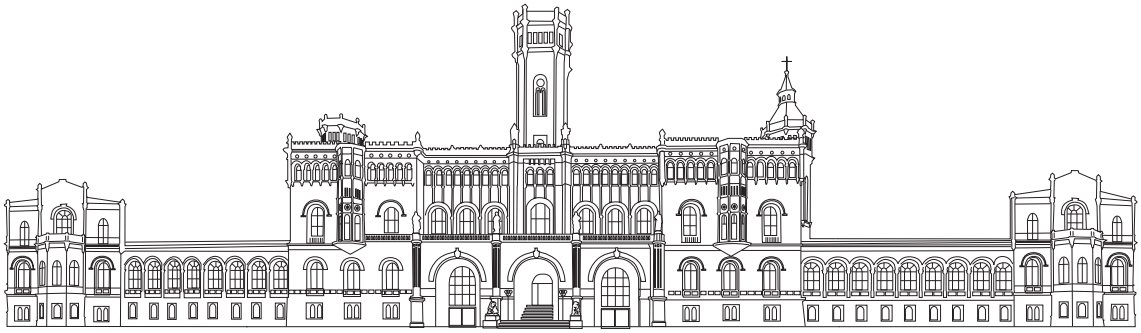


Gottfried Wilhelm Leibniz University Hanover
Institute for Theoretical Physics

Classical Simulation of Quantum Feed-forward Neural Networks

In partial fulfillment of the requirement
for the degree of the Bachelor of Science

submitted by
Marvin Schwiering
March 09, 2020

Examiner : Prof. Dr. Tobias J. Osborne
Supervisor : M.Sc. Kerstin Beer

Abstract

The recent emergence of machine learning with neural networks proved to have a stunning impact on technology by making use of big data to efficiently develop complex, new algorithms. As current advancements announce the impending rise of quantum computational technology, it is of great interest to realize a quantum adaptation of these procedures. In this thesis we review, develop and simulate a state-of-the-art proposal for quantum feed-forward neural networks, discussing their functionality, usage and assessing their capabilities.

Contents

1 Introduction

The conception of *thinking* machinery created by humans indeed long predates even the realisation of programmable computers. However now, with the extraordinary progress in sciences in the past century, it is not only of interest to science-fiction authors, but also to actual research. And while that search for artificial intelligence is predestined to prolong well into the future, the recent emergence of affiliated fields like machine learning proved to have a stunning impact on technology and automation. So called neural networks – computer models based on the human brain – can be *trained* to correctly perform intricate tasks by being fed with exemplifying instances, therefore providing an completely new way of developing advanced, complex machines and algorithms. In the age of big data, these neural networks have proven to be powerful tools, nowadays already commonly used in a variety of fields ranging from image and speech recognition to scientific research and medicine. [GBC16]

Quantum computation and quantum information is the study of algorithms and information processing tasks that can be accomplished using quantum mechanical systems. The usage of such systems in computer science offers fundamentally new opportunities and solutions beyond what is achievable with classical means – both in regard to the speed-up of principally attainable tasks, e.g. with Grover's algorithm, and realising classically impossible tasks, such as *safe* communication with quantum cryptography [NC10]. And while, generally speaking, one can consider the field of quantum computation to be still in its infancy, current advancements – such as Google's very recent demonstration of *quantum supremacy* [Aru+19] – already announce the impending rise of its technology.

It is therefore of great interest to realize a quantum adaptation of classical machine learning practices. In this thesis we inspect, develop and simulate a state-of-the-art proposal for quantum feed-forward neural networks (first introduced in [Bee+20]) discussing their architecture, functionality and usage and assessing their capabilities.

As always, a thorough understanding of basic concepts allows for a quick comprehension of continuative subjects, which is why we will review classical machine learning and neural networks in detail in Chapter 2. In the following Chapter 3 we will recapitulate basic principles of quantum mechanics and information to then introduce the quantum version of a feed-forward neural network in Chapter 4. We conclude by presenting crucial parts of the developed simulation code, as well as some findings, in Chapter 5.

2 Classical Machine Learning

2.1 What is Machine Learning?

The field of artificial intelligence (AI) deals with the creation of machines that imitate cognitive behaviour displayed by humans and other animals, with the goal of eventually artificially creating a mechanism able to independently learn and solve problems. Machine learning is one of its subfields, appointed with studying computer models that are able to perform certain tasks, without being given the specific instructions on how to do so, but instead by extorting or analysing patterns existing in given task-related data.

As research and innovation in machine learning and affiliated fields is thriving, and expected to be of relevance well into the future, it is neither achievable nor necessariliy desireable to completely classify and differentiate all of its subfields, as those definitions are likely to loose their accuracy as future innovations assumably will reform our understanding of the subject. Nonetheless, we do want to give a general overview of currently predominant efforts: Fairly broadly speaking, machine learning can be divided into three different types: Supervised learning, unsupervised learning and reinforced learning. While supervised learning deals with learning from labeled data, e.g. classification or regression tasks, unsupervised learning is concerned with finding patterns in unlabeled data. Reinforced learning is about learning a task by interaction with an environment. One can award points every time an algorithm makes a meaningful step towards the completion of a task and – in doing so – favor changes that achieve better performance. [Meh+19]

Neural networks are a potent tool in artificial intelligence and specifically supervised machine learning. They are computer models whose structure is inspired by the human brain, used in these fields because "the brain provides a proof by example that intelligent behavior is possible, and a conceptually straightforward path to building intelligence is to reverse engineeer the computational principles behind the brain and duplicate its functionality" [GBC16, p. 13].

Deep learning is a subfield of neural networks with slightly varying definitions. However a common one, that will be sufficient for us, would be the task of dealing with neural networks with multiple hidden layers [Nie15]. (We will reiterate this definition after having actually explained what a layer is.)

In this chapter, we discuss the classical feed-forward neural network[1] by first off

[1]Note that these (deep) feed-forward neural networks with fully-connected layers constitute only one class of neural networks. However, as they will remain the only type discussed in this thesis, we will not continuously burden ourselves with such an extensive nomenclature and

intruducing its architecture in an intuitively accessible way (Section 2.2), followed by a presentation of the prevalent training algorithms (Section 2.3). In doing so, we repeatedly follow [Nie15]. We conclude by giving a more mathematical introduction to supervised machine learning in Section 2.4, which will round off our grasp on the subject.

2.2 Classical Feed-Forward Neural Networks: Structure

Artificial neurons

The human brain is composed of cells called *neurons*. Billions of them, with trillions of connections between them, give humans the ability to do everything from seemingly mundane tasks, such as enjoying a walk in a park, to complex assignments, such as writing this thesis [Her09]. If one is to replicate the functionality of a human brain, understanding its smallest building block seems like a natural starting point.

Speaking in the broadest possible terms, a biological neuron, also called perceptron, is perceptive to signals incoming from connected neurons. When a certain threshold is met, it itself "fires", sending a signal to its connected neurons, thus again impacting their behavior. Human insights, decisions and actions stem from an enourmous chain of neural behavior. To illustrate how a neuron may model decision-making, consider the following example.

Say you want to decide if you want to attend a certain lecture next semester and you want to base your decision on three factors that are most important to you: Wheather ("1") or not ("0") you like the lecturer's style of presentation (input_1), you are personally interested in the subject (input_2) and you think the contents are overall useful to your studies (input_3). If you want two of those criteria to be met, your decision would look as follows:

$$\text{decision} = \begin{cases} \text{Yes}, & \text{if} \quad \text{input}_1 + \text{input}_2 + \text{input}_3 \geq \text{threshold} = 2 \\ \text{No}, & \text{else} \end{cases}$$

What if the factors aren't equally important to your decisions? As you emphazise the importance of passion to your doings, you factor the corresponding input by 3, while only weighing the others by 2, wanting an overall threshold of 3 to be met:

$$\text{output} = \begin{cases} 1, & \text{if} \quad 2 \cdot \text{input}_1 + 3 \cdot \text{input}_2 + 2 \cdot \text{input}_3 \geq \text{threshold} = 3 \\ 0, & \text{else} \end{cases}$$

Denoting the inputs as x_k, the factors as w_k, the threshold as t and using the

often merely refer to them as neural networks with layers.

Heaveside step function

$$\Theta : \mathbb{R} \to \{0, 1\},\ \Theta(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases}$$

we can mathematically describe the decision as

$$a = \begin{cases} 1, & \text{if } w_1\, x_1 + w_2\, x_2 + w_3\, x_3 \geq t \\ 0, & \text{else} \end{cases}$$

$$= \Theta(w_1\, x_1 + w_2\, x_2 + w_3\, x_3 - t)$$

Or, more generally, a decision based off of $m \in \mathbb{N}$ inputs:

$$a = \Theta(w_1\, x_1 + \cdots + w_m\, x_m + b)$$

where $b := -t$.

With that, we have nearly arrived at a common computer science adaptation of a biological neuron. First off, we drop the restriction of only allowing 0 and 1 to be viable inputs. (If you weren't sure wheather or not the previously mentioned lecture's contents were actually useful for your studies, you might just give the related input a value of e.g. $\frac{1}{2}$.) Moreover, we also drop the restriction of choosing the Heaveside step function as the used mapping. In theory – mathematically speaking –, any function f mapping from the real numbers is sufficiently equipped to be inserted at its place; in partice, there are various reasons for which one might select one different to Θ. This generalisation gives us an universal form of the *artificial neuron*

$$a = f\big(\underbrace{w_1\, x_1 + \cdots + w_m\, x_m + b}_{z}\big) \tag{2.1}$$

The term is denoted by a because it is called the neuron's *activation*. Furthermore, f is called the *activation function*, w_k are called *weights* and b is the *bias*. The *perceptron* then is an artificial neuron with $f = \Theta$. A widespread graphical depiction of such neurons is shown in Figure 2.1.

As implied, using the Heaveside step function Θ as the activation function comes with multiple idiosyncrasies which – depending on the task – might be considered displeasing shortcomings. Not only is one restricted to only getting results of exactly 0 and 1, but also can infinitesimal changes around $x = 0$ completely change said result. At that position, the smallest possible change in input can result in the biggest possible change in output. One can see how such sensitivity to minuscule variations in the argument can be problematic for a model possibly used for guiding decisions in applications – simply consider the conceivable occurance of inaccuracies, errors or even uncertainty in given input data. One might be inclined to inherently integrate a measure of how "sure" a system is of its decision in the result itself.

