

Master's thesis

November 22, 2023

On Variational Autoencoders: Theory and Applications

Maksym Sevkovych

Registration number: 3330007

In collaboration with: DevDuck 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, 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 Variational Autoencoders.

By dissecting the architecture of Variational Autoencoders, 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. Practical applications of Variational Autoencoders extend from anomaly detection to image generation. However, we will focus on the latter in the course of this thesis.

Contents

Introduction	4
1 Preliminaries	5
1.1 Probability and Statistics	5
1.2 Neural Networks	8
1.3 Training of Neural Networks	11
1.4 Neural Networks in Computer Vision	38
2 Autoencoders	43
2.1 Conceptional ideas	43
2.2 Training of Autoencoders	47
2.3 Applications	48
3 Variational Autoencoders	65
3.1 Probabilistic foundations	65
3.2 Variational Bayes meets Autoencoders	69
3.3 Applications	76
References	95
Deutsche Zusammenfassung	97

Introduction

In this thesis, our main goal is to understand Variational Autoencoders from a mathematical perspective. Since Variational Autoencoders bring together Bayesian statistics and Deep Learning, we embark on our journey by exploring the basics - covering probability theory, statistics and statistical learning theory, as well as neural networks along with how to optimize them.

After considering the fundamental basics, we continue our expedition by introducing regular Autoencoders. To get a solid grasp, we create various implementations of Autoencoders in different training approaches, which we introduced in the preliminary chapter. We consider all implementations on the MNIST dataset, which contains images of handwritten digits. This hands-on approach helps us build a strong foundation before moving on to Variational Autoencoders.

The shift from Autoencoders to Variational Autoencoders is a crucial part of this thesis. We take a close look at what sets them apart and consider the theory of Variational Autoencoders in depth, subsequently. Afterwards, we explore practical applications. Through implementing and training multiple Variational Autoencoders, we stumble upon some interesting findings. One notable discovery is that we can optimize the common approach of training Variational Autoencoders under certain conditions.

So, in a nutshell, this thesis aims to be a self-contained guide to Variational Autoencoders in a mathematical way, which combines Bayesian statistics with Deep Learning. Along the way, we share insights that could tweak the usual practices in training Variational Autoencoders, which hopefully makes this work not only interesting from a theoretical perspective, but also gives it some sort of practical relevance.

Chapter 1

Preliminaries

In order to comprehend the topic of variational autoencoders or even autoencoders in general, we need to consider a couple of preliminary ideas first. Those ideas consist mainly of concepts considering probability theory and statistics, statistical learning theory, neural networks and their optimization, which is often referred to as training of neural networks. In this chapter, we will begin with fundamental definitions in probability theory and afterwards, we will introduce quantities we need for the statistical learning setting. Then our goal is to formulate neural networks in a mathematical way and consider a handful of useful operations that neural networks are capable of doing. Moreover, 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 orientate ourselves towards [14, 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 $f : \Omega \rightarrow \bar{\mathbb{R}} := \mathbb{R} \cup \{-\infty, \infty\}$ 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

$$\begin{aligned} f \odot \mu : \mathcal{A} &\rightarrow \bar{\mathbb{R}}, \\ (f \odot \mu)(A) &:= \int_A f d\mu, \end{aligned}$$

as the **measure with density f (with regard to μ)**. The function $f : \Omega \rightarrow \bar{\mathbb{R}}_+$ 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 actually exists. In order to address this, we need to introduce a property which in a sense compares two measures on their null sets. The property we introduce in the following is called absolute continuity.

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

$$\mu(A) \implies \nu(A),$$

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. [14, 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 (Ω, \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 $(\Omega, \mathcal{A}, \mathbb{P})$ be a measure space. If $\mathbb{P}(\Omega) = 1$ holds, then we call \mathbb{P} a **probability measure** and $(\Omega, \mathcal{A}, \mathbb{P})$ 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. [14, Definition 1.13].

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

$$\begin{aligned} F_{\mathbb{P}} : \mathbb{R} &\rightarrow [0, 1], \\ x &\mapsto F_{\mathbb{P}}(x) := \mathbb{P}((-\infty, x)), \end{aligned}$$

the **distribution function of \mathbb{P}** . We will write shortly **distribution of \mathbb{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 $x \rightarrow -\infty$ and to 1 for $x \rightarrow \infty$. 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 $F : \mathbb{R} \rightarrow \mathbb{R}$ be a monotonously increasing function that is continuous on its right side. If for the function F holds

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

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. [14, Definition 1.42] or [9, Definition 1.102] for more details.

Definition 1.1.7. Let $(\Omega_1, \mathcal{A}_1, \mathbb{P})$ be a probability space and $(\Omega_2, \mathcal{A}_2)$ be a measure space. A function $X : \Omega_1 \rightarrow \Omega_2$ 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 $n = 1, 2, \dots$, we refer to X as an **n -dimensional real-valued random variable**. If $\Omega_2 = \mathbb{R}$, we refer to X as a **numerical random variable**.

Lastly, for $A_2 \in \mathcal{A}_2$ we denote $\{X \in A_2\} := X^{-1}(A_2)$ and $\mathbb{P}(X \in A_2) := \mathbb{P}(X^{-1}(A_2))$.

Lastly, we need to consider how the distribution of a random variable looks like. This we want to do in the following definition, which is inspired by [9, Chapter 1.5].

Definition 1.1.8. Let $(\Omega, \mathcal{A}, \mathbb{P})$ be a probability space. Furthermore, let X be a random variable. Then we define:

- (i) The probability measure $\mathbb{P}_X := \mathbb{P} \circ X^{-1}$ is called the **distribution** of X .
- (ii) For a real-valued random variable X , the map $F_X : x \mapsto \mathbb{P}(X \leq x)$ is called the **distribution function of X** (or, more accurately, of P_X). We write $X \sim \mu$ if $\mu = \mathbb{P}_X$ and say that X has distribution μ .
- (iii) A family $(X_i)_{i \in I}$ of random variables is called **identically distributed** if $\mathbb{P}_{X_i} = \mathbb{P}_{X_j}$ for all $i, j \in I$.

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 [22, 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 in 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 pointwise 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 1.1.9. 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** $\mathcal{L} : X \times Y \times \mathbb{R}^n \rightarrow [0, \infty)$ as a measurable function that compares a true value $y \in Y \subset \mathbb{R}^n$ to a predicted value $\hat{y} = t \in \mathbb{R}^n$ at some $x \in X$ in a suitable way.

The recently introduced loss function from Definition 1.1.9 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 [4, Chapter 4.3].

Example 1.1.10. Let $y \in Y := \mathbb{R}$ 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 $y \in Y := \mathbb{R}^n$ and $t \in \mathbb{R}^n$ with $n > 1$, where we denote $y = (y_1, \dots, y_n)$ and $t = (t_1, \dots, t_n)$. 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 1.1.11. Let X and Y be input and output spaces, \mathcal{L} be a supervised loss function and \mathbb{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

$$\mathcal{R}_{\mathcal{L}, \mathbb{P}}(f) := \int_{X \times Y} \mathcal{L}(x, y, f(x)) d\mathbb{P}(x, y).$$

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 1.1.12. Let X, Y be arbitrary input and output spaces, $D = ((x_1, y_1), \dots, (x_N, y_N))$ be a dataset consisting of $N \in \mathbb{N}$ 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

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

We will write $\mathcal{R} := \mathcal{R}_{\mathcal{L}, D}$, 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 1.1.13. *Let \mathcal{L} be a convex loss function and D be an arbitrary dataset of length $N \in \mathbb{N}$. Then the empirical risk function $\mathcal{R}_{\mathcal{L}, D}$ is also convex.*

Proof. We begin by considering the Definition 1.1.12 of the empirical risk

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

Since \mathcal{L} is assumed to be a convex loss function, so 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. \square

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 [15, 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 on the use-case what exactly is demanded from the activation function. Therefore, we define activation functions as generally as possible.

Definition 1.2.1. A non-constant function $\varphi : \mathbb{R} \rightarrow \mathbb{R}$ 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 [4, 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: $\varphi(t) = \frac{1}{1 + e^{-t}}.$

Now, having introduced activation functions we can introduce neurons.

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

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

We call $\theta := (w, b)$ 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 $\varphi : \mathbb{R} \rightarrow \mathbb{R}$ be an activation function and $W \in \mathbb{R}^{m \times k}, b \in \mathbb{R}^m$. Then a function $H : \mathbb{R}^k \rightarrow \mathbb{R}^m$ is called **φ -layer** of width m with **weights** W and **biases** b if for all $i = 1, \dots, m$ 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 $\hat{\varphi} : \mathbb{R}^k \rightarrow \mathbb{R}^m$ as the component-wise mapping of $\varphi : \mathbb{R} \rightarrow \mathbb{R}$, meaning $\hat{\varphi}(v) = (\varphi(v_1), \dots, \varphi(v_k))$, we can write the φ -layer $H : \mathbb{R}^k \rightarrow \mathbb{R}^m$ as

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

In the following, we will denote the weights W and biases b as **parameters of the neural layer** $\theta := (W, b)$. Moreover, we will not make the distinction between $\hat{\varphi}$ and φ .

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

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

Then we define a **φ -deep neural network** of depth L with parameters θ as

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

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

We depict a neural network in Figure 1.1. Each node represents a neuron and each line represents a connection between two neurons.

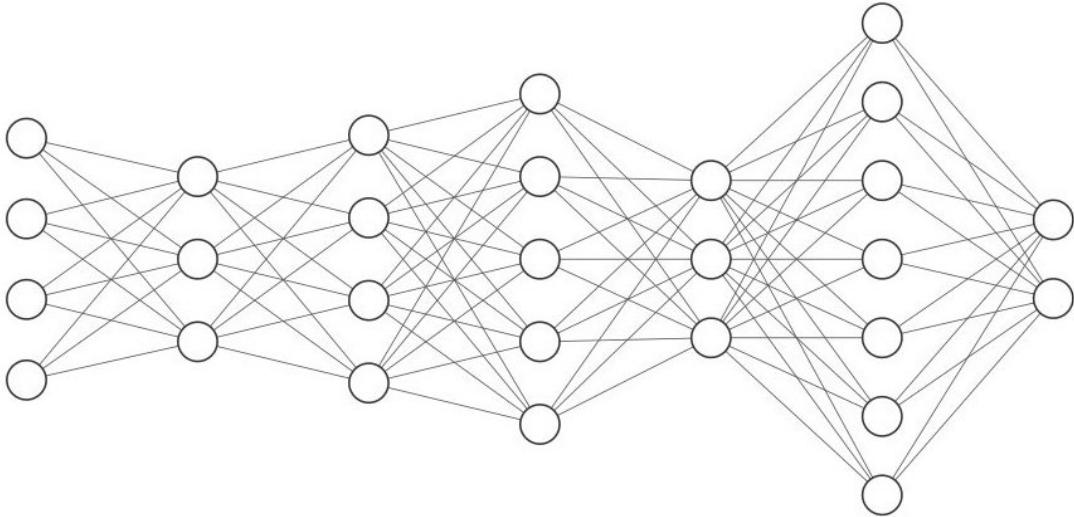


Figure 1.1: A neural network with input $x \in \mathbb{R}^4$ and output $y \in \mathbb{R}^2$. The five hidden layers have dimensions 3, 4, 5, 3 and 7 respectively. The graphic was generated with <http://alexlenail.me/NN-SVG/index.html>

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 throughout 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 $Y \subseteq \mathbb{R}^n$, which we will refer to as **input** and **output space**. Furthermore, let $D = \{(x_1, y_1), \dots, (x_N, y_N)\}$ be a **dataset** of length $N \in \mathbb{N}$. 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 ($i \in \{1, \dots, L - 1\}$) share the same activation function. The last layer, often referred to as output layer, usually has a different activation function. In some cases this activation function may even be the identity function.

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. The most common approach to train a neural network is to iteratively find the gradient - the direction of greatest ascent - of the risk function, and afterwards perform an optimization step in the opposite direction, thus the direction of the steepest descent. The method of the steepest descent dates way back to the early 19th century and was introduced by Augustin L. Cauchy in the year 1847. For a brief overview we refer to [11]. This method is immensely popular and thus, is very wide spread in literature. We will mostly draw inspiration from [6, Chapter XV]. For in-depth explanation, we kindly refer to 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 from 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

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

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

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

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 the function f **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 $f : X \rightarrow \mathbb{R}$ 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 n -th iterate $x^{(n)}$, then we define the $(n+1)$ -th iterate by

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

where z_{n+1} denotes the direction of steepest descent at $x^{(n)}$. The numerical parameter γ_{n+1} can be found in various ways, e.g. [6, Chapter XV]. However, we want to approach it by finding the optimal step size analytically.

We specify a sequence of positive numbers (γ_n) such that $\gamma_n \rightarrow 0$ and $\sum_{n \geq 1} \gamma_n = \infty$. Furthermore, we assume the z_n to be normalized for all $n \in \mathbb{N}$, i.e. $\|z_n\| = 1$. Hence, the step size at the n -th

iteration is γ_n . We do this in order to avoid making too great steps, which could possibly 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 bounded from below by $m > 0$ and bounded from above by $M > 0$, i.e. the following equations hold

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

which implies that U is an injective operator.

Now, let's 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 F in equation (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 equation (1.6). Then the gradient descent algorithm applied to the functional F with arbitrary initial iterate $x^{(0)} \in \mathcal{H}$ looks like*

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

where $z_n \in \mathcal{H}$ and $\gamma_n > 0$ look like

$$z_n = Ux^{(n-1)} - y, \quad \text{and} \quad \gamma_n = \frac{\langle z_n, z_n \rangle}{\langle Uz_n, z_n \rangle}.$$

Moreover, we note that we choose an optimal step size γ_n 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-valued 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 $\alpha > 0$.

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_n and z_n can be determined for all $n > 1$ and hence, all assertions from the theorem are proven. \square

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*

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

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

Proof. First, we rewrite the equation (1.5) for a $k > 0$ to

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

We chose the numerical factor k in a way, that the operator $T = \text{id} - kU$ has the smallest possible norm, where we denote id as the identity operator. Since we assumed U to be bounded from below by m and bounded from above by M , the operator T needs to be bounded from below by $1 - km$ and bounded from above by $1 - kM$, where the minimal norm will occur if we consider the sum of the upper bound and lower bound to be equal to zero, see [6, Chapter XV Theorem 2].

$$\begin{aligned} &1 - km + 1 - kM = 0, \\ &\iff 1 - km = -(1 - kM). \end{aligned}$$

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

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

Moreover, applying equation (1.11) and rewriting it gives us

$$x'_1 = Tx^{(0)} + ky = x^{(0)} - k(Ux^{(0)} - y) = x^{(0)} - kz_1. \quad (1.14)$$

Let us introduce the operator $V = U^{1/2}$, which is the root operator of U , see e.g [6, Theorem V.6.2]. This is the unique operator that fulfils the equation $V^2 = U$. Furthermore, we should note 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 equation (1.7), which gives us

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

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 equation (1.15) to

$$\|V(x^{(1)} - x^*)\| \leq \|V(x'_1 - x^*)\|. \quad (1.16)$$

Since equation (1.7) was the rewritten equation (1.5), we can use the definition of T to obtain

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

Subtracting this equation from equation (1.14) gives us

$$x'_1 - x^* = T(x^{(0)} - x^*).$$

Applying the operator V on both sides gives us

$$V(x'_1 - x^*) = VT(x^{(0)} - x^*). \quad (1.17)$$

With the easy calculation

$$VT = V(\text{id} - kU) = V - kVU = V - kVVV = (\text{id} - kU)V = TV,$$

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

$$V(x'_1 - x^*) = TV(x^{(0)} - x^*).$$

Considering the norm leads to

$$\begin{aligned} \|V(x'_1 - x^*)\| &= \|TV(x^{(0)} - x^*)\| \leq \|T\| \|V(x^{(0)} - x^*)\| \\ &= \frac{M-m}{M+m} \|V(x^{(0)} - x^*)\|. \end{aligned}$$

Therefore, with inequality (1.16) we have

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

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

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

which leads to the bound

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

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 [6, Theorem V.6.2] (which regards the spectrum S_U of the operator U) the following equations hold

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

Combining the above results, we receive

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

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

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

Plugging this result into equation (1.18), gives us

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

This is exactly the inequality asserted in the theorem. \square

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. 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 1.1.10.

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

$$f(x) = \varphi_\theta(wx + b) \quad x \in \mathbb{R}, \quad (1.19)$$

where $w \in \mathbb{R}$ is the networks weight and $b \in \mathbb{R}$ is the networks bias. Hence, we denote the networks parameters as $\theta = (w, b)$.

Since we are interested in computing the gradient of the empirical risk function, we want to formulate it explicitly.

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

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

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

Now, lets consider how the gradient of (1.20) with regard to the parameters θ of the neural network f_θ looks like. In mathematical notation, this means $\partial \mathcal{R}_{\mathcal{L},D}(f_\theta)/\partial \theta$.

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. [20, Chapter 19.6] which states that

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

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

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

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

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

where the derivative of the sigmoid function is $\varphi'(x) = \varphi(x)(1 - \varphi(x))$.

Furthermore, the second part $\partial \mathcal{L}(f_\theta(x), y)/\partial f_\theta(x)$ of the right-hand side (1.22) is

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

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

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

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

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

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 consists of one neuron. Hence for each $i = 1, \dots, L$ holds

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

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

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

where $\theta = (\theta_1, \dots, \theta_L)$ denotes the parameters of the neural network. Since we assumed $b_i = 0$ for all $i = 1, \dots, L$, we can denote the parameters as $\theta = (w_1, \dots, w_L)$.

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 generally 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 $x^{(i)} := H_i(x)$ for all $i = 0, \dots, L$, where $x \in \mathbb{R}^{i-1}$ and $x^{(0)} := x$. With this definition we can write $f_{\theta}(x) = H_L(x^{(L-1)})$ and therefore,

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

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 equation (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

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

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 regard 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 $f : \mathbb{R}^n \rightarrow \mathbb{R}$ and $g : \mathbb{R}^m \rightarrow \mathbb{R}^n$ 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

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

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 [4, Chapter 6.5.2]. 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 an arbitrary size of layers.

The back-propagation algorithm as already motivated, 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 in the entire neural network. However, formulating this idea formally correct is quite messy. Therefore, we want to introduce some quantities beforehand. As we defined in Definition 1.2.4, each layer H_i looks like

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

where the input and output dimensions of the i -th layer are described by d_{i-1} and d_i , respectively. 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 $W^{(i)} = (w_{jk}^{(i)})$, where each of the entries $w_{jk}^{(i)}$ 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

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

such that it is the squared sum over all entries of the output vector, which is the activation vector of the last layer $a^{(L)} := (a_0^{(L)}, \dots, a_{d_L-1}^{(L)})$. 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)

$$\frac{\partial \mathcal{L}(y, a^{(L)})}{\partial a_j^{(L)}} = \frac{\partial \sum_{l=0}^{d_L-1} (a_l^{(L)} - y_l)^2}{\partial a_j^{(L)}} = 2(a_j^{(L)} - y_j). \quad (1.25)$$

For the second term it holds

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

and for the third term it holds

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

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 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 parameters, 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 parameter 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 equation (1.27). Therefore, we can apply exactly the same computation until we reach the input layer. This procedure allows us to compute the gradient $\nabla_\theta \mathcal{L}(a^{(L)}, y)$ component-wise. 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 [21, Chapter 13.3.2], [19, Chapter 4.2] and [23]. We will focus on the latter reference.

Beforehand, we need to consider the empirical risk function, which we defined in Definition 1.1.12. 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

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

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

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

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

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

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.

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

for all $\theta \in \Theta$.

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

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

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. [10], 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 $(\Omega, \mathcal{A}, \mathbb{P})$ 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 defined in equation (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 $n \geq 0$, we denote for readability reasons $\theta^{(n)} := \theta^{(n)}(\omega_0, \dots, \omega_{n-1})$. since it depends on the choices of $\omega_1, \dots, \omega_{n-1}$. Furthermore, we denote d_n as

$$d_n = \mathbb{E} \left(\left\| \theta^{(n)} - \theta^* \right\|^2 \right) := \int_{(X \times Y)^n} \left\| \theta^{(n)} - \theta^* \right\|^2 d\omega_0 \dots d\omega_{n-1}. \quad (1.31)$$

Then for d_{n+1} holds

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

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

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

4. If the sequence (γ_n) meets the conditions

$$\gamma_n \rightarrow 0 \quad \text{and} \quad \sum_{n \geq 0} \gamma_n = \infty, \quad (1.34)$$

then $d_n \rightarrow 0$, that is $\lim_{n \rightarrow \infty} \theta^{(n)} = \theta^*$, where the convergence is the L^2 -convergence of random variables, see e.g. [9, 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

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

First, we consider that

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

what we can bound with the help of the second assumption, equation (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 equation (1.35) and use the first assumption, equation (1.29). This gives us

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

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

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

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 $k \rightarrow \infty$ we obtain

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

or equally

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

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

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

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 $\epsilon_n := \gamma_n B / \mu$, this gives us

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

Therefore, applying inequality (1.37) twice gives us

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

Applying inequality (1.37) k times iteratively leads to

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

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

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

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

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

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

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

which is exactly Assertion 4. \square

Lastly, we want to consider another powerful optimization algorithm that adapts learning rates based on past gradient magnitudes and momenta. Therefore, 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 [17, Chapter 5].

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

$$m_k := \mathbb{E}[X^k] = \int_{\Omega} X d\mathbb{P}.$$

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

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

where we denote $\mu := \mathbb{E}[X] = m_1$.

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

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

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 [14, Theorem 4.21].

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

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

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 [7] or [4, Chapter 8].

Algorithm 1 Adam optimizer

Let D be an arbitrary dataset and $g_t := \nabla_{\theta} \mathcal{R}_{\mathcal{L},D}(f_t(\theta))$ 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 $g_t^2 := g_t \odot g_t$ 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: $f_t(\theta)$: Prediction function with parameters θ at time step t

Require: θ_0 : Initial parameter vector

```

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

However, The Adam optimization algorithm we introduced in Algorithm 1 was later proven to be not converging in certain settings, see e.g. [18]. 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 the AMSGrad algorithm 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 $X : \mathbb{R}^d \rightarrow \mathbb{R}^d$ be a self-adjoint operator. Then we define the **X -projection of y onto \mathcal{F}** as

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

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

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 $x^* \in \mathcal{F}$.

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

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

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

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

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

But since we assumed that $x^* \in \mathcal{F}$, it follows that

$$\min_{x \in \mathcal{F}} \|X^{(1/2)}(x - x^*)\| = 0,$$

which in turn leads to $x' = x^*$ and the assertion holds. \square

Another quantity we need to introduce is the so called regret of an algorithm. It intuitively quantifies how much worse a prediction function performed compared to 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

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

Algorithm 2 Generic Adaptive Method Setup

Let D be an arbitrary dataset, $g_t := \nabla_{\theta} \mathcal{R}_{\mathcal{L}, D}(f_t(\theta))$ 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: $f_t(\theta_t)$: Prediction function with parameters θ_t at time step t
Require: $\theta_1 \in \Theta$: Initial parameter vector

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

$$\begin{aligned} \phi_t(g_1, \dots, g_t) &= g_t, \\ \psi_t(g_1, \dots, g_t) &= \mathbb{1}. \end{aligned}$$

Then the resulting update rule from Algorithm 2 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

$$\phi_t(g_1, \dots, g_t) = (1 - \beta_1) \sum_{i=1}^t \beta_1^{t-i} g_i,$$

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

Then the resulting update rule from Algorithm 2 looks like

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

which is the same as in Algorithm 1 (without the bias-correction, but the argument still holds, see [18]. 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: $f_t(\theta_t)$: Prediction function with parameters θ_t at time step t

Require: $\theta_1 \in \Theta$: Initial parameter vector

