Quantum Neural Networks with General Output

by

Nils Gerhard Renziehausen

Student Number: 10019578

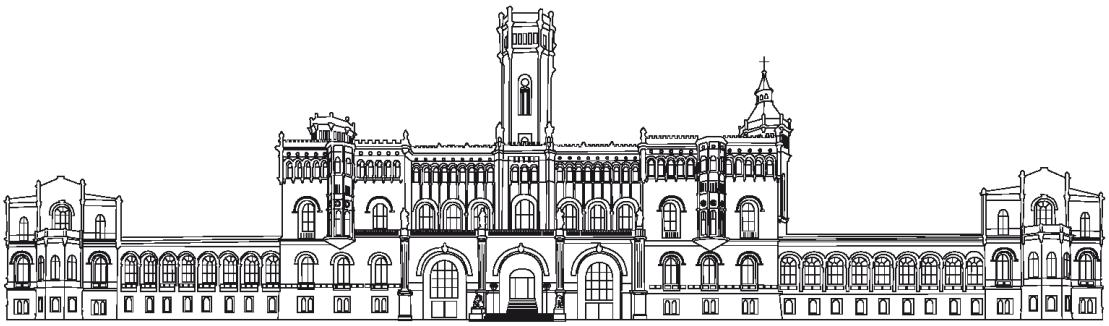
Supervised by

Prof. Dr. Tobias J Osborne

Viktoria Sophie Schmiesing

Bachelor Thesis

B.Sc. Physics



Gottfried Wilhelm Leibniz Universität Hannover

Institut für Theoretische Physik

Hannover

May 29, 2022

Erklärung

Hiermit versichere ich,

Vorname, Name: Renziehausen, Nils Gerhard ,

dass ich die anliegende Arbeit

Studienfach, Prüfer/in : Physik, Tobias J. Osborne

Titel der Arbeit : Quantum Neural Networks with General Output

selbst angefertigt und alle für die Arbeit verwendeten Quellen und Hilfsmittel in der Arbeit vollständig angegeben habe.

Ich habe die beigefügte Arbeit noch nicht zum Erwerb eines anderen Leistungsnachweises eingereicht.

Mit der Übermittlung meiner Arbeit auch an externe Dienste zur Plagiatsprüfung durch Plagiatssoftware erkläre ich mich einverstanden (bitte unten ankreuzen).

☒ ja

[ ] nein

Hannover, 29.05.2022
Ort, Datum

Nils Renziehausen
Unterschrift

Contents

1 Introduction

The terms *machine learning*, *neural networks*, and *artificial intelligence* are circulating in the media, business contexts, and also in research. In recent years, the most influential papers by citation in science all reported on machine learning techniques [21],[15],[8]. Their success arises from the wide variety of application domains for machine learning, ranging from science and technology [42] to everyday life [40].

Likewise, quantum computing has enjoyed great attention since Feynman [12] and Manin [24] showed that a quantum mechanical model of a computer [5] can solve problems not efficiently solvable in reasonable amounts of time for classical computers[27]. More current developments proved that these are not merely theoretical ideas but practical opportunities [13][39]. Promising new capabilities naturally attract interest from various fields[2], including cryptography[14], finance[32], biology[33], and machine learning[1].

Reciprocal applications of machine learning and quantum computing are manifold. Machine learning is applied to solve quantum problems [6][17]. Quantum computers can be deployed to enhance machine learning algorithms for classical problems [19], and finally, quantum computers can utilise machine learning techniques to tackle quantum problems [10]. Quantum neural networks capable of performing general quantum computations [4] form the basis of this thesis.

Beer et al. [4] developed an approach for a quantum neural network analogous to a classical deep neural network with a backpropagation learning algorithm. This thesis will generalise the present approach for mixed output states and present simulation results for learning several quantum channels.

Simulation results for a generalised version of the aforementioned learning algorithm will be presented in this thesis. Based on introductions to classical machine learning and artificial neural networks in section 2 and quantum computing in section 3, this thesis discusses quantum neural networks in section 4. Section 2 discusses artificial neurons, neural networks, and a supervised machine learning algorithm with backpropagation, as well as applications and limitations of machine learning techniques. Section 3 introduces quantum computing by discussing qubits, systems of qubits, quantum algorithms and mathematical tools to describe them. Based on that, section 4 elaborates on a quantum machine learning algorithm and the changes made to generalise it. Putting these theoretical results into practice, section 5 presents simulation results for the generalised algorithm.

2 Classical Machine Learning and Neural Networks

This section will begin with some conceptual clarifications. Subsequently, this section will show how machine learning can be utilised to teach a neural network a task. Finally, applications and limitations of neural networks and machine learning will be discussed.

Artificial intelligence is a term that broadly describes the intelligent behaviour of artificial systems. It is difficult to define this term further since there is not even a broadly accepted definition of intelligence. The term artificial intelligence will therefore not be used again in this thesis.

The term *Machine learning* is a little more precise. It describes the ability of artificial systems to learn from examples and generalize this experience. It often describes different statistical methods that allow learning from training data.
This section will focus on *supervised machine learning*, a frequently used technique. Training data containing the correct or desired outputs for an input provides *supervision* with this technique. In practice, this presents itself, for instance, as a set of images labeled with their contents.

Neural networks are one of the most frequently used methods for machine learning. They are networks of artificial neurons that are linked with each other and transform input data to output data.

2.1 The Neuron

A neuron is an element of a neural network. In such networks, neurons are linked to each other and can send and receive information to and from other neurons. One can therefore imagine a neuron as some kind of dot linked to other dots. The concept of neural networks dates back to the 1910s, when Llinas [23] discovered the synapsis in the human brain. Building on that, mathematicians proved several important abilities of theoretical neural networks, such as their ability to solve arithmetic problems [26].

This section is dedicated to a special type of neuron, the *Perceptron*, first introduced by Rosenblatt et. al in 1958 [37]. A perceptron takes input values $x_i \in \{0, 1\}$, multiplies them with factors called *weights* w_i and has a *bias* b (See fig. 1). The output o is either 1 or 0, depending on whether the sum of products of inputs and weights and the bias is larger or smaller than 0.

$$\mathrm{o} = \begin{cases} 1, \text{if } \sum_{\text{inputs } i} x_i w_i + b > \text{threshold} \\ 0, \text{else} \end{cases} \tag{1}$$

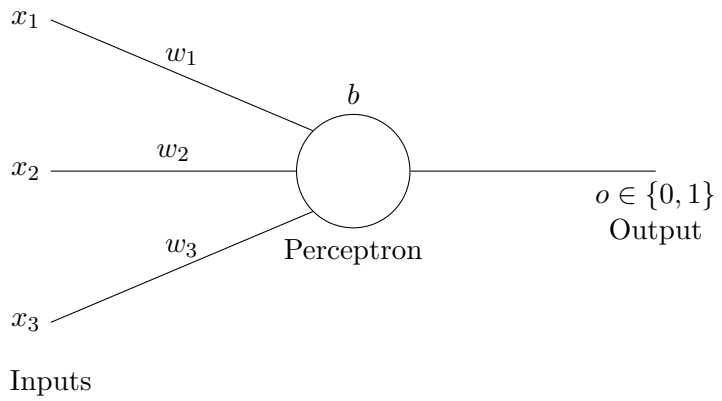


Figure 1: Inputs x_i into a perceptron with weights w_i and bias b.

For later applications it is necessary for the output function to be differentiable. One therefore chooses a continuous output function $o = \varphi(z)$ with $z = \sum_{\text{inputs } i} x_i w_i + b$. Functions φ can be defined in a way that they almost behave like (1) with the important distinction that φ can take on values from 0 to 1.

This quite abstract model is able to solve very practical problems. A common example is an everyday decision. Take, for instance, Alice, who owns a perceptron and faces the difficult question of whether to move to Hanover or not. Every factor influencing this decision is represented by one binary input. Such factors can be, for instance, friends of her living there (yes $\Rightarrow x_1 = 1$), whether she could afford an adequate apartment (yes $\Rightarrow x_2 = 1$) and the distance to her birthplace, which is too long ($\Rightarrow x_3 = 0$). Each of these inputs to her perceptron is weighted by a weight, 4 for the friends input, 3 for the apartment input and 1 for the distance input. The weights indicate the relative importance of the factors. Furthermore, the perceptron has a bias of -4 because Alice does not like moving. The threshold in this example is 0. Alice's perceptron now calculates the result:

$$\sum_{\text{inputs } i} x_i w_i + b = 1 \cdot 4 + 1 \cdot 3 + 0 \cdot 0 - 4 = 3 > 0 \tag{2}$$

As a result, her perceptron gives a 1 according to (1), so Alice will move to Hanover. For other types of problems, one may alter the binary output of the perceptron to a continuous output.

A single perceptron is still very limited despite its ability to also solve some trickier linear problems by using continuous output functions. For instance, it cannot simulate a logical *exclusive or* operation, as pointed out by Minsky and Papert [25]. The following section will detail some improvements to the single perceptron.

2.2 Networks

The simple, single perceptron has its limitations. Some of these can be circumvented by connecting several perceptrons to a network. It makes sense to introduce the *layer* at this point. A layer is a level of the network. The first layer (counted as 0th layer) is the input layer, consisting of all the perceptrons that receive the input data. All the perceptrons receiving the outputs of the input layer are in the first layer and so on until one reaches the output layer. Every layer that is neither input nor output layer is called a *hidden* layer (see fig. 2). Networks with several hidden layers are called *deep* neural networks. The jth neuron in the lth layer will be represented as neuron$_j^l$, and its weights and biases will be represented in the same fashion. Functions such as (1) are called activation functions. There are a lot of different possible activation functions, but the principle remains the same. The *network architecture* summarises the structure of the network: How each perceptron is connected to other ones, the number of layers, the number of perceptrons per layer etc. *Feedforward* is the data transformation process of the network. Input data is fed to the input layer of the network, the output of the first layer is calculated and is in turn fed into the first hidden layer as input and so on until one obtains the output of the output layer.

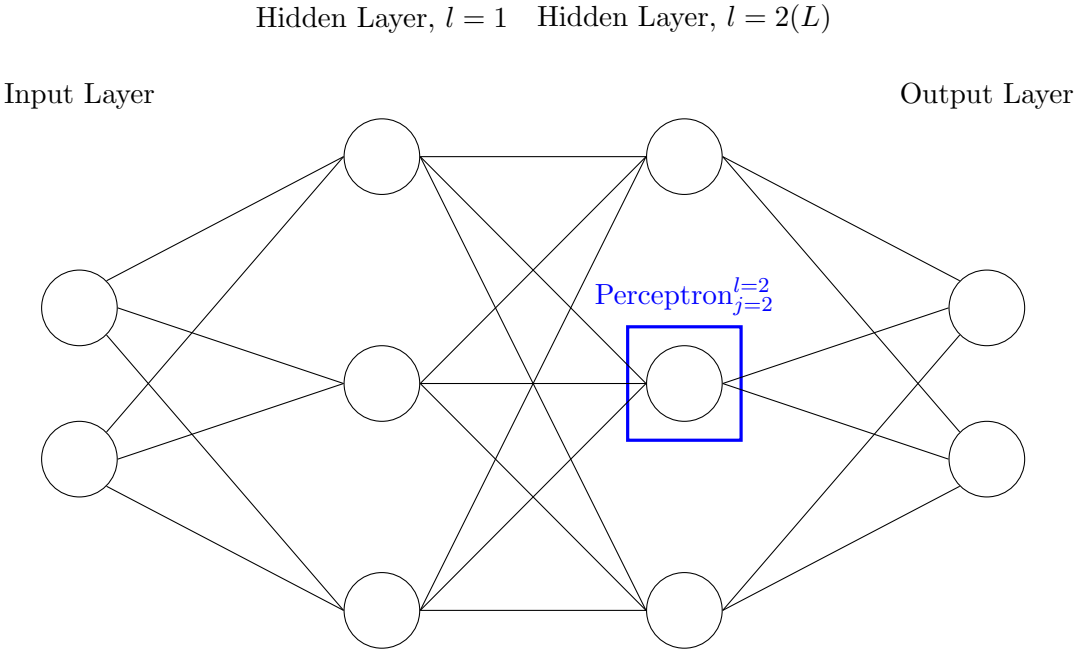


Figure 2: A 2-3-3-2 network: An input layer with two perceptrons, two hidden layers with three perceptrons each and an output layer with two perceptrons. The last hidden layer is often referred to as layer L.