Using the step function will also prove to be an insufficient practice for later on

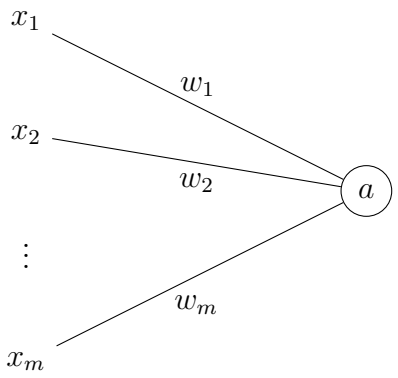


Figure 2.1: Visual representation of an artificial neuron.

training neural networks, which will be made up of such neurons. In training one will, simply put, gradually change the values of the weights and biases repeatedly, in such a way, that the output of the network approaches the desired results. As incremental improvements of it either show up in the results all of the sudden or not at all, the step function is making it hard to evaluate how much progress in training was made and how possible future changes would impact the model's performance.[2]

One way of solving these issues is the usage of a 'smoothened' step function, such as the sigmoid function

$$\sigma : \mathbb{R} \to (0,1),\ \sigma(x) = \frac{1}{1+e^{-x}} \tag{2.2a}$$

or the hyperbolic tangent

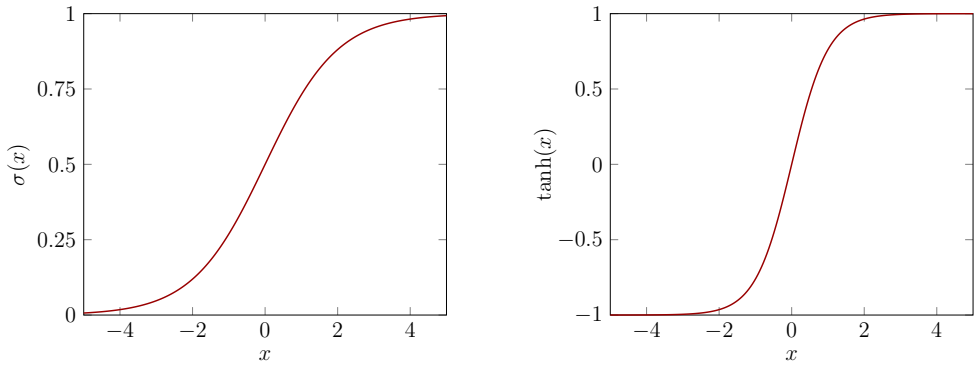
$$\tanh : \mathbb{R} \to (0,1),\ \tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^{2x}-1}{e^{2x}+1} \tag{2.2b}$$

as shown in Figures 2.2a and 2.2b.

Instead of only mapping to the integers 0 and 1, the sigmoid function maps into the open interval between them and, in doing so, the meets our previously raised demands: As it is continuous, small changes in the weighted sum z, wheather induced by changes in some input x_k, some weight w_k, or the bias b, only prompt small changes in the result $\sigma(z)$. Moreover, it satisfies $\lim_{x\to\pm\infty}\sigma(x) = \lim_{x\to\pm\infty}\Theta(x)$, so that, in cases where the weighted sum z differs strongly from 0, so $|z| \gg 0$, we get $\sigma(z) \approx \Theta(z)$. Additionally, in instances where this is not the case, we still obtain a "Yes" or "No" by simply rounding the result – said result now, however, also expresses how "sure" of a decision was made.

The hyperbolic tangent meets similar criteria with the most striking difference simply being that the "decision interval" can be considered streched from $[0,1]$ to

[2]We note that, for being able to use subsequently introduced training algorithms, it is not an advantage, but actually a necessity to use a differentiable activation function.



(a) Sigmoid function. (b) Hyperbolic tangent.

Figure 2.2: Common sigmoidal activation functions.

$[-1, 1]$. Then -1 and 1, rather than 0 and 1, would be considered the sure decisions for one result[3]. Both have been used in applications, as the designation of 0 and 1 as 'No' and 'Yes' – which was initially conceived when transitioning from biological perceptrons into artificial ones – is obtained solely by using standard practices from computer science with no more deep-rooted reason in mind.

Herewith we have adequately described artificial neurons to explain neural layers and neural networks.

Before we do so it is, however, worth mentioning, that even the selection of an activation function of sigmoidal shape, such that it resembles the Heaveside step function, is somewhat arbitrary. This can be exemplified when considering a newly established class of activation functions, the so called *Rectified Linear Unit*, or *ReLU*,

$$\mathcal{R} : \mathbb{R} \to \mathbb{R}_{\geq 0}, \ \mathcal{R}(x) = \max(0, x) \tag{2.3a}$$

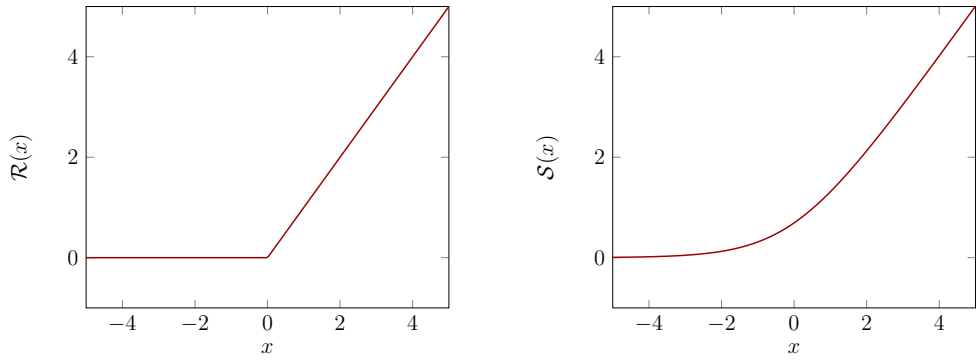
a common differentiable version of which, called *SmoothReLU* or *softplus*, is given by

$$\mathcal{S} : \mathbb{R} \to \mathbb{R}_{>0}, \ \mathcal{S}(x) = \ln(1 + e^x) \tag{2.3b}$$

These activation functions are shown in Figures 2.3a and 2.3b. In the past decade, they superceded these sigmoidal functions as the primarily used ones. "The use of ReLUs was a breakthrough that enabled the fully supervised training of state-of-the-art deep networks" [RZL17]. This is because the derivative of the activation function plays a significant role in the training of neural networks, as we will later see. The non-diminishing derivative of the ReLU for big arguments $x \gg 0$ favors it in training over sigmoidal ones.

Yet, the way these different activation functions work in implementations won't

[3]The relation $\sigma(x) = \frac{1}{2}(1+\tanh(\frac{x}{2}))$, which can be easily verified using the second given expression in the definition of tanh, shows how closely related both options are.



(a) Rectified Linear Unit. (b) Softplus.

Figure 2.3: Common ReLU activation functions.

fundamentally change the way a network functions conceptually – far from it, actually. When programmed well, the activation function can be interchanged by the alteration of a single line of code. Furthermore, the search for better activation functions is an ongoing one. With that we move on, having solely acknowledged ReLU funczions as the modern standard and having discussed the sigmoidal ones in a bit more detail, as an example to which we can turn when in need for intuition.

Now, having discussed fairly in detail the workings of artificial neurons, we are sufficiently equipped to smoothly explain how we can compose such artificial neurons into ...

Layers and networks

A layer is simply a set of $n \in \mathbb{N}$ artificial neurons, each with its own unique weights and bias, however normally all with the same activation function of one's choosing, which are all given the same inputs $x_1, \dots, x_m$. Instead of w_k being the weight of the k-th input to our *one* neuron, we denote by $w_{j,k}$ the weight of the k-th input to our j-th neuron. We apply similar reasoning to the bias-values, the j-th of which we denote by b_j. With that, the activation of our j-th of n artificial neurons is given by

$$a_j = f\big(\underbrace{w_{j,1}\, x_1 + \cdots + w_{j,m}\, x_m + b_j}_{z_j}\big) \tag{2.4}$$

This concept is likely more easily understood when considering a expanded graphical representation, as shown in Figure 2.4.

Remembering our intuitive example of thinking of artificial neurons as decision makers, a layer essentially models recieving m units of information and then making n decisions off of it simultaniously.

We reach the notion of a neural network by concluding with the glaring idea, that we may use the output of a layer as an input for another layer, just like the originally given inputs. Furthermore we can do so not only once, but $L \in \mathbb{N}$ times. Our mathematical notation is then expanded by an superscript index $l \in \{1, \dots, L\}$

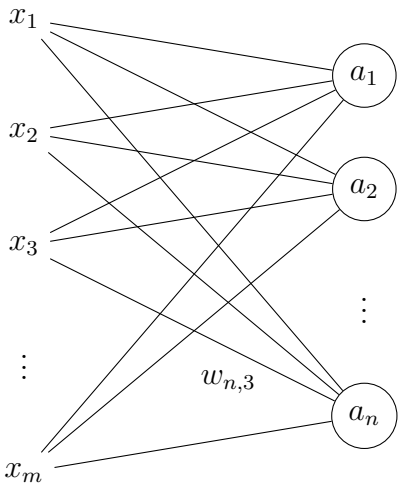


Figure 2.4: Visual representation of a layer.

indicating the number of the layer:

$$a_j^l(x_1, \ldots, x_{m_{l-1}}) = f\big(\underbrace{w_{j,1}^l\, x_1 + \cdots + w_{j,m_{l-1}}^l\, x_{m_{l-1}} + b_j^l}_{z_j^l}\big) \tag{2.5}$$

Here we denoted the number of neurons in the l-layer with m_l. With that, a_j^l is the activation of the j-th neuron of the l-th layer, b_j^l the bias of said neuron, and $w_{j,k}^1$ is now the weight of the connection arriving at that neuron from the k-th neuron of the previous layer. Again, a graphical representation of a small feed-forward neural network is shown in Figure 2.5.

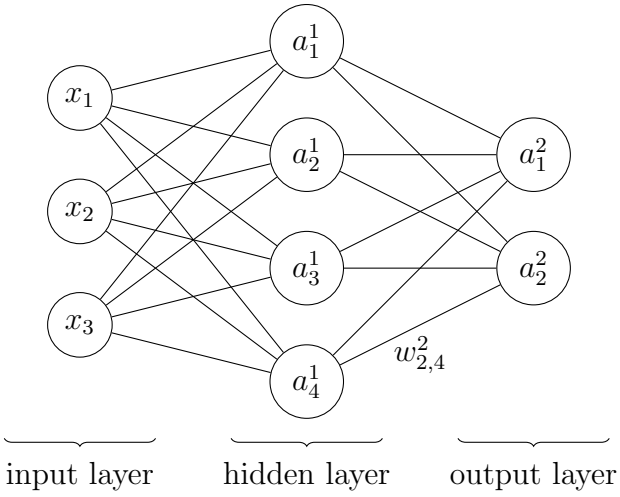


Figure 2.5: Visual representation of a 3-4-2 neural network.