- ```

1: $m_0, v_0 \leftarrow 0$ (Initialise 1st and 2nd moment vector)
2: for $t = 1, \dots, T$ do
3: $g_t \leftarrow \nabla_{\theta} \mathcal{R}_{\mathcal{L}, D}(f_t(\theta_t))$ \triangleright Get gradients w.r.t. prediction function at time step t ,
4: $m_t \leftarrow \beta_{1t}m_{t-1} + (1 - \beta_{1t})g_t$ \triangleright Update biased first moment estimate,
5: $v_t \leftarrow \beta_2v_{t-1} + (1 - \beta_2)g_t^2$ \triangleright Update biased second moment estimate,
6: $\hat{v}_t \leftarrow \max\{\hat{v}_{t-1}, v_t\}$ \triangleright Compute bias-corrected first moment estimate,
7: $\hat{V}_t \leftarrow \text{diag}(\hat{v}_t)$ \triangleright Compute bias-corrected second moment estimate,
8: $\theta_{t+1} \leftarrow \Pi_{\Theta, \sqrt{\hat{V}_t}} (\theta_t - \alpha_t m_t / \sqrt{\hat{v}_t})$ \triangleright Update parameters,
9: end for
10: return θ_{t+1} \triangleright Return resulting parameters.

```

The AMSGrad optimizer, defined in Algorithm 3 does indeed converge, as proven in [18, Theorem 4]. We take a quick look at the theorem and its proof. However, for a deeper understanding of the alternative algorithms, we kindly refer to [18], since this would go beyond the scope of this thesis' topic.

First, we need an auxiliary lemma, which we cite from [18, Lemma 4], but since their proof does not align with the proof of the original paper [13, Lemma 3] we will adjust the proof to the original one.

**Lemma 1.3.16.** Let  $\mathcal{S}_+^d$  denote the set of all positive definite  $d \times d$ -matrices and let  $Q \in \mathcal{S}_+^d$ . 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 by defining

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

Hence, we can write

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

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

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

Therefore, it holds that

$$\langle Q(u_1 - z_1), u_2 - u_1 \rangle \geq 0, \quad (1.42)$$

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 inequality (1.41) and inequality (1.42), we receive

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

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

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

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

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

Now, lets consider the inequality

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

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

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

Thus,

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

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  $\hat{u}$  and  $\hat{z}$  we achieved

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

Taking the square root of both sides leads to the assertion.  $\square$

**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  $\Theta$  has bounded diameter  $D_\infty < \infty$  and  $\max_{\theta \in \Theta} \|\nabla_\theta \mathcal{R}_{\mathcal{L}, D}(f_t(\theta))\|_\infty = G_\infty < \infty$  for all  $t \in \{1, \dots, T\}$ . Furthermore, let  $\mathcal{L}$  be a convex loss function. Then for  $\theta_t$  the following bound for the regret holds*

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

where we denote for readability reasons  $g_{1:t} := (g_1, \dots, g_t)$  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

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

Furthermore, with Lemma 1.3.13 it follows that for all  $\theta^* \in \Theta$  holds

$$\Pi_{\Theta, \sqrt{\hat{V}_t}}(\theta^*) = \theta^*.$$

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

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

If we now rearrange the last inequality, we receive

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

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

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

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 1.1.13. 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  $\theta_t$ . The inequality ultimately follows from the definition of the weak subgradient, see e.g. [5, Remark 16.11.].

This we can further bound by the previously considered inequality (1.43).

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

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

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

then this leads ultimately to

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

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

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

Now, considering the fact that  $\beta_{1t} \leq \beta_1$  for all  $t = 1, \dots, T$  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

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

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  $t = 1, \dots, T-1$  as we did for the last time step, we receive

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Since we assumed that the parameter space  $\Theta$  has bounded diameter  $D_\infty$ , we can write as well

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

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

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

where we additionally included the definition of  $\alpha_T$ .  $\square$

## 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 approaches that are 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 consider images from a mathematical perspective.

In order to do so, we need to introduce some quantities first. The first important quantity is a pixel. Intuitively, this is a single point in an image.

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

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

where for all  $i = 1, \dots, d$  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 stated 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  $\Omega$  defined by

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

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, which is often called digital image.

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

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

where each  $\psi_{ij}$  is a pixel with  $d$  channels.

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

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  $\Omega$  be a pixel domain and  $d \in \mathbb{N}$  a number of channels. Then we call the  $\Psi'$ , defined as

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

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

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

where  $\lceil \cdot \rceil$  denotes the ceil function (also known as Gaussian bracket).

*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 ceil function gives us the representation defined in Definition 1.4.1.  $\square$

Having formally defined what an image is, we can consider how a neural network operating on images looks like. In order to do so, 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 usually operate on vectors.

**Lemma 1.4.7.** Let  $\Omega$  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  $\psi$  be a matrix with entries  $\psi_{ij} \in \Psi_d$ , what follows from the Definition 1.4.4 of an image. Hence,  $\psi$  is an  $M \times N$ -matrix.

Define the rows of  $\psi$  by  $\widehat{\psi}_i := (\psi_{i1}, \dots, \psi_{iN})$  for all  $i \in \{1, \dots, M\}$ . Then the matrix representation of the picture  $\psi$  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 vector

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

$\square$

Lemma 1.4.7 allows us to feed images into a neural network by considering them as one large vector instead of a matrix. 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  $\Omega_0$  and  $\Omega_1$  be pixel domains with resolutions  $(M_0, N_0)$  and  $(M_1, N_1)$  respectively. Furthermore, let  $d \in \mathbb{N}$  be the number of channels.

Then we define an **image operator**  $T$  as the continuous mapping

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

where  $T$  does not change the number of channels  $d$ . Thus, we shorten  $\Psi_{d, \Omega_0}$  to  $\Psi_{\Omega_0}$  in the following.

The Definition 1.4.8 of an image operator 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.

However, the image operator helps us introduce neural networks operating on images.

**Definition 1.4.9.** Let  $\varphi$  be an arbitrary activation function and  $\Omega_0$  be an arbitrary pixel domain with resolution  $(M_0, N_0) \in \mathbb{N}^2$ . Let  $\psi$  be an image with number of channels  $d \in \mathbb{N}$ .

Then a neural layer that operates on images looks as follows

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

where  $T$  is an image operator as in Definition 1.4.8,  $b \in \Psi_{\Omega_0}$  is a bias and  $\Omega_1$  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  $\Psi_{\Omega_i}$ . This means 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 [4, Chapter 9].

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

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

This operator is defined component-wise by

$$(T_{\text{conv}} \psi_0)_{ij} = (\psi_1)_{ij} := (\psi_0 * k)_{ij} = \sum_{m,n=1}^s (\psi_0)_{m+i-1, n+j-1} k_{mn}, \quad (1.46)$$

where  $(i, j) \in \Omega_1 := \{1, \dots, M_0 - s + 1\} \times \{1, \dots, N_0 - s + 1\}$ . Hence,  $M_1 = M_0 - s + 1$  and  $N_1 = N_0 - s + 1$ .

Furthermore,  $k$  is called an  **$s$ -convolution kernel**, which is an image with resolution  $(s, s)$  with  $s \in \mathbb{N}$  and  $s \leq \min\{M_0, N_0\}$ .

Moreover, we want to mention that image convolution operators introduced in Definition 1.4.10, decrease the resolution of an image. However, since we will be considering neural networks, which reduce the dimension and increase it, afterwards, we need to introduce a so-called transposed convolution operator that increases the dimension of a given input. Intuitively, it does exactly the opposite of a convolution operator.

**Definition 1.4.11.** Let  $\Omega_0, \Omega_1$  be pixel domains with arbitrary but fixed number of channels and resolutions  $(M_0, N_0)$  and  $(M_1, N_1)$  respectively. Then the **transposed image convolution operator**  $T_{\text{conv}}^{\top} : \Psi_{\Omega_1} \rightarrow \Psi_{\Omega_0}$  is defined by

$$\begin{aligned} T_{\text{conv}}^{\top} : \Psi_{\Omega_1} &\rightarrow \Psi_{\Omega_0}, \\ \psi_1 &\mapsto \psi_0 := T_{\text{conv}}^{\top} \psi_1. \end{aligned}$$

This operator is defined component-wise by

$$(\psi_1)^{\top}_{ij} = (\psi_0)_{ij} := \sum_{m,n=1}^s (\psi_1)_{m+i-1, n+j-1} k_{mn},$$

where  $(i, j) \in \Omega_0 := \{1, \dots, M_1 + s - 1\} \times \{1, \dots, N_1 + s - 1\}$ . Hence,  $M_0 = M_1 + s - 1$  and  $N_0 = N_1 + s - 1$ .

Furthermore,  $k$  is called an **s-transposed convolution kernel**, which is an image with resolution  $(s, s)$  with  $s \in \mathbb{N}$ .

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

**Definition 1.4.12.** Let  $\Omega_0, \Omega_1$  be pixel domains 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  $I := \{1, \dots, p_1\} \times \{1, \dots, p_2\} \subset \mathbb{N}^2$  with  $p_1, p_2 \in \mathbb{N}$  is called **pooling**.

Then the **average-pooling operator**  $T_{\text{avg}} : \Psi_{\Omega_0} \rightarrow \Psi_{\Omega_1}$  with stride  $s$  and pooling  $I$  is defined by

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

This operator is defined component-wise by

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

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

**Definition 1.4.13.** Let  $\Omega_0, \Omega_1$  be pixel domains with arbitrary but fixed number of channels and resolutions  $(M_0, N_0)$  and  $(M_1, N_1)$  respectively.

Then the **min-pooling operator**  $T_{\text{min}} : \Psi_{\Omega_0} \rightarrow \Psi_{\Omega_1}$  with stride  $s$  and pooling  $I$  is defined by

$$\begin{aligned} T_{\text{min}} : \Psi_{\Omega_0} &\rightarrow \Psi_{\Omega_1}, \\ \psi_0 &\mapsto \psi_1 := T_{\text{min}} \psi_0. \end{aligned}$$

This operator is defined component-wise by

$$(\psi_1)_{ij} := \min_{(k,l) \in I} (\psi_0)_{kl},$$

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

**Definition 1.4.14.** Let  $\Omega_0, \Omega_1$  be pixel domains with arbitrary but fixed number of channels and resolutions  $(M_0, N_0)$  and  $(M_1, N_1)$  respectively.

Then the **max-pooling operator**  $T_{\max} : \Psi_{\Omega_0} \rightarrow \Psi_{\Omega_1}$  with stride  $s$  and pooling  $I$  is defined by

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

This operator is defined component-wise by

$$(\psi_1)_{ij} := \max_{(k,l) \in I} (\psi_0)_{kl},$$

where  $(i, j) \in \Omega_1 := \{1, \dots, M_0 - p_1 + 1\} \times \{1, \dots, N_0 - p_2 + 1\}$ . 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 definition.

**Definition 1.4.15.** Let  $T_{\text{conv}}$  be an image convolution operator with  $s$ -convolution kernel  $k$ . Furthermore, let  $\Psi_0$  and  $\Psi_1$  be image domains and  $\varphi$  be an activation function. Then the mapping

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

defines a neural layer with the parameters  $\theta = (k, b)$ , where  $k$  denotes the  $s$ -convolution kernel and  $b \in \Psi_1$  denotes the bias. We will call such a layer a **convolutional neural layer**.

Moreover, we note that for the transposed image convolution operator  $T_{\text{conv}}^\top$  with transposed  $s$ -convolution kernel  $k$  we define a convolutional layer analogously and make no distinction, since it is clear out of context due to increase/reduction of the dimensions.

Lastly, we highlight that in computer vision settings one commonly considers unbiased neural layers, i.e.  $b = 0$ .

# 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 described 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. In a nutshell, the goal of an autoencoding neural network is to learn how to compress data to a lower dimensional representation and afterwards, reconstruct the original representation as precisely as possible.

It is worth to mention that most state of the art Machine Learning models use autoencoding structures, since it is way more efficient to first encode the data and then process 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 hard drive and in memory) much more efficient.

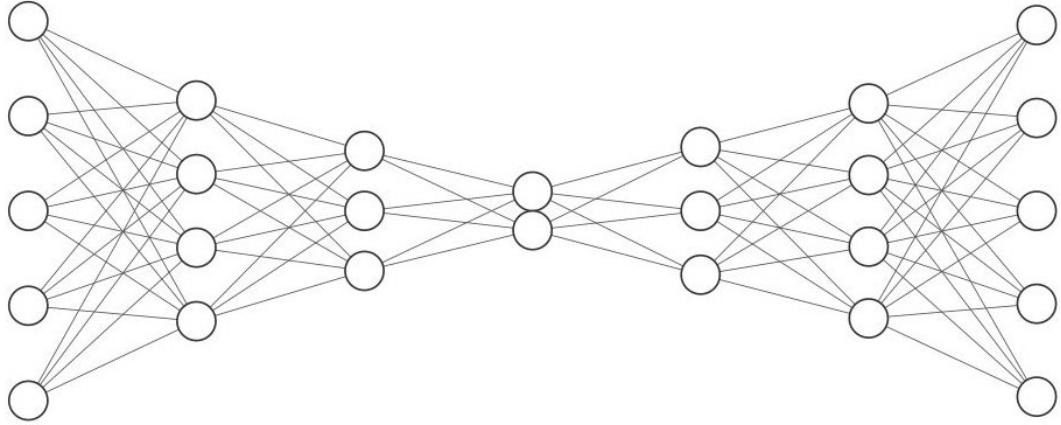
In this chapter we want to consider how to formulate autoencoding neural networks from a mathematical point of view, take a look at some important results and lastly, analyse the theory in multiple applications using Python.

### 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. We visualise an example of an autoencoding architecture in Figure 2.1. Each circle represents a neuron and each line represents a connection between neurons.

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

**Definition 2.1.1.** Let  $\Theta$  be a parameter space and  $\theta \in \Theta$  a set of parameters,  $L \in \mathbb{N}$  and  $d_1, \dots, d_L \in \mathbb{N}$ . Let further  $\varphi$  be an activation function and  $f_{\varphi, L, \theta}$  a neural network.



**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 the neural network  $f_{\varphi, L, \theta}$  fulfils the condition  $n_i = d_1 \geq \dots \geq d_L = n_o$  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 decoding structure, which we depict in Figure 2.3. We can define this architecture analogously to the encoding structure in Definition 2.1.1.

**Definition 2.1.2.** Let  $\Theta$  be a parameter space and  $\theta \in \Theta$  a parameter,  $L \in \mathbb{N}$  and  $d_1, \dots, d_L \in \mathbb{N}$ . Let further  $\varphi$  be an activation function and  $f_{\varphi, L, \theta}$  a neural network.

If the neural network  $f_{\varphi, L, \theta}$  fulfils the condition  $n_i = d_1 \leq \dots \leq d_L = n_o$  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 structures 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  $\Theta$  is an arbitrary parameter space. Furthermore, let the dimensions of each layer be  $d_1, \dots, d_{L_1} \in \mathbb{N}$  of  $f_1$  and  $\tilde{d}_1, \dots, \tilde{d}_{L_2} \in \mathbb{N}$  of  $f_2$ . Additionally, let  $d_{L_1} = \tilde{d}_1$ .

Then their composition  $f_2 \circ f_1$  is a neural network of depth  $L_1 + L_2$  with parameters  $(\theta_1, \theta_2)$ .

*Proof.* Since  $f_1$  is a neural network of depth  $L_1$  with parameters  $\theta_1$ , its architecture looks like

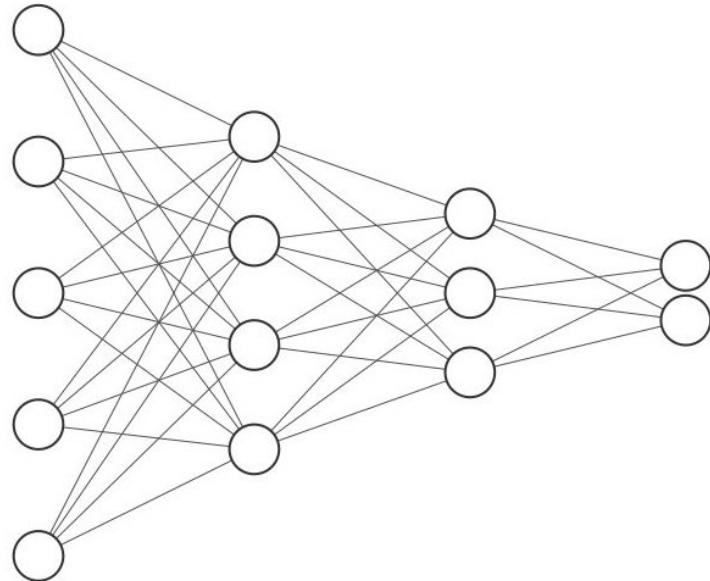
$$f_1(x) = H_{L_1} \circ H_{L_1-1} \circ \dots \circ H_2 \circ H_1(x), \quad x \in \mathbb{R}^{d_1}. \quad (2.1)$$

Analogously, we can write  $f_2$  as

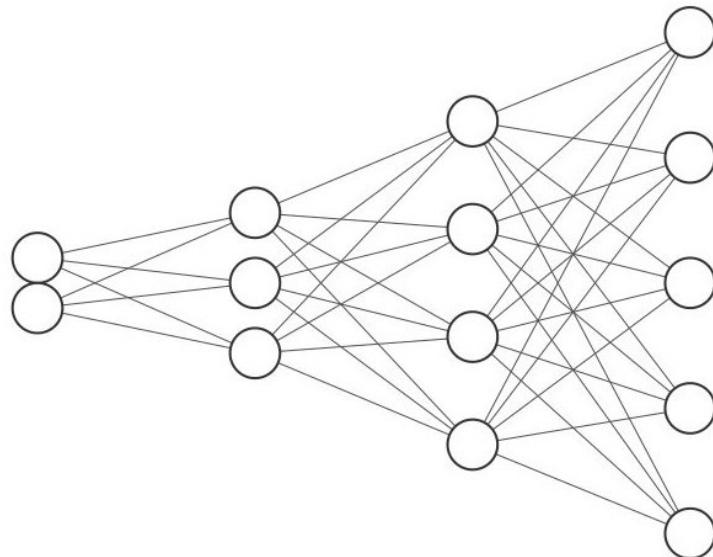
$$f_2(y) = \tilde{H}_{L_2} \circ \tilde{H}_{L_2-1} \circ \dots \circ \tilde{H}_2 \circ \tilde{H}_1(y), \quad y \in \mathbb{R}^{\tilde{d}_1}. \quad (2.2)$$

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

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



**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>

Hence, we obtain

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

Therefore, from equation (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  $\theta$  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 component is defined as  $\theta_i = (W_i, b_i)$  and denotes the weight and bias of each layer  $H_i$  or  $\tilde{H}_i$ , respectively, we can write the parameters of both neural networks as

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

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

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

□

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. Since this approach is so important, we want to formulate it as a theorem.

**Theorem 2.1.4.** *Let  $\Theta$  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*

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

*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  $f^{(1)} := f_2 \circ f_1$  with parameters  $\theta^{(1)} := (\theta_1, \theta_2)$  and depth  $L^{(1)} := L_1 + L_2$ .

If we now apply Lemma 2.1.3 once again to  $f^{(1)}$  and  $f_3$ , we receive  $f^{(2)} := f_3 \circ f^{(1)}$  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

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

Therefore the assertion is proven. □

With the help of Thereom 2.1.4 we can now formally introduce autoencoding neural networks as the composition of an encoding and a decoding structure. This we do in the following definition.

**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  $n_b \in \mathbb{N}$ . Then we define an **autoencoding neural network**  $f_a$  as the composition

$$f_a : \mathbb{R}^{n_i} \rightarrow \mathbb{R}^{n_i}, \\ x \mapsto (f_d \circ f_e)(x).$$

Moreover, we will refer to  $n_b$  as the **bottleneck** of the autoencoding neural network.

## 2.2 Training of Autoencoders

Now we want to tackle the question of how to train an autoencoding neural network. We realize that when training regular neural networks, we compare the output of the neural network to a label. In contrast, 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 afterwards, 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 we did 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  $p : X \rightarrow \mathbb{R}^n$  be a prediction function. Furthermore, let  $\hat{x} := p(x)$  be the predicted value of  $x \in X$ . Then a measurable function defined as

$$\mathcal{L} : X \times \mathbb{R}^n \rightarrow [0, \infty), \\ (x, \hat{x}) \mapsto \mathcal{L}(x, \hat{x}),$$

is called **unsupervised loss function**.

There are multiple important loss functions in the unsupervised learning setting. We will consider a couple of those in the following example. For further details and examples we refer to [3].

**Example 2.2.2.** Let  $\Omega$  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 expressions define unsupervised loss functions.

**Mean Squared Error (MSE):**

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

**Binary Cross-Entropy (BCE):**

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

where  $\psi$  denotes an image defined on  $\Psi_\Omega$ .

**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  $\Theta$  be a parameter space,  $L \in \mathbb{N}$  and  $d_1, \dots, d_L \in \mathbb{N}$ , where  $d_1 \geq \dots \geq d_L$ . Furthermore, let  $\varphi$  be an arbitrary activation function.

Then we define an encoder, where each layer  $H_1, \dots, H_L$  is defined as

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

where  $\theta_i = (W_i, b_i) \in \Theta$  are the parameters of the  $i$ -th layer. 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  $\Theta$  be a parameter space,  $L \in \mathbb{N}$  and  $d_1, \dots, d_L \in \mathbb{N}$ , where  $d_1 \leq \dots \leq d_L$ . Furthermore, let  $\hat{\varphi}$  be an arbitrary activation function.

Then we define a decoder, where each layer  $H_1, \dots, H_L$  is defined as

$$H_i(x) = \hat{\varphi}(W_i x + b_i), \quad x \in \mathbb{R}^{d_i}, i \in \{1, \dots, L\},$$

where  $\theta_i = (W_i, b_i) \in \Theta$  are the parameters of the  $i$ -th layer  $H_i$ . 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 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**  $f_{\text{lin}}$  is defined as the composition

$$f_{\text{lin}} := f_d \circ f_e.$$

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

Before giving thought to 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  $\Omega$  of  $D$  with  $\Omega = \{1, \dots, 28\} \times \{1, \dots, 28\}$  with only one channel.

Now we propose a specific example of how to train a linear autoencoder on the MNIST dataset in Algorithm 4.

Having proposed a possible training algorithm, we now want to take a look at the trained autoencoders. We used the Adam optimizer and the AMSGrad optimizer in different approaches. 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 onto. 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  $(28, 28)$  and hence, consists of 784 pixels. The first autoencoder we want to train will have a bottleneck dimension of 2, so it reduces the dimensions of the input from 784 pixels to merely 2 pixels. This resulting 2-dimensional vector we can easily visualise in a coordinate system, which we

---

**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: $\gamma \leftarrow 3 \times 10^{-4}$ ▷ Declare a learning rate.
1: for epoch in epochs do
2: for image in batch do
3: image = image.reshape(784) ▷ Convert the image from matrix to array.
4: encoded = encoder(image) ▷ Encode the image onto latent space.
5: reconstructed = decoder(encoded) ▷ Decode the encoded image.
6: loss = MSE(reconstructed, image) ▷ Compare the output to the input.
7: optimization(loss, γ) ▷ Perform an optimization step.
8: end for
9: end for

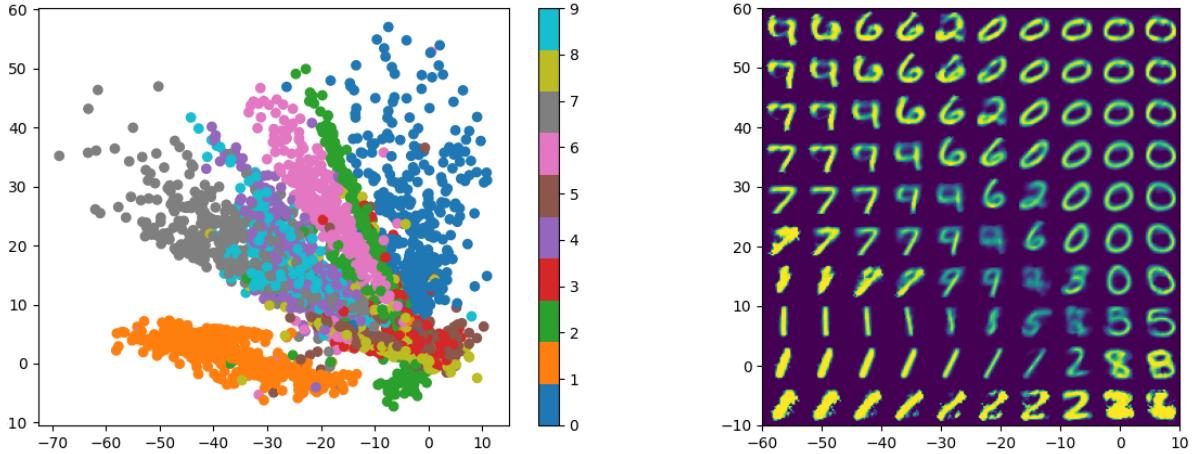
```

---

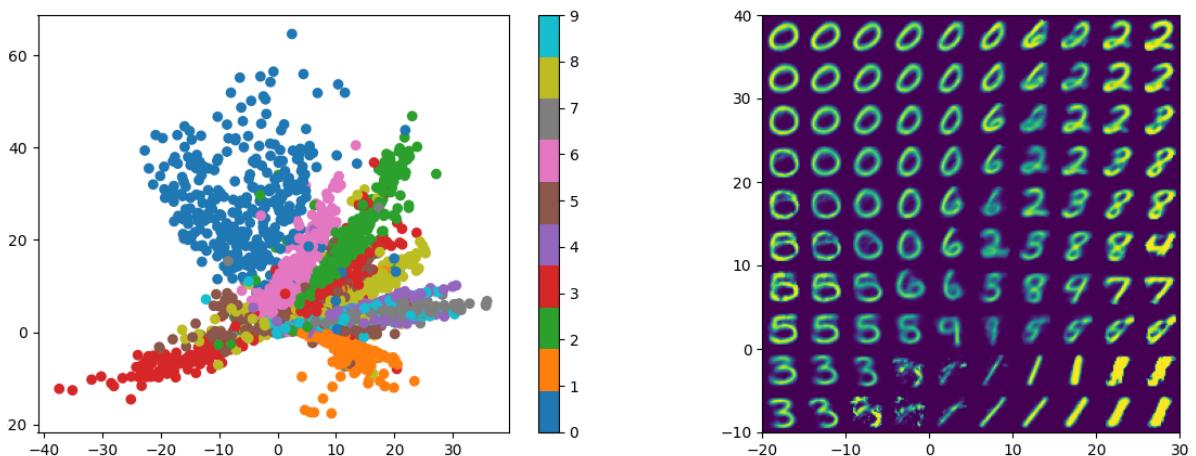
do in the left charts of Figure 2.4, where we used the Adam optimizer for optimization and Figure 2.5, where we used the AMSGrad optimizer for optimization, respectively. In these charts we see point clouds of ten different colours, where each point represents an encoding of a sample. The colours of the encodings represent what digit and thus label a 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 maps 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 spatially 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 image fed into the autoencoder. We see that the samples, which are encoded such 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 consisting of  $10 \times 10$  nodes in the coordinate system, where each node has the same distance to its neighbour. These nodes then are fed into the decoder. Intuitively, we see how each vector in this plane corresponds to a reconstructed image.

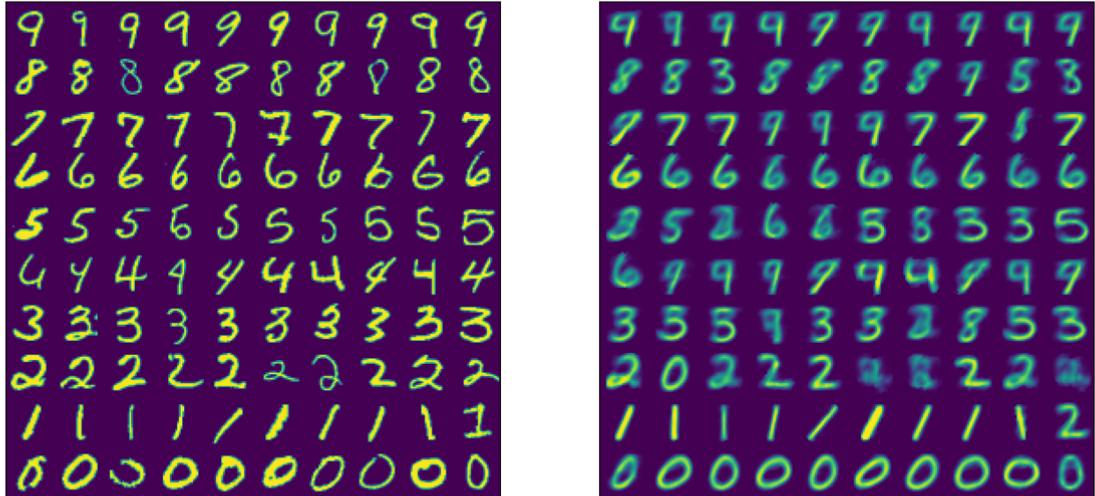
Furthermore, we want to take a quick glance at the training progress of the two autoencoders. These we can see in Figure 2.8 for the Adam optimizer and in Figure 2.9 for the AMSGrad optimizer. We see that in roughly the first 100 epochs the training loss falls dramatically. From around epoch 1000 on, the training loss decreases 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 monotonously. Since we do not consider the entire dataset in each iteration of the optimization, we do not chose the exact gradient of the risk function in each training



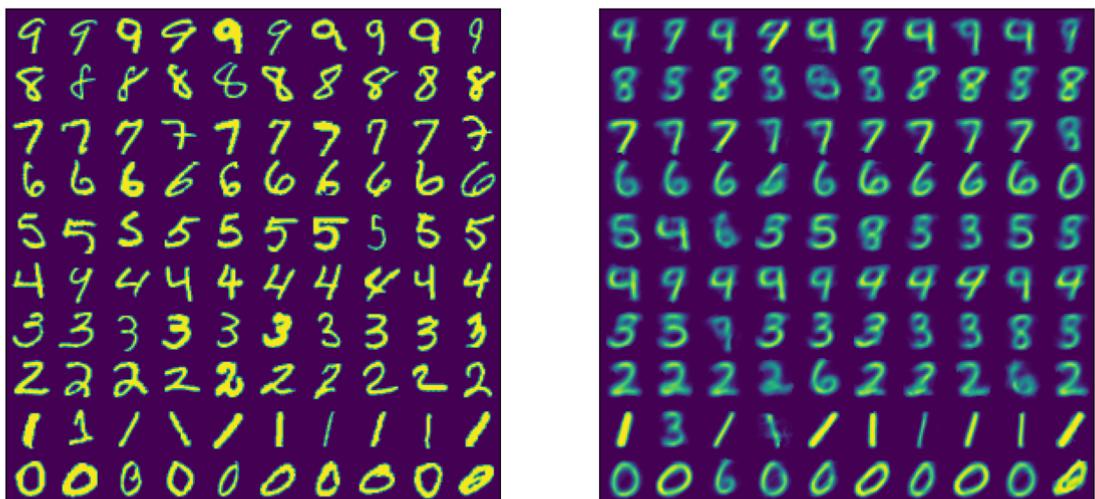
**Figure 2.4:** On the left-hand side, the figure illustrates the latent space of the linear autoencoder with bottleneck  $n_b = 2$  optimized with an Adam optimizer, where each dot is one encoded image of a digit. The color and the corresponding color map represent the digit that was encoded. On the right-hand side the figure illustrates the corresponding reconstruction through the autoencoder. Each node of the  $10 \times 10$  mesh is fed into the decoding architecture of the autoencoder to generate an image.