Deep neural networks, trained with machine learning algorithms, are able to solve much more complicated problems, most importantly, problems that are difficult or impossible to

solve with classical algorithms. A famous example of the ability to solve problems that are difficult to address by algorithms is the detection of cats in images. The input, in this case, is the pixels of the image, for instance, via the grey value (from 0 = white to 1 = black) for black and white images. A glance at the datasheet of a modern camera reveals that such networks need a lot (several million) of input perceptrons, one for each pixel. Via a few hidden layers such networks can identify cats with great precision.

2.3 Learning a Task

So far, only the networks themselves have been covered by this thesis. After selecting a network architecture - the number of layers and the number of perceptrons in each layer - weights and biases must be assigned to each perceptron. A training algorithm transforms these weights and biases to improve the network's performance on a task. In the following, such algorithms, machine learning algorithms, will be discussed.

In the beginning, all the weights and biases of the network are selected randomly. Furthermore, there is some training data. Training data are sets of training pairs $\{x_i, a_i\}$. A training pair consists of some input x (e.g. an image) and the desired output for the input a (e.g. whether the image shows a cat or not).

As the first step, one needs a way to compare the desired outputs from the training data to the actual outputs of the network. This comparison is the task of the *cost function*. Since the cost function depends on the output, which depends on all the perceptrons' weights and biases in the network, the cost function itself depends on the weights and biases. In the example of images of cats, the output might be a number between 0 and 1 which translates into a percentage. This percentage is the network's confidence that an image shows a cat. A training pair with an image of a cat has a 1 as desired output, and a training pair with another image, a dog for instance, has a 0 as the desired output. For each training pair in the training data of N pairs, the cost function averages the squared difference between desired output a_i and network output o_i for the input x_i:

$$C(w, b) = \frac{1}{N} \sum_{i=1}^{N} ||a_i - o_i||^2 \tag{3}$$

This type of cost function is called a *least square cost function*, and there are other possible types of cost functions that will not be discussed in this thesis. Also, note that some authors (e.g. [36]) refer to the cost function as *loss function* or *empirical risk*.

The goal is to find a network with a minimal cost function. In the example of pictures of cats, a network associated with a zero cost function correctly identifies every image from the training data correctly. This is hardly the case for the random weights and biases before the training. Therefore, one needs an algorithm that changes the weights and biases of the network to reduce the cost function: A supervised machine learning algorithm.

The cost function was introduced as a function depending on the weights and biases of the network. A naive approach to finding the optimal weights and biases is to take the derivative of the cost function with respect to every weight and every bias. For a function with just one parameter $f(x_1)$, the derivation of the function with respect to the parameter $\frac{df(x_1)}{dx_1}$ becomes 0 at the extreme point of the function. For a minimum, the second derivative of the function is positive. An intuition for this is a slope. At the extreme points of $f(x_1)$, there are low points (or high points). This intuition can be generalised for a function depending on two parameters $f(x_1, x_2)$ for which the low points are valleys in a two-dimensional plane. The goal of finding the lowest point of a multi-dimensional plane remains the same.

Calculating derivatives of the cost function for many perceptrons and thus many more weights and biases and performing the calculus to find the minima is computationally too exhaustive [31]. A technique that allows coming very close to the minima is called *gradient descent*. The gradient of a function $f(x)$ at any point x is a vector that points in the direction of the steepest ascent from this point. Going a small step in the opposite direction thus decreases the function fastest. After that step, one has to calculate the gradient at the new point and again move a small step to the opposite direction of the gradient. There is no general rule for the size of the steps. Therefore, one must experiment with it a little: With a large stepsize, one might accidentally jump over a minimum. Conversely, many steps are required to reach the minimum with a small step size. In this section, the stepsize will be referred to as η.

The gradient is composed of the partial derivatives of the cost function with respect to all the weights and biases:

$$\frac{\partial C}{\partial w^l_{jk}} \quad \text{and} \quad \frac{\partial C}{\partial b^l_j}. \tag{4}$$

The double index for the weights w_{jk} indicates the weight for outputs from the kth perceptron in layer $l-1$ to the jth perceptron in the lth layer. The output of a perceptron is

$$o^l_j = \varphi\Big(\sum_{\text{inputs } k} x^l_{jk} w^l_{jk} + b^l_j\Big) = \varphi(z^l_j) \tag{5}$$

with the *weighted input* z^l_j. *Backpropagation* is an algorithm to compute the gradient of the cost function. The change of the cost function depending on the weighted input z^l_j is referred to as *error* [38]:

$$\delta^l_j = \frac{\partial C}{\partial z^l_j}. \tag{6}$$

The backpropagation algorithm begins at the output layer L. With the cost function (3) depending on the output of the network, the error for the output layer is given as [31]:

$$\delta^L = \frac{\partial C}{\partial o_j^L} \varphi'(z_j^L). \tag{7}$$

It is useful to amalgamate the errors of each perceptron in one layer into a vector δ^l for the whole layer. An omitted index j in the following indicates a vector constructed in a similar fashion. Omitting both indices j and k indicates the construction of a matrix with the entries for each index pair. In order to further trace back the error, one needs an expression for the error of a layer l depending on the error of the next layer. This expression is defined as[31]:

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \varphi'(z^l). \tag{8}$$

Expression (8) contains an arithmetic operator $\odot$ which stands for a component-wise multiplication, and the superscript T, which transposes the matrix w^{l+1}. Equations (7) and (8) allow to compute the error for each layer in the network. The rates of change of the cost function with respect to a weight or a bias in the network can be expressed as [31]:

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l \tag{9}$$

and

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l. \tag{10}$$

Combining expressions (7),(8),(9) and (10) allows to compute the wanted information in (4) and thus the gradient. Having calculated the gradient, one can now take a step $\Delta w_j^l = -\eta \frac{\partial C}{\partial w_j^l}$ in the opposite direction of the gradient for each component. For every step s, every weight now has to be updated according to $w_j^l(s+1) = w_j^l(s) + \Delta w_j^l(s)$. Similarly, all the biases must be updated. The whole procedure is called *backpropagation.*

In summary, the algorithm to minimise the cost function consists of four steps:

1.) Feedforward the input data.

2.) Calculate the cost function.

3.) Run the backpropagation algorithm.

4.) Repeat the previous steps until the cost function reaches its minimum.

With large sets of training data, one can avoid performing the algorithm for each training pair by averaging the cost function over a subset of training data. This method is called *statistical gradient descent.*

2.4 Applications

Modern neural networks and learning techniques achieve impressive results in solving real-world problems. Already today, there are very promising applications. Programs can not only detect almost anything in images but also create entirely new images and videos from text [34]. Most smartphones have some kind of voice assistant that translates audio recordings into text. Some neural networks can already create texts on given topics, compose music or even develop computer chips [29]. Neural networks will penetrate into many more areas of everyday life.

2.5 Limitations

With all the current and future applications, there are several inherent limitations. Additionally, there are functional and technological limitations.

Three inherent limitations will be discussed. First, the networks, by design, never give definite answers. In general, that is not a problem. No one gets hurt if a network identifies a cat with only 99% certainty. That is literally not the case if a similar network is to classify medical images and some condition is not correctly identified. The technology itself cannot solve this problem. The ethical implications have to be discussed and taken into account by the people using and relying on the technology.

The second limitation is the explainability of decisions. One may not worry too much about why a network confused a cat with a dog. However, if a bank uses a neural network to decide who receives a loan, this may have legal, social and ethical implications. It is, therefore, an ongoing debate about how many responsibilities can or should be delegated to neural networks and machines.

The third limitation, in particular, affects supervised machine learning techniques. The training data has to be generated by humans. Humans decide which answer is correct. Therefore, training data is subject to human biases and errors. Furthermore, the generation of training data is time-consuming.

Other limitations are of a more technical nature. The best neural networks are incredibly complex, with many layers and even more perceptrons. They are very deep in that sense. Enormous amounts of calculations and computer hours have to be invested in training and using such large networks. For complex tasks networks can bring the most powerful computers to their knees. It is an open question whether so many resources, computationally and energy-wise, should be devoted to machine learning techniques [30]. Larger and more capable networks, in general, need more computational power. That was not much of a concern in the past because Moore's Law promised an exponential growth in the number of available transistors for the same price. Due to physical limitations, this growth is expected to flatten in the future [41]. Another limitation is the amount of training data needed to

perform some tasks. In comparison with humans, this is still very slow [16]. Despite some clever ideas to circumvent those problems, like using analog technologies to more efficiently calculate multiplications [9], one problem remains: Sustainability. Every calculation on a computer consumes electricity, and as of today, most of the electricity generated globally is generated by burning fossil fuels [18].

3 Quantum Computing

This section will introduce quantum computers and quantum computing. Building on the qubit, the fundamental building block of quantum computers, this section will continue by introducing many qubit systems and, finally, quantum algorithms.

3.1 Qubits

Three components are necessary to build a computer: *bits*, *gates*, and *wiring*. The fundamental building block of most current computer systems is the bit. Its defining property is already included in its name: it is a binary digit [11]. A bit can take on two values like 0 or 1, and on or off. By wiring together several bits and including gates, elements that can change the state of a bit, one can build a computer that can perform general calculations.

In a similar fashion, quantum computers rely on quantum bits or *qubits* for short. A qubit is a state in a two-dimensional quantum mechanical system that, like a classical bit, has two different readings, 0 or 1. Every time a measurement is made on the system, the measurement outcome is either 0 or 1. The difference to a classical bit lies in its behaviour before a measurement occurs. Until measured, a qubit can also be in a superposition state of two states $|0\rangle$ and $|1\rangle$. This can be expressed as a state $|\Psi\rangle$, a linear combination of the two possible states after a measurement: $|\Psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$. α and β are complex numbers called the *amplitudes*. A measurement of the state $|\Psi\rangle$ yields 0 with a probability $|\alpha|^2$ and 1 with the probability $|\beta|^2$. Since one always either measures 0 or 1, there is a constraint for α and β: $|\alpha|^2 + |\beta|^2 = 1$. Mathematically, a qubit state is a complex wave function. Therefore, it has a global *phase* φ, a real number. The phase is not detectable because the measurement results for $|\Psi\rangle$ are equal to $e^{i\varphi}\,|\Psi\rangle$. Another use of the word phase is encountered in the relative phase. Two amplitudes α and β with $\alpha = e^{i\varphi}\beta$ differ by a relative phase φ [28].

3.1.1 Tensor Product

Quantum states, like the qubit, are vectors in a particular type of mathematical vector space, the *Hilbert* space. Every sort of quantum system is assigned to a Hilbert space. Every qubit, for instance, lives in its own Hilbert space. It is often necessary to combine quantum systems and therefore combine their Hilbert spaces. The tensor product of two vector spaces $\mathcal{H}_A$ and $\mathcal{H}_B$ is again a vector space $\mathcal{H}_A \otimes \mathcal{H}_B$. There is a map assigning a vector in the combined vector space to each pair of vectors from the initial vector spaces. A basis of the tensor product vector space can be formed from combinations of all the basis elements of the initial vector spaces.

3.1.2 Pure and Mixed States

So far, only pure states have appeared in this thesis. A *pure state* can be represented by a single vector in a Hilbert space. Such a vector can be denoted as a *ket* in the bra-ket

notation like $|\Psi\rangle$. *Mixed states* are ensembles of pure states. A probability p_i is assigned to each pure state constituting a mixed state. In quantum mechanics, mixed states are represented by density operators ρ, which can be constructed with pure states $|\Phi_i\rangle$, their dual states $\langle\Phi_i|$ and probabilities p_i via:

$$\rho = \sum_i p_i \, |\Phi_i\rangle \, \langle\Phi_i|. \tag{11}$$

Therefore, a pure state can also be represented by a density operator with probability $p_i = 1$ for the corresponding pure state in the ket representation.

3.1.3 Partial Trace

The expectation value for a measurement of an observable A for a quantum state ρ is given as:

$$tr(A\rho). \tag{12}$$

Quantum mechanics and quantum computing often deal with systems composed of several subsystems. Those subsystems might be two qubits A and B in $\mathcal{H}_\mathcal{A}$ and $\mathcal{H}_\mathcal{B}$, for instance. There is a Hilbert space for each subsystem, and the combined system is represented by a Hilbert space $\mathcal{H}_A \otimes \mathcal{H}_B$. In order to retrieve information about one of the subsystems from a state in the combined Hilbert space, one needs to introduce the partial trace. For vectors ρ in the combined Hilbert space, the partial trace with respect to subsystem B is denoted by

$$tr_B(\rho). \tag{13}$$

Sometimes (13) is referred to as tracing out system B because it leaves a *reduced state* ρ_A living in subsystem A. A more technical definition for the partial trace is

$$\mathrm{tr}_B(\rho) = \sum_b (I_A \otimes \langle b|)\rho(I_A \otimes |b\rangle) \tag{14}$$

with an orthonormal basis $\{|b\rangle\}$ of subsystem B and I_A, the identity of subsystem A.