For the shown network we have $L = 2$ with $(m_1, m_2) = (4, 2)$.

Notice how we drew circles around the inputs aswell. We can think of our inputs as a layer aswell – a zeroth layer –, however with neurons of fixed, predetermined activations – the input values –, rather than calculated ones. With that, we denote the $m_0 := n$ inputs $x_1, \ldots, x_{m_0}$ as $a^0_1, \ldots, a^0_{m_l}$. This initial layer of a network is plainly called the input layer.

All $L - 1$ layers between the input and the last layer are called hidden layers[4]. In the example given in Figure 2.5 there is only one hidden layer. Nonetheless, in the general case there can be arbitrarily many. As mentioned in the introduction to this chapter, deep networks are ones with multiple hidden layers, so networks with $L > 2$.

The last layer of a network is called the output layer. It ultimately contains the values we wish to obtain, computed by repeatedly using the already given activations of a certain layer to calculate the next one's, determined by arrival at the output layer. This process is called the *feed-forward* algorithm, hence the name feed-forward neural network. (The feed-forward algorithm is also presented as Step 2 in `Algorithm 1` in Section 2.3.)

Recalling the example of the decision described by a single perceptron, one can easily imagine how sufficiently large networks can be capable of executing very complex tasks. To quantify this at least for a subset of neural networks: The existence of a NAND-Perceptron ($w_1 = -1$, $w_2 = -1$, $b = 0$ for $f = \Theta$) already implies that boolean networks are functionally complete, that is, any boolean function can be computed by a sufficiently large boolean neural network. We will expand this discussion on the capabilities of feed-forward neural networks towards the end of this chapter in Section 2.4.

At this point, one might be inclined to pause and ponder what one actually gains from the usage of neural networks. While we – up to this point, vaguely – argued that these networks appear capable to perform complex decision-making tasks, the prospect of manually setting up a network and configuring all its weights and biases does not seem like an improvement compared to conventional methods of designing algorithms.

The crux: If one sets up a network to perform a certain task – in such a way, that the input and output layer indeed have suitable dimensions to handle the desired in- and outputs – then one can automate the search for fitting weights and biases, provided one has access to so called training data. A training set is a set of labeled data, as mentioned in Section 2.1, consisting of pairs of inputs and the corresponding, desired outputs. The automation of this process (called training) is explained thoroughly in the following section.

We conclude this introduction of classical feed-forward neural networks by quickly presenting a more compact mathematical notation: By arranging the input arguments and weights, $x_1, \ldots, x_{m_l}$ and $w^l_{j,1}, \ldots, w^l_{j,m_{l-1}}$, in vectors, $\mathbf{x} = (x_1, \ldots, x_{m_{l-1}})^T$

[4]We note that this label might be a bit misleading, as there is nothing that technically prevents us from accessing these values.

and $\mathbf{w}_j^l = (w_{j,1}^l, \dots, w_{j,m_{l-1}}^l)$, we can write a neuron's activation as

$$a_j^l(\mathbf{x}) = f\big(\mathbf{w}_j^l\,\mathbf{x} + b_j^l\big) \tag{2.6}$$

Moreover, we can proceed by naturally composing the m_l biases b_j^l of the l-th layer into a vector $\mathbf{b}^l$, as well as the corresponding weight-vectors $\mathbf{w}_j^l$ into a matrix W^l. Eventually, we may write the activations of the neurons of the l-layer in a vector

$$\mathbf{a}^l(\mathbf{x}) = f\big(W^l\,\mathbf{x} + \mathbf{b}^l\big) \tag{2.7}$$

where the activation function of a vector $\mathbf{v} \in \mathbb{R}^n$ is (well-) defined by $f(\mathbf{v}) = f(\sum_{i=1}^n v^i\,\mathbf{e}_i) := \sum_{i=1}^n f(v^i)\,\mathbf{e}_i$ where $\{\mathbf{e}_i : i = 1, \dots, n\}$ is the standard basis of $\mathbb{R}^n$ and v^i the components of $\mathbf{v}$ regarding said basis.

Finally, including our understanding of the inputs as the network's 0-th layer, we obtain

$$\mathbf{a}^l = f\big(W^l\,\mathbf{a}^{l-1} + \mathbf{b}^l\big) \tag{2.8}$$

The output of a network then is

$$\mathbf{n}(\mathbf{a}^0) = \mathbf{a}^L\big(\mathbf{a}^{L-1}(\dots\mathbf{a}^1(\mathbf{a}^0)\dots)\big) \tag{2.9}$$

2.3 Classical Feed-Forward Neural Networks: Training

To automate the optimization of a network with a given set of training data $T = \big\{(\mathbf{x}_i, \mathbf{y}_i) : i \in \{1, \dots n\}\big\}$, we first want to create a measure showing how well it performs, to quantify how much better or worse the network gets for certain changes made to its weights and biases. To achieve that, we introduce a *cost function* called C, that takes all weights and biases of a network as arguments, and data on which to check the networks performance on as parameters. While there may exist various practical options, one of the most commonly used choices would certainly be

$$C_T(w_{1,1}^1, \dots, b_1^1 \dots) = \frac{1}{2\,|T|} \sum_{i=1}^{|T|} \|\mathbf{n}(\mathbf{x}_i) - \mathbf{y}_i\|^2$$

Gradient descent

The gradient descent is a numerical method for the minimization or maximization of a function. The idea is that, in order to find such an extremum, e.g. a minimum as we aim to identify, one must simply 'move' from an arbitarily chosen starting point in the direction of the negative derivative. For our multidimensional cost function C that is $-\nabla C$. In other words, having a function C measuring how well a network performs, the partial derivates of that function with regards to a weight or bias tells one, how changing that weight or bias changes the performance of the

network. Repeatedly updating weights and biases with $w^l_{j,k} \to w^l_{j,k} - \eta \frac{\partial C}{\partial w^l_{j,k}}$ and $b^l_j \to b^l_j - \eta \frac{\partial C}{\partial b^l_j}$, where $\eta > 0$ is called the *learning rate*, one ultimately obtains a network with a minimized cost function, a trained network. We note that this algorithm doesn't necessarily converge towards a possible global minimum, but only a local one.

Backpropagation

At its core, backpropagation is an efficient algorithm for the computation of the partial derivates of the gradient. In this section, the presentation of the algorithm is preceded by a quick derivation of its fundamental equations.

There are two neccessary requirements for the cost function C for the backpropagation algorithm to work. Foremost, the cost function of a whole training set C_T must be calculated as the average over the cost function for individual training pairs:

$$C_T = \frac{1}{|T|} \sum_{i=1}^{|T|} C_i$$

This e.g. is the case for the cost function, as defined in Equation 2.3, where $C_i = \frac{1}{2} \|\mathbf{n}(\mathbf{x}_i) - \mathbf{y}_i\|^2$

Furthermore, one needs to be able to write the cost function as a function of the activations of the output layer a^L. This automatically ensures that the cost function can be written as a function of all weights and biases (as done in 2.3), since one can always express the activations of a layer in terms of the layers weights, biases and the previous layer's activations, see 2.8.

In a search of the partial derivates $\partial C / \partial w^l_{j,k}$ and $\partial C / \partial b^l_j$, the partial derivative $\partial C / \partial z^l_j$ is key, as we can express both in terms of it by usage of the chain rule. As the latter partial derivative is that crucial, we denote it by

$$\delta^l_j := \frac{\partial C}{\partial z^l_j} \tag{2.10}$$

For $l = L$, using the chain rule, we find the following expression[5]:

$$\begin{aligned}\delta_j^L &= \sum_{k=1}^{m_L} \frac{\partial C}{\partial a_k^L} \frac{\partial a_k^L}{\partial z_j^L} \\ &= \sum_{k=1}^{m_L} \frac{\partial C}{\partial a_k^L} \left(\delta_{jk}\, f'(z_j^L)\right) \\ &= \frac{\partial C}{a_j^L}\, f'(z_j^L)\end{aligned}$$

The derivative f' of the activation function f is known to us. The concrete form of $\partial C / \partial a_j^L$ depends on our choice of the cost function, but nevertheless is still computable by our requirements. For the previously introduced one we would obtain $\partial C / \partial a_j^L = a_j^L - y_j$, where $a_j^L \cong \mathbf{n}(\mathbf{x})_j$ and y_j being the j-th component of $\mathbf{y}$ of the training pair $(\mathbf{x}, \mathbf{y})$, so that

$$\delta_j^L = \left(a_j^L - y_j\right)\, f'(z_j^L)$$

For $l < L$ we can find an expression of δ_j^l that is dependend on δ_j^{l+1} with which one can iteratively compute all δ_j^l.

$$\delta_j^l = \sum_{k=1}^{m_{l+1}} \frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} \overset{*}{=} \sum_{k=1}^{m_{l+1}} \delta_k^{l+1}\, w_{k,j}^{l+1}\, f'(j_j^l)$$

where, at $(*)$, we inserted the Definition 2.10 of δ_k^{l+1} and used

$$\begin{aligned}\frac{\partial}{\partial z_j^l}\, z_k^{l+1} &= \frac{\partial}{\partial z_j^l} \left(w_{k,1}^{l+1}\, a_1^l \;+\; \ldots \;+\; w_{k,m_l}^{l+1}\, a_{m_l}^l + b_k^{l+1}\right) \\ &= \frac{\partial}{\partial z_j^l} \left(w_{k,1}^{l+1}\, f(z_1^l) \;+\; \ldots \;+\; w_{k,m_l}^{l+1}\, f(z_{m_l}^l) + b_k^{l+1}\right) \\ &= w_{k,j}^{l+1}\, f'(z_j^l)\end{aligned}$$

Defining $\delta^l = \left(\delta_1^l, \ldots, \delta_{m_l}^l\right)^T$ we can make use of our matrix notation and write these equations in a more compact form[6].

$$\delta^L = \nabla_{\mathbf{a}^L} C \odot f'\left(\mathbf{z}^L\right) \tag{2.11a}$$

$$\delta^l = \left[\left(W^{l+1}\right)^T \delta^{l+1}\right] \odot f'\left(\mathbf{z}^l\right) \tag{2.11b}$$

[5] By δ_{jk} we denote the Kronecker delta defined by $\delta_{jk} = \begin{cases} 1, & \text{if } j = k \\ 0, & \text{if } j \neq k \end{cases}$.