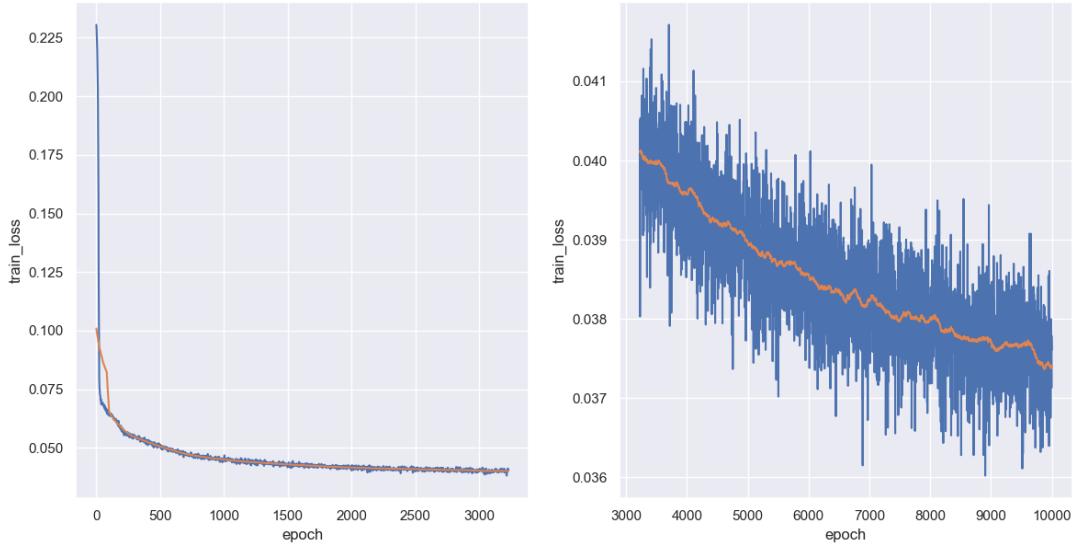
**Figure 2.5:** On the left-hand side, the figure illustrates the latent space of the linear autoencoder with bottleneck  $n_b = 2$  optimized with an AMSGrad optimizer, where each dot is one encoded image of a digit. The color and the corresponding color map represent the digit that was encoded. On the right-hand side the figure illustrates the corresponding reconstruction through the autoencoder. Each node of the  $10 \times 10$  mesh is fed into the decoding architecture of the autoencoder to generate an image.



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



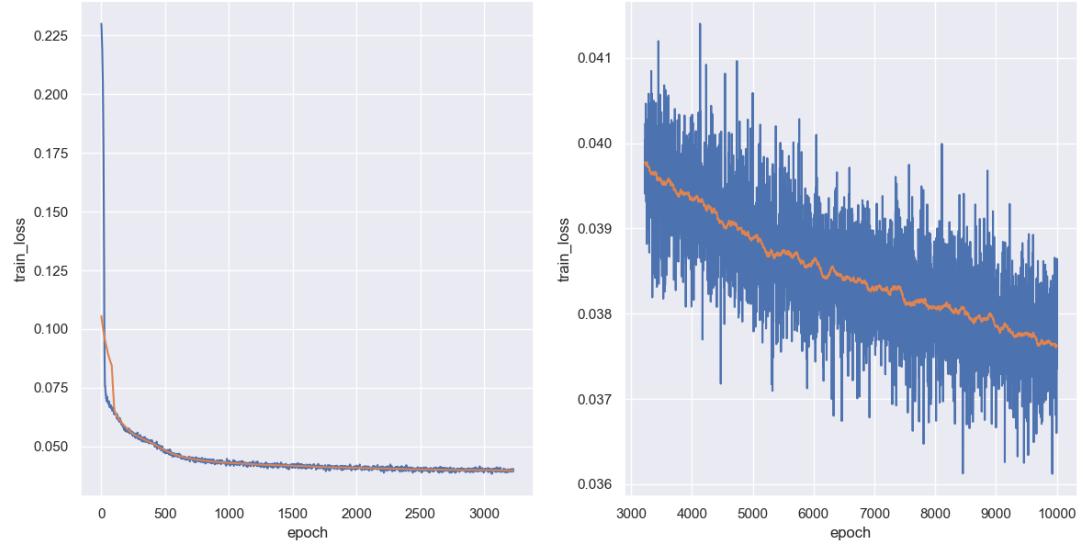
**Figure 2.8:** The figure illustrates the training progresses of the linear autoencoder with bottleneck  $n_b = 2$  optimized with an Adam optimizer with epochs on one axis and corresponding training loss on the other axis. On the left-hand side we see the first 3,500 epochs and on the right-hand side 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.

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 minimized.

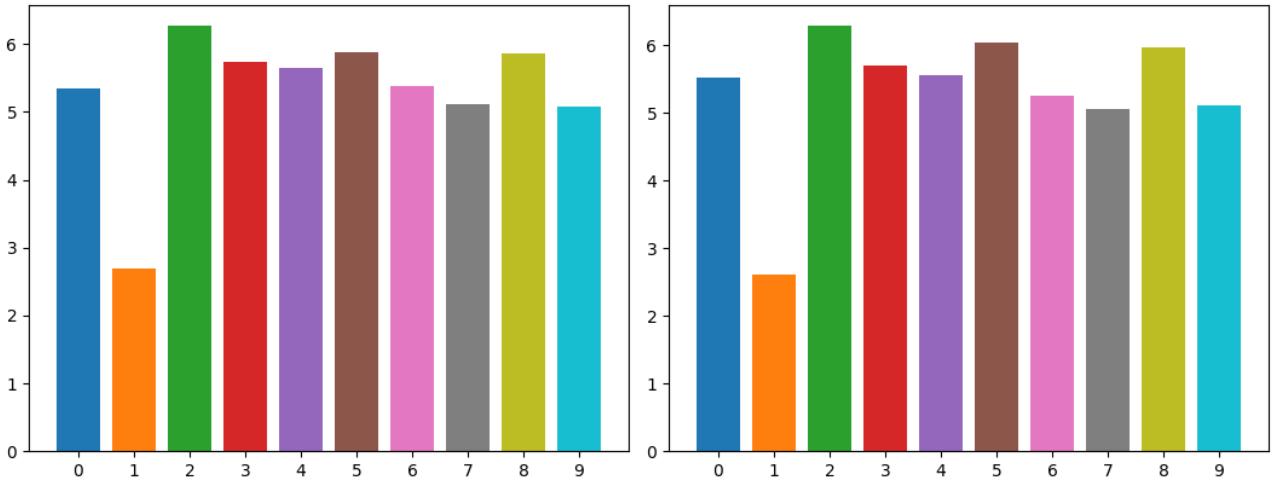
Lastly, we want to give thought about how high the reconstruction error actually is. To do so we compute the Euclidean distance of the error vector, i.e. we compute the difference between the reconstruction and the original image and afterwards, compute the root of the mean squared error over all pixels. Furthermore, we note that the reconstruction error is computed in the normed image domain, see Definition 1.4.5. In order to get a more accurate quantity, we compute this error for each of the ten digits. We average the errors over all samples in the dataset, i.e. over all 1s, 2s, etc. These errors we then depict in Figure 2.10, with the errors for the autoencoder trained with the Adam optimizer on the left-hand side and for the autoencoder trained with the AMSGrad optimizer on the right-hand side. It is worth to highlight, that we can barely see a difference between the reconstruction errors for the two optimizers with bottleneck  $n_b = 2$ . This will indeed be different for higher bottleneck dimensions.

Now we want to study what happens if we increase the bottleneck dimension. The next experiment we want to conduct is to create an autoencoder that has the same architecture as the previous one, but has a 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-dimensional 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 2.11 and Figure 2.12 we can see two different perspectives of the 3-dimensional latent space of an autoencoder trained with an Adam optimizer and an autoencoder trained with an AMSGrad optimizer, respectively.

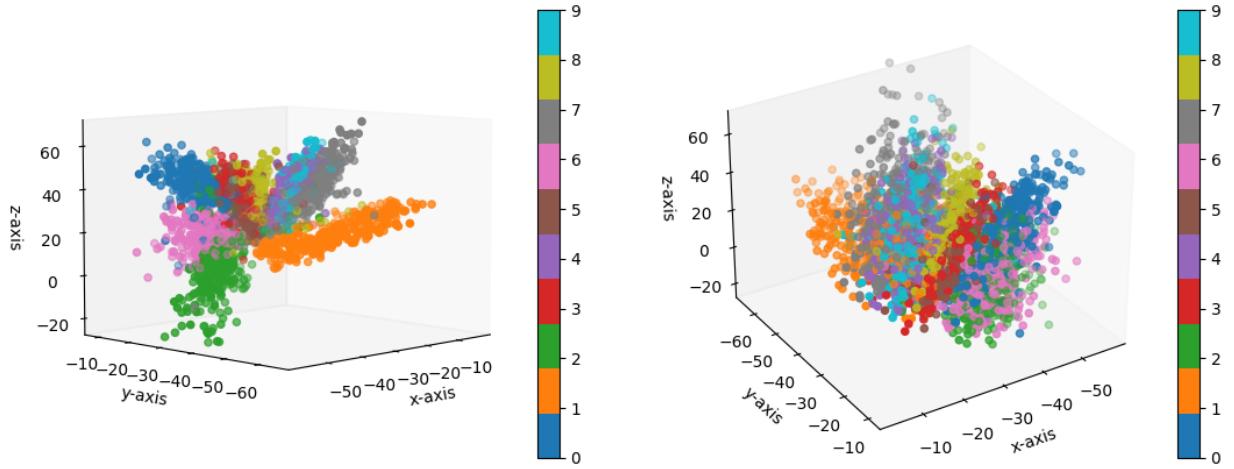
We can see in these figures clearly that the encoder behaves the same way as it did in the 2-dimensional setting, that is for each digit we can see that the encodings are all aligned on a ray for



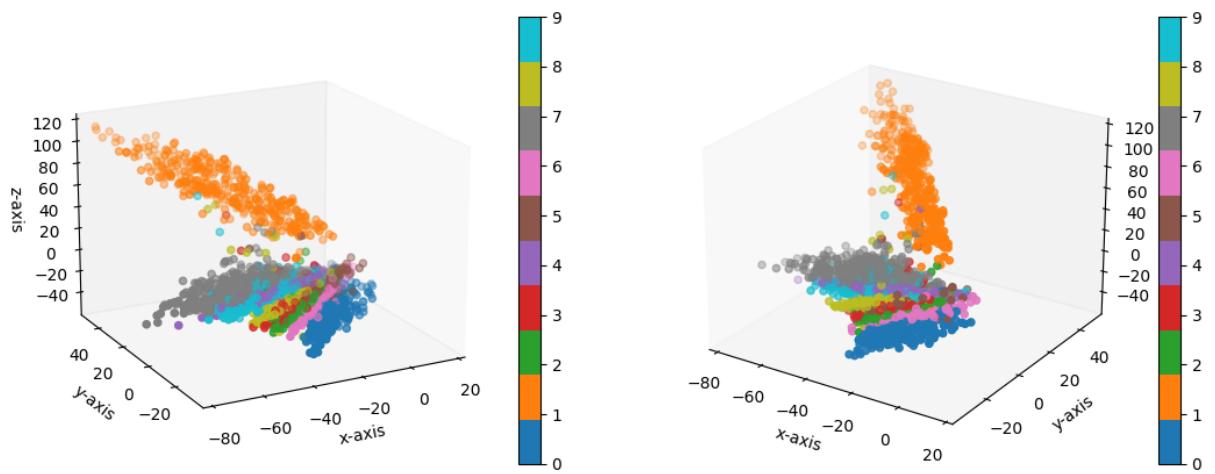
**Figure 2.9:** The figure illustrates the training progresses of the linear autoencoder with bottleneck  $n_b = 2$  optimized with an AMSGrad optimizer with epochs on one axis and corresponding training loss on the other axis. On the left-hand side we see the first 3,500 epochs and on the right-hand 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.



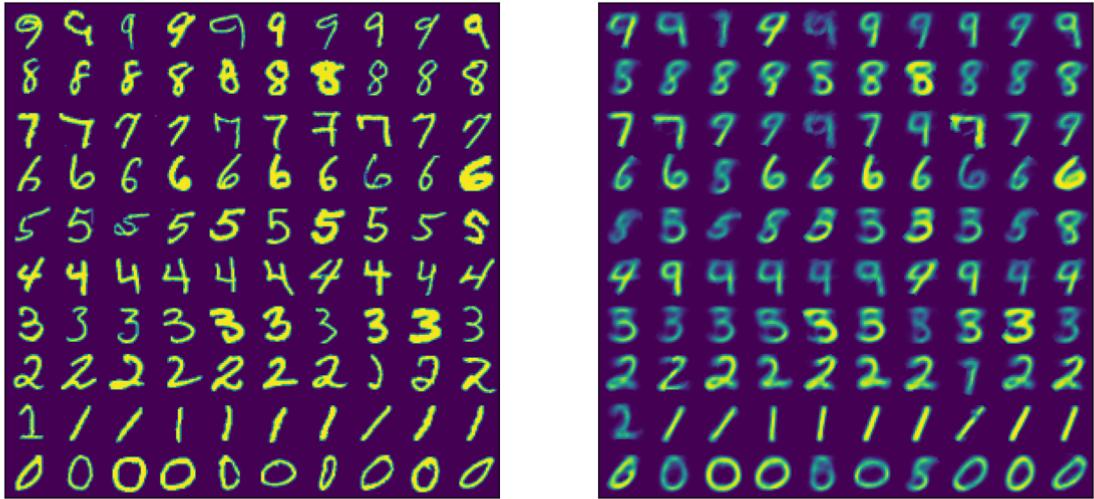
**Figure 2.10:** On the left-hand side, the figure illustrates the test errors of the linear autoencoder with bottleneck  $n_b = 2$  optimized with an Adam optimizer, where each bar represents the averaged test errors over the entire MNIST dataset for each of the ten digits. On the right-hand side the figure illustrates the same test errors of the linear autoencoder optimized with an AMSGrad optimizer.



**Figure 2.11:** 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.



**Figure 2.12:** 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 2.13:** On the left-hand side, the figure illustrates 100 original digits from the MNIST dataset. On the right-hand side, the figure illustrates the same digits after feeding them through the linear autoencoder with bottleneck  $n_b = 3$  optimized with an Adam optimizer.

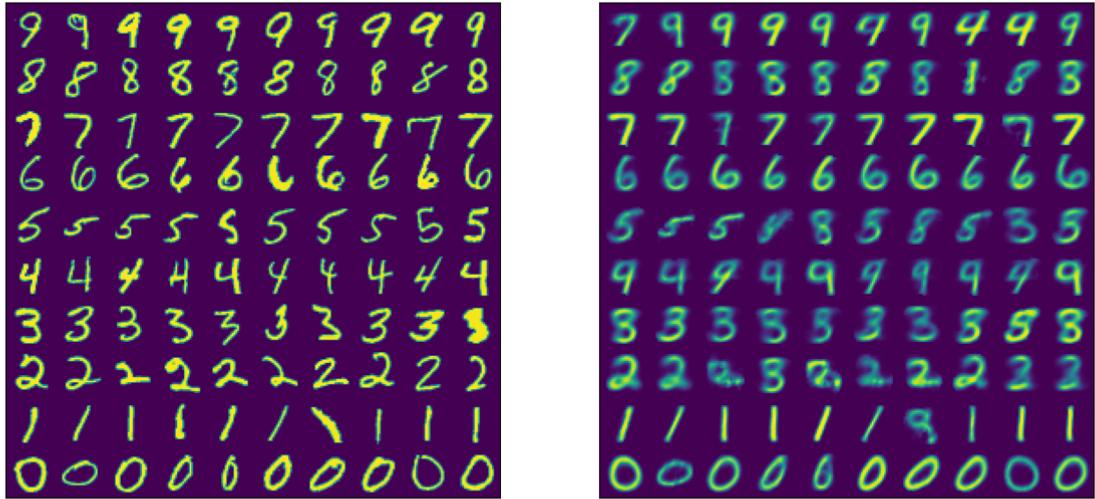
the same digit. The more the digits differ, the farther apart these rays are. For example, we can see clearly in Figure 2.11 that the cluster of encoded 2's is spatially separated well, as well as the cluster of encoded 1's. In contrast, the clusters of encoded 4's, 7's and 9's are heavily intertwined, which results in bad reconstructions, as we can see in Figure 2.13, which depicts the reconstructions of the linear autoencoder, where we used the Adam optimizer. The same behaviour we can see in Figure 2.12, which depicts the latent space of the autoencoder, where we used the AMSGrad optimizer. The clusters are indeed slightly better separated, but still not really good. This we can see in the reconstructions in Figure 2.14.

Lastly, we take a quick glance at the training progress of the two described autoencoders, which we can see in Figure 2.15 with the Adam optimizer and in Figure 2.16 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 2-dimensional setting, but the overall training progress is similar.

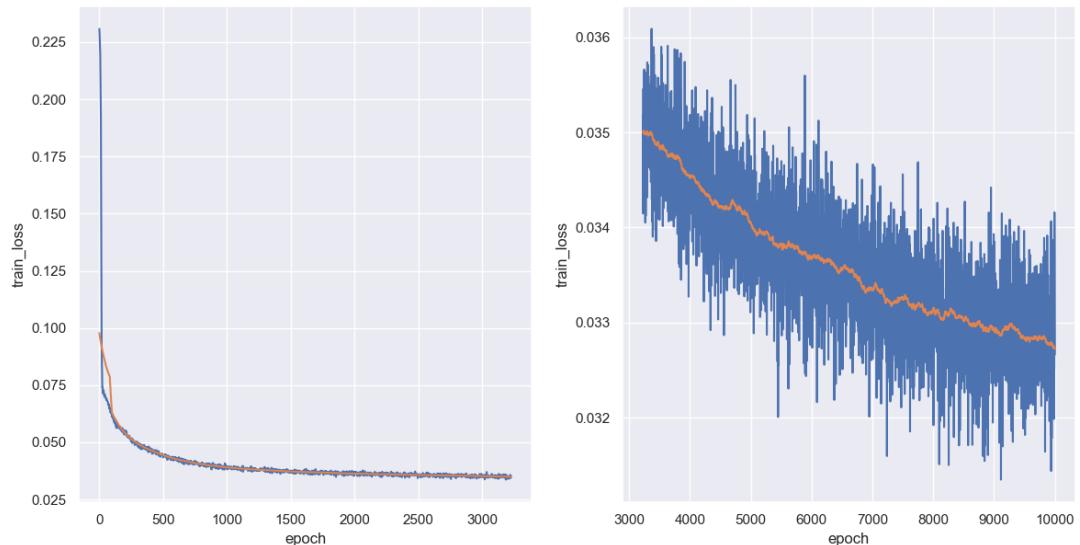
Furthermore, we now give thought to the reconstruction errors in the same way as we did in the 2-dimensional case. The errors are depicted in Figure 2.17, where the left chart represents the reconstruction errors for an autoencoder trained with an Adam optimizer and the right chart represents the reconstruction errors for an autoencoder trained with an AMSGrad optimizer. We can see a very slight difference between the reconstruction errors for the two neural networks, which additionally shows the fact that the AMSGrad optimizer does indeed perform better than the Adam optimizer.

Proceeding even further, we want to increase the bottleneck one last time to  $n_b = 64$  dimensions, because this is the bottleneck dimension we want to compare to other autoencoding structures, which we will consider in the course of this section. Since it is no longer possible to visualise the latent space in a coordinate system, as it was for the 2-dimensional and for the 3-dimensional case, we will omit this. Instead we want to take a look at the reconstruction capability of the neural network. Analogously to the lower dimensional cases, we depict 100 reconstructed images in Figure 2.18 optimized with the Adam optimizer and in Figure 2.19 optimized with the AMSGrad optimizer.

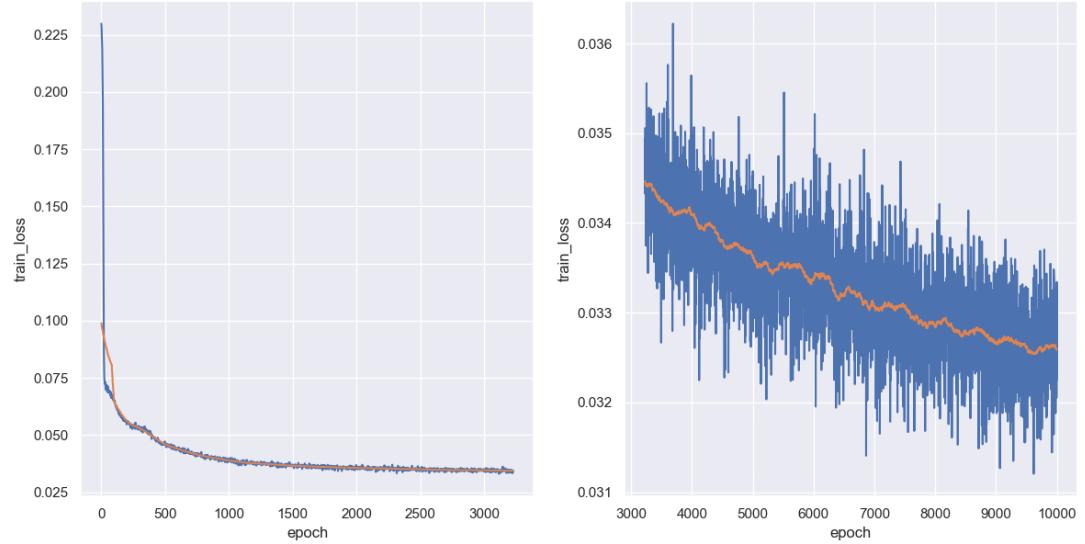
Furthermore, we depict the reconstruction errors in the same way as we did before, as well. These reconstruction errors are depicted in Figure 2.20. We can see multiple interesting things here. First,



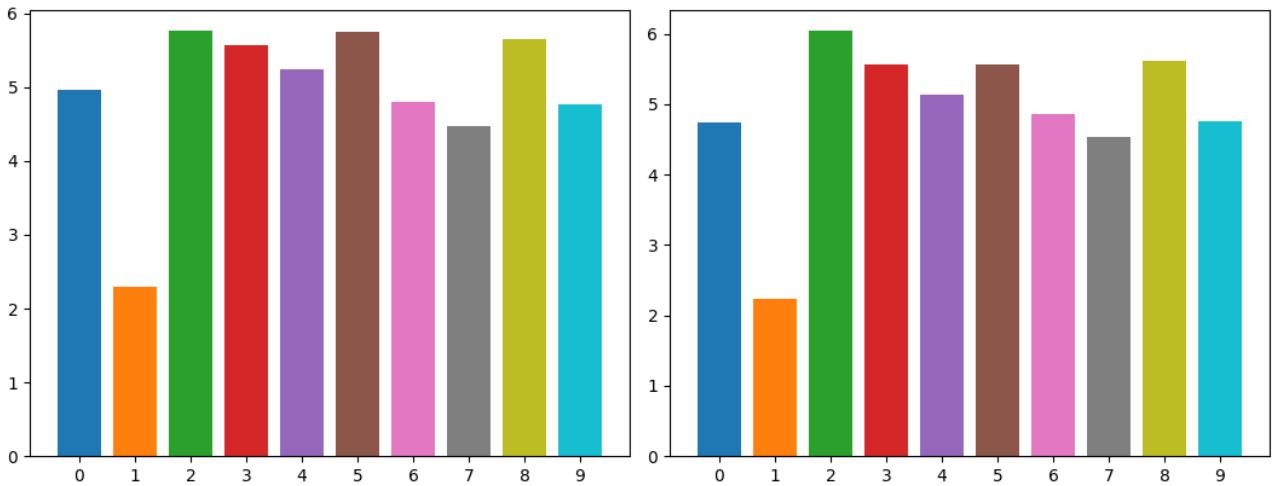
**Figure 2.14:** On the left-hand side, the figure illustrates 100 original digits from the MNIST dataset. On the right-hand side, the figure illustrates the same digits after feeding them through the linear autoencoder with bottleneck  $n_b = 3$  optimized with an AMSGrad optimizer.



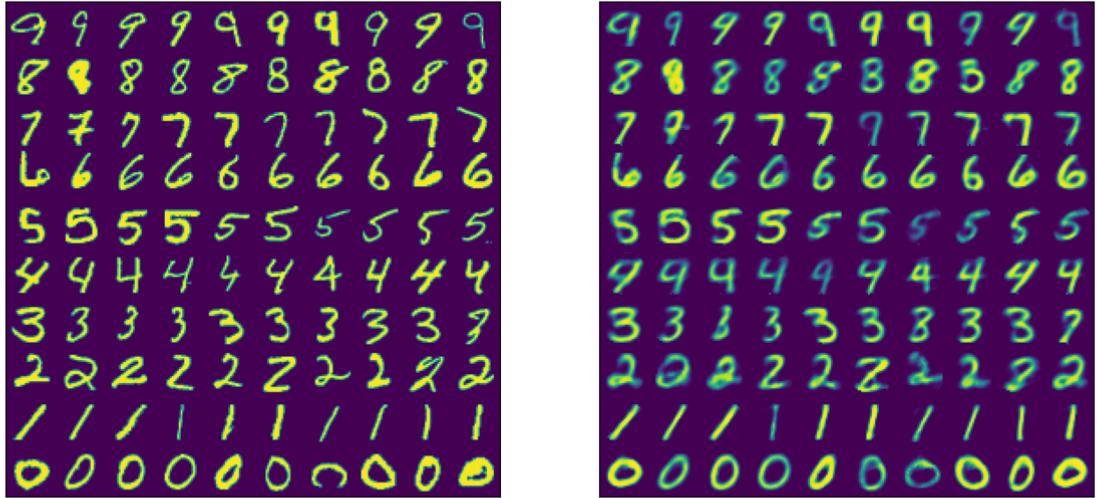
**Figure 2.15:** The figure illustrates the training progresses of the linear autoencoder with bottleneck  $n_b = 3$  optimized with an Adam optimizer with epochs on one axis and corresponding training loss on the other axis. On the left-hand side we see the first 3,500 epochs and on the right-hand side 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.



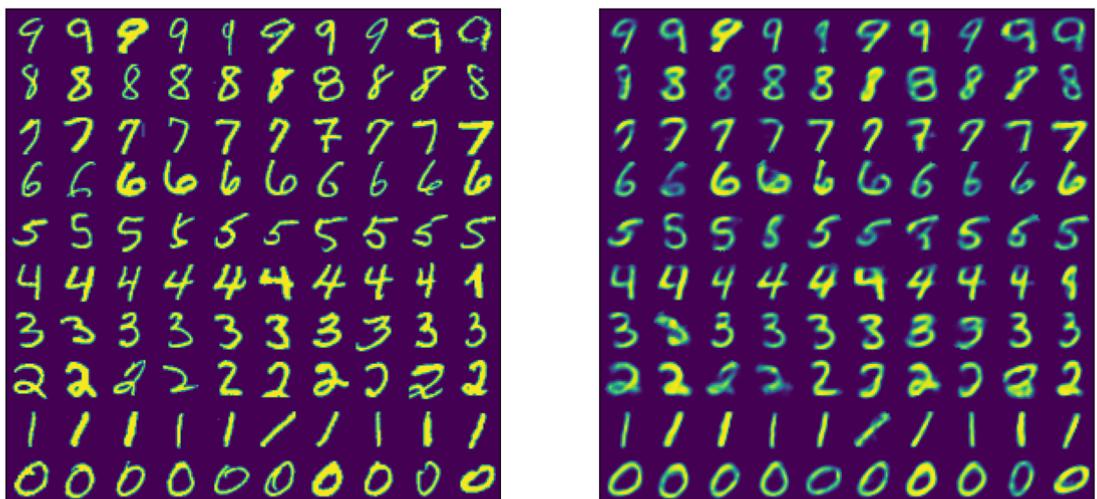
**Figure 2.16:** The figure illustrates the training progresses of the linear autoencoder with bottleneck  $n_b = 3$  optimized with an AMSGrad optimizer with epochs on one axis and corresponding training loss on the other axis. On the left-hand side we see the first 3,500 epochs and on the right-hand 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.