3.1.4 The Quantum Channel Formalism

In order to gain a better understanding of some mathematical concepts appearing in the following sections, it is necessary to introduce quantum channels. Quantum channels or quantum operations are a way to describe the dynamics of a quantum system. Dynamic evolution of quantum systems can be the effect of planned interactions but can also result from external noise. The initial state of a quantum system, given by its density operator ρ, may transform due to a dynamic process. Quantum operations are given by maps $\mathcal{E}$.

$$\rho' = \mathcal{E}(\rho) \tag{15}$$

The map $\mathcal{E}$ must further be trace-preserving and completely positive. The property of being trace-preserving means that $tr(\rho) = tr(\mathcal{E}(\rho))$. Complete positivity means that positive elements are mapped to positive elements. Both properties are necessary because ρ' must also be a density operator. A map can be represented in the operator sum representation as follows: [28]:

$$\mathcal{E} = \sum_k E_k \rho E_k^\dagger \tag{16}$$

E_k in this case is an operator acting on ρ, called a *Kraus Operator* [28]. Each E_k represents a unitary evolution of the density operator ρ occurring with probability p_k:

$$p_k = tr(E_k \rho E_k^\dagger) \tag{17}$$

3.2 Gates

One can only perform computations when it is possible to transform the state of a bit (or a qubit). Transformations of the state are performed by gates. For only one classical bit, a gate can change 0 to 1 and vice versa or leave the bit unchanged. The gate changing the state is called the NOT gate. There should also be a quantum NOT gate that changes $|0\rangle$ to $|1\rangle$ and vice versa. Indeed, there is a quantum NOT gate. It, however, is not that trivial. Take the superposition state Ψ. How should a quantum NOT gate transform this state? Here, α and β come into play. The quantum NOT gate X interchanges α and β. It is easy to convince oneself that X indeed has the desired effect on a state, for instance: $|0\rangle$ is just a superposition state Ψ with $\beta = 0$ and $\alpha = 1$. After applying the NOT gate, one receives a superposition state with a 0 as the factor for $|0\rangle$ and a 1 as the factor for $|1\rangle$, which is nothing else than $|1\rangle$.

$$\left|\Psi'\right\rangle = \beta\,|0\rangle + \alpha\,|1\rangle = X\,|\Psi\rangle \tag{18}$$

Gates, such as the NOT gate, can be represented by matrices. In this representation, states Ψ can be represented as vectors

$$\begin{bmatrix} \alpha \\ \beta \end{bmatrix}. \tag{19}$$

The gate has to be a 2 by 2 matrix:

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}. \tag{20}$$

Every matrix representing a gate must be unitary, which means that for any gate U, $UU^\dagger = U^\dagger U = I$. This is a consequence of the constraint that $|\alpha|^2 + |\beta|^2 = 1$.

There are infinitely many possible quantum gates. They must only be unitary. In practice,

however, there are some important gates that are used very often. One of those important one qubit gates is the Hadamard gate H. It is defined as

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{21}$$

The Hadamard gate turns the basis states $|0\rangle$ and $|1\rangle$ into superposition states that have a probability of $\frac{1}{2}$ for each possible measurement outcome.

$$|0\rangle \rightarrow \frac{1}{\sqrt{2}} |0\rangle + \frac{1}{\sqrt{2}} |1\rangle \tag{22}$$

$$|0\rangle \rightarrow \frac{1}{\sqrt{2}} |0\rangle - \frac{1}{\sqrt{2}} |1\rangle \tag{23}$$

3.3 Multiple Qubits

Building a (quantum)computer with only one (qu)bit is not very useful. One has to wire qubits together and add gates to the circuits of wired qubits. For many, for instance, n, qubits, there are 2^n possible measurement results. Therefore, there are 2^n possible basis states with their factors α_i. The basis vectors of a many-qubit system are given by listing the state of each qubit one after the other:

$$|0...0\rangle, |1...0\rangle, ..., |1...1\rangle. \tag{24}$$

The convention (24) is a short form for the tensor product of all the basis states. A difference to a classical computer is the partial measurement. One can measure the state of some qubits while not measuring others. Therefore, one obtains no information about the other qubits but still has to update the many-qubit state after measurement. Take a two-qubit state $|\Psi\rangle = \alpha_{00} |00\rangle + \alpha_{10} |10\rangle + \alpha_{01} |01\rangle + \alpha_{11} |11\rangle$ for instance. Performing a measurement only on the first qubit (one can retrieve the state of the first qubit by using the partial trace on $|\Psi\rangle \langle\Psi|$), one obtains 0 with probability $|\alpha_{00}|^2 + |\alpha_{01}|^2$ and 1 with probability $|\alpha_{10}|^2 + |\alpha_{11}|^2$. After a measurement on the first qubit with the result 0, for instance, the post measurement state $|\Psi'\rangle$ must be consistent with that information. Therefore, only $|00\rangle$ and $|01\rangle$ are left. Since the amplitudes of the remaining states are no longer normalised, a factor is introduced:

$$|\Psi'\rangle = \frac{\alpha_{00} |00\rangle + \alpha_{01} |01\rangle}{\sqrt{|\alpha_{00}|^2 + |\alpha_{01}|^2}}. \tag{25}$$

3.3.1 Many Qubit Gates

Section 3.2 introduced quantum gates as unitary operations. This has profound consequences. Unitary operators have inverse unitary operators. Therefore, every operation performed by a quantum gate can be inverted by another unitary operator and hence another quantum gate. To understand why this is important, one can look at classical multi-bit

gates. There is the *not and* gate NAND. It takes several inputs and has one output. It only outputs a 0 if all the inputs are 1; in all other cases, it gives 1. It can be shown that every possible computation on bits can be represented by combining NAND gates. This property is called universality [28]. The NAND gate, however, is not reversible: There is no way to know which input bits were 0 and which were 1 when the output of the NAND gate is 1. As a consequence, there is no quantum mechanical NAND gate. Despite this inconvenience, it can be shown that a combination of CNOT gates and single-qubit gates has the universality property. CNOT is short for *controlled not* gate. It has two input qubits: the control qubit and the target qubit. The CNOT gate flips the target qubit only if the control qubit is in state $|1\rangle$. Else it leaves both qubits unchanged. The *Toffoli* gate, a gate constructed from CNOT and single qubit gates, is able to simulate classical NAND gates. Therefore, every classical gate can be simulated by quantum gates, and hence, the circuitry of a classical computer can be simulated on a quantum computer.

3.4 Quantum Algorithms

Since quantum computers simulate any classical computer circuit (see section 3.3.1), the question remains whether quantum computers can outperform classical computers in some tasks. Indeed, they can. There are two categories where quantum computers offer advantages: quantum communication and quantum algorithms. This section will not cover quantum communication in detail, although the no-cloning theorem, a significant theoretical result, will be discussed. Subsequently, a few classes of algorithms will be introduced.

3.4.1 No-Cloning Theorem

One significant result in quantum theory is the no-cloning theorem. It states that it is impossible to construct a system copying every qubit perfectly to another set of qubits [43]. At first glance, this appears to be a major disadvantage for quantum computers. Many classical algorithms rely on copying data. In particular, this affects error correction codes [7]. Classical error correction codes do not work on a quantum computer if they involve the copying of data. However, on closer examination, the no-cloning theorem offers some exciting opportunities. Eavesdropping on encrypted communication always involves the eavesdropper copying the message. In quantum communication, this cannot happen unnoticed due to the no-cloning theorem. The receiver of the message will inherently always be able to find out whether a message has been eavesdropped on.

3.4.2 Algorithms

Three main classes of algorithms are known to be calculated faster on quantum computers [28]. Computational complexity theory is an open research area, and there may be unexplored opportunities in quantum computation.

The first class of algorithms is the quantum simulation of quantum mechanical systems.

The advantage quantum computers may offer in this area becomes obvious with the quantum states discussed. In order to completely describe $|\Psi\rangle = \alpha|0\rangle + \beta|1\rangle$, one needs two complex numbers. More complex systems with n distinct qubits, therefore, need 2^n complex numbers to simulate the behaviour of the system. Quantum computers only need n qubits to simulate the same system. This is an exponential improvement that comes with a caveat. It is impossible to access the information about the superposition of the base states, encoded in the complex numbers α and β, directly. When performing a measurement, the result will always be 0 or 1. Nevertheless, with a clever design of the quantum simulator for a quantum system, one might be able to extract useful information and benefit from exploiting the exponential improvement.

The second class of algorithms is based on the *Fourier transform*. It is not necessary to convince physicists of the enormous number of applications for the Fourier transform. The classical fast Fourier transform needs $\propto Nlog(N)$ steps to transform N numbers. Quantum computers only need $\propto log^2(N)$ steps [28]. Again, this advantage is not entirely usable, since measuring the output of a quantum circuit yields either $|0\rangle$ or $|1\rangle$ for each qubit.

Finally, the third class of algorithms computed faster by quantum computers are *quantum search algorithms*. For an unknown data set of size N, it takes approximately N operations to find a specific element. Quantum search algorithms take only about $\sqrt{N}$ operations to find a specific element, a quadratic improvement that has an enormous impact in the range of large data sets.

Section 2.5 discussed the limits of classical machine learning in part due to physical constraints. With the knowledge about the advantages of quantum computers, one might want to exploit these advantages for machine learning. There are two ways to do so: either by using quantum algorithms to speed up classical machine learning or by using quantum devices to learn tasks involving quantum data. Alongside machine learning techniques that speed up quantum simulations, the previous techniques are called *quantum machine learning* [4].

4 Quantum Neural Networks

This section provides a summary of an approach for a deep quantum neural network by Beer et al.[4]. This section further describes alterations to the network generalising it to work with mixed states.

The revolutionary ability of machine learning with neural networks to solve problems and the potential of quantum computing are discussed in sections 2 "Classical Machine Learning and Neural Networks" and 3 "Quantum Computing". Naturally, those possibilities make quantum machine learning a promising research topic.

Beer et al.[4] developed an approach analogous to a classical deep neural network with backpropagation. Hence, three main ingredients had to be developed for a quantum device. The first ingredient is a quantum perceptron and the network architecture; the second is a cost function, and the third is an optimisation algorithm.

4.1 The Perceptron and the Network

Beer et al. [4] defined the perceptron as a general unitary operator acting on qubits.

Each perceptron acts on m input qubits and one output qubit. This way, the unitaries can be seen as analogous to the weights and biases of the perceptron in a classical neural network.

The network receives the states of input qubits as input ρ^{in}. The unitaries of the first layer act on the input qubits and the output qubits which are in a reference state $|0...0\rangle_{out}$. Consequently the unitaries act on the tensored state $\rho^{in} \otimes |0...0\rangle_{out} \langle 0...0|$. The output state is obtained by tracing out the input layer: $\rho^{out} = tr_{in}(U(\rho^{in} \otimes |0...0\rangle_{out} \langle 0...0|)U^\dagger)$. This output state forms the input state for the next layer.

The layer of the network, counted with the index l, and the number of perceptrons in each layer, counted via the index j, are used to name the unitaries U_j^l. Since unitary operators generally do not commute, the order of application to the input state matters. With the number of unitaries in layer l, m_l and the total number of hidden layers L, one can represent the feedforward of the network as a composition of completely positive *layer to layer maps*.