[6] By $\odot$ we denote the Hadamard product of $\mathbf{v}, \mathbf{w} \in \mathbb{R}^n$, $n \in \mathbb{N}$, defined by $(\mathbf{v} \odot \mathbf{w})_i = v_i\, w_i$.

We may now use the obtained Equations 2.11 for the partial derivates with respect to the weighted sum to find useful expressions to compute the partial derivates with respect to the weights and biases. For that, consider

$$\begin{aligned}\frac{\partial C}{\partial w^l_{j,k}} &= \sum_{i=1}^{m_l} \frac{\partial C}{\partial z^l_i} \frac{\partial z^l_i}{\partial w^l_{j,k}} \\ &= \sum_{i=1}^{m_l} \frac{\partial C}{\partial z^l_i} \left(\delta_{ij}\, a^{l-1}_k\right) \\ &= \delta^l_j\, a^{l-1}_k\end{aligned}$$

$$\begin{aligned}\frac{\partial C}{\partial b^l_j} &= \sum_{i=1}^{m_l} \frac{\partial C}{\partial z^l_i} \frac{\partial z^l_i}{\partial b^l_i} \\ &= \sum_{i=1}^{m_l} \frac{\partial C}{\partial z^l_i} \delta_{ij} \\ &= \delta^l_j\end{aligned}$$

In summary:

$$\frac{\partial C}{\partial w^l_{j,k}} = \delta^l_j\, a^{l-1}_k \tag{2.12a}$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \tag{2.12b}$$

The Equations 2.11 and 2.12 are the fundamental equations of backpropagation. Using them is an effective way to compute ∇C. Considering the terms for δ^l given in 2.11, it becomes clear how the choice of the activation function impacts the training of a neural network, as a vanashing derivative, such as the sigmoidal ones for $|x| \gg 0$, causes the gradient to vanish aswell.

The backpropagation algorithms computing ∇C_i and ∇C_T are given by ...

`Algorithm 1`: Backpropagation for a training pair.

1. **Input:** A training pair $(\mathbf{x}, \mathbf{y})$.
2. **Feed-Forward:** For each $l \in \{1, \dots, L\}$ compute:

$$\mathbf{z}^l = W^l \, \mathbf{a}^{l-1} + \mathbf{b}^l \qquad \text{and} \qquad \mathbf{a}^l = f(\mathbf{z}^l).$$

3. **Backpropagation:** For each $l \in \{L, \dots, 1\}$ compute:

$$\delta^l = \begin{cases} \nabla_{\mathbf{a}^L} C \odot f'(\mathbf{z}^l), & \text{if } l = L \\ \left[(W^{l+1})^T \, \delta^{l+1}\right] \odot f'(\mathbf{z}^l), & \text{if } l < L \end{cases}$$

4. **Output:** The partial derivatives

$$\frac{\partial C}{\partial w^l_{j,k}} = \delta^l_j \, a^{l-1}_k \qquad \text{and} \qquad \frac{\partial C}{\partial b^l_j} = \delta^l_j$$

`Algorithm 2`: Backpropagation for a training set.

1. **Input:** A training set $T = \big\{(\mathbf{x}_i, \mathbf{y}_i) \, : \, i \in \{1, \dots, n\}\big\}$.
2. **Average 1:** For each $i \in \{1, \dots, n\}$ use `Algorithm 1` to compute ∇C_i.
3. **Average 2:** Compute $\nabla C_T = \frac{1}{|T|} \sum_{i=1}^{n} \nabla C_i$.

The training of a network via gradiant descent is thus done, by repeatedly applying the backpropagation algorithm to calculate the gradient and updating the network.

Stochastic gradient descent

The idea of the stochastic gradient descent is simple: Instead of calculating the gradient of the cost function with regards to the whole set of training data T, one does so for a subset $B \subset T$, expecting

$$\nabla C_T \approx \nabla C_B$$

thus essentially trading off computational accuracy against computational speed.

However, the number of training steps in the training of a neural network can easily exceed hundreds or thousands. If we only calculate the gradient with regards to one subset B for every repetition, we optimize the network only for that subset of data, therefore not only not utilizing all of the given data, but possibly, depending on the composition of said subset, not even effectively training for every case given in

T. As one solution to this, consider a randomly created partition[7] $P = \{B_1, \dots, B_n\}$ of T with *batches* B_i, that are of equivalent size – meaning the sets B_i have the same cardinality when neglecting the possible indivisibility of $|T|$ regarding the rank n of P. Then, for the first n training steps, one updates the network by ∇C_{B_i} and at every n-th training step a new randomly generated partition of such kind is created – and the process is repeated.

While implementations of the stochastic gradient descent may vary considerably, the speed-accuracy-tradeoff is generally seen as favorable when training neural networks, which is why, in practice, modified versions of it remain the prevailent training algorithm [GBC16, p. 14].

Further modifications

Even for the most basic case of training multilayered feed-forward neural networks, there exists a multitude of modifications to optimize training for given circumstances. As mentioned, the selection of the activation function plays a huge role in the training of a network. Furthermore, the values the weights and biases are initialized at, the value of the learning rate, and the overall chosen structure of the network all prove to be promising starting points for optimization. Also, there exist modifications to the training algorithm itself, such as adding momentum. However, the intricancies of these propositions far exeed the scope of this thesis, which is why we will leave it at a mere mentioning of them.

2.4 An Alternative Introduction to Supervised Machine Learning

In this section we aim to provide an alternative, more mathematical, introduction to the field of supervised machine learning. We do so to not only get an different viewpoint of the subject, but also at the premise of gaining additional insights in general. In theory, no prior knowledge given in the previous sections is necessary for the comprehension of this introductory way – it will however likely become clear why we chose to give the more picturesque introduction beforehand.

Suppose one has a set Ω and a mapping $I : \Omega \to \mathbb{R}$. As $\mathbb{R}$ is an ordered field, one can ask if there exist global extremums or, if Ω is equipped with a topology, even local ones. For e.g. $\Omega = \mathbb{R}^d$ this is a thoroughly discussed undertaking. However, Ω mustn't necessarily be a subset of the complex numbers. For example, the set Ω itself may be composed of mappings $f : A \to B$, so

$$\Omega = \{f \mid f : A \to B\}$$

[7] A partition of a set X is a set P, consisting of sets P_i, such that X is the disjoint union of all P_i.

The function $I : \Omega \to \mathbb{R}$ is then called a *functional*, the exemplary task of determining the existence of and finding extremums a *variational problem*.

A known, most important example of this to physicists is, of course, Hamilton's principle, which states that, for physically possible trajectories $\mathbf{q}$, the action functional

$$S[\mathbf{q}] = \int_{t_1}^{t^2} L\big(\mathbf{q}(t), \dot{\mathbf{q}}(t), t\big)\, dt$$

must take on a stationary value – and from the solution of which the Euler-Lagrange-equations, and thus analytical mechanics, arise.

To propose an example affiliated with computer science, suppose you have a set of data points $\{a_1, \ldots, a_n\}$, for each of which you have stored a category one may assign it to, denoted $\{b_1, \ldots, b_n\}$. If we are dealing with classical data, which can be stored on a hard drive, we can trivially assume that there exists a representation such that $\{a_1, \ldots, a_n\} \subset \{0,1\}^{d_1}$ for some suitable $d_1 \in \mathbb{N}$. We can also always enumerate the categories, to which we assign the data points, in a base-2 numeral system, so $\{b_1, \ldots, b_n\} \subset \{0,1\}^{d_2}$ for a suitable $d_2 \in \mathbb{N}$. When searching for a classifier function f, that assigns to each data point a_i its category b_i, we are thus looking for a function $f : \{0,1\}^{d_1} \to \{0,1\}^{d_2}$. However, as $\{0,1\}^d \subset [0,1]^d \; \forall\, d \in \mathbb{N}$, we may extend that search to the set $\mathfrak{F} := \big\{f \mid f : A \to B\big\}$ with $A = [0,1]^{d_1}$ and $B = [0,1]^{d_2}$. Then we can, with an appropriate distance measure d on B, assess the quality of a classifier f with the functional

$$C[f] = \frac{1}{n} \sum_{i=1}^{n} d\big(f(a_i),\, b_i\big) \tag{2.13}$$

A straightforward path to finding the optimal classifer mapping $\underline{f}$ would therefore be the minimization of the functional C:

$$\underline{f} = \arg\min_{f \in \mathfrak{F}} C[f] \tag{2.14}$$

To find an adequate solution to this problem, let's start by considering the following ...

Definition 2.1 (Squashing function, [HSW89])
A function $\phi : \mathbb{R} \to [0,1]$ is called a squashing function if it satisfies

(SF1) $\lim\limits_{x \to -\infty} \phi(x) = 0$

(SF2) $\lim\limits_{x \to +\infty} \phi(x) = 1$

(SF3) It's monotonically increasing

Additionally we define its generalisation to $n \in \mathbb{N}$ dimensions by

$$\Phi_\phi^n : \mathbb{R}^n \to [0,1]^n, \mathbf{x} = \big(x_1, \dots, x_n\big) \mapsto \big(\phi(x_1), \dots, \phi(x_n)\big)$$

Remark
Both Θ and σ, as introduced in Section 2.2, fall under this definition.

Definition 2.2 (Feed-Forward neural network)
For $L \in \mathbb{N}$, $\mathbf{m} = (m_0, m_1, \dots, m_L) \in \mathbb{N}^{L+1}$ we define a $\mathbf{m}$-feed-forward-neural-network to be a 3-touple (W, b, ϕ) that satisfies the following conditions:

(NN1) $W = (W^1, W^2, \dots, W^L)$ *with* $W^l = (w^l_{j,k})_{\substack{j=1,\dots m_l \\ k=1,\dots m_{l-1}}} \in \mathbb{R}^{m_l \times m_{l-1}}$

(NN2) $b = (\mathbf{b}^1, \mathbf{b}^2, \dots, \mathbf{b}^L)$ *with* $\mathbf{b}^l = (b^l_j)_{j=1,\dots m_l} \in \mathbb{R}^{m_l}$

(NN3) $\phi : \mathbb{R} \to [0,1]$ *is a squashing function*

We call a feed-forward neural network multilayered, if $L > 1$.
Furthermore, we define

$$\mathcal{L}^l : \mathbb{R}^{m_{l-1}} \to [0,1]^{m_l}, \mathbf{x} \mapsto \Phi_\phi^{m_l}(W^l \mathbf{x} + \mathbf{b}^l)$$

to be the neural network's l-th layer function,

$$\mathcal{N} = \mathcal{L}^L \circ \cdots \circ \mathcal{L}^1$$

to be its network function, and

$$\mathcal{N}_{lo} = h_{W^L} \circ \mathcal{L}^{L-1} \circ \cdots \circ \mathcal{L}^1$$

to be its linear output network function.