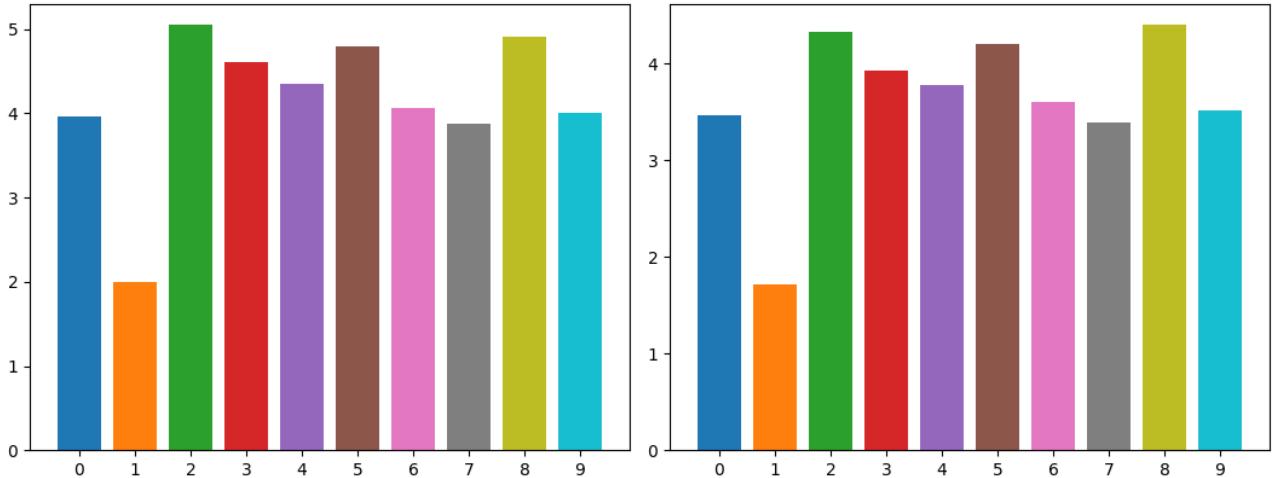
**Figure 2.17:** On the left-hand side, the figure illustrates the test errors of the linear autoencoder with bottleneck  $n_b = 3$  optimized with an Adam optimizer, where each bar represents the averaged test errors over the entire MNIST dataset for each of the ten digits. On the right-hand side the figure illustrates the test errors of the linear autoencoder optimized with an AMSGrad optimizer.



**Figure 2.18:** On the left-hand side, the figure illustrates 100 original digits from the MNIST dataset. On the right-hand side, the figure illustrates the same digits after feeding them through the linear autoencoder with bottleneck  $n_b = 64$  optimized with an Adam optimizer.



**Figure 2.19:** On the left-hand side, the figure illustrates 100 original digits from the MNIST dataset. On the right-hand side, the figure illustrates the same digits after feeding them through the linear autoencoder with bottleneck  $n_b = 64$  optimized with an AMSGrad optimizer.



**Figure 2.20:** On the left-hand side, the figure illustrates the test errors of the linear autoencoder with bottleneck  $n_b = 64$  optimized with an Adam optimizer, where each bar represents the averaged test errors over the entire MNIST dataset for each of the ten digits. On the right-hand side the figure illustrates the test errors of the linear autoencoder optimized with an AMSGrad optimizer.

the reconstruction errors do indeed become smaller the higher the bottleneck dimension becomes. This does absolutely make sense, since the higher the bottleneck the more information the neural network is able to keep and therefore, loses less information upon encoding the images. Second, the reconstruction error is clearly smaller, when we used the AMSGrad optimizer during training. Therefore, we will only consider the AMSGrad optimizer in the following.

As we saw in Figure 2.18 and in Figure 2.19, 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 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 formally as follows.

**Definition 2.3.5.** Let  $\Theta$  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 each layer, where  $M_1 \geq \dots \geq M_L$  and  $N_1 \geq \dots \geq N_L$ . Furthermore, let  $\varphi$  be an arbitrary activation function. Then we define an encoder, where each layer  $H_1, \dots, H_L$  is defined as

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

where  $\theta_i = (T_{\text{conv},i}, b_i) \in \Theta$  are the parameters and  $\Psi_{d_i, \Omega_i}$  the image domain of the  $i$ -th layer. Such an encoding neural network we refer to as **convolutional encoder**.

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

**Definition 2.3.6.** Let  $\Theta$  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 each layer, where  $M_1 \leq \dots \leq M_L$  and  $N_1 \leq \dots \leq N_L$ . Furthermore, let  $\varphi$  be an arbitrary activation function. Then we define a decoder, where each layer  $H_1, \dots, H_L$  is defined as

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

where  $\theta_i = (T_{\text{conv},i}, b_i) \in \Theta$  are the parameters and  $\Psi_{d_i, \Omega_i}$  the image domain of the  $i$ -th layer. Such a decoding neural network we refer to as **convolutional decoder**.

As we saw in Lemma 2.1.3, we can consider the composition of the convolutional encoder and the convolutional decoder, as long as the output dimension of the former matches the input dimension of the latter. This gives us the following autoencoding architecture.

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

$$f_{\text{conv}} := f_d \circ f_e.$$

Now we propose a specific example of how to train a convolutional autoencoder on the MNIST dataset in Algorithm 5 and afterwards take a look at its performance with some tangible visualisations.

---

#### 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.

```

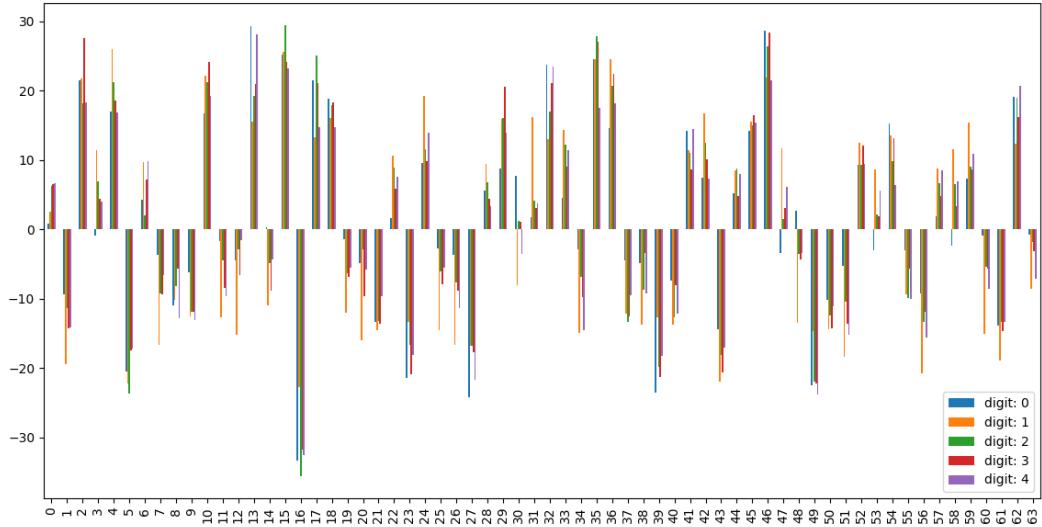
Require: $\gamma \leftarrow 3 \times 10^{-4}$ ▷ Declare a learning rate.
1: for epoch in epochs do
2: for image in batch do
3: encoded = encoder(image) ▷ Encode the image onto latent space.
4: reconstructed = decoder(encoded) ▷ Decode the encoded image.
5: loss = MSE(reconstructed, image) ▷ Compare the output to the input.
6: optimization(loss, γ) ▷ Perform an optimization step.
7: end for
8: end for

```

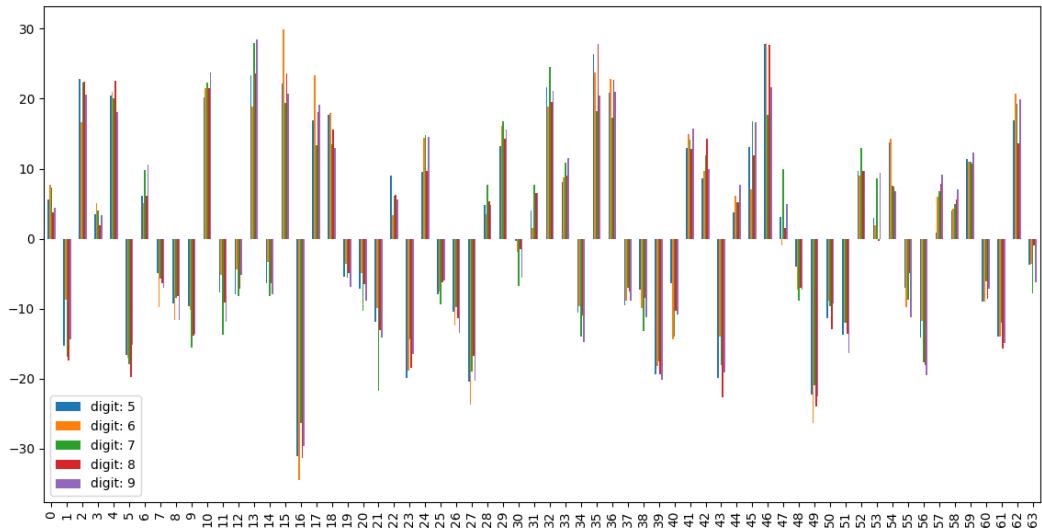
---

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 linear autoencoders, with bottleneck dimensions of 2 and 3, respectively. This choice 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. However, we still came up with an idea of how to visualise the encoded representation. We plot each channel in a separate bar in a bar plot, see Figure 2.21. 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. Intuitively, one can imagine this as a 64-dimensional vector mapping to the cluster center of each digit. Since all ten vectors have different activations, the cluster centers are spatially separated. This is exactly the behaviour that we wished to observe. Moreover, what is highly interesting to highlight here is that although the magnitude of the values in each channel might differ quite significantly, their signs match most of the time for all ten digits.

Another interesting result that we want to highlight here is the quality of the reconstructions produced by the convolutional autoencoder. These reconstructions we illustrate in Figure 2.22. We depict them analogously to the linear autoencoder, where we take ten images for each of the ten digits (shown on the left-hand side) and feed them into the convolutional autoencoder. The autoencoder then generates a reconstruction for each of the 100 images (shown on the right-hand side). Every single reconstruction is clear enough to be recognised as their original digit, which is a huge improvement

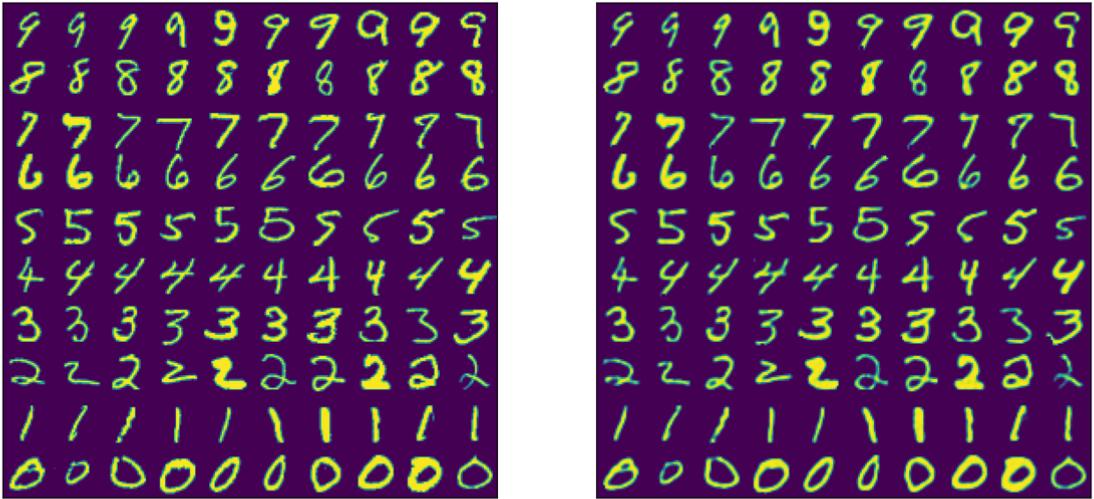


This figure illustrates the average activation value of each of the 64 channels in the latent space of the convolutional autoencoder for the digits 0, 1, 2, 3 and 4. The average has been taken over all images of the same digit in the entire dataset.



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 all images of the same digit in the entire dataset.

**Figure 2.21:** 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.

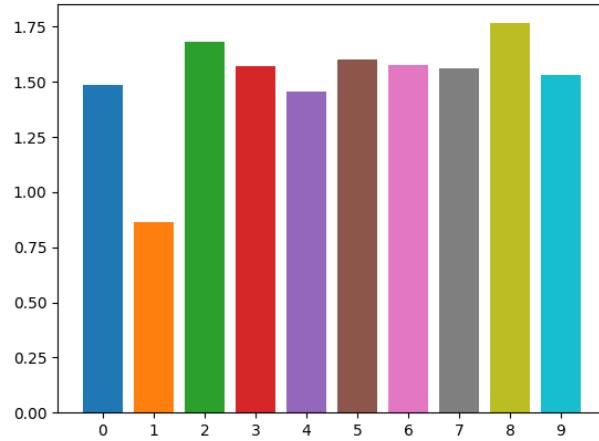


**Figure 2.22:** On the left-hand side, the figure illustrates 100 original digits from the MNIST dataset. On the right-hand side, the figure illustrates the same digits after feeding them through the convolutional autoencoder which was optimized with an AMSGrad optimizer.

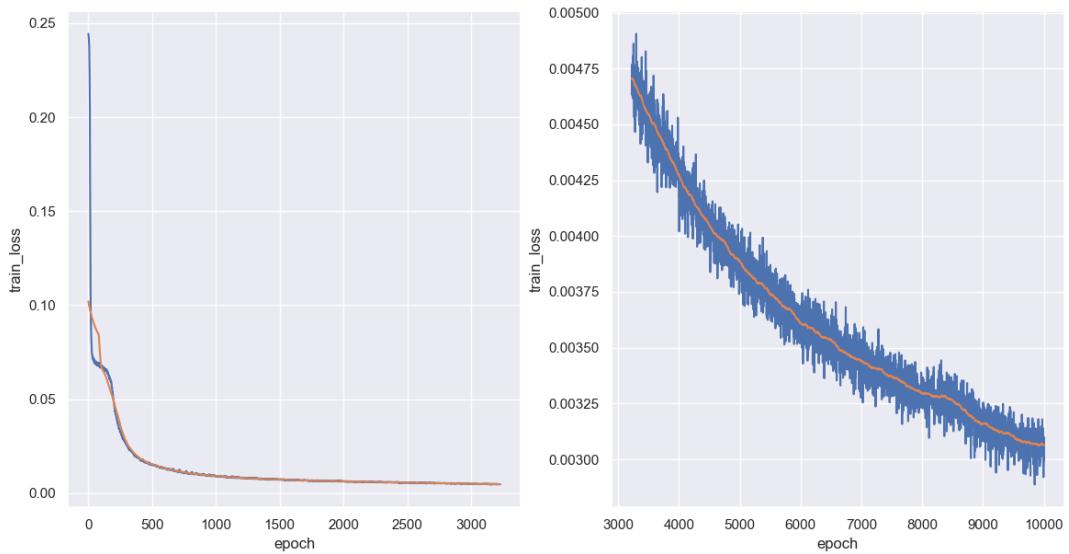
compared to the reconstructions of the linear autoencoder, see e.g. Figure 2.18 or Figure 2.18. This result clearly shows the fact that convolutional neural networks perform much better than linear neural networks, which is a well known fact in the realm of Machine Learning.

Moreover, we want to quantify the reconstruction error of the convolutional autoencoder. The reconstruction errors are computed analogously to the reconstruction errors for the linear autoencoder, where we average the Euclidean distance between reconstruction and original image over a digit in the entire dataset, i.e. for all 1s, 2s, etc. These reconstruction errors we depict in Figure 2.23. Compared to the linear autoencoder with bottleneck  $n_b = 64$ , see Figure 2.20, we realise that the reconstruction errors are roughly 3 times smaller. This result additionally shows that the convolutional autoencoder performs much better than the linear autoencoder.

Lastly, we want to take a quick glance at the training progress of the convolutional autoencoder, see Figure 2.24. 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 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 2.24 compared to 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 2.23:** The figure illustrates the test errors of the convolutional autoencoder, where each bar represents the averaged test errors over the entire MNIST dataset for each of the ten digits.



**Figure 2.24:** 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-hand side we see the first 3,500 epochs and on the right-hand 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

We already discussed ordinary autoencoding neural networks to be discriminative models in Chapter 2. In contrast, variational autoencoding neural networks, first introduced by Kingma and Welling in the year 2013, see [8], are so called generative models, see [2, 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. 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  $Y$ . Therefore, we somehow have to construct a probability distribution and at the same time, take the observed data into consideration. In order to do so, we introduce the concept of conditional probabilities. This we do in the following definition.

**Definition 3.1.1.** Let  $(X, \mathcal{A})$  and  $(Y, \mathcal{B})$  be measurable spaces and let  $\mathbb{P}$  be a probability measure on  $(X \times Y, \mathcal{A} \otimes \mathcal{B})$ . Then we call the map  $\mathbb{P}(\cdot | \cdot) : \mathcal{B} \times X \rightarrow [0, 1]$  a **regular conditional probability** of  $\mathbb{P}$  given  $X$ , if the following three conditions are satisfied:

- (i) For all  $x \in X$  the map  $\mathbb{P}(\cdot | x) : \mathcal{B} \rightarrow [0, 1]$  is a probability measure.
- (ii) For all  $B \in \mathcal{B}$  the map  $\mathbb{P}(B | \cdot) : X \rightarrow [0, 1]$  is  $\mathcal{A}$ -measurable.
- (iii) For all  $A \in \mathcal{A}$  and  $B \in \mathcal{B}$  we have

$$\mathbb{P}(A \times B) = \int_X \mathbb{1}_A(x) \mathbb{P}(B | x) d\mathbb{P}_X(x),$$

where  $\mathbb{P}_X$  is the marginal distribution of  $\mathbb{P}$  on  $X$ .

In Definition 3.1.1 we did not assume the existence of a probability density function. However upon doing so, the regular conditional probability can be quickly computed. Let  $\mu \otimes \nu$  be a  $\sigma$ -finite measure on  $\mathcal{A} \otimes \mathcal{B}$  such that the probability measure  $\mathbb{P}$  on  $\mathcal{A} \otimes \mathcal{B}$  has a  $\mu \otimes \nu$  density  $p : X \times Y \rightarrow [0, \infty]$ . We define  $p_X : X \rightarrow [0, \infty]$  by

$$p_X(x) := \int_Y p(x, y) d\nu(t), \quad x \in X.$$

Then  $p_X$  is  $\mathcal{A}$ -measurable by the theorem of Fubini-Tonelli, see e.g. [9, Theorem 14.16]. Moreover, it holds that

$$\begin{aligned} \int_X \mathbb{1}_A p_X d\mu &= \int_X \mathbb{1}_A(x) \int_Y p(x, y) d\nu(y) d\mu(x) = \int_{X \times Y} \mathbb{1}_{A \times Y} p d(\mu \otimes \nu) \\ &= \mathbb{P}(A \times Y) \\ &= \mathbb{P}_X(A), \end{aligned}$$

for all  $A \in \mathcal{A}$ . Therefore,  $p_X$  is a  $\mu$ -density of  $\mathbb{P}_X$ , which we will refer to as the **marginal  $\mu$ -density** of  $\mathbb{P}$  on  $X$ .

Furthermore, for an arbitrary but fixed  $\nu$ -probability density function  $h : Y \rightarrow [0, \infty)$  we define the **conditional probability density** of  $y \in Y$  given  $x \in X$  by

$$p(y|x) := \begin{cases} \frac{p(x,y)}{p_X(x)}, & \text{if } f_X(x) > 0, \\ h(y), & \text{otherwise.} \end{cases}$$

We note that the map  $(x, y) \mapsto p(y|x)$  is measurable, since  $p, p_X$  and  $h$  are measurable. Consequently, for every fixed  $x \in X$  the map  $(y, x) \mapsto p(y|x)$  is non-negative and  $\mathcal{B}$ -measurable. Thus, we can define a measure  $\mathbb{P}(\cdot | x)$  on  $\mathcal{B}$  by

$$\mathbb{P}(B|x) := \int_Y \mathbb{1}_B(y) f(y|x) d\nu(y), \quad B \in \mathcal{B}.$$

Our construction then gives  $\mathbb{P}(Y|x) = 1$  for all  $x \in X$ . Moreover, since  $(x, y) \mapsto p(y|x)$  is measurable,  $\mathbb{P}(B|\cdot) : X \rightarrow [0, 1]$  is  $\mathcal{A}$ -measurable for all  $B \in \mathcal{B}$ . Therefore,  $\mathbb{P}(\cdot | \cdot) : \mathcal{B} \times X \rightarrow [0, 1]$  satisfies conditions (i) and (ii) of Definition 3.1.1. Lastly, to verify that  $\mathbb{P}(\cdot | \cdot)$  is a regular conditional probability of  $\mathbb{P}$  given  $X$ , we fix some  $A \in \mathcal{A}$  and  $B \in \mathcal{B}$ . Then with the consideration

$$\mathbb{P}(\{p_X = 0\} \times Y) = \mathbb{P}_X(\{p_X = 0\}) = \int_{\{p_X=0\}} p_X d\mu = 0,$$

follows the fact that

$$\int_X \mathbb{1}_A(x) \mathbb{P}(B|x) d\mathbb{P}_X(x) = \int_{\{p_X>0\}} \mathbb{1}_A(x) \int_X \mathbb{1}_B(y) \frac{p(x,y)}{p_X(x)} d\nu(y) d\mathbb{P}_X(x).$$

If we now use the fact that  $p_X$  is a  $\mu$ -density, it holds that

$$\begin{aligned} \int_{\{p_X>0\}} \mathbb{1}_A(x) \int_X \mathbb{1}_B(y) \frac{p(x,y)}{p_X(x)} d\nu(y) d\mathbb{P}_X(x) &= \int_{\{p_X>0\}} \mathbb{1}_A(x) \int_X \mathbb{1}_B(y) p(x, y) d\nu(y) d\mu(x), \\ &= \int_{X \times Y} \mathbb{1}_{A \cap \{p_X>0\}} \mathbb{1}_B p d(\mu \otimes \nu), \\ &= \mathbb{P}((A \cap \{p_X > 0\}) \times B), \\ &= \mathbb{P}(A \times B). \end{aligned}$$

Consequently,  $\mathbb{P}(\cdot | \cdot) : \mathcal{B} \times X \rightarrow [0, 1]$  is indeed a regular conditional probability of  $\mathbb{P}$  given  $X$ . This result allows us to consider probability density functions instead of probability distributions in the Bayesian Inference setting.

Lastly, we want to introduce the famous Bayes' formula, see e.g [9, Theorem 8.7].

**Theorem 3.1.2.** *Let  $(\Omega, \mathcal{A}, \mathbb{P})$  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  $\mathbb{P}(\bigcup_{i \in I} B_i) = 1$ .*

*Then for any  $A \in \mathcal{A}$  with  $\mathbb{P}(A) > 0$  and any  $k \in I$  holds*

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

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 their probability density functions. In particular, to generate a family of densities 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 density. Therefore, we cite a theorem from [2, Theorem 2.1], which proposes exactly such a construction.

**Theorem 3.1.3.** *Let  $p(x)$  be a probability density function and  $\mu \in \mathbb{R}$ ,  $\sigma > 0$  constants. Then the following quantities define probability density functions*

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

We refer to the parameter  $\mu$  as **location parameter** and  $\sigma$  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**.

Before continuing with the theory, let us consider some examples.

**Example 3.1.4.** The following distributions allow us to define location-scale families.

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

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

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  $\mu$  and the scale parameter  $\sigma$ , respectively. This leads to

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

Therefore, we can write each normal distribution as an altered standard normal distribution.

The Bayes formula, introduced in Theorem 3.1.2, lead to an entire optimization ecosystem. The conceptional idea of this ecosystem is to iteratively update ones belief of the knowledge one possesses about some observed data. This means that one first assumes some kind of knowledge about their 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 the knowledge no longer (significantly) changes upon updating. This optimization setting is commonly known as Variational Bayes or Variational Inference and can be introduced as follows.

We assume to have some observations  $x = (x_1, \dots, x_n)$  that are generated from a random variable  $X$ , which we thus call the **observation random variable**. We assume that the distribution of  $X$  has a probability density function  $p(x)$ . We will refer to it as the **evidence**. Usually, this probability density function is highly complicated and thus, our goal is to achieve a better understanding of it. We do so, by assuming 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). For this unknown or (or latent) reason we introduce another random variable  $Z$  which we will refer to as **latent random variable**. Moreover, we assume that this latent random variable  $Z$  does have a known and „easily understandable“ probability distribution with density  $p(z)$  (e.g. a Gaussian distribution). This probability density we will refer to as **prior**, since we assume the density before considering the observations  $x$ . The next step is to introduce the conditional density  $p(x|z)$ , which is a function of the observed data  $x$ , given the realisations  $z$  of the

latent variable  $Z$ . It quantifies how likely the observed data are under the assumed statistical model and its specific parameter values. This quantity we will refer to as **likelihood**. If we now assume, that there exists a joint distribution of  $X$  and  $Z$ , we can represent the density  $p(x)$  as the marginal density

$$p(x) = \int p(x|z)p(z)dz, \quad (3.1)$$

where we integrate the likelihood function  $p(x|z)$  over all possible priors  $p(z)$ . This means that we asses how good the realisation  $z$  of the latent variable  $Z$  describes our observed data  $x$ , where we additionally consider how likely it is that  $z$  occurs. Lastly, we want to update our belief  $p(z)$  after considering the observations  $x$ . In order to do so, we need some kind of update rule which we can define using the Bayesian formula from Theorem 3.1.2. This leads to the conditional density

$$p(z|x) = \frac{p(x|z)p(z)}{p(x)}, \quad (3.2)$$

which is usually referred to as the **(true) posterior**. This completes the Bayesian Inference setting.

Furthermore, we must consider how to actually compare densities to one another. We want to update the prior density  $p(z)$  until it „no longer (significantly) changes“ or in other words, until it converges. But what does actually convergence mean in this setting? A common approach to quantify the discrepancy between two probability densities is the Kullback-Leibler divergence. It is a relative measure, which assesses the dissimilarity between two probability densities over the same random variable  $X$ .

**Definition 3.1.5.** Let  $(\Omega, \mathcal{A}, \mathbb{P})$  be a probability space and let  $X : \Omega \rightarrow \mathbb{R}^n$  be a random variable. Furthermore, let  $p$  and  $q$  be two probability densities of  $\mathbb{P}_X$ , respectively. Then the **Kullback-Leibler divergence** from  $q$  to  $p$  is defined as

$$D_{\text{KL}}(p \parallel q) = \int_{\Omega} p(x) \log \left( \frac{p(x)}{q(x)} \right) d\mathbb{P}_X(x) = \mathbb{E}_p [\log p(x) - \log q(x)]. \quad (3.3)$$

We note that the Kullback-Leibler divergence is not a symmetrical quantity and it holds that  $D_{\text{KL}}(p \parallel q) \geq 0$ , which follows directly from Jensen's inequality. Moreover,  $D_{\text{KL}}(p \parallel q) = 0$  holds if and only if  $p = q$ , see [9, Section 23.3].

In applications one usually computes the Kullback-Leibler divergence with regard to some kind of dataset consisting of observed data samples. Then the Kullback-Leibler divergence would look as in the following definition.