In the first step, the input state of a previous layer X^{l-1} has to be tensored to the outer product of the output states to match the dimension of the unitaries:

$$X^{l-1} \otimes |0...0\rangle_l \langle 0...0| \tag{26}$$

Now the unitaries of the layer can be applied to the tensored state. As mentioned before, the order of application matters. The first unitary of the layer has to be applied first,

followed by the second and so on:

$$U^l_{m_l}(...(U^l_1(X^{l-1} \otimes |0...0\rangle_l \langle 0...0|)U^{l\dagger}_1)...)U^{l\dagger}_{m_l} \tag{27}$$

The total quantum circuit $\mathcal{U}$ is the product of layer unitaries which are, in turn, the product of the unitaries from a layer l.

$$\mathcal{U} = U^{out}U^L U^{L-1}...U^1 \quad \text{with} \quad U^l = U^l_{m_l}U^l_{m_l-1}...U^l_1 \tag{28}$$

The layer to layer map is then obtained by tracing out the previous layer from (27):

$$\mathcal{E}^l(X^{l-1}) = tr_{l-1}(U^l_{m_l}(...(U^l_1(X^{l-1} \otimes |0...0\rangle_l \langle 0...0|)U^{l\dagger}_1)...)U^{l\dagger}_{m_l}) \tag{29}$$

In order to go into the opposite direction, i.e. from layer $l+1$ to layer l, one has to apply the adjoint to the layer channel $\mathcal{E}$, $\mathcal{F}$:

$$\mathcal{F}^l(X^l) = tr_l\Big(I_{l-1} \otimes |0...0\rangle_l \langle 0...0|\, U^{l\ \dagger}(I_{l-1} \otimes X^l)U^l\Big) \tag{30}$$

Expression (30) contains the layer unitaries as introduced in (28). Now, the feedforward results from applying the layer to layer maps one after the other to the input state:

$$\rho^{out} = \mathcal{E}^{out}(\mathcal{E}^L(...(\mathcal{E}^1(\rho_{in})...)) \tag{31}$$

4.2 The Cost Function

For the network to learn, it is necessary to compare the network's output to the desired output for the same input. Pairs of states, one input state and one desired output state, are called training pairs. A collection of training pairs is the training data. The output of the quantum neural network is the output state ρ_{out}, which is a quantum state. Hence, one has to find a suitable measure that measures the "closeness" of two quantum states. This is not a trivial task and is one of the subjects of quantum information theory.

4.2.1 Cost Function for Pure Output

Beer at al. chose the *fidelity*. Physically, the fidelity for two quantum states gives the probability of identifying one state with the other in a measurement. For pure quantum states $\sigma = |\psi\rangle \langle\psi|$ and ρ it is defined as [28]:

$$F(\sigma, \rho) = \sqrt{\langle\psi|\, \rho\, |\psi\rangle}. \tag{32}$$

The fidelity is 1 in the case of two identical pure quantum states:

$$F(\sigma, \sigma) = \sqrt{\langle\psi|\, \sigma\, |\psi\rangle} = \sqrt{\langle\psi|\psi\rangle\, \langle\psi|\psi\rangle} = 1 \tag{33}$$

and 0 for two orthogonal quantum states. This difference to most conventions for identity has to be considered when updating the network parameters in the learning algorithm later.

To have a cost function analogous to the one described in section 2.3, the fidelity has to be calculated for the output state of the network and the desired output state to the input state of the network. In general, the training data consists of several training pairs of input states ρ^{in} and desired output states $\left|\Phi^{out}\right\rangle\left\langle\Phi^{out}\right|$. Hence, the cost function yields the average "distance" of network output to desired output over all training pairs N:

$$C = \frac{1}{N}\sum_{x=1}^{N}\left\langle\Phi_x^{out}\right|\rho_x^{out}\left|\Phi_x^{out}\right\rangle. \tag{34}$$

4.2.2 Cost Function for Mixed Output

A more general quantum neural network works with mixed states. The fidelity, defined in (32), has a more general definition for mixed states. It is sometimes referred to as *Uhlmann fidelity*. For two mixed states ρ and σ it is given as [20]:

$$F(\rho,\sigma) = \left(tr\left(\sqrt{\sqrt{\sigma}\rho\sqrt{\sigma}}\right)\right)^2. \tag{35}$$

It behaves similarly to the fidelity in (32): It becomes 1 for identical states and 0 for orthogonal states. Due to its high nonlinearity, (35) is hard to optimise. Therefore, a new measure of distance is necessary for the cost function.

The *Schatten-2-norm* was selected for this purpose. In general, a Schatten-p-norm for a linear bounded operator $T\colon \mathcal{H}_1 \to \mathcal{H}_2$ for $p \in [1,\infty)$ is denoted by [22]

$$\|T\|_p = [tr(|T|^p)]^{\frac{1}{p}}. \tag{36}$$

As a distance measure for the cost function, the Schatten-2-norm is sufficient. The difference of the output state ρ^{out} to the desired output state σ acts as the linear bounded operator T. Now, the new cost function can be defined:

$$C = \frac{1}{N}\sum_{x=1}^{N}\left[tr(|\rho_x^{out}-\sigma_x|^2)\right] \tag{37}$$

$$= \frac{1}{N}\sum_{x=1}^{N} tr((\rho_x^{out}-\sigma_x)(\rho_x^{out}-\sigma_x)^\dagger) \tag{38}$$

It is important to note that this cost function yields 0 for identical states and has an upper limit. Furthermore, the exponent $\frac{1}{2}$ was omitted because the squared part of (37) is strictly monotonically increasing and easier to optimise.

4.3 The Learning Algorithm

In total, the learning algorithm described by Beer et al. [4] consists of five major parts: The initialisation of the network, a feedforward part, a backpropagation part, an update of the parameters and finally, repetition until the cost function reaches the maximum (Considering that the fidelity outputs the highest values for identical quantum states).

The generation of training data is not a part of the learning algorithm, but in the end, the network should be able to solve a task. In order to train the network, one needs training data containing examples of how the network should transform input data. The training data is a set of data that consists of N pairs of input states and desired output states $(\left|\Phi_x^{in}\right\rangle, \left|\Phi_x^{out}\right\rangle)$. The desired output states are generated from the input states by applying the specific transformation, which the network should learn, to the input states. In practical scenarios, however, this can also be experimental data.

1.) **Initialisation of the network:** First, some new parameters must be introduced: the *step* parameter s and the *step size* ϵ. After every execution of the backpropagation, the step parameter s will be increased by ϵ. Doing this allows to express the cost function as a function of the step: $C(s)$. With these two new parameters, one can initialise the network by setting $s = 0$ and randomly choosing the network unitaries $U_j^l(s = 0)$.

2.) **Feedforward:** For each pair of states in the training data and every network layer, the layer channel $\mathcal{E}^l$ is applied to the tensored output of the previous layer. After that, the previous layer is traced out. Doing that for every layer completes the feedforward according to (31).

3.) **Backpropagation:** Similar to the classical backpropagation algorithm, the goal is to update the network parameters in a way that increases the cost function the fastest. A change in the cost function is given as

$$\Delta C = \frac{\epsilon}{N} \sum_{x=1}^{N} \sum_{l=1}^{\text{Output Layer}} tr(\sigma_x^l \Delta \mathcal{E}^l(\rho_x^{l-1})). \tag{39}$$

The backwards part is in the σ_x^l, which is similar to ρ_x^l. One obtains ρ_x^l by applying the layer channels $\mathcal{E}$ to the input data and σ_x^l by applying the adjoint layer channels $\mathcal{F}$ to the output data.

4.) **Update of the network parameters:** Start with the calculation of the cost function for the output state of the network after the feedforward and the desired output state from the training data. Next, calculate the parameter matrices $K_j^l(s)$, which are used to update each perceptron according to

$$U_j^l(s + \epsilon) = e^{i\epsilon K_j^l(s)} U_j^l(s) \tag{40}$$

and can be derived using expression (39). The calculation of the parameter matrices is the subject of section 4.3.1. Having updated every perceptron, finish by updating the step parameter s to $s + \epsilon$.

5.) **Repetition** Repeat steps 2, 3 and 4 until the maximum of the cost function is reached.

4.3.1 Calculation of Parameter Matrices

In step 4 of the learning algorithm, the parameter matrices $K_j^l(s)$ have been introduced. This section will explain how the parameter matrices are calculated.

In a classical neural network as described in section 2, the perceptrons are updated by taking the derivative (gradient) of the cost function and moving a small step in the opposite direction. The derivative of the cost function in (34) is given via the difference quotient as:

$$\frac{dC}{dS} = \lim_{\epsilon \to 0} \frac{C(s+\epsilon) - C(s)}{\epsilon} \tag{41}$$

In order to calculate $C(s+\epsilon)$, one needs to know the output of the updated network $\rho_x^{out}(s+\epsilon)$. Every unitary in the network is updated according to the update rule introduced in (40). Changing the unitaries changes the layer channels in $\mathcal{E}$ (29). With the knowledge about the updated layer channels one can finally express $\rho_x^{out}(s+\epsilon)$ as a function of $\rho_x^{out}(s)$. Now, the derivative of the cost function can be calculated.

$$\begin{aligned}
\frac{dC}{dS} &= \lim_{\epsilon \to 0} \frac{C(s+\epsilon) - C(s)}{\epsilon} \\
&= \lim_{\epsilon \to 0} \frac{C(s) + \frac{i\epsilon}{N} \sum_x \left\langle \Phi_x^{out} \right| \left(\rho_x^{out}(s+\epsilon) - \rho_x^{out}(s) \right) \left| \Phi_x^{out} \right\rangle - C(s)}{\epsilon} \qquad (42) \\
&= \frac{1}{N} \sum_x tr\Big(\big((I_{in,hidden} \otimes \left| \Phi_x^{out} \right\rangle \left\langle \Phi_x^{out} \right|) \big(\Big[iK_{m_{out}}^{out}(s), \\
&U_{m_{out}}^{out}(s) ... U_1^1(s) (\rho_x^{in} \otimes |0...0\rangle_{hidden,out} \langle 0...0|) U_1^{1\dagger}(s) ... U_{m_{out}}^{out\ \dagger}(s) \Big] + ... \\
&+ U_{m_{out}}^{out}(s) ... U_2^1(s) \Big[iK_1^1(s), U_1^1(s) (\rho_x^{in} \otimes |0...0\rangle_{hidden,out} \langle 0...0|) U_1^{1\ \dagger}(s) \Big] \\
&U_2^{1\ \dagger}(s) ... U_{m_{out}}^{out\ \dagger}(s) \big) \big) \Big) \qquad (43)
\end{aligned}$$

A key feature of the algorithm is that, in order to calculate the parameter matrix, the algorithm only has to access two layers at a time. It can be written in terms of layer to layer channels. This key feature appears by defining

$$\begin{aligned}
M_j^l(s) = \Big[& U_j^l(s) U_{j-1}^l(s) ... U_1^1(s) (\rho_x^{l-1} \otimes |0...0\rangle_l \langle 0...0|) U_1^{1\ \dagger}(s) ... U_{j-1}^{l\ \dagger}(s) U_j^{l\ \dagger}(s), \\
& U_{j+1}^{l\ \dagger}(s) ... U_{m_{out}}^{out\ \dagger}(s) (I_{l-1} \otimes \sigma_x^l) U_{m_{out}}^{out}(s) ... U_{j+1}^l(s) \Big]
\end{aligned} \tag{44}$$

The backwards part shows itself in the second part of the commutator. For some calculations it makes sense to further compress the expression for $M_j^l(s)$ by defining

$$A_j^l(s) = U_j^l(s)U_{j-1}^l(s)...U_1^1(s)(\rho_x^{l-1} \otimes |0...0\rangle_l \langle 0...0|)U_1^{1\,\dagger}(s)...U_{j-1}^{l\,\dagger}(s)U_j^{l\,\dagger}(s) \tag{45}$$

$$B_j^l(s) = U_{j+1}^{l\,\dagger}(s)...U_{m_{out}}^{out\,\dagger}(s)(I_{l-1} \otimes \sigma_x^l)U_{m_{out}}^{out}(s)...U_{j+1}^l(s). \tag{46}$$

so that

$$M_j^l(s) = [A_j^l(s), B_j^l(s)]. \tag{47}$$

By rearranging the commutators in (43) and further writing $M_j^l(s)$ differently as

$$M_j^l(s) = \Big[U_j^l(s)U_{j-1}^l(s)...U_1^1(s)(\rho_x^{in} \otimes |0...0\rangle_{hidden,out} \langle 0...0|)U_1^{1\,\dagger}(s)...U_{j-1}^{l\,\dagger}(s)U_j^{l\,\dagger}(s),$$
$$U_{j+1}^{l\,\dagger}(s)...U_{m_{out}}^{out\,\dagger}(s)(I_{in,hidden} \otimes \left|\Phi_x^{out}\right\rangle \left\langle\Phi_x^{out}\right|)U_{m_{out}}^{out}(s)...U_{j+1}^l(s)\Big] \tag{48}$$

one can simplify the expression in (43) to

$$\frac{dC}{dS} = \frac{i}{N}\sum_x tr\big(M_{m_{out}}^{out}(s)K_{m_{out}}(s)^{out}\big) + ... + tr\big(M_1^1(s)K_1^1(s)\big). \tag{49}$$

