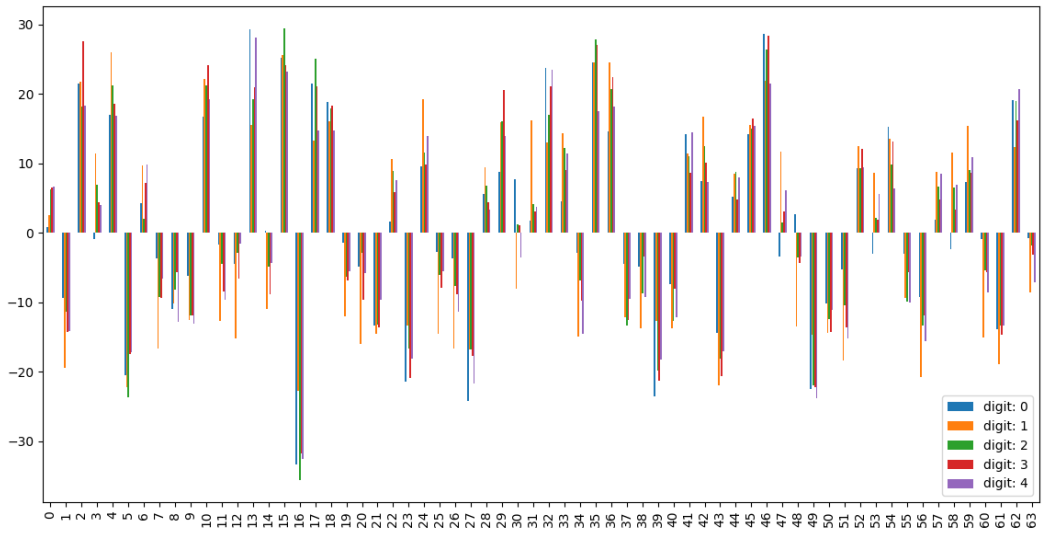
Require: $\gamma \leftarrow 3 \times 10^{-4}$

```
for epoch in epochs do
    for image in batch do
        encoded = encoder(image)              ▷ Encode the image onto latent space.
        reconstructed = decoder(encoded)      ▷ Decode the encoded image.
        loss = MSE(reconstructed, image)      ▷ Compare the output to the input.
        optimization(loss, γ)                 ▷ Perform an optimization step.
    end for
end for
```

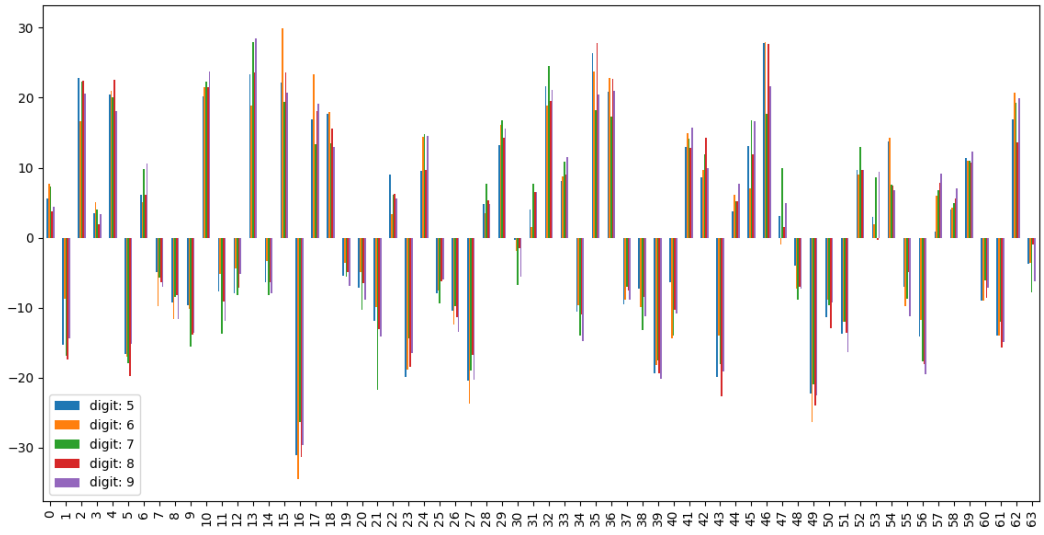
allowed us straightforward visualisation of the latent space, given the single channel nature of our data, which was not altered in the course of computation. Unfortunately, the convolutional neural network does alter the amount of channels in the course of computation, such that we encode the data onto a resolution of one single pixel that comprises of 64 channels and therefore, we have to visualise 64 channels.
We still came up with an idea of how to visualise the encoded representation, where we plot each channel in a separate bar in a bar plot, see Figure 2.16. It is to be read in the following way: each digit receives an own representation in the latent space, that can be uniquely described by the composition of activations in each channel. Here, by activation we mean the value that each channel has. We see that most of the channels have quite different average values and so can be distinguished in that way.
However, what is highly interesting to highlight here is that although the magnitude of the values might differ quite significantly, their signs match most of the time.