**Definition 3.1.6.** Let  $(\Omega, \mathcal{A}, \mathbb{P})$  be a probability space and let  $X : \Omega \rightarrow \mathbb{R}^n$  be a random variable and let  $p$  and  $q$  be two probability densities of  $\mathbb{P}_X$ , respectively. Furthermore, let  $D = \{x_1, \dots, x_L\}$  be an unsupervised dataset of length  $L \in \mathbb{N}$ . Then, the **Kullback-Leibler divergence** from  $q$  to  $p$  with respect to the dataset  $D$  is defined as

$$D_{\text{KL}}(p \parallel q) = \sum_{i=1}^L p(x_i) \log \left( \frac{p(x_i)}{q(x_i)} \right). \quad (3.4)$$

The Kullback-Leibler divergence, which we introduced in Definition 3.1.5 and Definition 3.1.6 is also known as relative entropy. The entropy intuitively describes how much „randomness“ a random variable possesses, i.e. the less we can predict the outcome of a random event, the more entropy the corresponding random variable has.

**Definition 3.1.7.** Let  $(\Omega, \mathcal{A}, \mathbb{P})$  be a probability space and let  $X : \Omega \rightarrow \mathbb{R}^n$  be a random variable and let  $p$  be a probability density of  $\mathbb{P}_X$ . Then the function

$$\mathcal{H}(p) = - \int_{\Omega} p(x) \log p(x) d\mathbb{P}_X(x) = \mathbb{E}_p [-\log p(X)], \quad (3.5)$$

is called **entropy** of  $p$ .

We considered the „randomness“ of a random variable, which we describes as entropy of a probability density in Definition 3.1.7. Another important quantity is the so-called cross-entropy between two probability densities. It intuitively quantifies how much information one density possesses over the random variable in opposition to the other density. This we will formally define in the following definition.

**Definition 3.1.8.** Let  $(\Omega, \mathcal{A}, \mathbb{P})$  be a probability space and let  $X : \Omega \rightarrow \mathbb{R}^n$  be a random variable and let  $p$  and  $q$  be two probability densities of  $\mathbb{P}_X$ , respectively. Then the function

$$\mathcal{H}_q(p) = - \int_{\Omega} p(x) \log q(x) d\mathbb{P}_X(x) = \mathbb{E}_p [-\log q(X)], \quad (3.6)$$

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

With the help of the entropy, which we defined in Definition 3.1.7 and cross-entropy, which we defined in Definition 3.1.8, we can write the Kullback-Leibler divergence as their difference, as we can easily compute

$$\begin{aligned} D_{\text{KL}}(p \| q) &= \int_{\Omega} p(x) \log \left( \frac{p(x)}{q(x)} \right) d\mathbb{P}_X(x) \\ &= \int_{\Omega} p(x) \log p(x) d\mathbb{P}_X(x) - \int_{\Omega} p(x) \log q(x) d\mathbb{P}_X(x) = \mathcal{H}_q(p) - \mathcal{H}(p). \end{aligned} \quad (3.7)$$

## 3.2 Variational Bayes meets Autoencoders

In Section 3.1 we introduced the Variational Inference, also known as the Variational Bayes setting. This setting allows us to iteratively improve our understanding of the observed data, which we model through the observation generating probability density function  $p(x)$ . This approach can be applied to autoencoding neural networks, as proposed in [8]. The idea is to consider an autoencoding neural network with parameters  $\theta$ , where we assume the parameters to be variable. In the Bayesian learning setting it is a common approach to not distinguish between latent variables and model parameters, since they both are unknown quantities. Therefore, we can consider the likelihood function  $p(x|z)$  as the output of a neural network, where the parameters  $\theta$  are determined by the realisation  $z$  of the latent variable  $Z$ . Thus, we denote the likelihood function as  $p_{\theta}(x|z)$ . Moreover, since we consider generative models the output of the neural network is not a vector but a probability density. If we assume it to be Gaussian, this means that the likelihood function would look like

$$p_{\theta}(x|z) = \mathcal{N}(\mu_{nn}(z), \sigma_{nn}(z)), \quad (3.8)$$

where  $\mu_{nn}(z)$  denotes the mean of the likelihood function and  $\sigma_{nn}(z)$  denotes the variance of the likelihood function. If we now define the dimensions of the Gaussian distribution from equation (3.8) to be smaller than the dimensions of the data samples  $x_i$ , we speak of a **probabilistic encoding neural network**, or simply a **probabilistic encoder**. Furthermore, if we consider that the evidence is the marginal density as in equation (3.1) we realise that this integral can not be computed in a closed-form. Therefore, the posterior density, which is defined as the fraction (3.2), can not be

computed either and we need to approximate it. For this manner, we introduce the density  $q_\phi$  with parameters  $\phi$ , which approximates

$$q_\phi(z|x_i) \approx p(z|x_i),$$

and assume that it is normally distributed with parameters  $\mu_i$  and  $\sigma_i$ , i.e.  $q_\phi(z|x_i) \sim \mathcal{N}(\mu_i, \sigma_i)$ . At this point it is worth highlighting that the assumed distribution of the approximate posterior  $q_\phi$  depends on the the assumed distribution on the true posterior  $p(z|x)$ . Since in applications one usually assumes the true posterior to be normally distributed, one therefore assumes the approximate posterior to be normally distributed as well.

Considering the approximate posterior  $q_\phi$  leads to the following consideration of the log of the evidence  $p(x)$

$$\log p(x_i) = \log \int p_\theta(x_i|z)p(z)dz = \log \int p_\theta(x_i|z)p(z) \frac{q_\phi(z|x_i)}{q_\phi(z|x_i)} dz = \log \mathbb{E}_{q_\phi(z|x_i)} \left[ \frac{p_\theta(x_i|z)p(z)}{q_\phi(z|x_i)} \right]. \quad (3.9)$$

If we now apply Jensen's inequality, see e.g. [9, Theorem 7.9], which states that  $\mathbb{E}[\varphi(X)] \geq \varphi(\mathbb{E}[X])$ , if  $\varphi$  is convex and hence,  $\mathbb{E}[\varphi(X)] \leq \varphi(\mathbb{E}[X])$ , if  $\varphi$  is concave. Due to the fact that  $\log$  is a concave function, we receive

$$\log p(x_i) \geq \mathbb{E}_{q_\phi(z|x_i)} \left[ \log \left( \frac{p_\theta(x_i|z)p(z)}{q_\phi(z|x_i)} \right) \right] = \mathbb{E}_{q_\phi(z|x_i)} [\log p_\theta(x_i|z) + \log p(z) - \log q_\phi(z|x_i)].$$

This inequality is very important, since it allows us to maximize the likelihood  $p_\theta(x_i|z)$  in the following. Since it is so important, it is commonly referred to as the **Evidence Lower Bound (ELBO)**. Furthermore, we can make use of the linearity of the integral, which leads to

$$\log p(x_i) \geq \mathcal{L}(\theta, \phi; x_i) := \mathbb{E}_{q_\phi(z|x_i)} [\log p_\theta(x_i|z)] + \mathbb{E}_{q_\phi(z|x_i)} [\log p(z)] - \mathbb{E}_{q_\phi(z|x_i)} [\log q_\phi(z|x_i)]. \quad (3.10)$$

Considering the fact that we introduced the density  $q_\phi(z|x_i)$  in order to approximate the posterior  $p(z|x_i)$ , we realise that our goal is to make this approximation as tight as possible. Therefore, we consider the Kullback-Leibler divergence between them. This gives us

$$D_{\text{KL}}(q_\phi(z|x_i) \| p(z|x_i)) = \mathbb{E}_{q_\phi(z|x_i)} \left[ \log \left( \frac{q_\phi(z|x_i)}{p(z|x_i)} \right) \right].$$

If we now apply the Bayes formula to the right-hand side, see Theorem 3.1.2, we receive

$$\mathbb{E}_{q_\phi(z|x_i)} \left[ \log \left( \frac{q_\phi(z|x_i)}{p(z|x_i)} \right) \right] = \mathbb{E}_{q_\phi(z|x_i)} \left[ \log \left( \frac{q_\phi(z|x_i)p(x_i)}{p(x_i, z)} \right) \right] = \mathbb{E}_{q_\phi(z|x_i)} \left[ \log \left( \frac{q_\phi(z|x_i)p(x_i)}{p_\theta(x_i|z)p(z)} \right) \right].$$

The next step is to consider the linearity of the integral and the fact that holds  $\log(ab) = \log a + \log b$ , which yields

$$\begin{aligned} & \mathbb{E}_{q_\phi(z|x_i)} \left[ \log \left( \frac{q_\phi(z|x_i)p(x_i)}{p_\theta(x_i|z)p(z)} \right) \right] \\ &= \mathbb{E}_{q_\phi(z|x_i)} [\log q_\phi(z|x_i) + \log p(x_i)] - \mathbb{E}_{q_\phi(z|x_i)} [\log p_\theta(x_i|z) + \log p(z)]. \end{aligned}$$

Comparing this equation to the ELBO, determined in equation (3.10), we realise that we can express the Kullback-Leibler divergence  $D_{\text{KL}}(q_\phi(z|x_i) \| p(z|x_i))$  as

$$D_{\text{KL}}(q_\phi(z|x_i) \| p(z|x_i)) = -\mathcal{L}(\theta, \phi; x_i) + \mathbb{E}_{q_\phi(z|x_i)} [\log p(x_i)] = -\mathcal{L}(\theta, \phi; x_i) + \log p(x_i),$$

where the last equation holds true due to the fact that  $\log p(x_i)$  does not depend on  $z$ .

At this point, it is worth mentioning that the Kullback-Leibler divergence is the difference between

the ELBO, see (3.10), which is the lower bound on the log of the evidence  $p(x_i)$  and the log of the evidence  $p(x_i)$  itself. Therefore, the tighter the ELBO is, the smaller the Kullback-Leibler divergence between approximation  $q_\phi(z|x_i)$  and posterior  $p(z|x_i)$  becomes. This leads to the representation

$$\mathcal{L}(\theta, \phi; x_i) = -D_{\text{KL}}(q_\phi(z|x_i) \| p(z)) + \mathbb{E}_{q_\phi(z|x_i)} [\log p_\theta(x_i|z)], \quad (3.11)$$

where we swapped the prior  $p(z)$  and posterior  $p_\theta(z|x_i)$  between Kullback-Leibler divergence and expectation. This is allowed due to the fact that we consider  $\mathbb{E}_{q_\phi(z|x_i)}[\cdot]$  and  $D_{\text{KL}}(q_\phi(z|x_i) \| \cdot)$ .

We want to optimize the ELBO  $\mathcal{L}(\theta, \phi; x_i)$  with regard to both the variational parameters  $\phi$  and generative parameters  $\theta$ . The common approach for the variational parameters  $\phi$  would be to determine the gradient of  $\mathcal{L}(\theta, \phi; x_i)$  with respect to the parameters and optimize it with a gradient ascent algorithm, i.e. an algorithm that iteratively maximizes  $\mathcal{L}(\theta, \phi; x_i)$ . However, computing the gradient is usually not possible in closed-form due to the size of data and the complexity of  $\mathcal{L}(\theta, \phi; x_i)$ . Thus, one has to approximates the gradient with e.g. a Monte Carlo Gradient estimator, see e.g. [1, Section 11.1.6], which would look like

$$\nabla_\phi \mathbb{E}_{q_\phi(z)} [f(z)] = \mathbb{E}_{q_\phi(z)} \left[ f(z) \nabla_{q_\phi(z)} \log q_\phi(z) \right] \simeq \frac{1}{L} \sum_{l=1}^L f(z_l) \nabla_{q_\phi(z_l)} \log q_\phi(z_l),$$

where  $z_l \sim q_\phi(z|x_i)$ , see [8, Section 2.2]. Unfortunately, this gradient estimator has shown to exhibit very high variance, see [16] and therefore, it is impractical for our purposes. To address this matter we choose a slightly different approach. We assume an (auxiliary) noise variable  $\epsilon$ , which we assume to be distributed with density  $\epsilon \sim p(\epsilon)$ . Then we can reparameterize  $\tilde{z} \sim q_\phi(z|x)$  using a differentiable transformation  $g_\phi(\epsilon, x)$ , such that

$$\tilde{z} = g_\phi(\epsilon, x) \quad \epsilon \sim p(\epsilon).$$

Moreover, we consider that we can form Monte Carlo estimates of expectations of some function  $f(z)$  with regard to the approximate posterior  $q_\phi(z|x_i)$  as

$$\mathbb{E}_{q_\phi(z|x)} [f(z)] = \mathbb{E}_{p(\epsilon)} [f(g_\phi(\epsilon, x_i))] \simeq \frac{1}{L} \sum_{l=1}^L f(g_\phi(\epsilon_l, x_i)).$$

If we apply this technique to the ELBO from equation (3.10), then we receive the **Stochastic Gradient Variational Bayes (SGVB)** estimator  $\tilde{\mathcal{L}}^A(\theta, \phi; x_i) \simeq \mathcal{L}(\theta, \phi; x_i)$ :

$$\tilde{\mathcal{L}}^A(\theta, \phi; x_i) = \frac{1}{L} \sum_{l=1}^L \log p_\theta(x_i|z_{i,l}) + \log p(z_{i,l}) - \log q_\phi(z_{i,l}|x_i),$$

where  $z_{i,l} = g_\phi(\epsilon_l, x_i)$  and  $\epsilon_l \sim p(\epsilon)$ . Moreover,  $A$  denotes the first version of the SGVB estimator. Often, the Kullback-Leibler divergence  $D_{\text{KL}}(q_\phi(z|x_i) \| p(z))$  can be integrated analytically (e.g. if we assume  $q_\phi(z|x_i)$  and  $p(z)$  to be normally distributed), which we will consider later in this section. Therefore, we do only need to estimate by sampling the last term of the right-hand side of equation (3.11). This can be interpreted in the way that the Kullback-Leibler divergence encourages the approximate posterior  $q_\phi(z|x_i)$  to be close to the prior  $p(z)$  and thus, regularize  $\phi$ . This leads to a second version of the SGVB estimator

$$\tilde{\mathcal{L}}^B(\theta, \phi; x_i) = -D_{\text{KL}}(q_\phi(z|x_i) \| p(z)) + \frac{1}{L} \sum_{l=1}^L \log(p_\theta(x_i|z_{i,l})), \quad (3.12)$$

where  $z_{i,l} = g_\phi(\epsilon_l, x_i)$  and  $\epsilon_l \sim p(\epsilon)$ . Moreover,  $B$  denotes the second version of the SGVB estimator. At this point, we want to mention that if we have a dataset  $D$  with  $N \in \mathbb{N}$  observations, i.e.  $D =$

$\{x_1, \dots, x_N\}$  and randomly choose mini-batches  $B$  of length  $M \in \mathbb{N}$ , then we can construct an SGVB estimator of the full dataset  $D$  by

$$\mathcal{L}(\theta, \phi; D) \simeq \mathcal{L}(\theta, \phi; B) = \frac{N}{M} \sum_{i=1}^M \mathcal{L}(\theta, \phi; x_i).$$

We can determine the gradients  $\nabla_{\theta, \phi} \mathcal{L}(\theta, \phi; B)$  and thus, use these gradients in optimization techniques, see Section 1.3, to optimize the parameters  $\theta$  and  $\phi$ .

Furthermore, we want to highlight at this point that equation (3.12) clearly shows the connection to auto encoding neural networks. The function  $g_\phi(\cdot)$  is chosen such that it maps an observation  $x_i$  and a random noise vector  $\epsilon_l$  to a sample of the approximate posterior  $z_{i,l} = g_\phi(\epsilon_l, x_i)$ , where  $z_{i,l} \sim q_\phi(z|x_i)$ . Subsequently, the drawn sample  $z_{i,l}$  is then fed into  $\log p(x_i|z_{i,l})$ , which therefore equals the probability density of the observation  $x_i$  under the generative model  $p_\theta(x_i|z_{i,l})$ , given  $z_{i,l}$ . We can interpret the first term of the right-hand side of equation (3.12) (the Kullback-Leibler divergence of approximate posterior  $q_\phi(z|x_i)$  to prior  $p(z)$ ) as a regularizer and the second-term of the right hand side as an expected negative reconstruction error.

The next obstacle we want to consider is the fact that it is unclear of how to actually compute the gradient in the sampling operation  $z \sim q_\phi(z|x)$ . In order to solve this problem, the authors Kingma and Welling of the original paper, see [8] applied a simple trick to omit this sampling with regard to distributions with different parameters. At this point we want to remember Example 3.1.4, where we saw that we can represent each Gaussian density as a scaled standard Gaussian density. This is exactly why we introduced the differentiable transformation  $g_\phi(\epsilon, x)$  with some auxiliary noise variable  $\epsilon$ . First, we can control the probability density  $p(\epsilon)$  of  $\epsilon$  and second,  $g_\phi(\cdot)$  is some vector-valued function that we control through the parameter  $\phi$  as well. With these considerations we can now rewrite the Monte Carlo estimate of the expectation  $\mathbb{E}_{q_\phi(z|x)}[\cdot]$  such that it is differentiable with regard to  $\phi$ . Due to the fact that  $q_\phi(z|x)$  and  $p(\epsilon)$  are both probability density functions, this leads to

$$\int q_\phi(z|x) dz = \int p(\epsilon) d\epsilon.$$

Therefore, given the mapping  $z = g_\phi(\epsilon, x)$ , if we compute the expectation of a function  $f(z)$  with respect to the approximate posterior  $q_\phi(z|x)$ , we receive

$$\int q_\phi(z|x) f(z) dz = \int p(\epsilon) f(g_\phi(\epsilon, x)) d\epsilon.$$

This leads to the differentiable Monte Carlo estimator

$$\int q_\phi(z|x) f(z) dz \simeq \frac{1}{L} \sum_{l=1}^L f(g_\phi(\epsilon_l, x)),$$

where  $\epsilon_l \sim p(\epsilon)$  for all  $l = 1, \dots, L$ .

For example, we assume the true posterior to be normally distributed, i.e.  $z \sim p(z|x) = \mathcal{N}(\mu, \sigma^2)$  with some mean  $\mu \in \mathbb{R}$  and some standard deviation  $\sigma > 0$ . Therefore, a valid reparameterization would be  $z = \mu + \sigma \epsilon$ , where the auxiliary noise variable  $\epsilon$  is standard normally distributed, i.e.  $\epsilon \sim p(\epsilon) = \mathcal{N}(0, 1)$ . This then leads to the computation of the expectation

$$\mathbb{E}_{z \sim \mathcal{N}(\mu, \sigma^2)}[f(z)] = \mathbb{E}_{\epsilon \sim \mathcal{N}(0, 1)}[f(\mu + \sigma \epsilon)] \simeq \frac{1}{L} \sum_{l=1}^L f(\mu + \sigma \epsilon_l),$$

where  $\epsilon_l \sim \mathcal{N}(0, 1)$  for all  $l = 1, \dots, L$ .

Having introduced the general setting of variational autoencoders, we now want to consider a specific example of a maximum likelihood estimation for a Gaussian distribution.

**Example 3.2.1.** Let  $(\Omega, \mathcal{A}, \mathbb{P})$  be a probability space and for  $i = 1, \dots, N$  let  $X_i : \Omega \rightarrow \mathbb{R}^D$  be observation random variables, where all random variables  $X_i$  are assumed to be independent and underly a multivariate Gaussian distribution. Hence for  $X = (X_1, \dots, X_N)$  it holds that  $X \sim \mathcal{N}(\mu, \Sigma)$ , where  $\mu$  denotes the mean and  $\Sigma$  denotes the covariance matrix of the multivariate normal probability distribution. We can estimate the parameters of this Gaussian distribution through maximum likelihood estimation, see e.g. [1, Section 2.3.4]. First, we consider that the likelihood function for each  $X_i$  is given by

$$p(X_i | \mu, \Sigma) = (2\pi)^{-D/2} \det(\Sigma)^{-1/2} \exp\left(-\frac{1}{2}(x_i - \mu)^\top \Sigma^{-1}(x_i - \mu)\right).$$

Moreover, since all  $X_i$  are assumed to be independent and identically distributed, we can denote the probability density function of  $X$  as the product of the marginal densities

$$p(X | \mu, \Sigma) = \prod_{i=1}^N (2\pi)^{-D/2} \det(\Sigma)^{-1/2} \exp\left(-\frac{1}{2}(x_i - \mu)^\top \Sigma^{-1}(x_i - \mu)\right)$$

Therefore,  $\log p(X | \mu, \Sigma)$  looks like

$$\begin{aligned} \log p(X | \mu, \Sigma) &= \log \prod_{i=1}^N p_{X_i}(x_i | \mu, \Sigma) \\ &= \log \prod_{i=1}^N (2\pi)^{-D/2} \det(\Sigma)^{-1/2} \exp\left(-\frac{1}{2}(x_i - \mu)^\top \Sigma^{-1}(x_i - \mu)\right) \\ &= \sum_{i=1}^N \left( -\frac{D}{2} \log(2\pi) - \frac{1}{2} \log \det(\Sigma) - \frac{1}{2}(x_i - \mu)^\top \Sigma^{-1}(x_i - \mu) \right) \\ &= -\frac{DN}{2} \log(2\pi) - \frac{N}{2} \log \det(\Sigma) - \frac{1}{2} \sum_{i=1}^N ((x_i - \mu)^\top \Sigma^{-1}(x_i - \mu)). \end{aligned} \quad (3.13)$$

We realise that we can rewrite the last term of the right-hand side of equation (3.13) to

$$\begin{aligned} (x_i - \mu)^\top \Sigma^{-1}(x_i - \mu) &= \langle x_i - \mu, \Sigma^{-1}(x_i - \mu) \rangle = \langle x_i, \Sigma^{-1}(x_i - \mu) \rangle - \langle \mu, \Sigma^{-1}(x_i - \mu) \rangle \\ &= \langle x_i, \Sigma^{-1}x_i \rangle - \langle x_i, \Sigma^{-1}\mu \rangle - \langle \mu, \Sigma^{-1}x_i \rangle + -\langle \mu, \Sigma^{-1}\mu \rangle \end{aligned}$$

We see that the likelihood function depends on the data  $x_1, \dots, x_N$  only through the quantities

$$\sum_{i=1}^N x_i, \quad \text{and} \quad \sum_{i=1}^N x_i^\top x_i.$$

These quantities are known as **sufficient statistics** for the Gaussian distribution, see e.g. [14, Definition 15.8]. Moreover, if we consider the derivative of the log likelihood with respect to the mean  $\mu$ , we receive

$$\begin{aligned} \nabla_\mu \log p(X | \mu, \Sigma) &= \nabla_\mu \left( -\frac{DN}{2} \log(2\pi) - \frac{N}{2} \log \det(\Sigma) - \frac{1}{2} \sum_{i=1}^N ((x_i - \mu)^\top \Sigma^{-1}(x_i - \mu)) \right) \\ &= \sum_{i=1}^N \Sigma^{-1}(x_i - \mu). \end{aligned}$$

Since we are interested in maximizing the log likelihood, we set this derivative to zero. Thus, we obtain the solution for the maximum likelihood estimate  $\mu_{\text{ML}}$  of the mean  $\mu$  given by

$$\mu_{\text{ML}} = \frac{1}{N} \sum_{i=1}^N x_i,$$

which is the mean of all observed data points  $x_1, \dots, x_N$ . The maximization of the log likelihood with respect to the covariance matrix  $\Sigma_{\text{ML}}$  is rather more involved. Therefore, we kindly refer to [12, Chapter 15.3, Theorem 1], which states that the maximum likelihood estimate for the covariance matrix  $\Sigma_{\text{ML}}$  looks like

$$\Sigma_{\text{ML}} = \frac{1}{N} \sum_{i=1}^N (x_i - \mu_{\text{ML}}) (x_i - \mu_{\text{ML}})^T.$$

It is worth to highlight that the computation of the maximum likelihood estimator of the covariance matrix  $\Sigma_{\text{ML}}$  relies on knowing the maximum likelihood estimator of the mean  $\mu_{\text{ML}}$ . However, since we do not need  $\Sigma_{\text{ML}}$  to compute  $\mu_{\text{ML}}$ , we can simply compute  $\mu_{\text{ML}}$  first in each iteration.

Lastly in this chapter, we want to consider how to actually define a statistical learning setting. First, we remember that in statistical learning our aim is to find a prediction function (in our case a neural network), which minimizes the empirical risk function from Definition 1.1.12 over a given dataset  $D = \{(x_1, y_1), \dots, (x_N, y_N)\}$  of length  $N \in \mathbb{N}$ , i.e. find a neural network  $f$  that minimizes the empirical risk

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

Obviously, we need to consider how to adapt the loss function  $\mathcal{L}$  to the setting in this chapter, since we are not only interested in producing good point estimates, but also in minimizing the Kullback-Leibler divergence and hence, smoothing the latent space. In order to do that, we redefine the loss function  $\mathcal{L}$  as composition of reconstruction term and regularisation term. The reconstruction term is the same as in the previous chapters, e.g. the mean squared error (MSE) or the binary cross-entropy (BCE). The regularisation term on the other hand, we newly introduce at this point and choose it to be the Kullback-Leibler divergence between approximate posterior  $q_\phi(z|x)$  and true posterior  $p(z|x)$ . Moreover, we introduce the Kullback-Leibler coefficient  $\lambda \in [0, \infty)$ , which controls the weighting of each loss term. This results in

$$\mathcal{L}(x_i, y_i, f(x_i)) = \mathcal{L}_{\text{rec}}(x_i, y_i, f(x_i)) + \lambda D_{\text{KL}}(q_\phi(z|x_i) \| p(z|x_i)),$$

where  $\mathcal{L}_{\text{rec}}(x_i, y_i, f(x_i))$  denotes the reconstruction loss and  $D_{\text{KL}}(q_\phi(z|x_i) \| p(z|x_i))$  denotes the regularisation loss.

Furthermore, we note that the Kullback-Leibler divergence  $D_{\text{KL}}(q_\phi(z|x_i) \| p(z|x_i))$  can be computed in closed-form if one poses some assumptions about approximate posterior  $q_\phi(z|x_i)$  and true posterior  $p(z|x_i)$ . This we want to consider in the following lemma.