This characterization of classical feed-forward neural networks is equivalent to the way they were introduced in the Section 2.2. We will similarly call $w^l_{j,k}$, b^l_j and ϕ, as defined in Definition 2.2, the weights, biases and activation function. If we denote by $\mathfrak{N}$ to set of all network functions, or rather say their restriction from $\mathbb{R}^{m_0}$ to $[0,1]^{m_0}$, then $\mathfrak{N}$ is a subset of $\mathfrak{F}$. The following theorem indicates how we can utilize said subset to solve our variational problem.

Theorem 2.1 (Universal approximation theorem, 1-dim.)
Let $\phi : \mathbb{R} \to [0,1]$ be a squashing function and $A \subset \mathbb{R}^m$ be compact.
Then, for every function $f \in \mathcal{C}(A)$ and every $\epsilon > 0$, there exist $N \in \mathbb{N}$ as well

as $v_i, b_i \in \mathbb{R}$ and $\mathbf{w}_i \in \mathbb{R}^m$ for $i = 1, \dots, N$ such that the function F defined by

$$F(\mathbf{x}) = \sum_{i=1}^{N} v_i \, \phi\big(\mathbf{w}_i \cdot \mathbf{x} + b_i\big)$$

approximates f with an error ϵ, meaning

$$|F(\mathbf{x}) - f(\mathbf{x})| < \epsilon \quad \forall \mathbf{x} \in A$$

Proof
This version of the theorem was proved in [HSW89]. ∎

Remark
There exists multiple versions of the universal approximation theorem, differing e.g. in their requirements for the activation function. The one mentioned here was preceded by Cybenko [Cyb89] and succeeded e.g. by further work by Hornik, et.al. [HSW90; Hor91; Hor93]. Furthermore, the theorem has been proven for a wider classes of functions, that include the mentioned ReLU functions, see [Les+93].

Before elaborating the impact of this theorem, we quickly affirm that this statement, as well as its afterward discussed consequences, hold true when generalized to an n-dimensional output space.

Corollary 2.2 (Universal approximation theorem, n-dim.)
The universal approximation Theorem 2.1 holds true for any continuous function $f : \mathbb{R}^m \to \mathbb{R}^n$.

The function F defined in Theorem 2.1 correlates to a linear output network function of a $(m, N, 1)$-*feed-forward-neural-network*. In other words, this theorem states that $\mathfrak{N}$ is a dense subset of $\mathfrak{F}$. What is notable about this dense subset is that we can fully parameterize any function in it with a set of arguments – the weights and biases. Provided that there exists an error $\epsilon > 0$, to which we can ignore deviations in our application, we have reduced the problem of finding the minimum of a functional to finding the minimum of a function:

$$C[f] \simeq C(w^1_{1,1}, \dots, b^1_1, \dots) \tag{2.15}$$

That transition is remarkable, as we now have available to us a huge toolset developed for that task, amongst them also numerical methods – such as the gradient descent. In its essence, supervised machine learning can be seen as a handy way of – approximately – solving a variational problem.

Remaining Problems

As previously stated, the developed conclusion extends to cases where one seeks to find classifiers with multidimensional outputs. Moreover, the described approximation capabilities also hold true for networks with $L > 2$ [GBC16, p. 194].

Outstanding issues were described by Hornik, et. al. as follows: "This [result] implies that any lack of success in applications must arise from inadequate learning, insufficient numbers of hidden units or the lack of a deterministic relationship between input and target" [HSW89]. In addition, the performance of the found classifier f on data outside the given set $\{a_1, \ldots, a_n\}$ is of prime importance.

In applications it is often the case that one be certain that the "lack of a deterministic relationship between input and output" is non-existing. While one will often see improvement in learnability, required network width and generelization error when using deeper networks [GBC16], finding adequate network structures which can be trained satisfactorily is, at its heart, the chief undertaking and pursuit of machine learning work and research – a deep dive into which is, as previously indicated in the conclusion of Section 2.3, beyond the scope of this thesis.

Universal approximation for non-linear output

We did not yet acknowledge that most results in this section assume the neural network to have a linear output layer – which is contrary to how we introduced them in previous sections, where every neuron of every layer had the same, usually non-linear, activation function. This begs the question if the universal approximation theorem even holds true for how we implement neural networks in practice.

Let's assume that the activation function ϕ is not only continuous, but also bijective with a continuous inverse function – so a homeomorphism. While this is a noteable restriction on possible activation functions it holds true for common ones, such as the discussed sigmoid or hyperbolic tangent functions. If we then want to approximate a continuous function f with a non-linear network, we can instead find a linear output function F that approximates the continuous function $\phi^{-1} \circ f$, such that:

$$F \approx \phi^{-1} \circ f$$

We can then rudimentarily conclude

$$\phi \circ F \approx f$$

where

$$\big(\phi \circ F\big)(\mathbf{x}) = \phi\Bigg(\sum_{i=1}^{N} v_i \, \phi\big(\mathbf{w}_i \cdot \mathbf{x} + b_i\big) + \underbrace{0}_{\substack{\text{output} \\ \text{layer} \\ \text{bias}}} \Bigg)$$

is equivalent to a network with a non-linear output layer. This argumentation is given in [Nie15] to illustrate how feed-forward neural networks without linear output layers are universal approximators aswell. It is, however, not such an elegant solution as the universal approximation theorem gives for linear output networks, as we cannot limit the difference $|\phi \circ F - f|$ by any given number $\epsilon \in \mathbb{R}^+$.

In practice, this shortcoming is believed to often be rather insignificant compared to the previously discussed problems of learnability, and thus often neglected.

2.5 Usage and Limits of Supervised Machine Learning

In this chapter, we introduced the fundamental class of neural networks, the (deep) feed-forward neural network with fully connected layers.

Following Sections 2.2 and 2.3, we can understand machine learning with neural networks both as an automated way of designing an algorithm *by* a machine or, when one considers the backpropagation algorithm to be an inherent part of the network itself, as a machine training *itself* – a machine *learning* – to properly perform a task. In Section 2.4, we further extended our understanding by considering supervised machine learning as an inventive way of solving a variational problem.

Machine learning works best when given a big, representative subset of the set of all possible inputs and their desired outputs. "Big" because one needs an adequate amount of examples to generalize the training beyond those given in the training set; "representative" because the neural network may only *learn* cases occuring in its training data.

Machine learning does not completely replace traditional programming in computer science, as its usage serves little purpose when dealing with assignments accompanied by thorough descriptions of necessary procedures – or ones, that can be broken down into solvable ones. As an example of this, consider the task of computing the n-th Fibonacci number. Furthermore, as mentioned the successful training of neural networks necessitates sufficiently large data sets and thus presupposes, that the acquisition of such data is a less complicated endeavor than simply designing the algorithm yourself.

Nonetheless, in the age of big data there is an enormous number of domains in which machine learning and affiliated fields thrive[8] and, as was made clear at the beginnings of this thesis, one can definitely expect that amount to be ever-growing.

[8] Common examples often refered to are image recognition – the identification of an image's contents by machine – and speech recognition – the automated transcription and understanding of verbal speech.

3 Quantum Mechanics

In this chapter, we aim to provide swift summary of the essential elements of quantum mechanics to provide a foundation of knowledge that is sufficient to follow the transition of classical feedforward neural networks to its quantum counterparts in the following chapter. Almost all of this chapter's content will be found in any ordinary quantum mechanics textbook, the one largely consulted by us was [NC10].

We start by reciting the postulates of quantum mechanics in a way that assumes that the reader is essentially already familiar with the linear algebra underlying it. As the number, phrasing and, to a slight extend, substance of said postulates will often vary depending on the source of one's choosing, we will not concern ourselves with the discussion of a sensible selection, but simply give an account of their contents necessary to us. Subsequently, we promptly draw attention to some peculiarities of the theory of quantum information and quantum computation, that are crucial for the succeeding work. (Any comparison to material on these studies, such as [NC10], will quickly show, that this thesis has no chance at doing the subject justice in its entirety.)

3.1 Fundamental Postulates of Quantum Mechanics

In quantum mechanics any isolated physical system is described by an Hilbert space $\mathcal{H}$. A vector space is called a Hilbert space, if it is equipped with an inner product $\langle \cdot, \cdot \rangle$ and complete with regards to the norm induced by that inner product. The concrete form of the Hilbert space varies with the system it aims to describe. An often encountered one is $\mathbb{C}^d$, $d \in \mathbb{N}$. Such is also the case for a qubit system, which is described by $\mathcal{H} = \mathbb{C}^d$ with $d = 2$, or, for composite qubit systems, with $d = 2^n$, $n \in \mathbb{N}$. A qubit system can be thought of as the quantum computational equivalent to the classical bit system. Another frequently come upon Hilbert space is the Lebesgue space of square-integrable functions, L^2.

Given the suitable Hilbert space, a system then is fully defined by a vector $\psi \in \mathcal{H}$, a so called *pure state*. For such states, quantum mechanics uses the Dirac notation, or bra-ket notation, denoting them by $|\psi\rangle$ (a "ket"). Additionally, $\langle\psi|$ (a "bra") denotes the element in the dual space $\mathcal{H}^*$ assigned to ψ by the mapping $\psi \mapsto \langle \psi, \cdot \rangle$.

A density matrix ρ is a positive operator on $\mathcal{H}$ with $\mathrm{tr}(\rho) = 1$, characterizing a so called *mixed state*. If a system has a chance $0 \leq p_i \leq 1$ of being in a pure state

$|\psi_i\rangle$, its density operator is given by

$$\rho = \sum_{i=1}^{m} p_i \, |\psi_i\rangle \, \langle\psi_i| \tag{3.1}$$

From the obvious condition $\sum_{i=1}^{m} p_i = 1$, it automatically follows that $\mathrm{tr}(\rho) = 1$. Positivity also directly follows from Equation 3.1. Note that a sum of pure states, such as $|\psi\rangle = \big(\,|0\rangle + |1\rangle\,\big)/\sqrt{2}$, with $|0\rangle$, $|1\rangle$ e.g. being basis vectors of a qubit system, represents a quantum superposition of states, while a density operator represents classical probabilities of a system being in any given state $|\psi_i\rangle$. If a system is in a pure state $|\psi\rangle$, its density matrix ρ is a onedimensional projector given by $\rho = |\psi\rangle \, \langle\psi|$. Pure states can be seen as a subset of the more general mixed states.