Another interesting result that we want to highlight here is the quality of the reconstructions produced by the convolutional autoencoder, which we illustrate in Figure 2.17. We depict them analogously to the linear autoencoder, where we take ten images for each of the ten digits (shown on the left side) and feed them into the convolutional autoencoder. The autoencoder then generates a reconstruction for each of the 100 images (shown on the right side). Every single reconstructions is sharp enough to be recognised as their original digit, which is a huge improvement compared to the reconstructions of the linear autoencoder, see e.g. Figure 2.12 or Figure 2.13. This result clearly shows the fact that convolutional neural networks perform much better than linear neural networks, which is a well known fact in the realm of Machine Learning.

Lastly, we want to take a quick look at the training progress of the convolutional autoencoder, see Figure 2.18. We highlight that compared to the training progress of the linear autoencoder, see e.g. Figure 2.9, the training loss is much smaller. The convolutional autoencoder achieves a training loss of roughly 0.003 after 10.000 epochs, where the linear autoencoder reaches a training loss of roughly 0.038 after the same amount of epochs. Therefore, the convolutional autoencoder performs better by magnitudes. Another fact that is worth to mention is



This figure illustrates the average activation value of each of the 64 channels in the latent space of the convolutional autoencoder for the digits $0, 1, 2, 3$ and 4. The average has been taken over 5.000 different images of the same digit.



This figure illustrates the average activation value of each of the 64 channels in the latent space of the convolutional autoencoder for the digits $5, 6, 7, 8$ and 9. The average has been taken over 5.000 different images of the same digit.

Figure 2.16: The figure illustrates the activations of all 64 channels in the latent space of our trained convolutional autoencoder, which was optimized with an AMSGrad optimizer, see Algorithm 5.

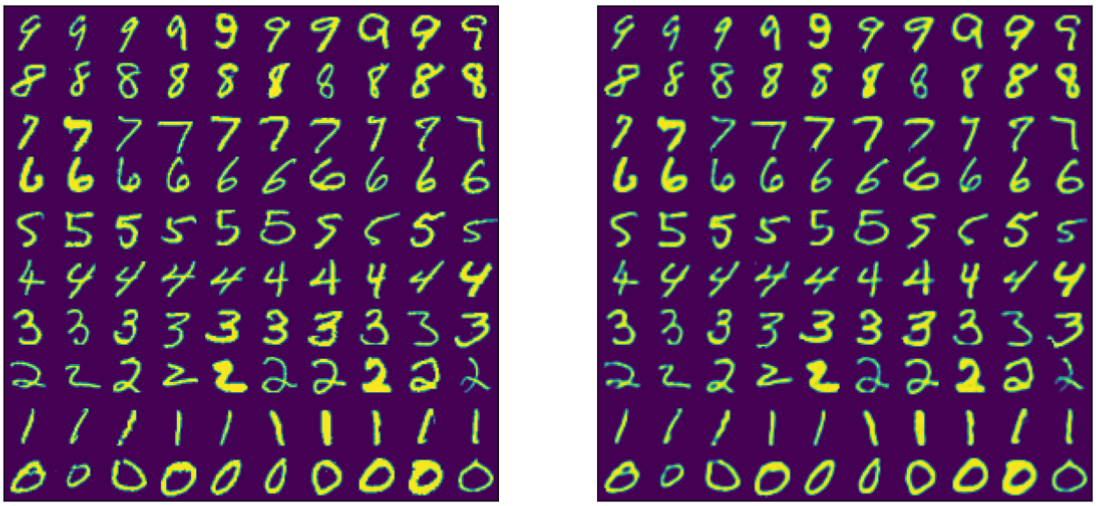


Figure 2.17: On the left side, the figure illustrates 100 original digits from the MNIST dataset. On the right side, the figure illustrates the same digits after feeding them through the convolutional autoencoder which was optimized with an AMSGrad optimizer.

that the training progress of the convolutional autoencoder is much smoother than the training progress of the linear autoencoders. This we can see in the right charts of the Figure 2.18 and e.g. Figure 2.8, respectively. The training loss of the convolutional autoencoder oscillates far less than the training loss of the linear autoencoder.