**Lemma 3.2.2.** *Let  $(\Omega, \mathcal{A}, \mathbb{P})$  be a probability space and let  $X : \Omega \rightarrow \mathbb{R}^n$  be a random variable. Furthermore, let  $p, q$  be probability density functions of  $\mathbb{P}_X$  with probability distributions  $p \sim \mathcal{N}(\mu_p, \sigma_p)$  and  $q \sim \mathcal{N}(\mu_q, \sigma_q)$ , respectively. Then for the Kullback-Leibler divergence from  $p$  to  $q$  holds*

$$D_{\text{KL}}(q \| p) = \log \frac{\sigma_q}{\sigma_p} - \frac{\sigma_q^2 + (\mu_p - \mu_q)^2}{2\sigma_p^2} + \frac{1}{2}.$$

*Proof.* First, we want to remember the Definition 3.1.5 of the Kullback-Leibler divergence to be

$$D_{\text{KL}}(q \parallel p) = - \int_{\Omega} q(x) \log \left( \frac{q(x)}{p(x)} \right) d\mathbb{P}_X(x) = - \int_{\Omega} q(x) \log q(x) d\mathbb{P}_X(x) + \int_{\Omega} q(x) \log p(x) d\mathbb{P}_X(x).$$

For the next step we want to consider  $\log q(x)$

$$\begin{aligned} \log q(x) &= \log \left( \frac{1}{2\sqrt{\pi\sigma_q^2}} \exp \left( -\frac{1}{2} \left( \frac{x - \mu_q}{\sigma_q} \right)^2 \right) \right) = -\frac{1}{2} \log 2\pi\sigma_q^2 - \frac{1}{2} \left( \frac{x - \mu_q}{\sigma_q} \right)^2 \\ &= -\frac{1}{2} \left( \log 2\pi\sigma_q^2 + \left( \frac{x - \mu_q}{\sigma_q} \right)^2 \right). \end{aligned}$$

Therefore, using the recently computed  $\log q(x)$ , we receive

$$-\int_{\Omega} q(x) \log q(x) d\mathbb{P}_X(x) = \frac{1}{2} \log 2\pi\sigma_q^2 \int_{\Omega} q(x) d\mathbb{P}_X(x) + \frac{1}{2} \int_{\Omega} q(x) \left( \frac{x - \mu_q}{\sigma_q} \right)^2 d\mathbb{P}_X(x).$$

We note that the second term of the right-hand side is exactly the expectation operator under the probability density  $q$  with respect to the centered and normalized random variable  $X$ . Therefore, it holds that

$$\int_{\Omega} q(x) \left( \frac{x - \mu_q}{\sigma_q} \right)^2 d\mathbb{P}_X(x) = \mathbb{E}_q \left[ \left( \frac{X - \mu_q}{\sigma_q} \right)^2 \right] = 1,$$

which again leads to

$$-\int_{\Omega} q(x) \log q(x) d\mathbb{P}_X(x) = \frac{1}{2} (\log 2\pi\sigma_q^2 + 1).$$

Moreover, we consider  $\log p(x)$  analogously to the previous computation of  $\log q(x)$ , which gives us

$$\begin{aligned} \int_{\Omega} q(x) \log p(x) d\mathbb{P}_X(x) &= -\frac{1}{2} \log 2\pi\sigma_p^2 \int_{\Omega} q(x) d\mathbb{P}_X(x) - \frac{1}{2} \int_{\Omega} q(x) \left( \frac{x - \mu_p}{\sigma_p} \right)^2 d\mathbb{P}_X(x) \\ &= -\frac{1}{2} \log 2\pi\sigma_p^2 - \frac{1}{2} \int_{\Omega} q(x) \left( \frac{x - \mu_p}{\sigma_p} \right)^2 d\mathbb{P}_X(x). \end{aligned}$$

Now we need to compute the second term of the right-hand side. In order to do so, we expand  $((x - \mu_p)/\sigma_p)^2$ , which then allows us to use the linearity of the integral. This gives us

$$\int_{\Omega} q(x) \left( \frac{x - \mu_p}{\sigma_p} \right)^2 d\mathbb{P}_X(x) = \frac{\int_{\Omega} q(x)x^2 d\mathbb{P}_X(x) - 2\int_{\Omega} q(x)x\mu_p d\mathbb{P}_X(x) + \int_{\Omega} q(x)\mu_p^2 d\mathbb{P}_X(x)}{\sigma_p^2}.$$

If we now denote  $\mathbb{E}_q[\cdot]$  as the expectation operator under  $q$ , we receive

$$\int_{\Omega} q(x) \left( \frac{x - \mu_p}{\sigma_p} \right)^2 d\mathbb{P}_X(x) = \frac{1}{\sigma_p^2} (\mathbb{E}_q[X^2] - 2\mu_p \mathbb{E}_q[X] + \mu_p^2).$$

Moreover, we know that  $\sigma_q^2 = \mathbb{E}_q[X^2] - \mathbb{E}_q[X]^2$ , which leads to

$$\int_{\Omega} q(x) \left( \frac{x - \mu_p}{\sigma_p} \right)^2 d\mathbb{P}_X(x) = \frac{1}{\sigma_p^2} (\sigma_q^2 + \mu_q^2 - 2\mu_p\mu_q + \mu_p^2) = \frac{\sigma_q^2 + (\mu_p - \mu_q)^2}{\sigma_p^2}.$$

Combining all results then gives us

$$\begin{aligned} D_{\text{KL}}(q \parallel p) &= \frac{1}{2} (\log 2\pi \sigma_q^2 + 1) - \frac{1}{2} \log 2\pi \sigma_p^2 - \frac{\sigma_q^2 + (\mu_p - \mu_q)^2}{2\sigma_p^2} \\ &= \log \frac{\sigma_q}{\sigma_p} - \frac{\sigma_q^2 + (\mu_p - \mu_q)^2}{2\sigma_p^2} + \frac{1}{2}. \end{aligned}$$

□

If we now apply Lemma 3.2.2 to the regularisation loss  $D_{\text{KL}}(q_\phi(z|x) \parallel p(z|x))$ , we can write it explicitly as

$$D_{\text{KL}}(q_\phi(z|x) \parallel p(z|x)) = \log \frac{\sigma_q}{\sigma_p} - \frac{\sigma_q^2 + (\mu_p - \mu_q)^2}{2\sigma_p^2} + \frac{1}{2},$$

where we assumed that the approximate posterior  $q_\phi(z|x)$  and the true posterior  $p(z|x)$  are both normally distributed. This in turn, allows us to denote the loss function  $\mathcal{L}$  explicitly as

$$\mathcal{L}(x, y, f(x)) = \mathcal{L}_{\text{rec}}(x, y, f(x)) + \lambda \left( \log \frac{\sigma_q}{\sigma_p} - \frac{\sigma_q^2 + (\mu_p - \mu_q)^2}{2\sigma_p^2} + \frac{1}{2} \right), \quad (3.14)$$

for an arbitrary sample  $(x, y) \in D$  and some fixed Kullback-Leibler coefficient  $\lambda \in [0, \infty)$ . Therefore, the empirical risk function can be written as

$$\mathcal{R}_{\mathcal{L}, D}(f) = \frac{1}{N} \sum_{i=1}^N \left( \mathcal{L}_{\text{rec}}(x_i, y_i, f(x_i)) + \lambda \left( \log \frac{\sigma_{q,i}}{\sigma_{p,i}} - \frac{\sigma_{q,i}^2 + (\mu_{p,i} - \mu_{q,i})^2}{2\sigma_{p,i}^2} + \frac{1}{2} \right) \right),$$

where each sample  $(x_i, y_i) \in D$  is encoded onto an own density  $q_\phi(z|x_i) \sim \mathcal{N}(\mu_{q,i}, \sigma_{q,i})$  and  $p(z|x_i) \sim \mathcal{N}(\mu_{p,i}, \sigma_{p,i})$ , respectively.

### 3.3 Applications

Finally, we want to consider some applications of variational autoencoders, similar to Section 2.3. We trained linear and convolutional autoencoders on the MNIST dataset, a dataset which consists of handwritten digits. As discussed in Section 3.2, in the course of training we want to minimize the (empirical) risk function with respect to given data, iteratively. In order to do so we can apply some kind of training algorithm proposed in Section 1.3, e.g. the stochastic gradient descent (SGD) or the AMSGrad algorithm. 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 how the Kullback-Leibler coefficient (KL-coefficient)  $\lambda$  affects the learned probability distributions, since this coefficient intuitively controls how the ratio between reconstruction loss and Kullback-Leibler divergence is considered during training. The resulting variational autoencoder is „more similar“ to a discriminative model if the coefficient is chosen to be smaller. Lastly, we will take a look at the training progress of each model we train. Additionally, we came up with an idea of our own of how to improve the reconstruction capability of the variational autoencoder and make it more stable with regard to the KL-coefficient in lower dimensions.

However, before actually looking at trained examples, we want to formulate a general approach of how a training algorithm for a variational autoencoder looks like, see Algorithm 6.

Now we want to go ahead and consider some examples. We begin by with model where the probabilistic encoder encodes the data onto a two dimensional normal distribution. Furthermore, we assume the true posterior  $p(z|x)$  to be a standard normal distribution, what is a common approach in

---

**Algorithm 6** Variational 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, respectively. Furthermore, let the probabilistic encoder map onto a  $k$ -dimensional probability density, where we call  $k \in \mathbb{N}$  the bottleneck.

Let the chosen optimizer be AMSGrad with a learning rate  $\gamma > 0$  and KL-coefficient  $\lambda \in [0, \infty)$ . Furthermore, let the chosen loss function be the MSE loss function. Then the training of a variational autoencoder looks as follows.

```

Require: $\gamma \leftarrow 3 \times 10^{-4}$ ▷ Declare a learning rate.
Require: $\lambda \in [0, \infty)$ ▷ Declare a KL-coefficient.
Require: $p(z|x) \sim \mathcal{N}(\mu_p, \Sigma_p)$ ▷ Declare a k -dimensional true posterior density.
Require: $q(z|x) \sim \mathcal{N}(\mu_q, \Sigma_q)$ ▷ Declare a k -dimensional approximative posterior density.

1: for epoch in epochs do
2: for image in batch do
3: $\mu, \Sigma = \text{encoder}(\text{image})$ ▷ Encode the image onto mean and variance.
4: $z \sim q(z|x)$ ▷ Sample from approximative posterior density.
5: reconstructed = decoder(z) ▷ Decode the generated sample.
6: KL-divergence = $D_{\text{KL}}(q(z|x) \parallel p(z|x))$ ▷ Compute the regularisation loss.
7: $\mathcal{L}_{\text{rec}} = \text{MSE}(\text{reconstructed}, \text{image})$ ▷ Compute the reconstruction loss.
8: loss = $\mathcal{L}_{\text{rec}} + \lambda \cdot \text{KL-divergence}$ ▷ Compute the overall loss.
9: optimization(loss, γ) ▷ Perform an optimization step.
10: end for
11: end for

```

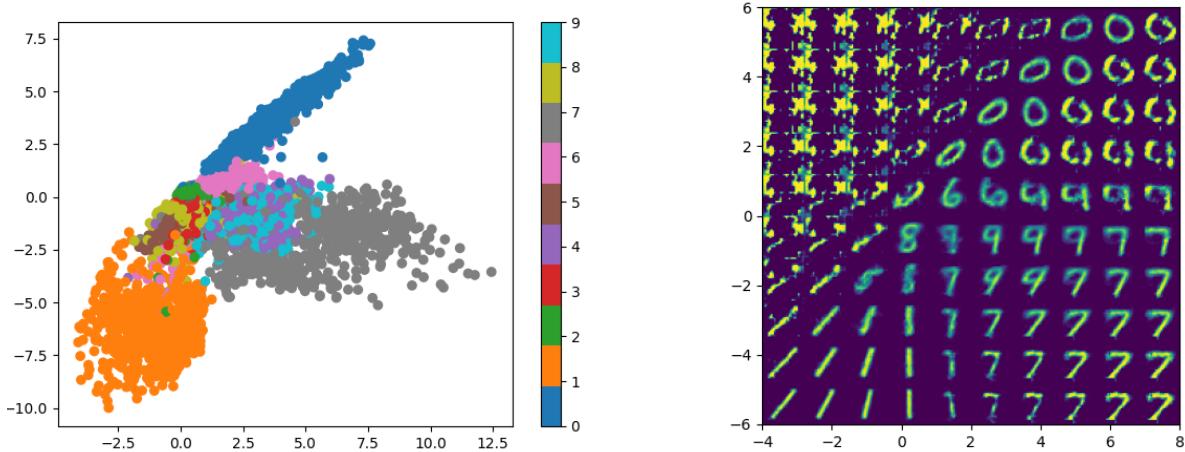
---

practice in order to control the learned approximations  $q(z|x)$  to be as close to the standard normal distribution as possible. Moreover, we choose the KL-coefficient to be  $\lambda = 4 \times 10^{-2}$ . At this point it is worth highlighting that the KL-coefficient depends on the specific implementation. There are various ways to implement the loss function (e.g. compute the mean or just sum the single values) and thus, the coefficient needs to be chosen according to the rest of the neural network architecture. Upon training such a variational autoencoder, we then visualise its latent space on the left-hand side of Figure 3.1, where each dot represents a sample from the encoded distribution of a data point. The colour of the dot represents the digit, which was encoded in the first place. We can see that the encodings in the latent space are located in the interval  $[-4, 8] \times [-6, 6]$ , whereas comparing to ordinary autoencoders, see e.g. Figure 2.4, the encodings can not be limited locally as easily. This is a huge benefit of variational autoencoders, since it makes generating new samples much easier. Moreover, on the right-hand side of Figure 3.1 we can see the corresponding reconstruction of the latent space, where we define a mesh in the 2-dimensional plane which consists of  $10 \times 10$  points and feed them into the decoding structure of the variational autoencoder to obtain a reconstructed image.

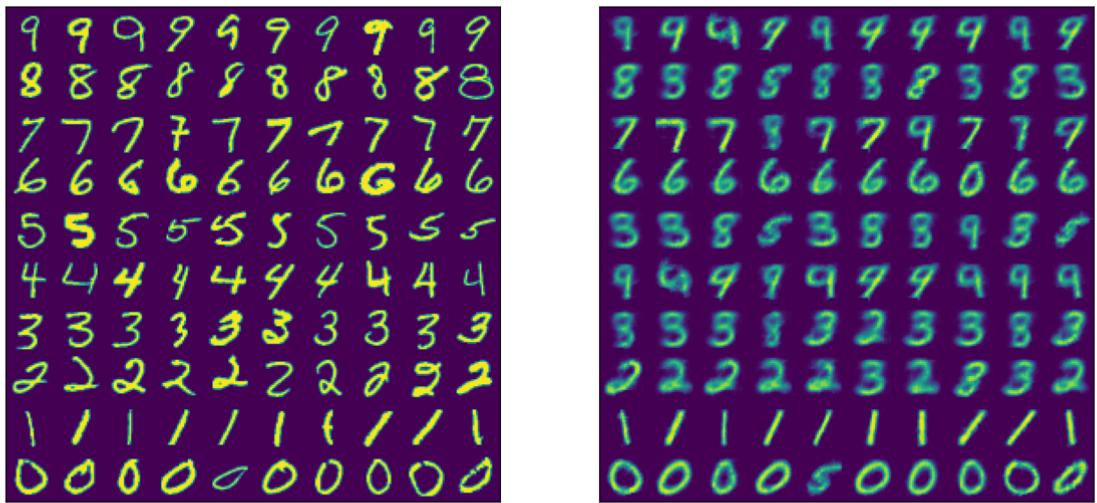
Another visualisation we want to take a look at is the reconstruction capability of the variational autoencoder. As we did with ordinary autoencoders, we randomly select ten images for each of the ten digits and feed them into the variational autoencoder in order to produce a reconstruction. These reconstructions we can see in Figure 3.2. We can see that the neural network does behave as desired. However, the bottleneck of  $n_b = 2$  is too small for good reconstructions.

Furthermore, we want to take a look at the training progress, which is depicted in Figure 3.3. The blue line represents the training loss in each epoch and the orange line represents the moving average of the training loss with regard to the 100 surrounding epochs to give an intuition of the trend of the training progress. We see that similarly to the autoencoders, see e.g. Figure 2.8, the training loss first falls rapidly and then after roughly 1000 epochs barely changes.

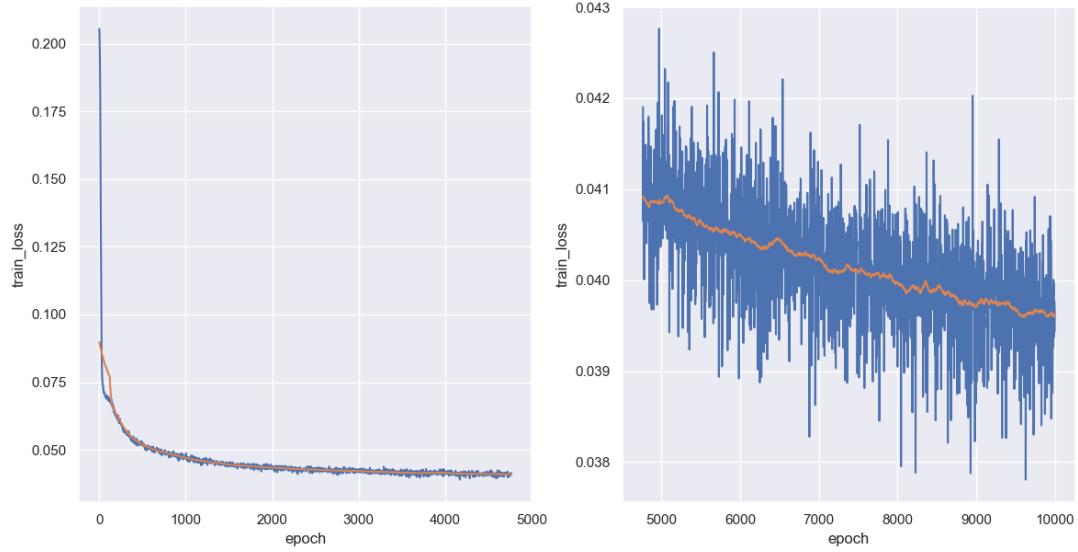
Lastly, we visualise the reconstruction errors of the neural network similarly as we did in Chapter 2.



**Figure 3.1:** On the left-hand side, the figure illustrates the latent space of the variational autoencoder with bottleneck  $n_b = 2$  and KL-coefficient  $\lambda = 4 \times 10^{-5}$ , where each dot is a sample from an encoded image of a digit onto a probability density. The color and the corresponding color map represent the digit that was encoded. On the right-hand side the figure illustrates the corresponding reconstruction through the variational autoencoder, where each point of the partitioned interval  $[-4, 8] \times [-6, 6]$  is fed into the decoding architecture of the neural network to generate an image.



**Figure 3.2:** On the left-hand side, the figure illustrates 100 original digits from the MNIST dataset. On the right-hand side, the figure illustrates the same digits after feeding them through the variational autoencoder with bottleneck  $n_b = 2$  and KL-coefficient  $\lambda = 4 \times 10^{-5}$  to produce a reconstruction.



**Figure 3.3:** The figure illustrates the training progresses of the variational autoencoder with bottleneck  $n_b = 2$  and KL-coefficient  $\lambda = 4 \times 10^{-5}$  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.

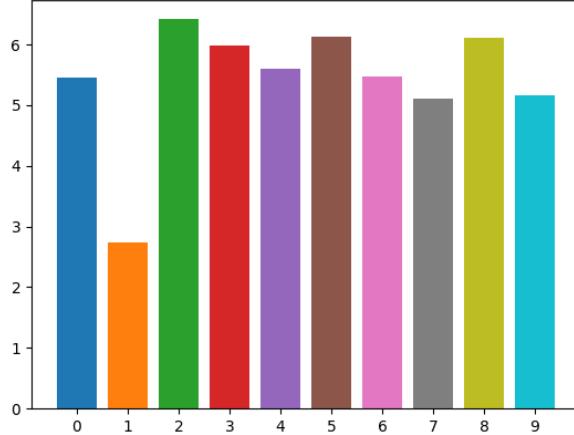
For each of the ten digits we compute the average Euclidean distance between reconstruction and original image over the entire dataset. These reconstruction errors we portray in Figure 3.4. We can see that similarly to the discriminative model, a bottleneck dimension of  $n_b = 2$  is too small to produce good reconstructions. Therefore, the resulting reconstruction error is similar for both neural networks as well.

To address the problem of having chosen a too small bottleneck we want to increase the bottleneck to  $n_b = 3$  and consider its latent space as well as the reconstructions. Since the training progress looks similar to the 2-dimensional case, we omit its depiction here and instead kindly refer to the Python code appended to this thesis.

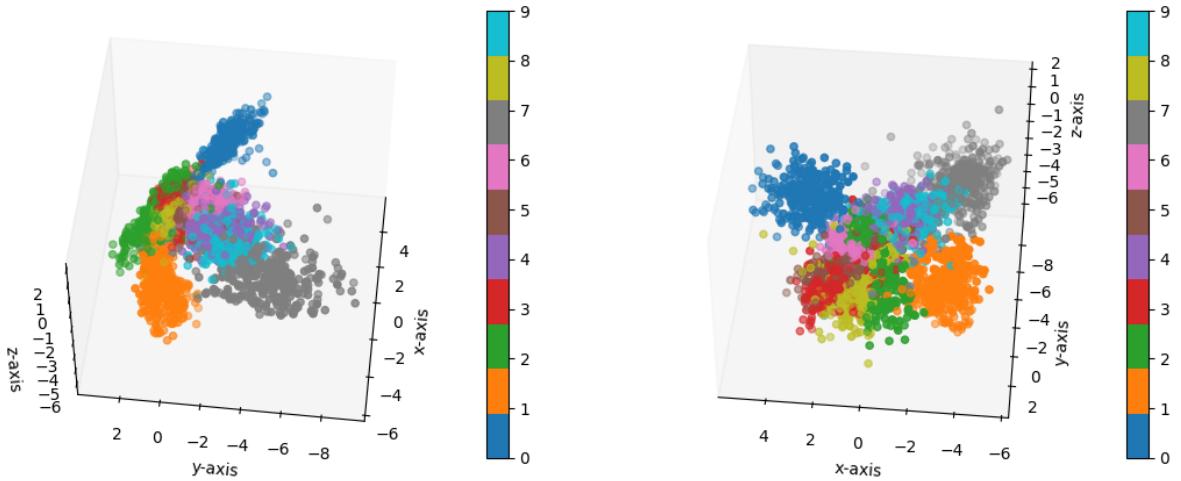
Having trained the variational autoencoder with bottleneck  $n_b = 3$ , we depict the latent space from two different perspectives in Figure 3.5. We can see that the clusters of each digit are well separated for digits that look differently, e.g. the digits 1 and 0 form clear clusters. However, digits that look similar, e.g. 4 and 9 are not separated at all and are mapped onto the same region in the latent space. This will affect the reconstructions, which we can see in Figure 3.6. In this figure we can see analogously to the 2-dimensional case, on the left-hand side 100 samples from the MNIST dataset, where we randomly chose 10 samples for each of the ten digits. On the right-hand side we can see the corresponding reconstructions produced by the neural network. As anticipated, the digits which are well separated are reconstructed quite well, e.g. 0, 1 and 7, whereas the digits that are not separated at all can hardly be distinguished, e.g. 4 and 9.

Lastly, we visualise the reconstruction errors in Figure 3.7. We can see that the errors are slightly smaller comparing to the 2-dimensional case, see Figure 3.4.

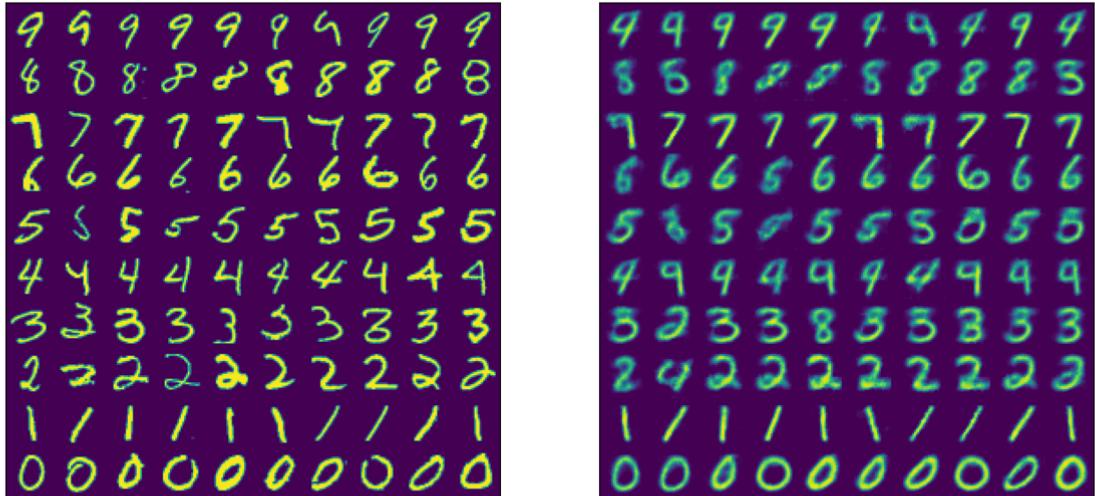
Furthermore, to be able to compare the reconstruction capability of the convolutional autoencoder, see Figure 2.22, to the variational autoencoder in a sensible way, we now want to increase the bottleneck dimension to  $n_b = 64$  and depict its reconstructions. These can be found in Figure 3.8. We realise that the reconstructions of the variational autoencoder are slightly worse than compared to the



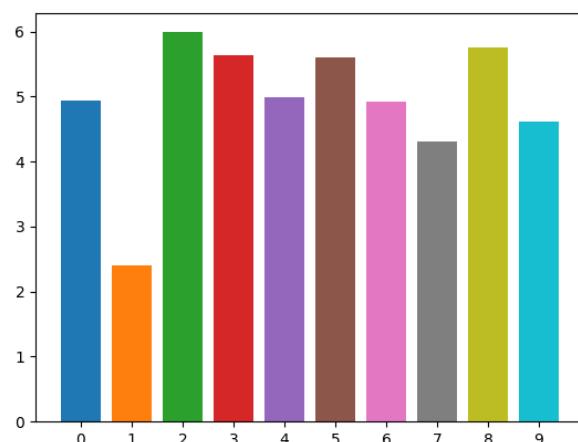
**Figure 3.4:** The figure illustrates the test errors of the variational autoencoder with bottleneck  $n_b = 2$  and KL-coefficient  $\lambda = 4 \times 10^{-5}$ , where each bar represents the averaged test errors over the entire MNIST dataset for each of the ten digits.



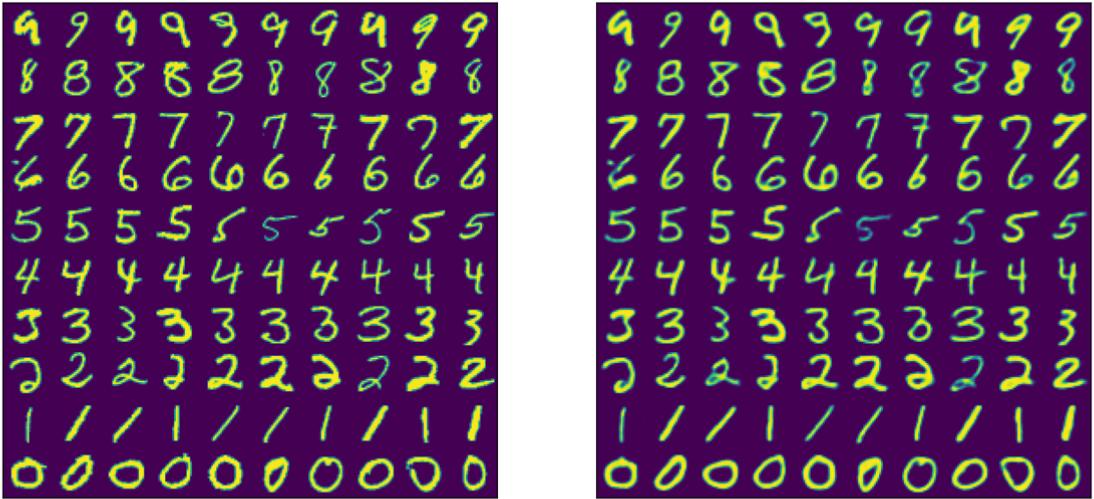
**Figure 3.5:** The figure illustrates the latent space of the variational autoencoder with bottleneck  $n_b = 3$  and KL-coefficient  $\lambda = 4 \times 10^{-5}$  from two different perspectives. Each dot is a sample from the encoded image of a digit onto a probability density. The color and the corresponding color map represent the digit that was encoded.



**Figure 3.6:** On the left-hand side, the figure illustrates 100 original digits from the MNIST dataset. On the right-hand side, the figure illustrates the same digits after feeding them through the variational autoencoder with bottleneck  $n_b = 3$  and KL-coefficient  $\lambda = 4 \times 10^{-5}$  to produce a reconstruction.



**Figure 3.7:** The figure illustrates the test errors of the variational autoencoder with bottleneck  $n_b = 3$  and KL-coefficient  $\lambda = 4 \times 10^{-5}$ , where each bar represents the averaged test errors over the entire MNIST dataset for each of the ten digits.



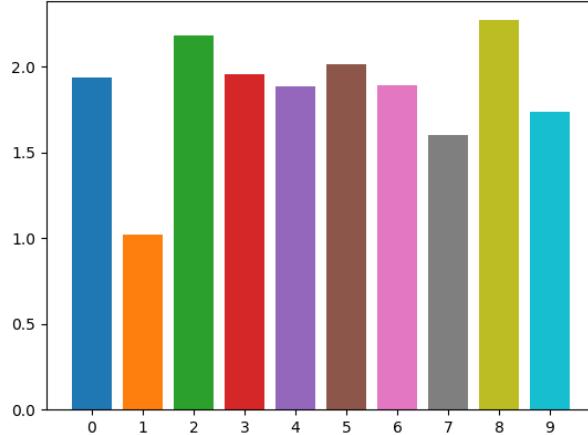
**Figure 3.8:** On the left-hand side, the figure illustrates 100 original digits from the MNIST dataset. On the right-hand side, the figure illustrates the same digits after feeding them through the variational autoencoder with bottleneck  $n_b = 64$  and KL-coefficient  $\lambda = 4 \times 10^{-5}$  to produce a reconstruction.

convolutional autoencoder.