Expression (49) contains the parameter matrices $K_j^l(s)$, which still have not been defined any further as in $U_j^l(s+\epsilon) = e^{i\epsilon K_j^l(s)}U_j^l(s)$. The gradient of the cost function in classical neural networks gives the direction of the steepest ascent. To find the steepest ascent in the cost function with regard to the parameters, the maximum of the derivative of the cost function has to be found. Remember that the cost function (34) reaches 1 for identical states. Since $\frac{dC}{ds}$ is a linear function, its maxima are at $\pm\infty$. Therefore, a boundary condition has to be applied. In order to construct the boundary condition, one can enumerate the qubits. For the jth perceptron in the lth layer, one can enumerate all the qubits in the previous layers as α_i and denote the current qubit as β. Now, one can parametrise the parameter matrices $K_j^l(s)$:

$$K_j^l(s) = \sum_{\alpha_1,\alpha_2,...,\alpha_{m_{l-1}},\beta} K_{j,\alpha_1,\alpha_2,...,\alpha_{m_{l-1}},\beta}^l(s)(\sigma^{\alpha_1} \otimes ... \otimes \sigma^{\alpha_{m_{l-1}}} \otimes \sigma^\beta). \tag{50}$$

The sum of the square of the factors $K_{j,\alpha_i,...,\beta}^l(s)$ in (50) has to be a constant ν. Therefore, with the method of *Lagrange multipliers* λ, one obtains an expression

$$\max_{K_{j,\alpha_i,...,\beta}^l} \Big(\frac{dC(s)}{ds} - \lambda\Big(\sum_{\alpha_i,\beta} K_{j,\alpha_i,...,\beta}^l(s)^2 - \nu\Big)\Big). \tag{51}$$

Additional algebra for this expression, taking the derivative with respect to $K_{j,\alpha_i,...,\beta}^l$ setting it to zero and performing even more algebraic operations yields the parameter matrix:

$$K_j^l(s) = \frac{2^{n_{\alpha_i,...,\beta}}i}{2N\lambda}\sum_x tr_{rest}(M_j^l(s)). \tag{52}$$

With λ in expression (52), the learning rate $\eta = \frac{1}{\lambda}$ can be defined. Now there is a definition for the parameter matrices $K_j^l(s)$ that allows to perform the learning algorithm from section 4.3.

4.3.2 Alterations for Mixed States

The learning algorithm for a network with mixed states is, in most parts, similar to the one in section 4.3. Changes must be made in the generation of training data and in steps 3 and 4.

The alterations in step 4 are the most obvious: A different cost function was introduced for mixed output in section 4.2.2. Equation (37) describes a cost function that reaches 0 for identical states. Therefore, the algorithm has to be repeated until the cost function reaches its minimum, in contrast to the maximum.

The training data used in the algorithm may also look different. Since the network should work with mixed states as training data, the training output can contain mixed states.

Finally, the calculation of the update matrices must be altered. Section 4.2.2 already presented the cost function for mixed output states ρ and the "correct" output state σ:

$$\begin{aligned} C &= \frac{1}{N}\sum_{x=1}^{N} tr(|\rho_x^{out} - \sigma_x^{out}|^2) \\ &= \frac{1}{N}\sum_{x=1}^{N} tr(\rho_x^{out\ 2} - 2\rho_x^{out}\sigma_x^{out} + \sigma_x^{out\ 2}) \end{aligned} \tag{53}$$

Similar to what has been done in (42), one has to first take a closer look at $\rho_x^{out}(s+\epsilon)$. In order to find an expression for the derivative of the cost function with respect to the parameter s, one has to evaluate an expression of the cost function as a function of $\rho_x^{out}(s+\epsilon)$ which, in turn, is a function of $s+\epsilon$. $\rho_x^{out}(s+\epsilon)$ can be expressed in terms of the updated unitaries $U_j^l(s+\epsilon) = e^{i\epsilon K_j^l(s)} U_j^l(s)$. Inserting this into the expression for the layer channels (29) and the resulting expression into (31) one obtains:

$$\begin{aligned} \rho_x^{out}(s+\epsilon) = tr_{in,hidden}\Big(& e^{i\epsilon K_{m_{out}}^{out}(s)} U_{m_{out}}^{out}(s) ... e^{i\epsilon K_1^1(s)} U_1^1(s) (\rho_x^{in} \otimes |0...0\rangle_{hidden,out} \langle 0...0|) \\ & U_1^{1\ \dagger}(s) e^{i\epsilon K_1^1(s)} ... U_{m_{out}}^{out\ \dagger}(s) e^{i\epsilon K_{m_{out}}^{out}(s)} \Big) \qquad (54) \\ = \rho_x^{out}(s) + i\epsilon tr_{in,hidden}\Big(& \Big[iK_{m_{out}}^{out}(s), U_{m_{out}}^{out}(s) ... U_1^1(s) (\rho_x^{in} \otimes |0...0\rangle_{hidden,out} \langle 0...0|) \\ & U_1^{1\dagger}(s) ... U_{m_{out}}^{out\ \dagger}(s) \Big] + ... + U_{m_{out}}^{out}(s) ... U_2^1(s) \\ & \Big[iK_1^1(s), U_1^1(s) (\rho_x^{in} \otimes |0...0\rangle_{hidden,out} \langle 0...0|) U_1^{1\ \dagger}(s) \Big] \Big) + \mathcal{O}(\epsilon^2) \qquad (55) \\ := \rho_x^{out}(s) + i\epsilon\gamma & \qquad (56) \end{aligned}$$

In (56), γ is introduced solely for better readability. With the expression for $\rho_x^{out}(s+\epsilon)$, one can insert that into expression (41) to obtain

$$\begin{aligned}\frac{dC}{dS} &= \lim_{\epsilon\to 0}\frac{C(s+\epsilon)-C(s)}{\epsilon}\\ &= \frac{1}{N\epsilon}\lim_{\epsilon\to 0}\sum_{x=1}^{N}\Big(tr\big((\rho_x^{out}+i\epsilon\gamma)^2-2(\rho_x^{out}-i\epsilon\gamma)\sigma_x^{out}+\sigma_x^{out\ 2}\big)\\ &- tr\big(\rho_x^{out\ 2}-2\rho_x^{out}\sigma_x^{out}+\sigma_x^{out\ 2}\big)\Big).\end{aligned} \tag{57}$$

Further simplifications lead to

$$\frac{dC}{dS} = \frac{-2i}{N}\sum_{x=1}^{N} tr\big(\gamma(I_{in,hidden}\otimes(\rho_x^{out}-\sigma_x^{out})\big). \tag{58}$$

Reintroducing γ yields an expression similar to (49) with $M_j^l(s) = [A_j^l(s), B_j^l(s)]$, with only a different definition of $B_j^l(s)$ because it is the only part of $M_j^l(s)$ which contains the output of the network.

$$B_j^l(s) = U_{j+1}^{l\ \dagger}(s)...U_{m_{out}}^{out\ \dagger}(s)(I_{in,hidden}\otimes(\rho_x^{out}-\sigma_x^{out}))U_{m_{out}}^{out}(s)...U_{j+1}^{l}(s) \tag{59}$$

5 Results

Having derived a training algorithm for mixed states in section 4, this section presents the results of simulations with the network. The code used can be found in [35]. It is based on code written by Beer et. al [3]. First, this section will compare the performance of the derived algorithm for mixed states to the initial algorithm for pure states. Afterwards, the algorithm is used to learn several different quantum channels for which the influence of network parameters like network architecture, learning rate and generalisation behaviour are examined.

5.1 Learning a Random Unitary

A sensible thing to do is to check how the algorithm for mixed states performs compared to the algorithm for pure states. In order to do so, the initial algorithm and the altered algorithm have been used to train the network with the same set of training data. The training data is generated by a random unitary operator acting on random states. The random unitary acts on states that are used as input states for the network and thus generates the output states the network should learn. There are three types of states to which the unitary is applied. First, pure states as used in the initial training algorithm for pure states. Secondly, projections of these pure states to which the algorithm for mixed states is applied. Thirdly, random mixed states are generated for the algorithm for mixed states.

During the training, all three algorithms used their respective cost functions, introduced in section 4.2. The performance metric is the cost function for previously unseen data, called test data. *Generalisationability* is the ability of the network to generalise its learning from a training pair to a set of new test data. In other words: With a certain amount of training data, the network can perform a task at some level with new data. It is a desirable property to need as few as possible training data sets to perform adequately on new data.

The new data, in this case, is more data of the same type as the training data. The same unitary acts on new input states and thus creates the new desired outputs (test data). The two network's outputs for the new input states (from the test data) are later compared to the desired outputs. For this comparison, it was reverted to the more general definition of the fidelity, which can be applied to the mixed output.

In order to also apply the more general fidelity to the pure outputs, one has to use their projection operators. $\rho = |\Psi\rangle \langle\Psi|$. Furthermore, the training data consists of random qubit states. In order to input them into the network for mixed qubit states, their projection operators have to be used.

A comparison of the performances for pure states and for mixed states yields the results shown in figure (3). There is no significant difference in performance for either type of states for the test and training data. The training algorithm for mixed states is comparable in

performance to its counterpart for pure states, justifying its application to other quantum channels. The following subsections will present some examples.

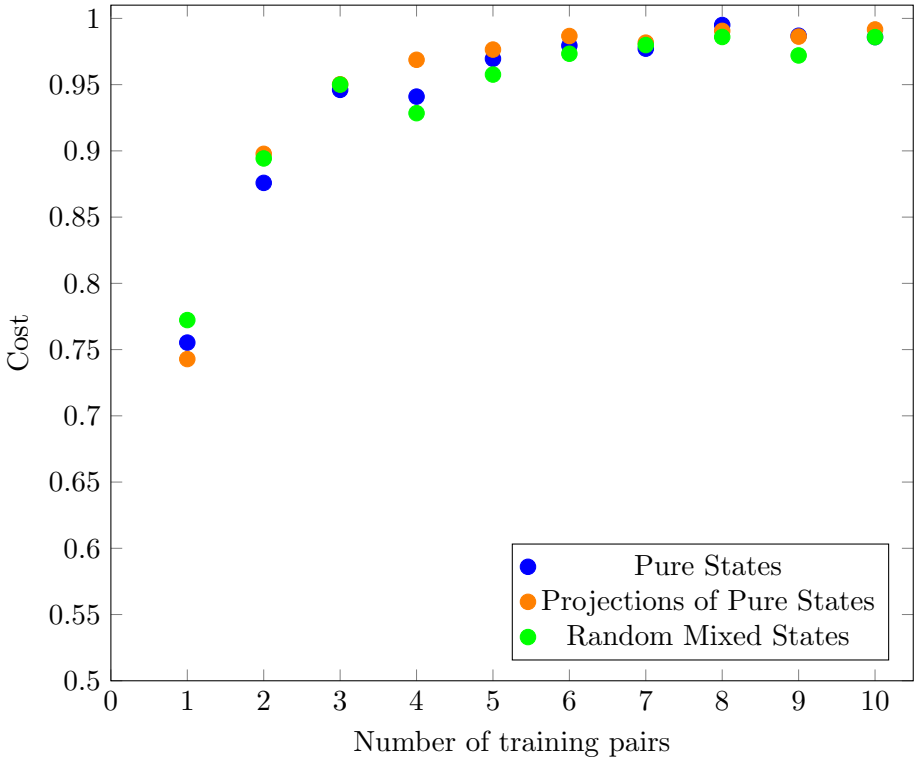


Figure 3: Average cost function for a 1-2-1 network with pure states and a network for mixed states with projectors of pure states and random mixed states depending on the number of training pairs. Each data point represents the average cost function for 10 sets of test data after 100 training rounds with 10 sets of training data.

5.2 Learning a Bit Flip

The *bit flip* channel flips a single qubit from its first state to its second state, e.g. from $|0\rangle$ to $|1\rangle$ and vice versa with probability $1-p$.
The Channel describing the bit flip is given by $\mathcal{E}(\rho) = \sum_i E_i \rho E_i^\dagger$ with operation elements [28]:

$$E_0 = \sqrt{p}\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad E_1 = \sqrt{1-p}\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{60}$$

The probability p chosen for the simulations was 0.7. The network that was trained had a 1-2-1 architecture which means one input qubit, one output qubit and one hidden layer

with two qubits.
First, some different learning rates η were tested. Five hundred training rounds were performed with a stepsize of 0.1.