There exist multiple equivalent formulations of quantum mechanics regarding the time evolution of physical system. In the one we will consider mostly, the Schrödinger picture, the time evolution of a system is formulated in terms of its state $|\psi\rangle$, which is determined by a unitary transformation, meaning there exists a unitary operator U, dependent only on t_1 and t_2, such that

$$|\psi(t_2)\rangle = U(t_2, t_1) \; |\psi(t_1)\rangle \tag{3.2a}$$

The differential equation determining the time evolution operator is the *Schrödinger equation*

$$i\hbar \frac{\partial}{\partial t} \, |\psi\rangle = H \, |\psi\rangle \tag{3.3}$$

where $\hbar$ is the real-valued Planck's constant and H an hermitian operator called *Hamiltonian*, aswell defining the system. The time evolution of a mixed state is then given by

$$\rho(t_2) = U(t_2, t_1) \; \rho(t_1) \; U^\dagger(t_2, t_1) \tag{3.2b}$$

Recalling Equation 3.1, this directly follows from 3.2a with usage of

$$\langle\psi(t_2)| = |\psi(t_2)\rangle^\dagger = \big(U(t_2, t_1) \, |\psi(t_1)\rangle\,\big)^\dagger = \langle\psi(t_1)| \; U^\dagger(t_2, t_1)$$

Measurements of a quantum system are described by a set of operators $\{A_j\}$, where j refers to the possible measurement outcomes. If a quantum system is prepared in the pure state $|\psi\rangle$, or in the mixed state ρ, before measurement, the probability of measuring result j is given by

$$p(j) = \langle\psi| \, A_j \, |\psi\rangle \tag{3.5a}$$

or

$$p(j) = \mathrm{tr}(\,\rho\, A_j\,) \tag{3.5b}$$

Afterwards, the system is in the state

$$\frac{A_j\,|\psi\rangle}{\sqrt{p(j)}} \tag{3.6a}$$

or

$$\frac{A_j\;\rho\;A_j^\dagger}{p(j)} \tag{3.6b}$$

Therefore, in quantum mechanics, measuring, or *observing*, a system changes its state. Classical systems, which remain well-described by the laws of classical mechanics, are characterized as the limit of quantum systems, in which measurements do not disturb the system itself.

Any composite quantum system is described by the tensor product of the component physical systems, so $\mathcal{H} = \mathcal{H}_1 \otimes \cdots \otimes \mathcal{H}_n$, $n \in \mathbb{N}$. Having the i-th system prepared in the pure state $|\psi_i\rangle$, or mixed state ρ_i, the state of the overall system is then given by

$$|\psi\rangle = |\psi_1\rangle \otimes \cdots \otimes |\psi_n\rangle \tag{3.7a}$$

$$\rho = \rho_1 \otimes \cdots \otimes \rho_n \tag{3.7b}$$

Density operators supply a convenient way of mathematically describing subsystems of composite system with the *partial trace*. The partial trace of a two-part composite quantum system is defined by

$$\mathrm{tr}_B(\rho^{AB}) = \mathrm{tr}_B(\rho^A \otimes \rho^B) = \mathrm{tr}(\rho^B)\,\rho^A \tag{3.8}$$

and gives rise to the *reduced* density matrix $\big(\mathrm{tr}(\rho^B)\,\rho^A\big)$, which characterizes the state of the subsystem A. This definition naturally extends to an n-part system.

Lastly, in general, a quantum channel can be thought of as a communication channel with the ability to transfer quantum information. In the following chapter, these channels can be thought of as the transition mapping between adjacent layers of the quantum neural network.

3.2 Quantum Computation and Quantum Information

The usage of quantum mechanics in computer science offers fundamentally new opportunities and solutions with regards to computation, information and cryptography, beyond what is possible with classical means. This circumstance often is

labeled *quantum supremacy*.

The result of quantum information theory that is most notable to our undertaking is the *no-cloning theorem*, which, simply put, states that it is not possible to construct a copy of a quantum state. This immediately tells us, that directly translating the structure of classical feedforward neural networks to their quantum counterparts will not be possible, as their operation is depenend on the ability to transmit a neuron's activation to all neurons of the succeding layer, therefore necessitating the capability to make copies of its value.

To clarify why: When working with quantum computational elements, one must consider which part of the procedure fundamentally incorporates quantum mechanics, as both the data and the algorithm or only one of them may make use of it. To exemplify, we consider the following table.

		Type of algorithm	
		classical	quantum
Type of data	classical	CC	CQ
	quantum	QC	QQ

Table 3.1: Overview of quantum machine learning approaches, [ABG06].

In Chapter 2 we handled machine learning for the CC case, in Chapter 4 we will deal with the QQ case. As that includes dealing with quantum data, the no-cloning theorem applies.

4 Quantum Machine Learning

4.1 Quantum Feed-Forward Neural Networks: Structure

In this chapter, we finally establish the architecture for quantum feed-forward neural network, as formulated in [Bee+20].

With classical neural networks, we started out by considering perceptrons – so by considering inputs and activations that take the form of a *bit*, assuming values in $\{0, 1\}$. (We later expanded that to artificial neurons with generalized activation functions.) With quantum neural networks, we consider quantum perceptrons that take the form of a *qubit* and are therefore characterized by states

$$|\psi\rangle \in \mathbb{C}^2$$

A quantum layer with n perceptrons is a composite quantum mechanical system of n component qubit systems, so defined by states

$$|\psi\rangle \in (\mathbb{C}^2)^{\otimes n} = \mathbb{C}^{2^n}$$

However, as we do not wish to limit ourselves to pure states, we will mathematically express the layer transition mappings in terms of the corresponding mixed states ρ.

A perceptron is now determined by a unitary operation U_j^l – where $l \in \{1, \dots, L\}$ again denotes the layer index and $j \in \{1, \dots m_l\}$ the perceptron index – acting on all (layer-) input qubits and all (layer-) output qubits. So for a layer-mapping, subsequently called the layer channel $\mathcal{E}^l$, with an input state ρ, a perceptron's unitary acts on the composite system of both layers. As the latter is prepared in the state $|0 \dots 0\rangle_l \langle 0 \dots 0|$, where

$$|0 \dots 0\rangle_l = \underbrace{|0\rangle \otimes \cdots \otimes |0\rangle}_{m_l \text{ - times}}$$

the state of said composite system before any operation is given by

$$\rho \otimes |0 \dots 0\rangle_l \langle 0 \dots 0|$$

Consecutive application of all perceptron unitaries U_j^l results in obtaining the new composite state. To then extract the output state, we simply take the partial trace

with regards to the input layer. Therefore, the layer channel is ultimately given by

$$\begin{aligned}\mathcal{E}^l\left(X^{l-1}\right) &= \mathrm{tr}_{l-1}\left(\prod_{j=m_l}^{1} U_j^l \left(X^{l-1}\otimes|0\ldots0\rangle_l\langle 0\ldots0|\right)\prod_{j=1}^{m_l} U_j^{l\,\dagger}\right)\\ &= \mathrm{tr}_{l-1}\left(U^l\left(X^{l-1}\otimes|0\ldots0\rangle_l\langle 0\ldots0|\right)U^{l\,\dagger}\right)\end{aligned} \tag{4.1}$$

where $U^l = U^l_{m_l}\ldots U^l_1$ is the so called layer unitary. In terms of our mathematical expressions, the dimension change between layers now stems from taking the partial trace, rather than a linear transformation.

With that, the output of the neural network for a given input ρ^{in} is given by

$$\rho^{\text{out}} = \mathcal{E}^{\text{out}}\left(\mathcal{E}^L\left(\ldots\mathcal{E}^2\left(\mathcal{E}^1\left(\rho^{\text{in}}\right)\right)\ldots\right)\right) \tag{4.2}$$

Just as the calculation of partial derivatives in classical backpropagation was dependent on the neuron's activations in each layer (see Equation 2.12), quantum backpropagation will also rely on the perceptron state in every layer. We must therefore include the storage of these states in our feed-forward algorithm to enable training. The final feed-forward algorithm is shown as `Algorithm 3`.

`Algorithm 3`: Quantum feed-forward.

For each element $\left[\,|\phi_x^{in}\rangle\,,\,|\phi_x^{out}\rangle\,\right]$ in trainingData do:

1. Calculate the network input $\rho_x^{in} = |\phi_x^{in}\rangle\langle\phi_x^{in}|$
2. For every layer l in qnnArch do:
   a. Apply the channel $\mathcal{E}^l$ to the output of the previous layer $l-1$
   b. Store the result ρ_x^l

Furthermore, as it will also be useful in training, we already formulate the *adjoint layer channel*, denoted $\mathcal{F}^l$:

$$\begin{aligned}\mathcal{F}^l(X^l) &= \mathrm{tr}_l\left(\left(\mathbb{1}_{l-1}\otimes|0\ldots0\rangle_l\langle 0\ldots0|\right)\prod_{j=1}^{m_l} U_j^{l\,\dagger}\left(\mathbb{1}_{l-1}\otimes X^l\right)\prod_{j=m_l}^{1} U_j^l\right)\\ &= \mathrm{tr}_l\left(\left(\mathbb{1}_{l-1}\otimes|0\ldots0\rangle_l\langle 0\ldots0|\right)U^{l\,\dagger}\left(\mathbb{1}_{l-1}\otimes X^l\right)U^l\right)\end{aligned} \tag{4.3}$$

4.2 Quantum Feed-Forward Neural Networks: Training

As was already indicated in `Algorithm 3`, in the quantum case the training data is given as the set $\{|\psi^{\text{in}_x}\rangle\,,|\psi_x^{\text{out}}\rangle\}$. As the cost function, we choose the so called *fidelity* between the output of the quantum neural network ρ^{out}, determined by Equation 4.2

for $\rho^{\text{in}} = |\psi^{\text{in}}\rangle \langle\psi^{\text{in}}|$, and the desired output $|\psi^{\text{out}}\rangle$, again averaged over all training pairs:

$$C = \frac{1}{N} \sum_{x=1}^{N} \langle \psi_x^{\text{out}}| \, \rho_x^{\text{out}} \, |\phi_x^{\text{out}}\rangle \tag{4.4}$$

This cost function takes on values between 0 and 1. Whereas previously 0 was the optimal – the best – value, with this definition it is now 1. Accordingly, we now aim to maximize the cost function.