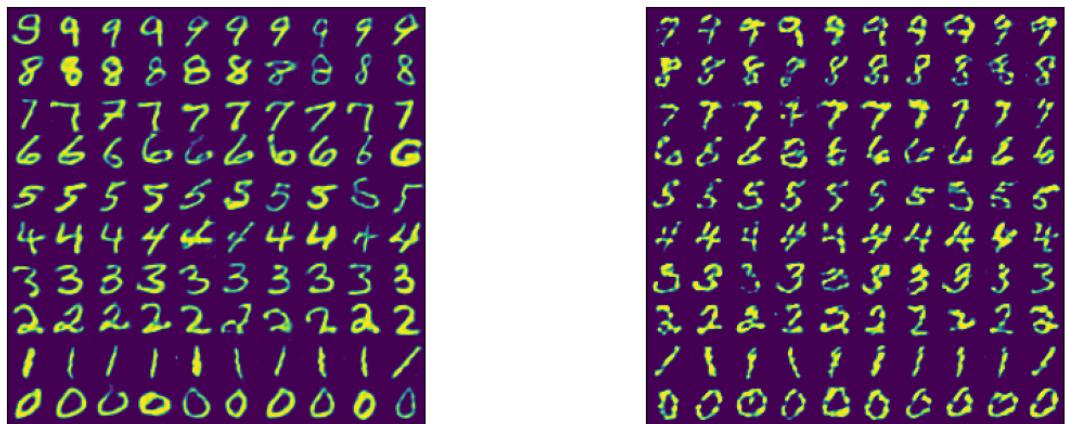
Moreover, we portray the reconstruction errors in Figure 3.9. We highlight that the reconstruction errors for the variational autoencoder are slightly higher than compared to the convolutional autoencoder, see Figure 2.23. This is due to the fact that producing single point evaluations, which these reconstructions essentially are, are tasks that discriminative models are tailored for. Therefore, the convolutional autoencoder performs slightly better. However, if we want to generate new samples, e.g. by adding some noise onto an encoded representation, the variational autoencoder performs much better. To prove this, we randomly chose one sample for each digit and then add some  $\mathcal{N}(0, 1)$ -noise onto the encoding. Then we reconstruct the images with the help of the corresponding decoding architectures of each neural network. These newly generated samples we depict in Figure 3.10, where on the left-hand side we can see the generated samples for the variational autoencoder and on the right-hand side for the convolutional autoencoder.

We can clearly see that the variational autoencoder is much more stable when applying noise onto the encoded representations and thus, generates much better samples. At this point it is worth mentioning that there indeed are some generated images that do not look as good as other ones, even for the variational autoencoder. This is due to the fact that we simply chose an arbitrary sample from the dataset to begin with. If we were to first determine the cluster center, by e.g. averaging the mean over a certain number of samples, then the generations would be much more stable. This we do in Figure 3.11. We clearly see that every single newly generated sample is a good reconstruction and thus, this experiment succeeded. We managed to generate new samples with the help of the variational autoencoder.

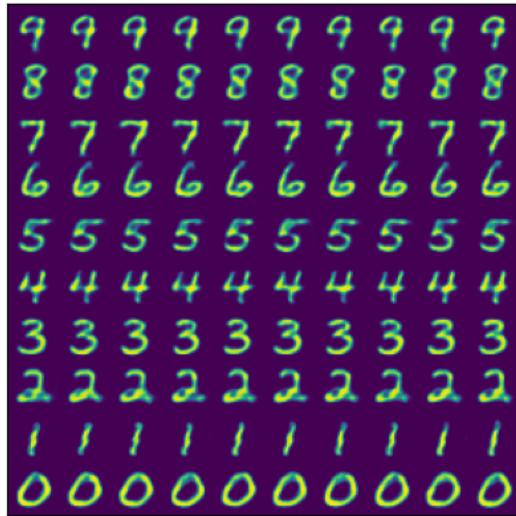
We analysed some variational autoencoders with different bottleneck dimensions. However, we always chose the KL-coefficient to be  $\lambda = 4 \times 10^{-5}$ , which is comparatively small. We now want to analyse the influence of the said coefficient on the resulting variational autoencoder. We begin by increasing the coefficient to  $\lambda = 4 \times 10^{-4}$ . The corresponding latent space is illustrated in Figure 3.12. We highlight that the clusters are much more ball shaped compared to the case with the lower KL-coefficient, see Figure 3.5. This phenomenon is a consequence of the normally distributed density,



**Figure 3.9:** The figure illustrates the test errors of the variational autoencoder with bottleneck  $n_b = 64$  and KL-coefficient  $\lambda = 4 \times 10^{-5}$ , where each bar represents the averaged test errors over the entire MNIST dataset for each of the ten digits.



**Figure 3.10:** On the left-hand side, the figure illustrates 100 generated samples by the variational autoencoder with bottleneck  $n_b = 64$ . On the right-hand side, the figure illustrates 100 generated samples by the convolutional autoencoder with bottleneck  $n_b = 64$  as well.



**Figure 3.11:** The figure illustrates 100 generated samples by the variational autoencoder with bottleneck  $n_b = 64$ . To make the generation more stable, we first averaged the encoded means and variances over all digits in the dataset and afterwards applied some noise to the averaged mean.

which the samples are encoded to. As we already described at the beginning of this section, a higher KL-coefficient leads to the fact that the autoencoder takes the KL-divergence more into account during training. This in turn enforces the encoding to be more similar to the chosen true posterior density.

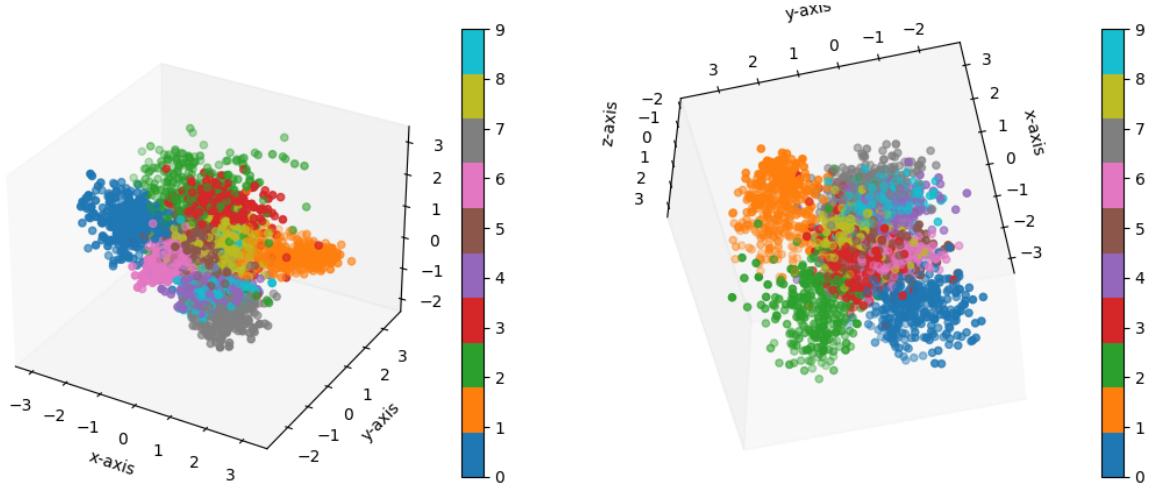
If we now take a look at the reconstructions of the variational autoencoder, see Figure 3.13, we see that they still are not good, even though they are a little smoother than compared to the  $\lambda = 4 \times 10^{-5}$  case, see Figure 3.6. Moreover, we illustrate the reconstruction errors in Figure 3.14. We highlight that the result is that the tradeoff for a smoother latent space is a higher reconstruction error.

We proceed to increase the KL-coefficient even further and consider the case  $\lambda = 4 \times 10^{-2}$ . This is a very interesting example for different reasons. The first reason is that if we take a look at the latent space of this variational autoencoder, see Figure 3.15, we realise that every digit is encoded onto a 3-dimensional normal distribution with basically the same density function. There can not be determined a single cluster. Therefore, the reconstructions, see Figure 3.16, can not be identified as digits either. The reconstruction error, see Figure 3.17, is quite high and pretty similar for all ten digits.

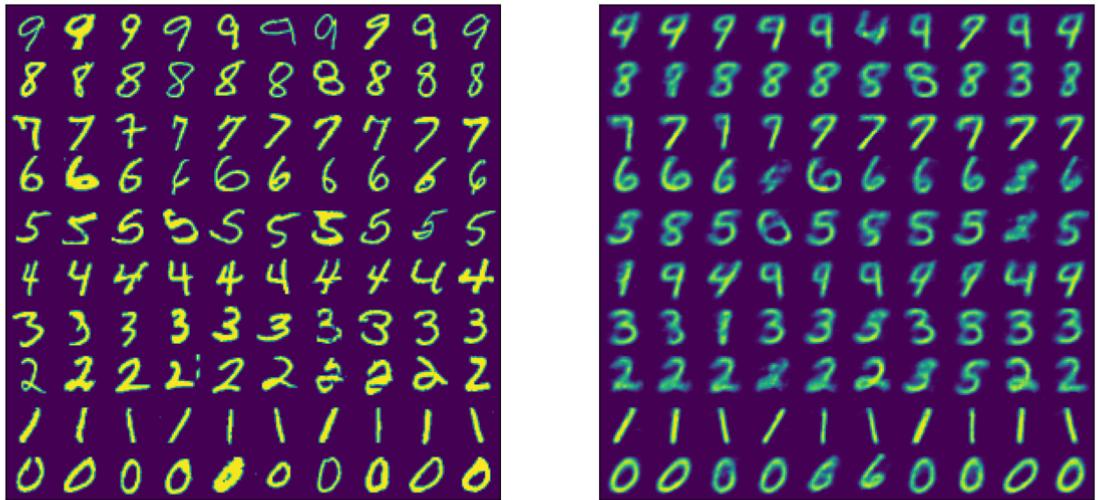
However, if we increase the bottleneck to  $n_b = 10$ , the reconstructions, see Figure 3.18 are recognizable again. This means that if the network has enough „spare dimensions“ it can find itself a mapping such that the encodings form clusters in the latent space. The reconstruction error, see Figure 3.26, is again comparatively small, which verifies the fact that the neural network encodes the samples onto distinct densities now.

If we now increase the bottleneck to  $n_b = 64$ , see Figure 3.20, we realise that the reconstructions are quite good and hence, the neural network behaves as it is supposed to. Moreover, taking a look at the reconstruction error, see Figure 3.20 supports this claim.

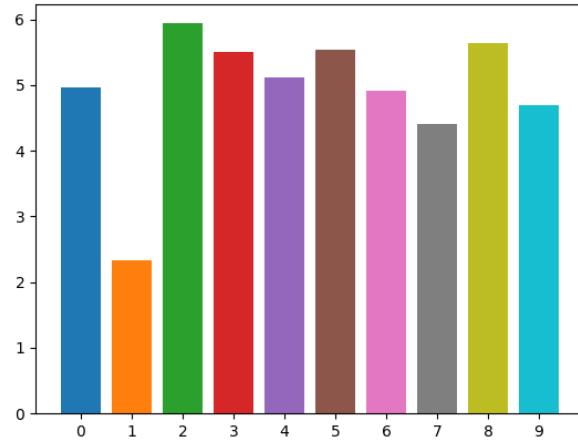
As the last experiment we want to analyse a slightly different approach than proposed in literature. As previously described, the common approach in literature is to choose the posterior density to be a standard normal distribution and allow the neural network to find a latent representation such that the encoded densities for each digit are close to a standard normal distribution but still differ enough to form distinct clusters for each of the ten digits. We now propose the following. Since we know that we have ten digits and thus, wish to have ten clusters in the latent space, we define for each of the digits a different vector which will be the mean of its chosen posterior density. In particular,



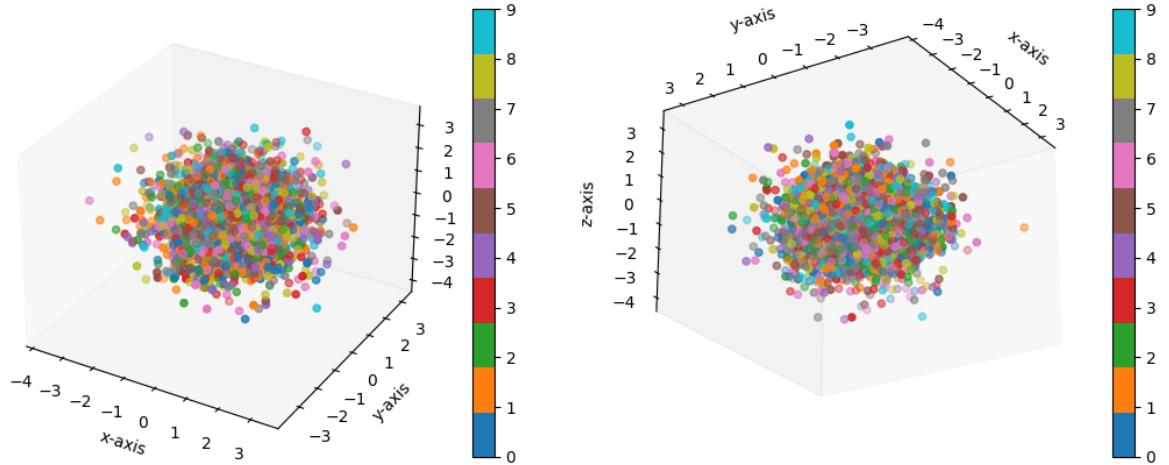
**Figure 3.12:** The figure illustrates the latent space of the variational autoencoder with bottleneck  $n_b = 3$  and KL-coefficient  $\lambda = 4 \times 10^{-4}$  from two different perspectives. Each dot is a sample from the encoded image of a digit onto a probability density. The color and the corresponding color map represent the digit that was encoded.



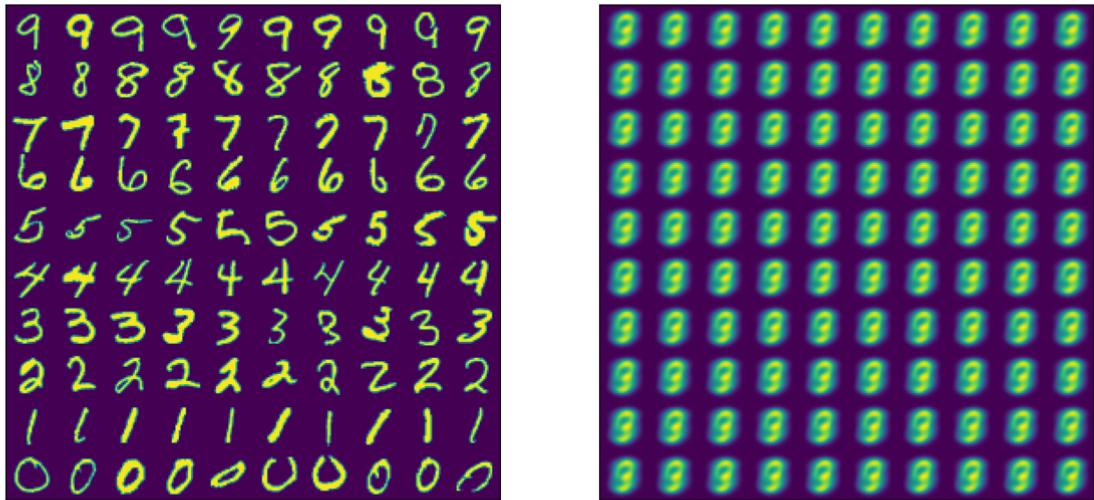
**Figure 3.13:** On the left-hand side, the figure illustrates 100 original digits from the MNIST dataset. On the right-hand side, the figure illustrates the same digits after feeding them through the variational autoencoder with bottleneck  $n_b = 3$  and KL-coefficient  $\lambda = 4 \times 10^{-4}$  to produce a reconstruction.



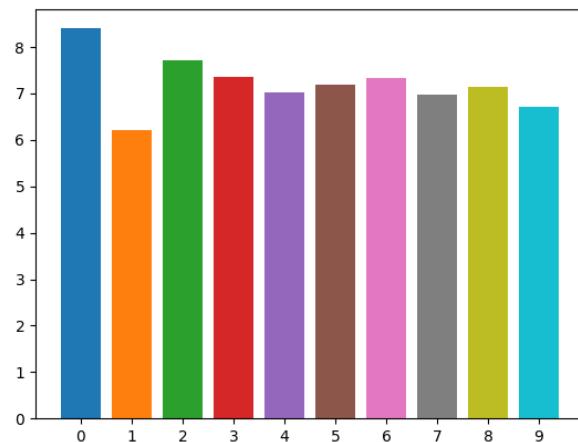
**Figure 3.14:** The figure illustrates the test errors of the variational autoencoder with bottleneck  $n_b = 3$  and KL-coefficient  $\lambda = 4 \times 10^{-4}$ , where each bar represents the averaged test errors over the entire MNIST dataset for each of the ten digits.



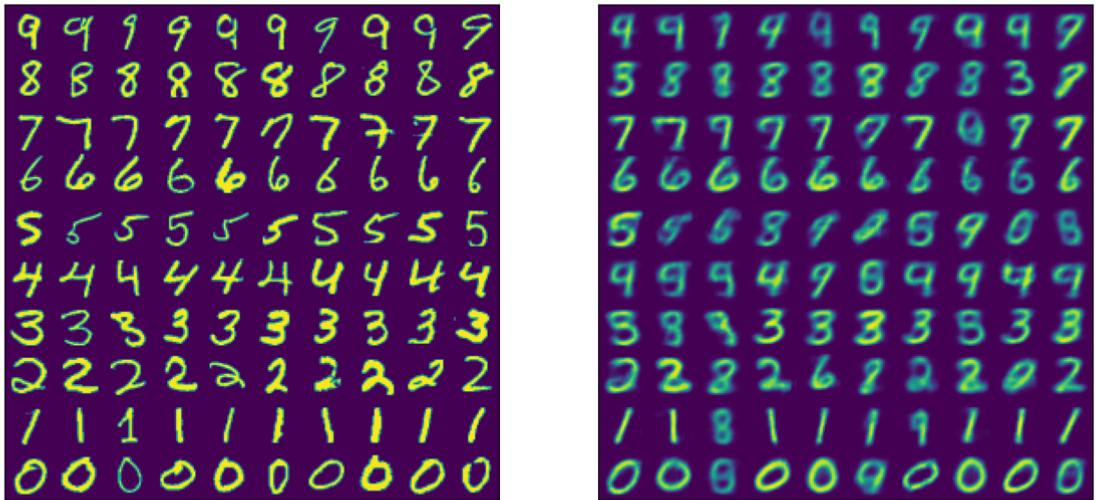
**Figure 3.15:** The figure illustrates the latent space of the variational autoencoder with bottleneck  $n_b = 3$  and KL-coefficient  $\lambda = 4 \times 10^{-2}$  from two different perspectives. Each dot is a sample from the encoded image of a digit onto a probability density. The color and the corresponding color map represent the digit that was encoded.



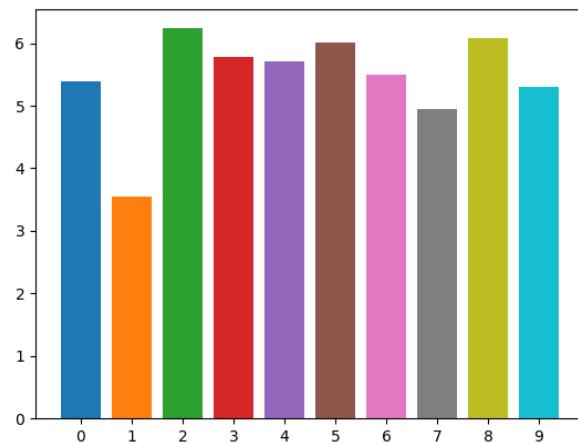
**Figure 3.16:** On the left-hand side, the figure illustrates 100 original digits from the MNIST dataset. On the right-hand side, the figure illustrates the same digits after feeding them through the variational autoencoder with bottleneck  $n_b = 3$  and KL-coefficient  $\lambda = 4 \times 10^{-2}$  to produce a reconstruction.



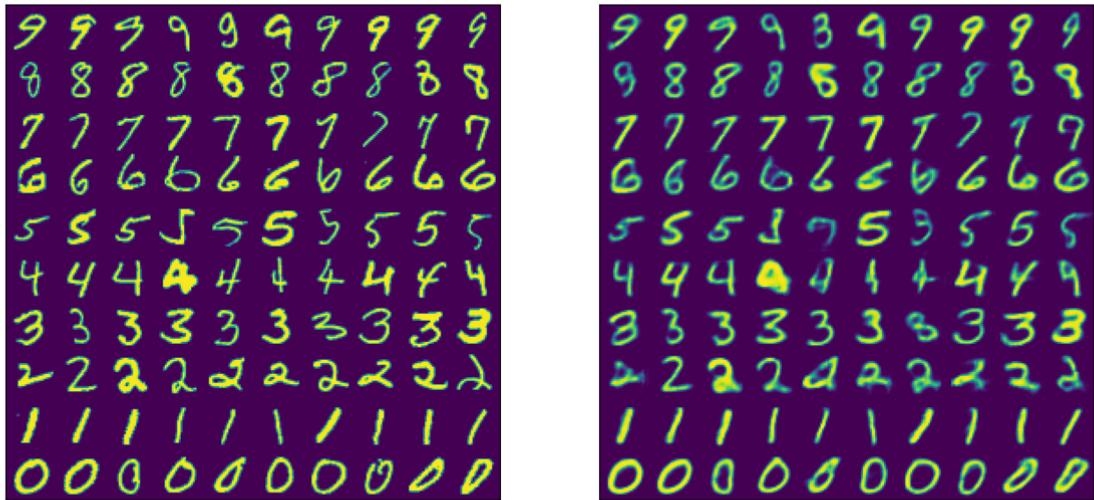
**Figure 3.17:** The figure illustrates the test errors of the variational autoencoder with bottleneck  $n_b = 3$  and KL-coefficient  $\lambda = 4 \times 10^{-2}$ , where each bar represents the averaged test errors over the entire MNIST dataset for each of the ten digits.



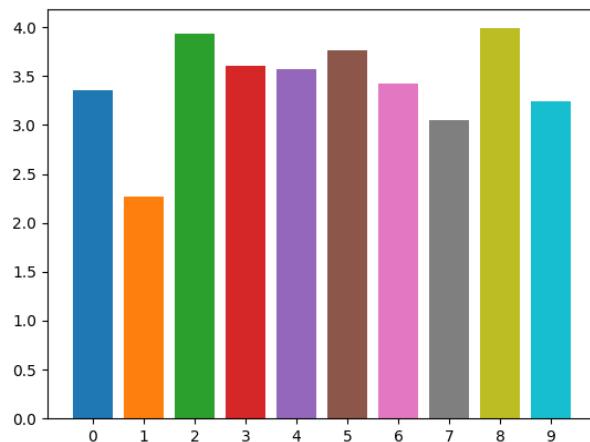
**Figure 3.18:** On the left-hand side, the figure illustrates 100 original digits from the MNIST dataset. On the right-hand side, the figure illustrates the same digits after feeding them through the variational autoencoder with bottleneck  $n_b = 10$  and KL-coefficient  $\lambda = 4 \times 10^{-2}$  to produce a reconstruction.



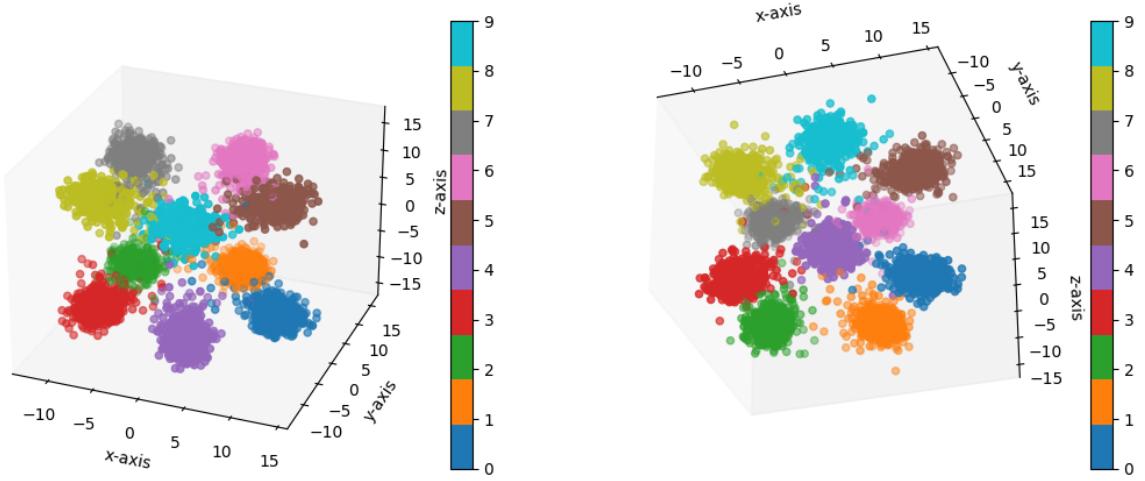
**Figure 3.19:** The figure illustrates the test errors of the variational autoencoder with bottleneck  $n_b = 10$  and KL-coefficient  $\lambda = 4 \times 10^{-2}$ , where each bar represents the averaged test errors over the entire MNIST dataset for each of the ten digits.



**Figure 3.20:** On the left-hand side, the figure illustrates 100 original digits from the MNIST dataset. On the right-hand side, the figure illustrates the same digits after feeding them through the variational autoencoder with bottleneck  $n_b = 64$  and KL-coefficient  $\lambda = 4 \times 10^{-2}$  to produce a reconstruction.



**Figure 3.21:** The figure illustrates the test errors of the variational autoencoder with bottleneck  $n_b = 64$  and KL-coefficient  $\lambda = 4 \times 10^{-2}$ , where each bar represents the averaged test errors over the entire MNIST dataset for each of the ten digits.



**Figure 3.22:** The figure illustrates the latent space of the variational autoencoder with the posterior chosen customly, where we choose the bottleneck  $n_b = 3$  and KL-coefficient  $\lambda = 4 \times 10^{-2}$ , from two different perspectives. Each dot is a sample from the encoded image of a digit onto a probability density. The color and the corresponding color map represent the digit that was encoded.

we define two circles in the  $z = -10$  plane and the  $z = 10$  plane, respectively. Then we choose 5 equidistant points on each of the two circles. These resulting 10 points we then uniquely assign to each of the 10 digits. Upon running the training algorithm we now choose the true posterior to be a normal distribution with variance  $\sigma = 1$  as in the previous case. However, for the mean  $\mu$  we choose the points we uniquely assigned to each of the 10 digits. This way we can force the neural network to spatially separate the clusters. We depict the resulting latent space for a variational autoencoder with bottleneck  $n_b = 3$  and KL-coefficient  $\lambda = 4 \times 10^{-2}$  in Figure 3.22.