Note that the cost function for the following examples is of the form introduced in 4.2.2. It becomes 0 for identical states; the algorithm therefore aimed at minimising the cost function.

Most of the difference in the learning behaviour for the tested learning rates can be seen until the step parameter s reaches 10. Hence, the graph is cut off at 10 in figure 4. The learning algorithm worked best with higher learning rates. Even higher learning rates than $\eta = 2$ can further increase the learning speed. This increase in speed, however, comes with the risk of overshooting. For large learning rates like $\eta = 10$, the cost increases again after about 25 steps (step 2.5 in (4) is step 25 due to the stepsize of 0.1) which illustrates the overshooting.

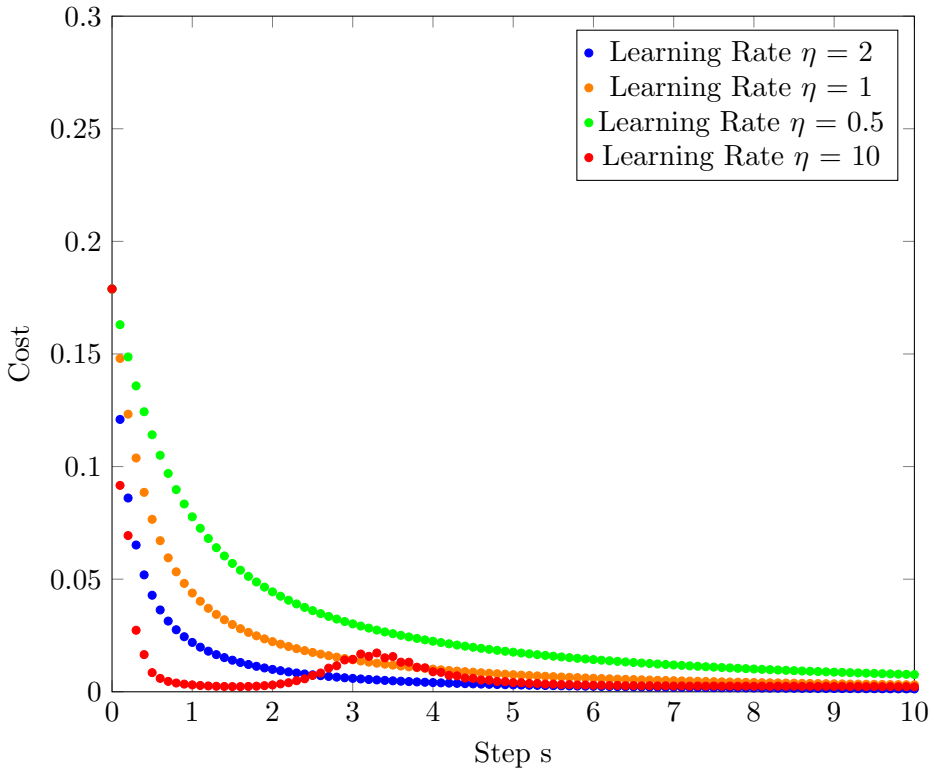


Figure 4: Cost function depending on the number of steps in the training algorithm for different learning rates. 500 training rounds to learn a bit flip were performed on a 1-2-1 network with a step size of 0.1.

Another parameter affecting the learning behaviour is the network architecture. One can

add more hidden layers with more qubits. Neural networks with wider layers are more elaborate to train. Therefore, it makes sense to find an efficient architecture for each problem. The task of learning a bit flip works best for a 1-2-1 network, as shown in figure (5).

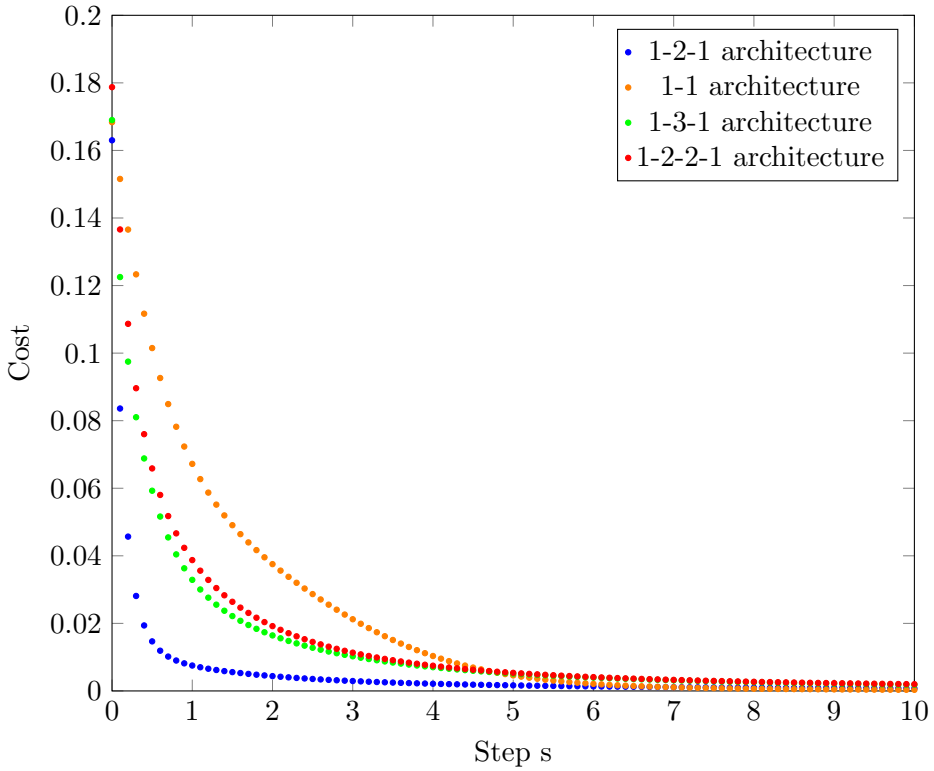


Figure 5: Cost function depending on training steps for different network architectures for learning a bit flip. 100 of 500 training rounds with a step size of 0.1 and a learning rate $\eta = 2$ are shown.

The previous assessments of the network are based on the cost function for the data with which the network has been trained. In order to evaluate the usefulness of the network, it is important to find out how good the network generalises. In real-world use cases for neural networks, this is important information to use and train neural networks efficiently.
In case of the bit flip, this was tested with a 1-2-1 network, a learning rate η of 2 and a stepsize of 0.1. The network was trained for 100 rounds with one training pair and then tested on a different set of 20 pairs of test data. Subsequently, two, three and higher numbers of pairs of training data were used to train the network. After the training, each network was tested with 20 sets of test data. Figure 6 shows the results for the bit flip. For comparison, the cost function for the training data was also plotted. After about seven training data pairs, the network performs almost as well for new test data as for training

data.

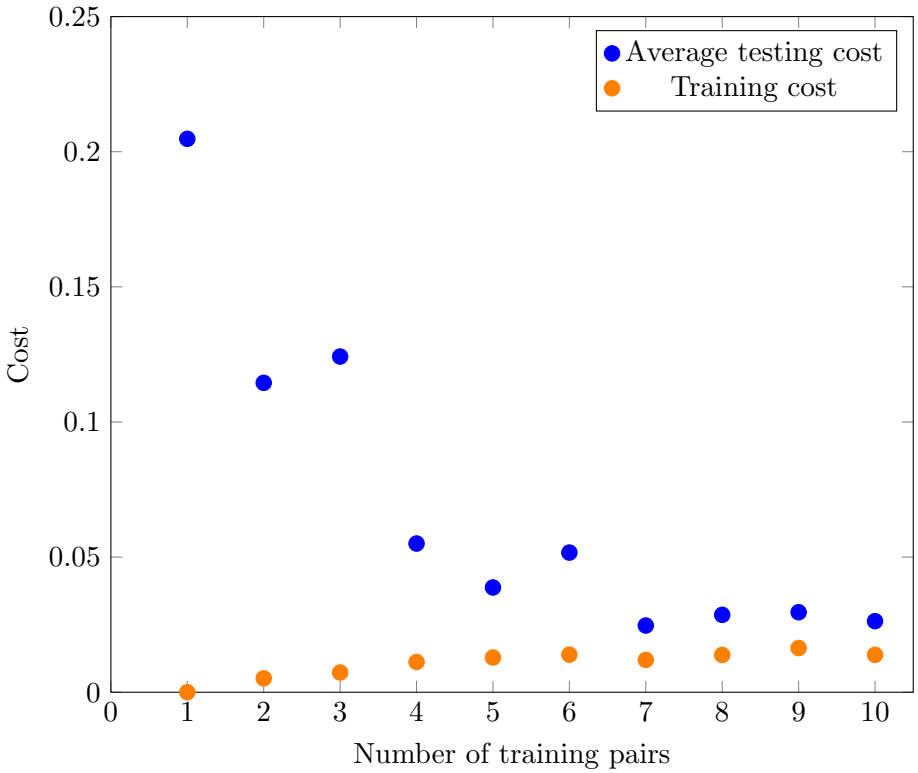


Figure 6: Generalisation for the Bit Flip: Average cost function for 20 sets of test data and cost function for training data depending on the number of training pairs after training for 100 rounds with stepsize of 0.1, a learning rate of $\eta = 2$ with a 1-2-1 network and performance on the training data after 100 rounds.

5.3 Learning a Phase Flip

Another standard channel is the *phase flip* channel. Similar to the bit flip channel, the learning algorithm was examined for different learning rates, network architectures, and generalisation behaviour. The phase flip channel flips the phase of a qubit. The phase of a quantum state is not a directly physically observable property, but it has some consequences in interference patterns, for instance. The operation elements for the phase flip are [28]:

$$E_0 = \sqrt{p}\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad E_1 = \sqrt{1-p}\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}. \tag{61}$$

This channel flips the phase with probability $1-p$. For all following phase flip simulations, p was set to 0.7.

5.3.1 Phase Damping Channel

The operation elements of the phase flip channel (61) are mathematically identical to the operation elements for a *phase damping* channel. Hence, there is no difference in the learning algorithms' performance for the phase damping channel. The phase damping channel can be understood as a noise event affecting the phase [28]. The equivalence can be derived by describing the interaction of a qubit with its environment as an interaction between two harmonic oscillators with an interaction Hamiltonian $H = \chi a^\dagger a(b + b^\dagger)$. a, $a^\dagger$ and b, $b^\dagger$ are the well known ladder operators for the quantum mechanical harmonic oscillator. The time evolution operator of this Hamiltonian is given by $U = e^{iH\Delta t}$. The qubit is represented as a quantum mechanical harmonic oscillator oscillating between the states $|0\rangle$ and $|1\rangle$. The environment is represented as a quantum mechanical harmonic oscillator starting in the state $|0\rangle$. In general, the operation elements E_k in the channel formalism for a system interacting with its environment can be expressed as [28]:

$$E_k = \langle e_k | U | e_0 \rangle \tag{62}$$

with the unitary evolution operator U, the basis states of the environment $|e_k\rangle$ and the initial state of the environment $|0\rangle$. In the example of the two oscillators

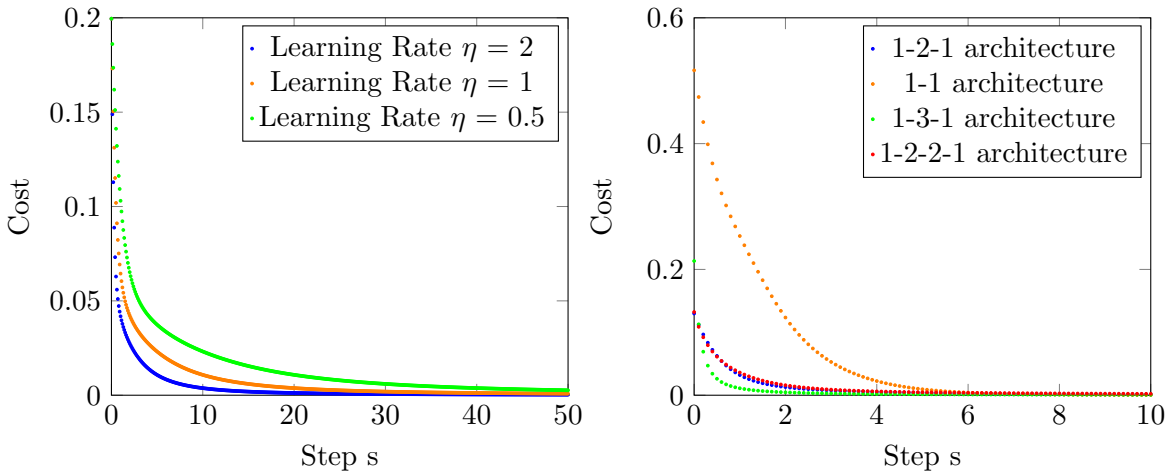
$$E_k = \langle k_{\text{environment}} | U | 0_{\text{environment}} \rangle . \tag{63}$$

Since the environment is also represented by an harmonic oscillator, the operation elements are given as:

$$E_0 = \sqrt{1-\lambda} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad E_1 = \sqrt{\lambda} \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} . \tag{64}$$

By further introducing $\lambda = 1 - \cos^2 \chi \Delta t$ and interpreting it as the probability of interaction of the first oscillator, the qubit, with the second one, the equivalence becomes obvious.