To capture the progress of the training, we introduce a corresponding parameter s. Unitaries and layer channels may now be denoted $U_j^l(s)$ or $\mathcal{E}_s^l$ respectively. The training process is detailed in `Algorithm 4` (where `trainingRounds` $\in \mathbb{N}$ is an argument given to determine when to terminate the training algorithm). The essential equation of updating unitaries is given by

$$U_j^l(s+\epsilon) = e^{i\epsilon K_j^l(s)} \, U_j^l(s) \tag{4.5}$$

where ϵ is the chosen step size.

	`Algorithm 4`: Quantum training.
1.	**Feed-forward:** Perform feed-forward algorithm (`Algorithm 3`).
2.	**Updating:** Update every unitary of every layer via $U_j^l \to e^{i\,\epsilon\,K_j^l} \, U_j^l$.
X.	**Termination:** Repeat Steps 1 and 2 `trainingRounds` times.

The update matrix $K_j^l(s)$ is calculated as follows:

$$K_j^l(s) = \frac{2^{n_{a_1,\ldots,\beta}} \, i}{2N\lambda} \sum_x \text{tr}_{\text{rest}} M_j^l(s) \tag{4.6}$$

Here $\eta = \frac{1}{\lambda}$ is a chosen parameter called the *learning rate*. Moreover $M_j^l(s)$ is defined by

$$M_j^l(s) = \big[A_j^l(s), B_j^l(s)\big] \tag{4.7}$$

with

$$A_j^l(s) = U_j^l(s) \ldots U_1^l(s) \left(\rho_x^{l-1}(s) \otimes |0\ldots0\rangle_l \langle 0\ldots0| \right) U_1^l(s)^\dagger \ldots U_j^l(s)^\dagger \tag{4.8a}$$

$$B_j^l(s) = U_{j+1}^l(s)^\dagger \ldots U_{m_l}^l(s)^\dagger \left(\mathbb{1}_l \otimes \sigma_x^l(s)\right) U_{m_l}^l(s) \ldots U_{j+1}^l(s) \tag{4.8b}$$

and

$$\sigma_x^l(s) = \mathcal{F}_s^{l+1}\big(\ldots \mathcal{F}_s^{out}\big(|\phi_x^{out}\rangle \langle \phi_x^{out}| \big) \ldots \big) \tag{4.9}$$

For a detailed derivation of these formulas used in training, we refer to the original paper [Bee+20].

5 Classical Simulation of Quantum Machine Learning

5.1 On the Simulation of Quantumcomputational Procedures

While the potentially crucial advantages of quantum computation have been known for quite some time – and intense interest in the realisation of such computers had developed as a result of it – efforts to build respective machinery have only resulted in modest success until very recently. And in despite of Google's demonstration of quantum supremacy [Aru+19], which assuredly marks a milestone in the development of quantum technology, widespread attainable access to these computers for researchers still fails to be available. The good news is that with a complete mathematical description, as established in the last chapter, one can use numerical methods to simulate these quantum computations. The bad news is that with the linear growth of an n-part composite qubit system, the size of its Hilbert space $\mathcal{H} = (\mathbb{C}^2)^{\otimes n} = \mathbb{C}^{2^n}$ grows exponentially, therefore making it infeasible to calculate results for systems bigger than a handful of qubits. "It turns out that while an ordinary computer can be used to simulate a quantum computer, it appears to be impossible to perform the simulation in an efficient fashion." [NC10]

With these limitations in mind, the numerical results obtained nevertheless remain of essential value. The simulations written in python as a pivotal part of this thesis' work were conceived based on preexisting code in Mathematica produced for [Bee+20].

In them, we not only showed that efficient training of deep quantum feed-forward neural networks is possible, but furthermore for larger networks we repeatedly achieve better performance than the Mathematica code when comparing runtimes on the same machines. Additionally, we reproduce the findings of [Bee+20] showing remarkable capabilities of these networks to learn unitaries, even when given training set which primarily consists of noisy data.

In the rest of this chapter, we will take but a mere glance at the most central parts of this simulating code. For a better understanding – as well as concrete depictions of the just stated results – we encourage to examine the well-documented, full version of it, which is available at `https://github.com/MarvinSchwi/qmlthesis`.

(Also note that, in an effort to improve readability, we reiterate already introduced formulae when presenting the corresponding code.)

5.2 The Simulation of Quantum Feed-forward Neural Networks

For many of the quantum mechanical computations the simulation inherently uses the open source package `qutip`, some nuances of which are explained in more detail in the full version of the code.

In the following code, the parameter `qnnArch` describes the architecture of a quantum neural network. It is expected to be a 1-dimensional list of natural numbers which refer to the number of perceptrons in the corresponding layer. E.g. a 2-3-2 network would be given by qnnArch = [2, 3, 2]. Any parameter `unitaries` is expected to be a 2-dimensional list of the networks perceptron unitaries given in a tensored state like `unitaries`$[l][j] = U^l_{j+1}$. (The plus one stems from the fact, that python, like almost all common programming languages, use zero-indexing, which happens to not be used by us in our denation of perceptron unitaries.) `trainingData` is a list of training pairs such that its x-th element `trainingData`$[x]$ is given by $(|\psi_x^{\text{in}}\rangle, |\psi_x^{\text{out}}\rangle)$. Whenever these parameters are encountered in the following code, the algorithms assume to be given them in the just described way.

Feed-forward and affiliated algorithms

The function `makeLayerChannel` calculates the layer channel (Equation 4.1)

$$\mathcal{E}^l_s(X^{l-1}) = \text{tr}_{l-1}\Big(U^l_{m_l}(s)\dots U^l_1(s)\,\big(X^{l-1}\otimes|0\dots0\rangle_l\langle0\dots0|\big)\,U^l_1(s)^\dagger\dots U^l_{m_l}(s)^\dagger\Big)$$

for `inputState` $= X^{l-1}$.

```
def makeLayerChannel(qnnArch, unitaries, l, inputState):
    numInputQubits = qnnArch[l-1];
    numOutputQubits = qnnArch[l];

    #Tensor input state
    state = qt.tensor(inputState, tensoredQubit0(numOutputQubits));

    #Calculate layer unitary
    layerUni = unitaries[l][0].copy();
    for i in range(1, numOutputQubits):
        layerUni = unitaries[l][i] * layerUni;

    #Multiply and tensor out input state
    return partialTraceRem(layerUni * state * layerUni.dag(),
    ↪ list(range(numInputQubits)));
```

`makeAdjointLayerChannel` calculates the adjoint layer channel (Equation 4.3):

$$\mathcal{F}^l_s(X^l) = \text{tr}_l\Big(\big(\mathbb{1}_{l-1}\otimes|0\dots0\rangle_l\langle0\dots0|\big)\,U^l_1(s)^\dagger\dots U^l_{m_l}(s)^\dagger\,\big(\mathbb{1}_{l-1}\otimes X^l\big)\,U^l_{m_l}(s)\dots U^l_1(s)\Big)$$

for `inputState` $= X^l$. (This function will actually not be needed until the training algorithms are presented.)

```
def makeAdjointLayerChannel(qnnArch, unitaries, l, outputState):
    numInputQubits = qnnArch[l-1];
    numOutputQubits = qnnArch[l];

    #Prepare needed states
    inputId = tensoredId(numInputQubits);
    state1 = qt.tensor(inputId, tensoredQubit0(numOutputQubits));
    state2 = qt.tensor(inputId, outputState);

    #Calculate layer unitary
    layerUni = unitaries[l][0].copy();
    for i in range(1, numOutputQubits):
        layerUni = unitaries[l][i] * layerUni;

    #Multiply and tensor out output state
    return partialTraceKeep(state1 * layerUni.dag() * state2 *
    ↪ layerUni, list(range(numInputQubits)) );
```

The `feedforward` function carries out the feed-forward process, as described in `Algorithm 3`, making use of the previously defined `makeLayerChannel` function to perform step 2.a.

`Algorithm 3`: Quantum feed-forward.

For each element $\left[\, |\phi_x^{in}\rangle, |\phi_x^{out}\rangle \,\right]$ in trainingData do:

1. Calculate the network input $\rho_x^{in} = |\phi_x^{in}\rangle \langle\phi_x^{in}|$
2. For every layer l in qnnArch do:
   a. Apply the channel $\mathcal{E}_s^l$ to the output of the previous layer $l-1$
   b. Store the result $\rho_x^l(s)$

```
def feedforward(qnnArch, unitaries, trainingData):
    storedStates = [];
    for x in range(len(trainingData)):
        currentState = trainingData[x][0] *
    ↪ trainingData[x][0].dag();
        layerwiseList = [currentState];
        for l in range(1, len(qnnArch)):
            currentState = makeLayerChannel(qnnArch, unitaries, l,
    ↪ currentState);
            layerwiseList.append(currentState);
        storedStates.append(layerwiseList);
    return storedStates;
```

Training algorithms

`costFunction` assumes to be given `trainingData` and `outputStates` so that

$$\texttt{trainingData}[x][1] = |\phi_x^{out}\rangle$$

$$\texttt{outputStates}[x] = \rho_x^{out}(s)$$

and computes the cost function (as previously introduced in Equation 4.4):

$$C(s) = \frac{1}{N}\sum_{x=1}^{N} \langle \phi_x^{out}| \, \rho_x^{out}(s) \, |\phi_x^{out}\rangle$$

```
def costFunction(trainingData, outputStates):
    costSum = 0;
    for i in range(len(trainingData)):
        costSum += trainingData[i][1].dag() * outputStates[i] *
    ↪ trainingData[i][1];
    return costSum.tr()/len(trainingData);
```

`makeUpdateMatrix` assumes to be given the parameters λ, $\epsilon \in \mathbb{R}$, and calculates the update matrix

$$\exp\bigl(i\,\epsilon\,K_j^l(s)\bigr)$$

for the j-th perceptron of the l-th layer with

$$K_j^l(s) = \frac{2^{n_{a_1,\ldots,\beta}}\, i}{2N\lambda}\sum_x \mathrm{tr}_{\mathrm{rest}} M_j^l(s)$$

and

$$M_j^l(s) = \bigl[A_j^l(s), B_j^l(s)\bigr]$$

making use of the subsequently introduced function `updateMatrixFirstPart` and `updateMatrixSecondPart`.