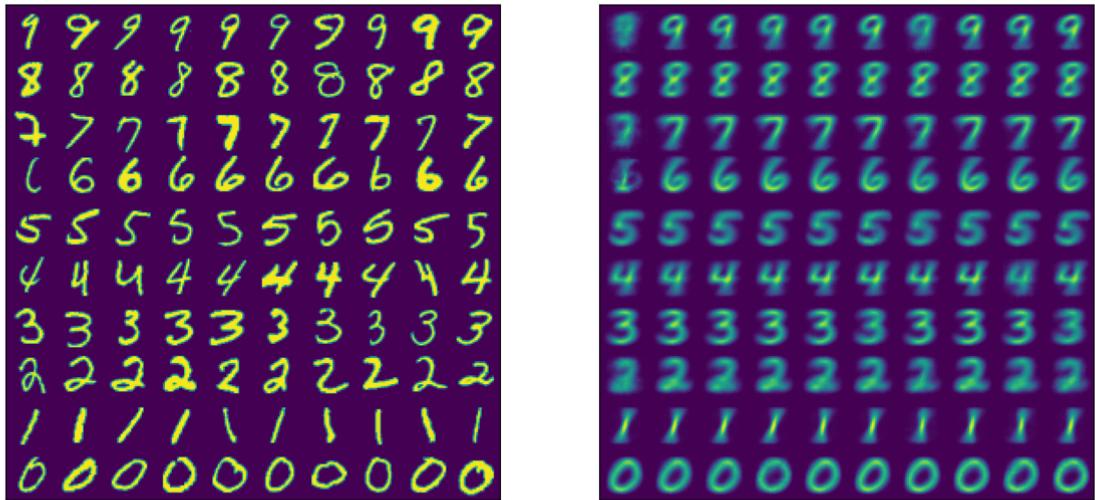
Furthermore, we visualise the reconstructions for each digit in Figure 3.23 and the corresponding reconstruction errors in Figure 3.24.

We realise that we managed to successfully train a variational autoencoder with a comparatively low KL-coefficient of  $\lambda = 4 \times 10^{-2}$ , where in contrast, the common approach failed, see Figure 3.16 or Figure 3.17. Moreover, our network even outperforms the discriminative model (with the same bottleneck), see e.g. Figure 2.14. Now, we want to take a look at the reconstruction capability for this approach with bottleneck  $n_b = 10$ , see Figure 3.25 with the corresponding reconstruction errors, see Figure 3.26.

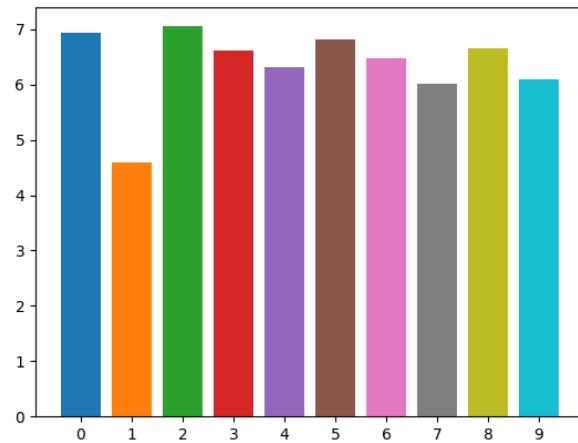
Increasing the bottleneck one last time to  $n_b = 64$  gives us the reconstructions in Figure 3.27 and reconstruction errors in Figure 3.28. We see that the reconstructions are comparable to the common approach of choosing the standard normal density as the density the KL-divergence refers to. This is clearly visible if one compared the reconstruction errors, see Figure 3.28 and 3.21.

Lastly, we want to analyse how the variational autoencoder behaves if we try to generate new samples. To do so, we add some  $\epsilon \sim \mathcal{N}(0, 1)$ -noise to the mean of the encoded probability density and then feed a sample through the decoding part of the neural network. This way we generate ten images for each of the ten digits. The generated samples are visualised in Figure 3.29.

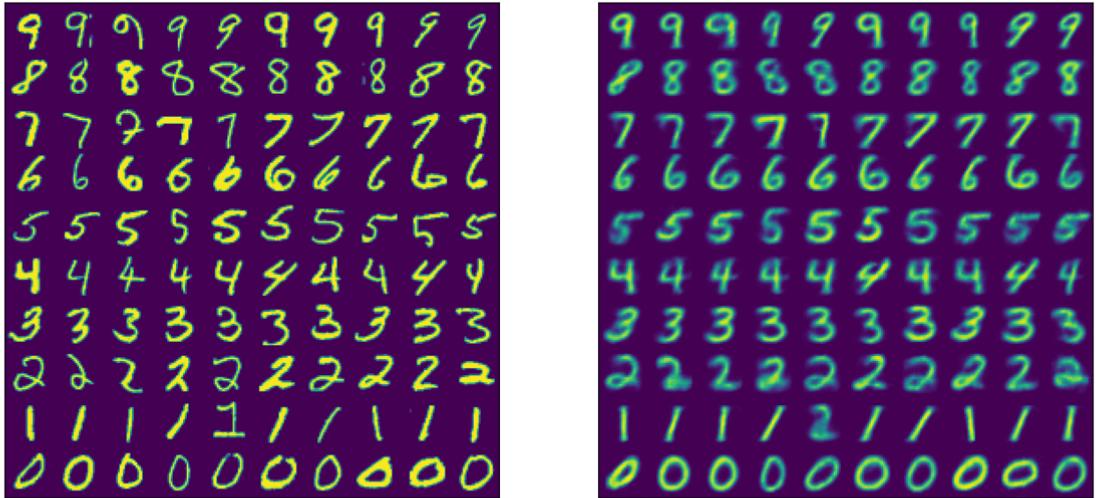
As we can clearly see, the generated samples look much better than with the common approach in literature, see Figure 3.11. Furthermore, since we chose the mean of the encoded distribution in advance, we do not have to determine it by computing the average of the encoded means, which is quite expensive since one has to consider many samples in order to compute an accurate average. Therefore, we can simply take mean chosen a priori and apply some noise to it. This is a nice byproduct we yield from our novel approach.



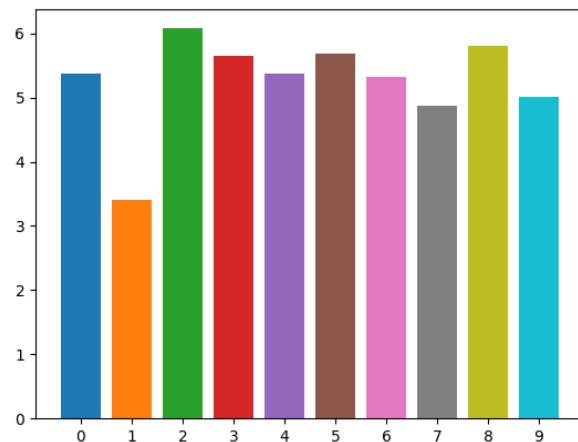
**Figure 3.23:** On the left-hand side, the figure illustrates 100 original digits from the MNIST dataset. On the right-hand side, the figure illustrates the same digits after feeding them through the variational autoencoder with the posterior chosen customly, where we choose the bottleneck  $n_b = 3$  and KL-coefficient  $\lambda = 4 \times 10^{-4}$  to produce a reconstruction.



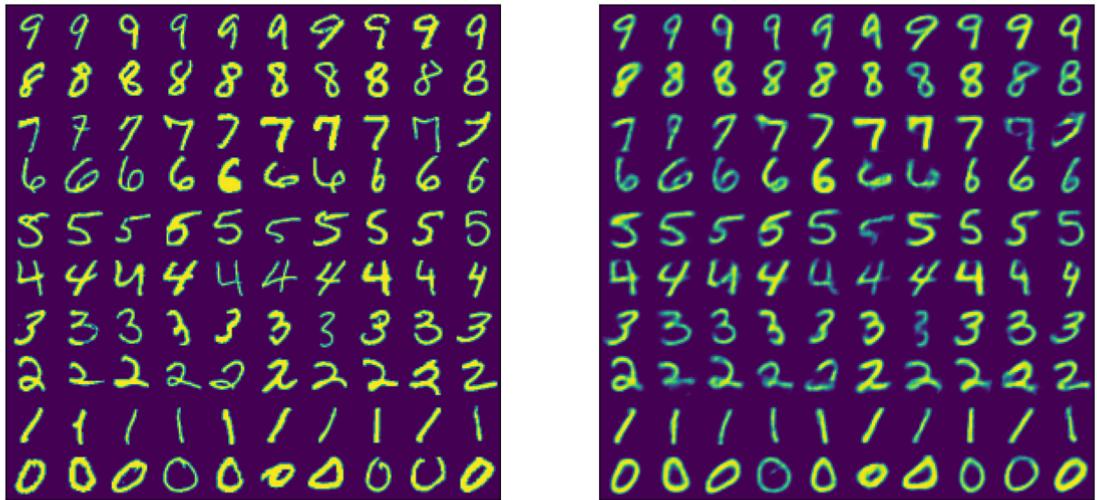
**Figure 3.24:** The figure illustrates the test errors of the variational autoencoder with the posterior chosen customly with bottleneck  $n_b = 3$  and KL-coefficient  $\lambda = 4 \times 10^{-2}$ , where each bar represents the averaged test errors over the entire MNIST dataset for each of the ten digits.



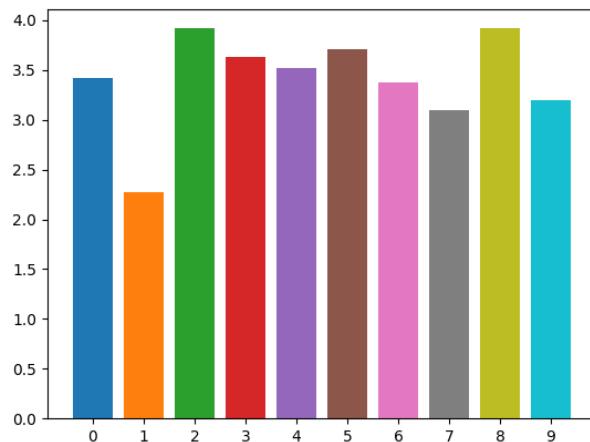
**Figure 3.25:** On the left-hand side, the figure illustrates 100 original digits from the MNIST dataset. On the right-hand side, the figure illustrates the same digits after feeding them through the variational autoencoder with the posterior chosen customly with bottleneck  $n_b = 10$  and KL-coefficient  $\lambda = 4 \times 10^{-2}$  to produce a reconstruction.



**Figure 3.26:** The figure illustrates the test errors of the variational autoencoder with the posterior chosen customly with bottleneck  $n_b = 10$  and KL-coefficient  $\lambda = 4 \times 10^{-2}$ , where each bar represents the averaged test errors over the entire MNIST dataset for each of the ten digits.



**Figure 3.27:** On the left-hand side, the figure illustrates 100 original digits from the MNIST dataset. On the right-hand side, the figure illustrates the same digits after feeding them through the variational autoencoder with the posterior chosen customly with bottleneck  $n_b = 64$  and KL-coefficient  $\lambda = 4 \times 10^{-2}$  to produce a reconstruction.



**Figure 3.28:** The figure illustrates the test errors of the variational autoencoder with the posterior chosen customly with bottleneck  $n_b = 64$  and KL-coefficient  $\lambda = 4 \times 10^{-2}$ , where each bar represents the averaged test errors over the entire MNIST dataset for each of the ten digits.



**Figure 3.29:** The figure illustrates 100 generated samples by the variational autoencoder with bottleneck  $n_b = 64$  and customly chosen posterior. Where we applied some noise to the mean of the encoded densities.

# Bibliography

- [1] C. M. BISHOP AND N. M. NASRABADI, *Pattern Recognition and Machine Learning*, Springer, 2006.
- [2] 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.
- [3] D. FOSTER, *Generative Deep Learning*, O'Reilly Media, Inc., 2022.
- [4] I. GOODFELLOW, Y. BENGIO, AND A. COURVILLE, *Deep Learning*, MIT Press, 2016.
- [5] J. JAHN, *Vector Optimization*, Springer, 2009.
- [6] L. V. KANTOROVICH AND G. P. AKILOV, *Functional Analysis*, Elsevier, 2016.
- [7] D. P. KINGMA AND J. BA, *Adam: A method for stochastic optimization*, International Conference on Learning Representations, (2014).
- [8] D. P. KINGMA AND M. WELLING, *Auto-encoding variational Bayes*, International Conference on Learning Representations, (2013).
- [9] A. KLENKE, *Probability Theory: A Comprehensive Course*, Springer, 2013.
- [10] 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.
- [11] C. LEMARÉCHAL, *Cauchy and the gradient method*, Doc. Math. Extra, (2012), p. 251–254.
- [12] J. R. MAGNUS AND H. NEUDECKER, *Matrix Differential Calculus with Applications in Statistics and Econometrics*, John Wiley & Sons, 2019.
- [13] H. B. McMAHAN AND M. STREETER, *Adaptive bound optimization for online convex optimization*, Conference on Learning Theory, (2010).
- [14] D. MEINTRUP AND S. SCHÄFFLER, *Stochastik: Theorie und Anwendungen*, Springer, 2006.
- [15] N. MÜCKE AND I. STEINWART, *Empirical risk minimization in the interpolating regime with application to neural network learning*, arXiv:1905.10686, (2019).
- [16] J. W. PAISLEY, D. M. BLEI, AND M. I. JORDAN, *Variational Bayesian inference with stochastic search*, Proceedings of the 29th International Conference on Machine Learning, (2012), p. 1367–1374.
- [17] A. PAPOULIS AND S. U. PILLAI, *Probability, Random Variables and Stochastic Processes*, McGraw Hill, 2002.

- [18] S. J. REDDI, S. KALE, AND S. KUMAR, *On the convergence of adam and beyond*, International Conference on Learning Representations, (2019).
- [19] D. SAAD, *On-line Learning in Neural Networks*, Cambridge Univ. Press, 2009.
- [20] G. SIMMONS, *Calculus with Analytic Geometry*, McGraw Hill, 1995.
- [21] S. SRA, S. NOWOZIN, AND S. J. WRIGHT, *Optimization for Machine Learning*, MIT Press, 2012.
- [22] I. STEINWART AND A. CHRISTMANN, *Support Vector Machines*, Springer, 2008.
- [23] G. TURINICI, *The convergence of the stochastic gradient descent (sgd): a self-contained proof*, arXiv:2103.14350, (2021).

## Deutsche Zusammenfassung

Diese Thesis mit dem Titel „Variational Autoencoders: Theory and Applications“ behandelt die Kombination aus Bayesianischer Statistik und der Theorie von Autoencodern, einer bestimmten Architektur neuronaler Netze. Die Theorie der neuronalen Netze ist in der Literatur auch bekannt als „Deep Learning“. Das Ziel dieser Thesis ist ein grundlegendes mathematisches Verständnis der Variational Autoencoder zu erlangen. Um dies zu erzielen ist die Thesis in drei Teile aufgeteilt. Im ersten Teil werden essentielle Grundlagen aus den Gebieten der Maß- und Wahrscheinlichkeitstheorie, der Statistik und Statistischen Lerntheorie, der neuronalen Netze und deren Optimierung behandelt. Im zweiten Teil werden Autoencoder eingeführt und an praktischen Beispielen ausgeführt. Im dritten und letzten Teil werden schließlich VAEs eingeführt und an unterschiedlichen praktischen Beispielen analysiert und veranschaulicht.

Im ersten Kapitel werden dem Lesenden die wichtigsten Grundlagen ausführlich erläutert. Dabei zitieren wir weitestgehend aus renomierter Literatur unterschiedliche Resultate. Zunächst führen wir aus der Maß- und Wahrscheinlichkeitstheorie wichtige Größen wie unter anderem Wahrscheinlichkeitsmaße und -dichten, sowie Zufallsvariablen ein. Im Anschluss führen wir die Statistische Lerntheorie ein. Wir erläutern, wie man aus vorhandenen Daten sinnvolle Schlüsse ableiten kann. Genauer gesagt erläutern wir, wie wir eine Funktion finden können, die unsere Daten möglichst gut beschreibt. Unter anderem erläutern wir dabei Größen wie beispielsweise Verlustfunktionen und (empirisches) Risiko. Als nächstes führen wir neuronale Netze ein und verbinden diese mit der Statistischen Lerntheorie. Dabei erläutern wir ausführlich, wie unterschiedliche Trainingsalgorithmen funktionieren. Wir betrachten im Detail den „Gradient Descent“ Algorithmus und formulieren dazu wichtige Resultate wie beispielsweise den „Backpropagation“ Algorithmus - eine Methode um Gradienten von hochgradig verketteten Funktionen zu berechnen. Diese Methode ist allgegenwärtig in der Praxis. Mithilfe des Backpropagation Algorithmus stellen wir fest, wie rechnerisch aufwändig das Training neuronaler Netze sein kann und führen deshalb einen weiteren Optimierungsalgorithmus ein. Dieser neue Algorithmus heißt „Stochastic Gradient Descent“. Er funktioniert ähnlich wie der Gradient Descent Algorithmus, ist allerdings deutlich effizienter, da er im Gegensatz zum Gradient Descent Algorithmus nur einen Bruchteil der Daten in jeder Iteration berücksichtigt. Somit benötigt dieser Algorithmus auch nur einen Bruchteil des Rechenaufwands. Im Zuge der Diskussion über Effizienz führen wir einen weiteren Optimierungsalgorithmus ein, der meist als aktuell optimal angesehen wird. Wir reden dabei über den „Adaptive Moment Estimation“ (Adam) Algorithmus. Allerdings wurde 2019 bewiesen, dass dieser Optimierungsalgorithmus in manchen Szenarien doch nicht optimal ist. Stattdessen wurde ein neuer Optimierungsalgorithmus angeregt, welcher „AMSGrad“ bezeichnet wird. Diesen führen wir ebenfalls ein und analysieren diesen ausführlich. In den darauffolgenden Kapiteln stellen wir ebenfalls anhand von Anwendungsbeispielen fest, dass der AMSGard Algorithmus bessere Ergebnisse liefert als der Adam Algorithmus. Als letzte Überlegung im ersten Teil der Thesis führen wir ein, wie neuronale Netze auf Bildern operieren können. Vorangehend führen wir neuronale Netze ein, welche nur auf Vektoren operieren. Durch geschickte Wahl der Operatoren in einem neuronalen Netz ermöglicht es uns, die Struktur von Bildern aufrecht zu erhalten. Dies ist von großem Vorteil, da wir in den darauffolgenden Anwendungen neuronalen Netze auf Bildern operieren lassen wollen.

Im zweiten Kapitel der Thesis betrachten wir Autoencoder. Dabei handelt es sich um eine spezielle Architektur neuronaler Netze, die Daten zunächst in ihrer Dimension reduzieren und im Anschluss wieder auf die ursprüngliche Dimension zurückführen. Diese Reduktion der Dimensionen wird im Englischen von Softwareentwicklern meistens als „Feature Extraction“ und von Theoretikern als „Dimensionality Reduction“ bezeichnet. Sie bietet einige interessante Vorteile und Anwendungsmöglichkeiten. Beispielsweise ist es dadurch möglich Bilder erst zu verkleinern, also deren Auflösung zu reduzieren, und im Anschluss abzuspeichern. Dabei ist eine Verkleinerung von mehreren Größenordnungen möglich. Dadurch lassen sich dementsprechend auf demselben Speicherplatz um Größenordnungen mehr Bilder abspeichern. eine weitere interessante Anwendung ist die Tatsache, dass Machine Lear-

ning Modelle genauso auf den „verkleinerten“ Daten, wie auch auf den Originalen, arbeiten können. Sind die Daten nun aber um Größenordnungen kleiner, so ermöglicht es dem Machine Learning Modell um Größenordnungen mehr Daten in nahezu derselben Zeit zu verarbeiten. Die meisten Modelle in der Industrie derzeit sind so groß, dass sie Tage und manchmal sogar Wochen brauchen, um trainiert zu werden. Beispielsweise wurde Meta’s Segment Anything Model (SAM) „3-5 Tage auf 256 A100 GPUs“ trainiert, siehe <https://segment-anything.com>. Dies verdeutlicht die Relevanz solcher Optimierungen durch Dimensionsreduktion. Die letzte Anwendungsmöglichkeit, die wir an dieser Stelle benennen wollen, ist die Möglichkeit von semantischer Suche in verschiedenen Medien. Damit ist folgendes gemeint. Beispielsweise trainiert man einen Autoencoder, der Bilder als niedrig-dimensionale Vektoren abspeichern, und wieder reproduzieren kann. Als nächstes trainiert man einen weiteren Autoencoder, der in der Lage ist dasselbe mit Audio-Dateien zu verwirklichen. Nun ist es möglich diese beiden Autoencoder gemeinsam zu trainieren, sodass die niedrig-dimensionalen Vektoren des einen autoencoders mit den Vektoren des zweiten Autoencoders übereinstimmen. Dadurch ermöglicht es uns, dem neuronalen Netz beispielsweise ein Bild von einem Tier bereitzustellen und im Anschluss eine Audio-Datei zu erhalten, welche Geräusche dieses Tier von sich gibt. Beispielsweise gibt man dem neuronalen Netz ein Bild eines Löwen und erhält als Ergebnis das Brüllen eines Löwen als Audio-Datei.

Konkret führen wir im zweiten Kapitel einen wichtigen Ansatz in Anwendungen neuronaler Netze ein. Wir zeigen, dass man neuronale Netze problemlos miteinander verknüpfen kann, sofern die Eingabe- und Ausgabedimension der beiden neuronalen Netze übereinstimmen. Dies lässt sich auf beliebig viele neuronale Netze verallgemeinern. In der Praxis wird dies als „modularer Ansatz“ beschrieben. Man entwickelt einzelne Module und fügt sie im Nachhinein zusammen. Als nächstes erläutern wir, inwiefern sich das Training von Autoencodern zum Training allgemeiner neuronaler Netze unterscheidet. Da man bei Autoencodern die Eingabe mit der Ausgabe vergleicht, spricht man hierbei von unüberwachtem Lernen. Im Englischen ist dies als „unsupervised learning“ bekannt. Zuletzt betrachten wir einige Autoencoder, die wir selber implementiert und trainiert haben. Wir betrachten zu Beginn lineare Autoencoder. Das heißt, dass die Operatoren innerhalb des neuronalen Netzes alle linear und dementsprechen Matrixmultiplikationen sind. Wir evaluieren die Autoencoder auf unterschiedlichste Art und Weise. Zum Beispiel visualisieren wir die niedrig-dimensionalen Darstellungen der Daten, die Rekonstruktionsfehler, den die Autoencoder machen und zeigen zudem, wie die rekonstruierten Bilder aussehen. Des Weiteren betrachten wir den Trainingsverlauf der neuronalen Netze.

Im dritten und damit letzten Kapitel der Thesis betrachten wir Variational Autoencoder. Wir beginnen damit, dass wir den grundlegenden Unterschied zu Autoencodern erläutern. Autoencoder bilden dabei einen Datenpunkt auf einen niedrig-dimensionalen Vektor ab. Im Gegensatz dazu bilden Variational Autoencoder einen Datenpunkt auf eine Wahrscheinlichkeitsdichte, oder genauer gesagt auf eine Wahrscheinlichkeitsverteilung, ab. Dies wird dadurch erreicht, dass die Daten erst auf einen niedrig-dimensionalen Vektor abgebildet werden und dieser im Anschluss als Parameter für eine Wahrscheinlichkeitsverteilung dient. In der Industrie ist dabei üblich Normalverteilungen zu betrachten, da diese Wahl einige Berechnungen im Laufe des Trainings deutlich vereinfacht (nicht für jeden Wahrscheinlichkeitsdichte lassen sich gewisse Integrale explizit berechnen).

Im Zuge dieses Kapitels erläutern wir zunächst wie Wahrscheinlichkeitsverteilungen und Wahrscheinlichkeitsdichten zusammenhängen, was uns im Folgenden erlaubt ausschließlich Wahrscheinlichkeitsdichten zu betrachten. Im Anschluss betrachten wir das Grundlegende Resultat der Bayesianischen Statistik, die sogenannte „Bayesianische Formel“. Diese ist grundlegend für die Idee der Variational Autoencoder. Des Weiteren führen essentielle Begriffe aus der Bayesianischen Statistik, beziehungsweise Bayesianischen Inferenz ein. Konkret lauten diese Begriffe „Prior“, „Evidence“, „Likelihood“ und „Posterior“. Mit diesen Begriffen motivieren wir den Bayesianischen Zugang zur Lerntheorie und formulieren einen allgemeinen Lernalgorithmus. Als nächstes beziehen wir diesen allgemeinen Lernalgorithmus auf unseren speziellen Fall der Variational Autoencoder. Um dies zu tun, führen wir die Kullback-Leibler Divergenz ein, welche ein relatives Maß ist, welches den Unterschied zwischen Wahrscheinlichkeitsdichten quantifiziert. Demnach ermöglicht dieses Maß uns Wahrschein-

lichkeitsdichten miteinander zu vergleichen und dadurch das Training von Variational Autoencoder zu formulieren. Wir beschreiben welche Probleme der Bayesianische Ansatz mit sich bringt und bieten gängige Lösungsansätze. Zusammengefasst lässt sich der Posterior nicht allgemein berechnen und muss deshalb approximiert werden. Diese Approximation lässt sich durch unterschiedliche Art und Weise ermöglichen. Wir betrachten den allgemeinen Ansatz der Literatur und schlagen einen eigenen Ansatz vor, welchen wir mit praktischen Beispielen analysieren. Als nächstes ermitteln wir verschiedene relevante Resultate der Variational Autoencoder, wie beispielsweise die „Evidence Lower Bound“ (ELBO). Dies ist, wie der Name vermuten lässt, eine untere Schranke der Evidence, welche eine Große Rolle im Training der Variational Autoencoder spielt. Sie beschreibt sinngemäß den Grenzwert der Kullback-Leibler Divergenz, falls das Training des Variational Autoencoder erfolgreich ist. Abschließend formulieren wir einen konkreten Ansatz eines Variational Autoencoders, wobei wir konkrete Wahlen für den approximierten Posterior und den eigentlichen Posterior treffen. Dadurch ermöglicht es uns eine explizite Darstellung des Trainingsalgorithmus zu formulieren.

Abschließend im letzten Kapitel implementieren und trainieren wir einige unterschiedliche Variational Autoencoder. Dabei erzeugen wir unterschiedliche Resultate. Wir implementieren einige Variational Autoencoder und analysieren diese. Im Zuge der Analyse zeigen wir deren Rekonstruktionsfähigkeit und die damit verbundenen Rekonstruktionsfehler. Zudem vergleichen wir diese mit Architekturen des vorangegangenen Kapitels, der herkömmlichen Autoencoder. Außerdem verdeutlichen wir durch Visualisierungen, wie die erzeugten Wahrscheinlichkeitsdichten unterschiedlicher Variational Autoencoder aussehen. Des Weiteren betrachten wir den Einfluss einer Stellschraube, die wir Kullback-Leibler Koeffizient bezeichnen. Diese Stellschraube kontrolliert das Verhältnis des Einflusses von Kullback-Leibler Divergenz und Verlustfunktion auf das Training des Variational Autoencoder. Dabei betrachten wir die Verlustfunktion als Rekonstruktions-Verlustfunktion und die Kullback-Leibler Divergenz als Regularisierungs-Verlustfunktion. Je höher wir diesen Koeffizient wählen, desto stärker werden die Datenpunkte auf eine Wahrscheinlichkeitsdichte „gedrückt“. Das heißt, desto weniger Flexibilität wird dem neuronalen Netz gelassen, selbst eine geeignete niedrig-dimensionale Darstellung zu finden. Zuletzt hinterfragen wir den in der Literatur üblichen Ansatz des Trainings eines Variational Autoencoder. Dabei wird als approximativer Posterior üblicherweise eine standard Normalverteilung gewählt. Wir hingegen treffen eine andere Wahl. Wir ermitteln dadurch ein interessantes Resultat, welches belegt, dass der allgemeine Ansatz deutlich verbessert werden kann. Durch unterschiedlichste Visualisierungen zeigen wir, dass unsere individuelle Wahl des approximativen Posteriors das Training eines Variational Autoencoder konvergieren lässt, wohingegen der allgemeine Ansatz divergiert.