Back to the simulation results, figure 7 shows the learning behavior for the phase flip. In contrast to the bit flip channel, the fastest learning network architecture is the 1-3-1 architecture. Additionally, higher learning rates lead to faster learning rates, which is similar to the learning of the bit flip. Another observation is that an additional hidden layer does not increase the learning performance: The 1-2-2-1 network has almost the same learning curve as the 1-2-1 network. Moreover, the generalisation works slightly better than for the bit flip.



(a) Cost function depending on the number of steps in the training algorithm for different learning rates. 500 training rounds to learn a phase flip were performed on a 1-2-1 network with a step size of 0.1.

(b) Cost function depending on training steps for different network architectures for learning a phase flip. 100 of 500 training rounds with a step size of 0.1 and a learning rate $\eta = 2$ are shown.

Figure 7: Learning behaviour for the phase flip channel.

The generalisation behaviour is very similar to the one of the bit flip. This suggests that the learning algorithm works similarly well for both channels.

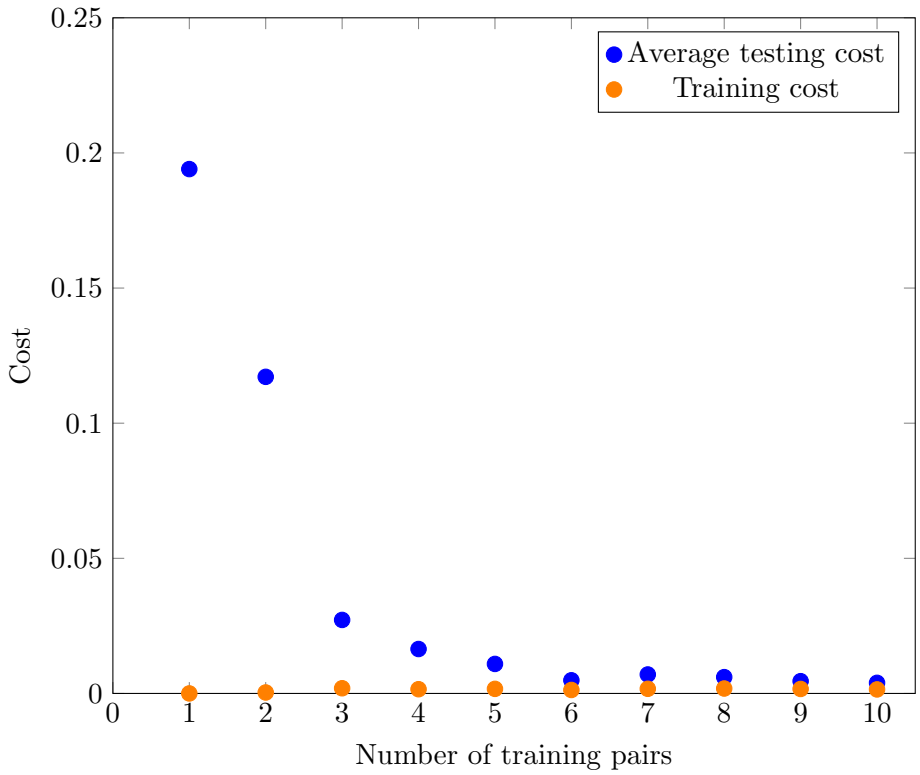


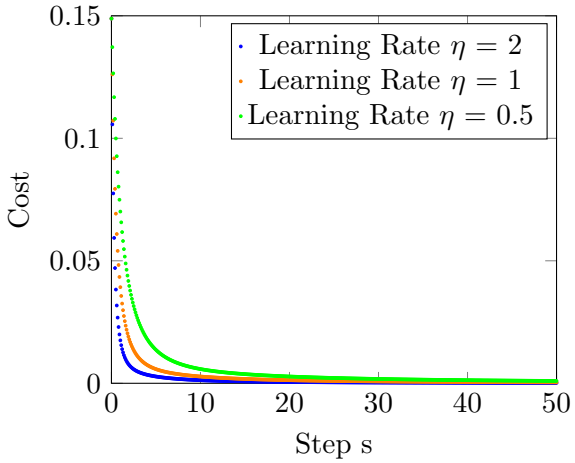
Figure 8: Generalisation for the Phase Flip: Average cost function for 20 sets of test data and cost function for training data depending on the number of training pairs after training for 100 rounds with stepsize of 0.1, a learning rate of $\eta = 2$ with a 1-2-1 network and performance on the training data after 100 rounds.

5.4 Learning a Bit Phase Flip

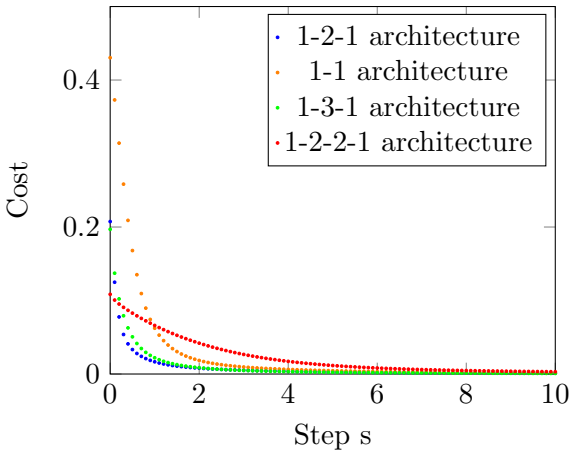
The *bit phase flip* channel combines the two previous flip operations. It flips the bit and the phase. Its operation elements are:

$$E_0 = \sqrt{p}\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad E_1 = \sqrt{1-p}\begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}. \tag{65}$$

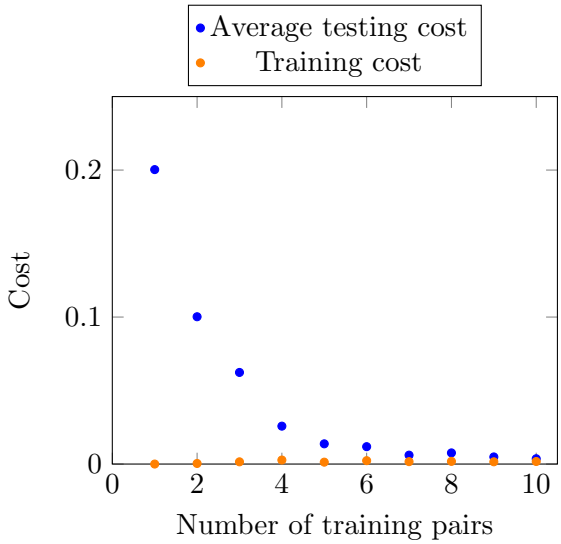
The results for learning the bit phase flip are akin to the ones of the bit flip channel. The learning algorithm has been tested for the same settings as for the previous channels. Higher learning rates decreased the cost function faster (see fig. 9a), the network with a 1-2-1 architecture performed slightly better than the network with a 1-3-1 architecture (see fig. 9b). Moreover, with only a handful of training sets, a trained network is able to generalise up to a degree which is in the range of scattering compared to the performance on training data (see fig. 9c).



(a) Cost function depending on the number of steps in the training algorithm for different learning rates. 500 training rounds to learn a bit phase flip were performed on a 1-2-1 network with a step size of 0.1.



(b) Cost function depending on training steps for different network architectures for learning a bit phase flip. 100 of 500 training rounds with a step size of 0.1 and a learning rate $\eta = 2$ are shown.



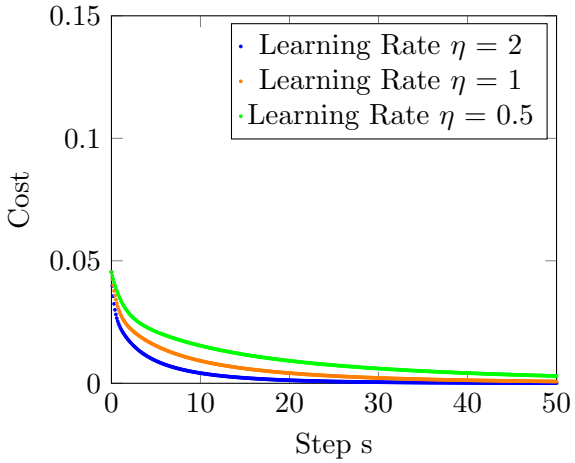
(c) Generalisation for the Bit Phase Flip: Average cost function for 20 sets of test data and cost function for training data depending on the number of training pairs after training for 100 rounds with stepsize of 0.1, a learning rate of $\eta = 2$ with a 1-2-1 network and performance on the training data after 100 rounds.

Figure 9: Results for the bit phase flip channel

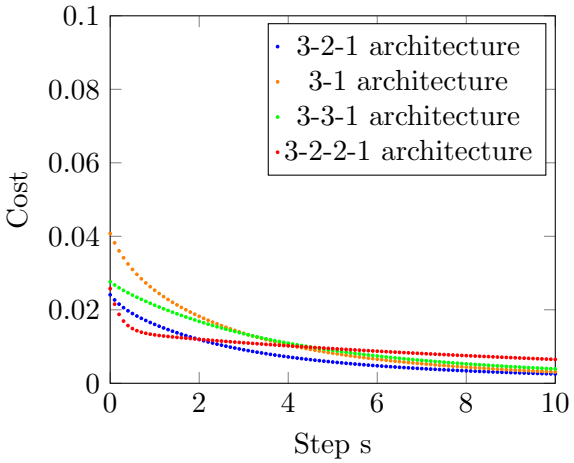
5.5 Learning a Partial Trace Channel

The partial trace has been introduced in section 3.1.3. Making calculations for the partial trace is different to what has been done for the previous channels. Calculating the partial trace for a one qubit input is not very interesting. Hence, the network was trained with a three-qubit input. Two qubits were traced out, leaving a one qubit output.

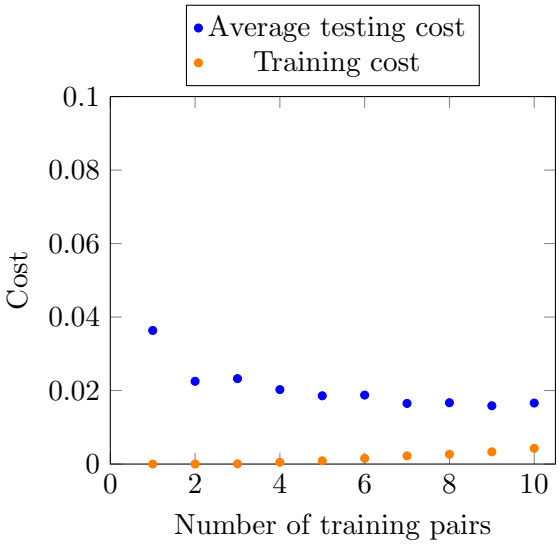
Figure 10 shows the results for the calculations. Higher learning rates decrease the cost faster, even more: For a learning rate of $\eta = 0.5$, 50 steps are not enough to fall below 0.1 in cost. Figure 10b illustrates that the initially fastest network architecture is not always the fastest if it comes to reaching certain thresholds. A 3-2-2-1 architecture enables the fastest learning in the beginning until the step parameter s reaches 2. However, a 3-2-1 architecture is inferior to the other tested network types in the long run. This effect might be a result of the initially randomly generated unitaries for the network: The initial set of unitaries may be already well suited for the task. Therefore, an advantage over other network architectures for which the learning algorithm works faster can be maintained for some steps despite a worse learning performance. Multiple runs for the calculation confirm the assumption: In all runs of the calculation, the 3-2-2-1 network was the slowest. A glance at figure 10c illustrates that the trained networks work well for the partial trace with only a few training examples while lacking improvement with more training data compared to other channels.