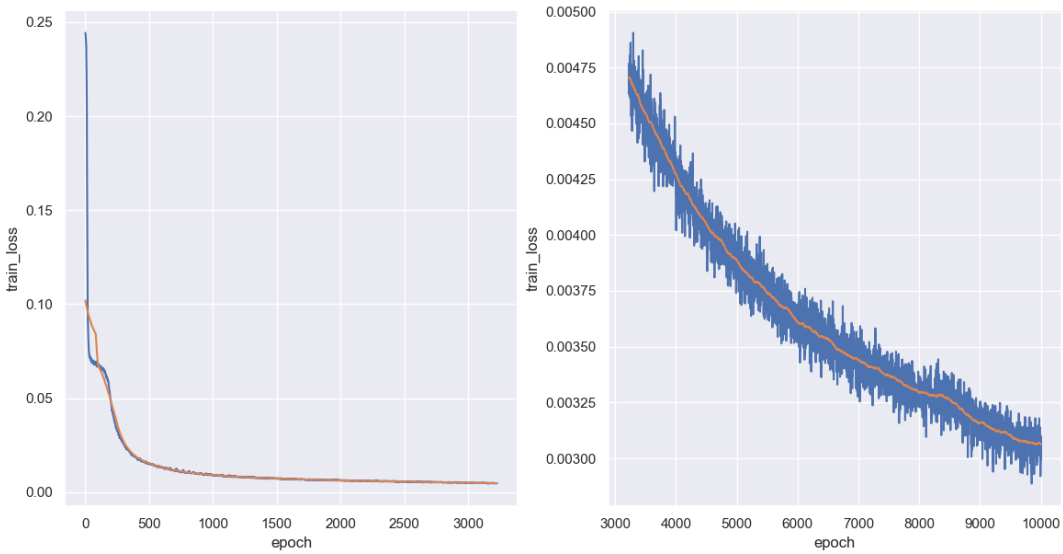


Figure 2.18: The figure illustrates the training progresses of the convolutional autoencoder optimized with an AMSGrad optimizer with epochs on one axis and corresponding training loss on the other axis. On the left side we see the first 3.500 epochs and on the right side the following epochs until 10.000. The blue line represents the loss in each epoch and the orange line every 300 epochs to point out the trend of the training progress.

Chapter 3

Variational Autoencoders

Variational autoencodering neural networks, first introduced by Kingma and Welling in the year 2013, see [7], differently to ordinary autoencoding neural networks that we already mentioned to be discriminative models, are so called generative models, see [1, Chapter 5]. This means that instead of trying to estimate the conditional distribution of $y|x$, where y is a predicted label to an observation x, variational autoencoders attempt to capture the entire probability distribution of Y. This is very interesting for multiple reasons, since this means that we can simulate and anticipate the evolution of the model output. Hence, we could generate new data based on the captured probability distribution. This will be our ultimate goal in this chapter, considering applications at last.

3.1 Probabilistic foundations

Before diving into the depths of variational autoencoders, we want to begin by laying the essential probabilistic foundations. We already introduced the Bayes formula in Theorem 1.1.11. This will be the fundamental idea in this chapter, since this idea lead to an entire optimization ecosystem. The conceptional idea of this ecosystem is to iteratively update ones belief of the knowledge one possesses about the data. This means that one first assumes some kind of knowledge about the data, e.g. a probability distribution. Then by considering a sample (or a batch of samples) of the data, update the assumed probability distribution. Upon repeating this process, one iteratively updates the knowledge about the data until it is adapted to the data. This optimization setting is commonly known as Variational Bayes or Variational Inference. There are some fundamental quantities such as evidence which as the name suggests, describes the observations that we can witness; prior, which describes our belief about the data before considering new data; likelihood, which conceptionally describes how well our current belief describes the data and lastly, the posterior which as the name suggests as well, describes our belief about the data after considering the newly observed evidence. These quantities we will now introduce formally.
As already mentioned, in contrast to ordinary autoencoding neural networks which we discussed in Chapter 2 variational autoencoding neural networks attempt to capture the entire probability distribution of Y. So the emerging question is how can we infer the probability distribution of Y, if we only have the observations x given? We assume that the single data points x_i are not independent - what is no grave assumption, since the observations are assumed to have a reason to be shaped the way they are (e.g. having the same underlying probability distribution). This