`makeUpdateMatrixTensored` tensors the calculated as above update matrix in such a way, that it can be applied to the already tensored perceptron unitaries.

```
def makeUpdateMatrix(qnnArch, unitaries, trainingData,
    ↪ storedStates, lda, ep, l, j):
    numInputQubits = qnnArch[l-1];

    #Calculate the sum:
    summ = 0;
    for x in range(len(trainingData)):
        #Calculate the commutator
        firstPart = updateMatrixFirstPart(qnnArch, unitaries,
    ↪ storedStates, l, j, x);
        secondPart = updateMatrixSecondPart(qnnArch, unitaries,
    ↪ trainingData, l, j, x);
        mat = qt.commutator(firstPart, secondPart);

        #Trace out the rest
        keep = list(range(numInputQubits));
        keep.append(numInputQubits + j);
        mat = partialTraceKeep(mat, keep);

```

```
        #Add to sum
        summ = summ + mat;

    #Calculate the update matrix from the sum
    summ = (-ep * (2**numInputQubits)/(lda*len(trainingData))) *
    ↪ summ;
    return summ.expm();
```

```
def makeUpdateMatrixTensored(qnnArch, unitaries, lda, ep,
    ↪ trainingData, storedStates, l, j):
    numInputQubits = qnnArch[l-1];
    numOutputQubits = qnnArch[l];

    res = qt.tensor(makeUpdateMatrix(qnnArch, unitaries, lda, ep,
    ↪ trainingData, storedStates, l, j),
    ↪ tensoredId(numOutputQubits-1));

    return swappedOp(res, numInputQubits, numInputQubits + j);
```

`updateMatrixFirstPart` calculates $A_j^l(s)$ according to

$$A_j^l(s) = U_j^l(s) \dots U_1^l(s) \left(\rho_x^{l-1}(s) \otimes |0 \dots 0\rangle_l \langle 0 \dots 0| \right) U_1^l(s)^\dagger \dots U_j^l(s)^\dagger$$

```
def updateMatrixFirstPart(qnnArch, unitaries, storedStates, l, j,
    ↪ x):
    numInputQubits = qnnArch[l-1];
    numOutputQubits = qnnArch[l];

    #Tensor input state
    state = qt.tensor(storedStates[x][l-1],
    ↪ tensoredQubit0(numOutputQubits));

    #Calculate needed product unitary
    productUni = unitaries[l][0];
    for i in range(1, j+1):
        productUni = unitaries[l][i] * productUni;

    #Multiply
    return productUni * state * productUni.dag();
```

`updateMatrixSecondPart` calculates $B_j^l(s)$ according to

$$B_j^l(s) = U_{j+1}^l(s)^\dagger \dots U_{m_l}^l(s)^\dagger \left(\mathbb{1}_l \otimes \sigma_x^l(s)\right) U_{m_l}^l(s) \dots U_{j+1}^l(s)$$

with

$$\sigma_x^l(s) = \mathcal{F}_s^{l+1}\left(\dots \mathcal{F}_s^{out}\left(|\phi_x^{out}\rangle \langle \phi_x^{out}| \right) \dots \right)$$

```
def updateMatrixSecondPart(qnnArch, unitaries, trainingData, l, j,
    ↪ x):
    numInputQubits = qnnArch[l-1];
    numOutputQubits = qnnArch[l];

    #Calculate sigma state
    state = trainingData[x][1] * trainingData[x][1].dag();
    for i in range(len(qnnArch)-1,l,-1):
        state = makeAdjointLayerChannel(qnnArch, unitaries, i,
    ↪ state);
    #Tensor sigma state
    state = qt.tensor(tensoredId(numInputQubits), state);

    #Calculate needed product unitary
    productUni = tensoredId(numInputQubits + numOutputQubits);
    for i in range(j+1, numOutputQubits):
        productUni = unitaries[l][i] * productUni;

    #Multiply
    return productUni.dag() * state * productUni;
```

`qnnTraining` assumes to be given λ, $\epsilon \in \mathbb{R}$ and `trainingRounds` $\in \mathbb{N}$, and trains the given quantum neural network as detailed in `Algorithm 4`.

`Algorithm 4`: Quantum training.

1. **Feed-forward:** Perform feed-forward algorithm (`Algorithm 3`).
2. **Updating:** Update every unitary of every layer via $U_j^l \to e^{i\,\epsilon\,K_j^l}\,U_j^l$.

X. **Termination:** Repeat Steps 1 and 2 `trainingRounds` times.

```
def qnnTraining(qnnArch, initialUnitaries, trainingData, lda, ep,
    ↪ trainingRounds, alert=0):

    ### FEEDFORWARD
    #Feedforward for given unitaries
    s = 0;
    currentUnitaries = initialUnitaries;
    storedStates = feedforward(qnnArch, currentUnitaries,
    ↪ trainingData);

    #Cost calculation for given unitaries
    outputStates = list(sc.array(storedStates)[:,-1]);
    plotlist = [[s], [costFunction(trainingData, outputStates)]];

    #Optional
    runtime = time();

    #Training of the Quantum Neural Network
    for k in range(trainingRounds):
        if alert>0 and k%alert==0: print("In training round
    ↪ "+str(k))
```

```
19
20          ### UPDATING
21          newUnitaries = unitariesCopy(currentUnitaries);
22
23          #Loop over layers:
24          for l in range(1, len(qnnArch)):
25              numInputQubits = qnnArch[l-1];
26              numOutputQubits = qnnArch[l];
27
28              #Loop over perceptrons
29              for j in range(numOutputQubits):
30                  newUnitaries[l][j] =
    ↪ (makeUpdateMatrixTensored(qnnArch, currentUnitaries,
    ↪ trainingData, storedStates, lda, ep, l, j) *
    ↪ currentUnitaries[l][j]);
31
32          ### FEEDFORWARD
33          #Feedforward for given unitaries
34          s = s + ep;
35          currentUnitaries = newUnitaries;
36          storedStates = feedforward(qnnArch, currentUnitaries,
    ↪ trainingData);
37
38          #Cost calculation for given unitaries
39          outputStates = list(sc.array(storedStates)[:,-1]);
40          plotlist[0].append(s);
41          plotlist[1].append(costFunction(trainingData,
    ↪ outputStates));
42
43      #Optional
44      runtime = time() - runtime;
45      print("Trained "+str(trainingRounds)+" rounds for a
    ↪ "+str(qnnArch)+" network and "+str(len(trainingData))+"
    ↪ training pairs in "+str(round(runtime, 2))+" seconds");
46
47      #Return
48      return [plotlist, currentUnitaries];
```

6 Conclusion

In this thesis, we provided an elaborate walkthrough of the fields supervised machine learning and classical feed-forward neural networks and – following a quick recapitulation of fundamental quantum mechanics – then explained how these concepts can be applied to accommodate and make use of future quantum computational possibilities. We produced an extensive simulation of these deep networks, proving their capabilities to be effectively trained and resisting problems posed by flawed or noisy training data. As both machine learning and quantum computation can be expected to be of enormous technological importance well into the future, their combination surely won't be an exception.

One might be inclined to initially presume that not much work remains to be done in the near future – after all, we just thoroughly described a quantum neural network capable of universal quantum computation with corresponding, efficient training algorithms.

However, that could not be further from the truth.

For starters, there are many optimizations left to incorporate in the existing, presented code. Even though no optimization could possibly overcome the exponential growth in required computational power to an extend which makes *efficient* simulation of quantum neural networks possible, continuous advancements will likely result in the ability to simulation larger networks and will thus – in the absence of attainable access to actual quantum computers – plausibly help and speed up further research in the field. A more comprehensive search for optimization-potential hidden in intricacies of the python programming language seems like an appropriate starting point. Furthermore from a more high-level, language-independent standpoint, there remains obvious room for improvements in the general design of the simulating algorithms. E.g. the calculation of the layer unitary by multiplying perceptron unitaries – which is currently redone in the feedforward algorithm for every single training pair – is doubtlessly unnecessary. Moreover, if one would store all intermediate results when calculating the layer unitary, giving an array of these partially multiplied unitaries to the training algorithms would further save a considerable amount of computational steps.

In addition to code optimization tasks and in a more general sense – and, likely, also more captivating – many areas of quantum machine learning remain little explored or even untouched. Within supervised machine learning, many other network structures, such as recurrent neural networks, exist, awaiting translation to quantum counterparts. Furhtermore, whole subfields of machine learning outside supervised machine learning anticipate exploration.

With those remarks, we conclude this thesis, facing an intriguing and encourag-

ing future in quantum machine learning whose further possibilities remain to be explored.

Bibliography

[ABG06] Esma Aimeur, Gilles Brassard, and Sebastien Gambs. "Machine learning in a quantum world". In: *Conference of the Canadian Society for Computational Studies of Intelligence*. Springer. 2006, pp. 431–442.

[Aru+19] Frank Arute et al. "Quantum supremacy using a programmable superconducting processor". In: *Nature* 574.7779 (2019), pp. 505–510.

[Bee+20] Kerstin Beer et al. "Training deep quantum neural networks". In: *Nature Communications* 11.1 (2020), pp. 1–6.

[Cyb89] George Cybenko. "Approximation by superpositions of a sigmoidal function". In: *Mathematics of control, signals and systems* 2.4 (1989), pp. 303–314.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL: `https://www.deeplearningbook.org/`.

[Her09] Suzana Herculano-Houzel. "The human brain in numbers: a linearly scaled-up primate brain". In: *Frontiers in human neuroscience* 3 (2009), p. 31.

[Hor91] Kurt Hornik. "Approximation capabilities of multilayer feedforward networks". In: *Neural networks* 4.2 (1991), pp. 251–257.

[Hor93] Kurt Hornik. "Some new results on neural network approximation". In: *Neural networks* 6.8 (1993), pp. 1069–1072.

[HSW89] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural networks* 2.5 (1989), pp. 359–366.

[HSW90] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks". In: *Neural networks* 3.5 (1990), pp. 551–560.

[Les+93] Moshe Leshno et al. "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function". In: *Neural networks* 6.6 (1993), pp. 861–867.

[Meh+19] Pankaj Mehta et al. "A high-bias, low-variance introduction to machine learning for physicists". In: *Physics Reports* 810 (2019).

[NC10] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. 10th Anniversary Edition. Cambridge University Press, 2010.

[Nie15] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015. URL: `http://neuralnetworksanddeeplearning.com`.

[RZL17] Prajit Ramachandran, Barret Zoph, and Quoc V Le. "Searching for Activation Functions". In: *arXiv preprint arXiv:1710.05941* (2017). URL: `https://arxiv.org/abs/1710.05941`.

Declaration

Herewith I declare that this thesis is the result of my independent work and that no sources, auxiliary materials or means other than those indicated were used. Furthermore, all passages of the work that make reference to other sources, whether through direct quotation or paraphrasing, have been indicated accordingly. This paper has not previously been submitted to an examining authority in the same or a similar form.

Place, Date: *Signature:*