reason, since it is unknown or *latent* to us, we will try to consider in more detail. In order to model this hidden cause, we introduce another random variable Z, a so called *latent variable* and thus by marginalizing over the joint distribution $\mathrm{P}_{X,Z}$ of X and Z obtain the marginal distribution

$$\mathrm{P}_X(x) = \int \mathrm{P}_{X,Z}(x,z)dz = \int \mathrm{P}_{X|Z}(x|z)\,\mathrm{P}_Z(z)dz = \mathbb{E}_Z\left(\mathrm{P}_{X|Z}(x|z)\right), \tag{3.1}$$

as we considered in Section 1.1. At this point, we want to note that in the Bayesian setting one commonly does not distinguish between latent variables and model parameters, as they are all random variables, whose values we wish to infer.
The probability distribution from equation (3.1) we now want to introduce formally, since it will be of crucial importance in this chapter. However, we first need to introduce the latent variables and the observation variables, which we will do in the following definition.

Definition 3.1.1. Let $(\Omega, \mathcal{A}, \mathrm{P})$ be a probability space. Furthermore, let $X : \Omega \to \mathbb{R}$, $Z : \Omega \to \mathbb{R}$ be random variables with joint probability distribution $\mathrm{P}_{X,Z}$. Then we call X the **observation random variable** and its marginal probability distribution P_X the **evidence**, as well as Z the **latent random variable** and its marginal probability distribution P_Z **prior**.

In Definition 3.1.1 we introduced the evidence, which describes the probability distribution of our observable data. Moreover, we introduced in the same definition the prior, which describes the probability distribution of the latent variables, e.g. the model parameters. With these definitions we can introduce the likelihood.

Definition 3.1.2. Let $(\Omega, \mathcal{A}, \mathrm{P})$ be a probability space, Θ be a parameter space and let X be a observation random variable with known evidence P_X and Z be a latent random variable with prior P_Z. Then we define the **likelihood** as the conditional probability $\mathrm{P}(X|Z)$.
Moreover, lets assume that the evidence P_X has the probability density $p(x;\theta)$, where $\theta \in \Theta$ denotes the parameters of the probability distribution. Then the likelihood can be expressed as the function in the parameters

$$L(\theta|x) = p(x;\theta).$$

Since our ultimate goal is to find a probability distribution that describes our observations as good as possible, we need to define an update rule, how we want to adapt our current knowledge of the data with respect to newly observed samples. This update rule we already mentioned to be called posterior. It considers the new samples and corrects our current belief as follows.

Definition 3.1.3. Let $(\Omega, \mathcal{A}, \mathrm{P})$ be a probability space, X be a observation random variable with known evidence P_X and Z be a latent random variable with prior P_Z. Then we define the **posterior** as the conditional probability $\mathrm{P}(Z|X)$. This conditional probability can be expressed with the help of the Bayesian rule (Theorem 1.1.11) as

$$\mathrm{P}(Z|X) = \frac{\mathrm{P}(X|Z)\,\mathrm{P}_Z(z)}{\mathrm{P}_X(x)}.$$

The next step is to maximize the likelihood (Maximum Likelihood Estimator MLE) and then maximize the posterior (Maximum A Posteriori (MAP). This would conclude one iteration of the Bayesian learning setting (I guess??). BUT: what do we need ELBO for??

FROM HERE ON THE TEXT IS NOT REWORKED YET

Firstly, we want to define the specific structure of variational autoencoding neural networks and their key differences to ordinary autoencoding neural networks. As already motivated, variational autoencoders are generative models. This means, that they intend to capture the entire probability distribution of the observation space Y. Hence, they are statistical models and depend on certain probabilistic approaches.

In order to compute the gradient of a variational autoencoding neural network, we need to determine a way to quantify the distance between probability distributions. A popular measure for this case is the so called Kullback-Leibler divergence. It assesses the dissimilarity between two probability distributions over the same random variable X.

Definition 3.1.4. Let X be a random variable and p, q two probability density functions over X. Then, the **Kullback-Leibler divergence** is defined as

$$D_{\mathrm{KL}}(p \parallel q) = \int p(x) \log \left(\frac{p(x)}{q(x)} \right) dx. \tag{3.2}$$

If we consider the Kullback-Leibler divergence with regard to a dataset consisting of observed data samples, we can write equivalently.

Definition 3.1.5. Let X be a random variable and p, q two probability density functions over X. Furthermore, let $D = \{x_i\}_{i=i}^{L}$ be an unsupervised dataset of length $L \in \mathbb{N}$.
Then, the **Kullback-Leibler divergence** is defined as

$$D_{\mathrm{KL}}(p \parallel q) = \sum_{i=1}^{L} p(x_i) \log \left(\frac{p(x_i)}{q(x_i)} \right). \tag{3.3}$$

Definition 3.1.6. Let X be a random variable and p a probability density function over X. Then, the nonnegative measure of the expected information content of X under correct distribution p, defined as

$$\mathcal{H}(X) = -\int p(x) \log p(x) dx = \mathbb{E}_p \left[-\log p(x) \right], \tag{3.4}$$

is called **entropy** of p.

Definition 3.1.7. Let X be a random variable and p, q two probability density functions over X. Then, the nonnegative measure of the expected information content of X under incorrect distribution q, defined as

$$\mathcal{H}_q(X) = -\int p(x) \log q(x) dx = \mathbb{E}_p \left[-\log q(x) \right], \tag{3.5}$$

is called **cross-entropy** between p and q.

Remark 3.1.8. Let X be a random variable and p, q probability density functions over X. Then the Kullback-Leibler divergence between p and q can be written as follows

$$D_{\mathrm{KL}}(p \parallel q) = \mathcal{H}_q(p) - \mathcal{H}(p),$$

where $\mathcal{H}_q$ and $\mathcal{H}$ denote the cross-entropy and entropy, respectively.

3.2 Training of Variational Autoencoders

3.3 Applications

Finally, we want to consider some applications of variational autoencoders, similar to Section 2.3. As discussed in Section 3.2, in the course of training we want to minimize the risk function, iteratively. In order to do so we can apply some kind of training algorithm proposed in Section 1.3. Our goal will be to visualise the latent space of a variational autoencoder and show how its reconstruction capability looks like. Furthermore, we will depict the training progress of each model we train. Lastly, we came up with an idea of our own of how to improve the reconstruction capability of the variational autoencoder.

Bibliography

[1] L. P. Cinelli, M. A. Marins, E. A. B. Da Silva, and S. L. Netto, *Variational Methods for Machine Learning with Applications to Deep Networks*, Springer, 2021.

[2] D. Foster, *Generative Deep Learning*, O'Reilly Media, Inc., 2022.

[3] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, MIT Press, 2016.

[4] J. Jahn et al., *Vector Optimization*, Springer, 2009.

[5] L. V. Kantorovich and G. P. Akilov, *Functional Analysis*, Elsevier, 2016.

[6] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, International Conference on Learning Representations, (2014).

[7] D. P. Kingma and M. Welling, *Auto-encoding variational bayes*, International Conference on Learning Representations, (2013).

[8] A. Klenke, *Probability Theory: a Comprehensive Course*, Springer, 2013.

[9] Y. Lei, T. Hu, G. Li, and K. Tang, *Stochastic gradient descent for nonconvex learning without bounded gradient assumptions*, IEEE Trans. Neural Netw. and Learn. Syst., 31 (2019), pp. 4394–4400.

[10] C. Lemaréchal, *Cauchy and the gradient method*, Doc. Math. Extra, 251 (2012), p. 10.

[11] H. B. McMahan and M. Streeter, *Adaptive bound optimization for online convex optimization*, Conference on Learning Theory, (2010).

[12] D. Meintrup and S. Schäffler, *Stochastik: Theorie und Anwendungen*, Springer, 2006.

[13] N. Mücke and I. Steinwart, *Empirical risk minimization in the interpolating regime with application to neural network learning*, arXiv:1905.10686, (2019).

[14] A. Papoulis and S. U. Pillai, *Probability, Random Variables and Stochastic Processes*, McGraw Hill, 2002.

[15] S. J. Reddi, S. Kale, and S. Kumar, *On the convergence of adam and beyond*, International Conference on Learning Representations, (2019).

[16] D. Saad, *On-line Learning in Neural Networks*, Cambridge Univ. Press, 2009.

[17] G. Simmons, *Calculus With Analytic Geometry*, McGraw Hill, 1995.

[18] S. Sra, S. Nowozin, and S. J. Wright, *Optimization for Machine Learning*, MIT Press, 2012.

[19] I. Steinwart and A. Christmann, *Support Vector Machines*, Springer, 2008.

[20] G. Turinici, *The convergence of the stochastic gradient descent (sgd): a self-contained proof*, arXiv:2103.14350, (2021).

Stuttgart, November 3, 2023