(a) Cost function depending on the number of steps in the training algorithm for different learning rates. 500 training rounds to learn the partial trace were performed on a 3-2-1 network with a step size of 0.1.



(b) Cost function depending on training steps for different network architectures for learning the partial trace. 100 of 500 training rounds with a step size of 0.1 and a learning rate $\eta = 2$ are shown.



(c) Generalisation for the Partial Trace: Average cost function for 20 sets of test data and cost function for training data depending on the number of training pairs during training for 100 rounds with stepsize of 0.1, a learning rate of $\eta = 2$ with a 3-2-1 network and performance on the training data after 100 rounds.

Figure 10: Learning and Generalisation behaviour for the partial trace.

5.6 Learning a Depolarising Channel

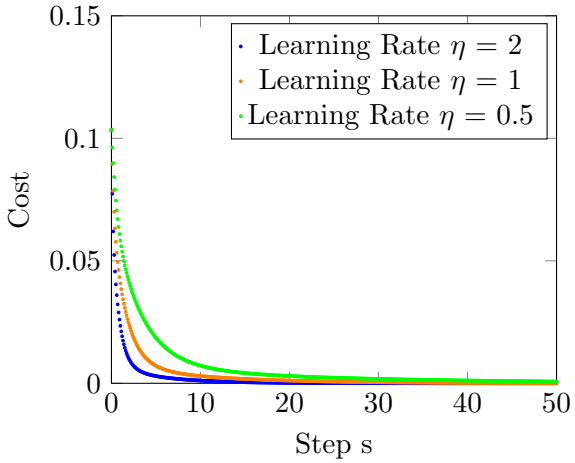
Another type of quantum channel used to model noise is the *depolarising* channel. With probability $1 - p$, the depolarising channel leaves the qubit unchanged, with probability p, it changes the qubits' state to the completely mixed state. The completely mixed state $\rho_{\text{completely mixed}}$ is defined as follows:

$$\rho_{\text{completely mixed}} = \frac{1}{d} \sum_i p_i \left|\Psi_i\right\rangle \left\langle\Psi_i\right| = \frac{1}{d} I. \tag{66}$$

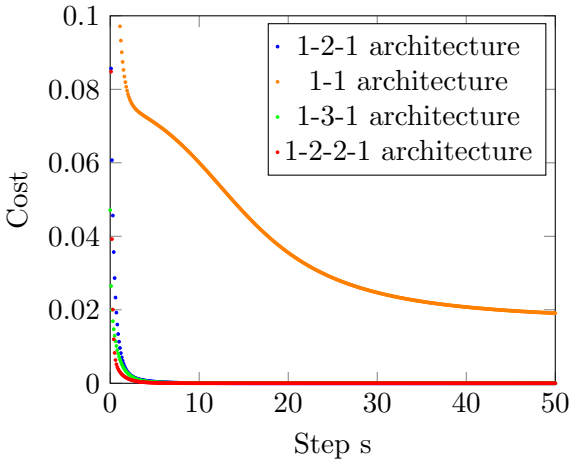
For a single qubit with two basis states, it is $d = 2$. The depolarising channel can therefore be defined as [28]:

$$\mathcal{E}(\rho) = p\frac{I}{2} + (1 - p)\rho \tag{67}$$

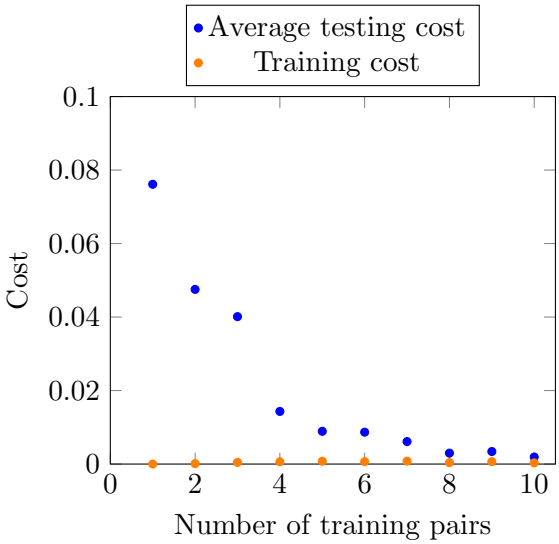
The depolarising channel is the first example where the network needs at least one hidden layer to learn the task properly. Figure 12b shows that a 1-1 network architecture performs significantly worse than other network architectures. Apart from the network architectures, the learning behaviour and the generalisation behaviour are similar to the bit flip, phase flip and bit phase flip channels.



(a) Cost function depending on the number of steps in the training algorithm for different learning rates. 500 training rounds to learn the depolarising channel were performed on a 1-2-1 network with a step size of 0.1.



(b) Cost function depending on training steps for different network architectures for learning the depolarising channel. 500 training rounds with a step size of 0.1 and a learning rate $\eta = 2$ are shown.



(c) Generalisation for the Depolarising Channel: Average cost function for 20 sets of test data and cost function for training data depending on the number of training pairs during training for 100 rounds with stepsize of 0.1, a learning rate of $\eta = 2$ with a 1-2-1 network and performance on the training data after 100 rounds.

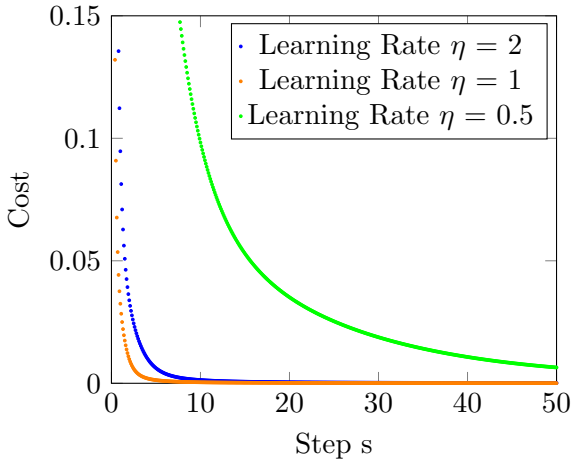
Figure 11: Learning and Generalisation behaviour for the Depolarising Channel.

5.7 Learning an Amplitude Damping Channel

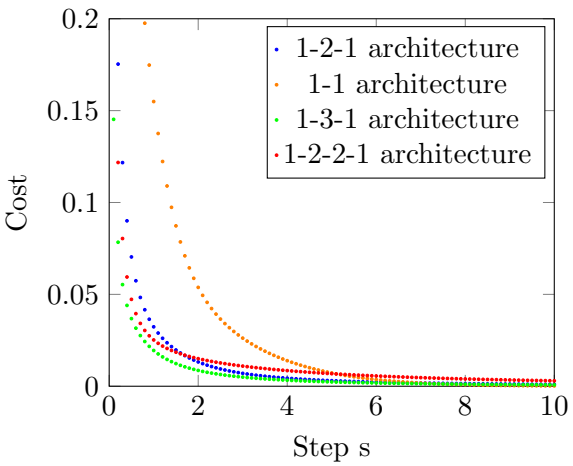
The energy of a state of a qubit has not been considered so far. In real-world applications, however, it is essential to consider the energy. Qubits are realized through quantum mechanical systems with two states, for instance, the energy level of a particle. A change in this energy level corresponds to the emission or absorption of a photon. Consider the $|0\rangle$ state as the ground state and the $|1\rangle$ state as the excited state. Interactions with the environment may result in the dissipation of energy. In the example of a particle, this means emitting a photon. The *amplitude damping channel* models this loss of a photon. The probability for the loss of a proton is given by $\gamma = \sin^2\theta$, the operation elements are given as [28]:

$$E_0 = \sqrt{p}\begin{bmatrix}1 & 0\\ 0 & \sqrt{1-\gamma}\end{bmatrix} \quad E_1 = \sqrt{1-p}\begin{bmatrix}0 & \sqrt{\gamma}\\ 0 & 0\end{bmatrix}. \tag{68}$$

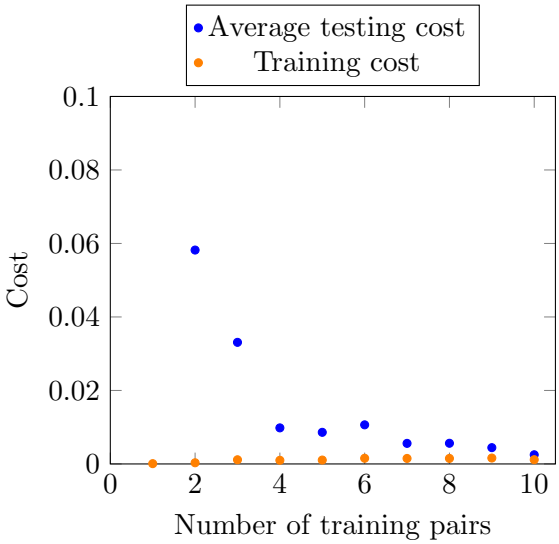
The amplitude damping channel can be taught to the network equally well as the other tested quantum channels. Higher learning rates yield faster reductions (see fig -12a) in the cost function up to a level where the cost function fluctuates for to large learning rates, as discussed in section 5.2. The learning rate of 0.5 further seems relatively low since there is a difference in the cost for higher learning rates after 500 steps. Nevertheless, the cost for a learning rate of 0.5 still decreases. A network with a 1-3-1 architecture is slightly superior to a 1-2-1 network (see fig. 12b), suggesting the addition of a qubit is beneficiary to the learning performance. The generalisation behaviour is also similar to the behaviour for the bit flip, phase flip or bit phase flip. A single-digit amount of training pairs is sufficient to train the network to perform almost as well on test data as on training data (see 12c).



(a) Cost function depending on the number of steps in the training algorithm for different learning rates. 500 training rounds to learn the amplitude damping channel were performed on a 1-2-1 network with a step size of 0.1.



(b) Cost function depending on training steps for different network architectures for learning the amplitude damping channel. 100 from 500 training rounds with a step size of 0.1 and a learning rate $\eta = 2$ are shown



(c) Generalisation for the Amplitude Damping Channel: Average cost function for 20 sets of test data and cost function for training data depending on the number of training pairs during training for 100 rounds with stepsize of 0.1, a learning rate of $\eta = 2$ with a 1-2-1 network and performance on the training data after 100 rounds.

Figure 12: Amplitude Damping: Learning and Generalisation behaviour

6 Conclusion

Having developed a quantum neural network and a learning algorithm, Beer et al. [4] already showed the remarkable ability of the quantum neural network to generalise. This thesis addresses two remaining questions. First: Does a slightly adapted learning algorithm for mixed states as network output impair the networks' generalisation behaviour? Second: Can such an altered learning algorithm train the network to perform quantum channel operations acting on mixed states or generating mixed quantum states?

Section 5.1 denies the first question. For random unitary operators, there is almost no difference in generalisation behaviour. The chosen approach to alter the learning algorithm is therefore justified.
Sections 5.2, 5.3, 5.4, 5.6 and 5.7 tested the network's ability to perform quantum channel operations with success. The ability to generalise manifested for every tested quantum channel. A single-digit number of training pairs suffices for the network to perform well on previously unseen test data. Results for the partial trace in section 5.5 are not as promising as for the other quantum channels. With an increasing amount of test data, however, the cost for test data comes closer to the cost for training data after training.

Additionally, this thesis examined the networks' learning behaviour for each quantum channel depending on certain network parameters. In general, higher learning rates increased the declination speed of the cost function depending on the step parameter. In other words, with higher learning rates, the network could learn a task in fewer training rounds. Moreover, section 5.2 exemplary illustrates that learning rates cannot be chosen arbitrarily high because of the risk of overshooting. The cost as a function of the training step is not necessarily a monotonously decreasing function. Learning rates η of 2 to 5 offered rapid declines in the cost function while staying monotonous.

Network Architectures have a great influence on the learning behaviour. Despite neither offering a concise result nor deriving a rule of thumb on which type of network performs best, it can be stated that it is worth experimenting with network architectures for each problem. More elaborate networks do not always perform better, take sections 5.2, 5.4 and 5.7 for instance.

In summary, the results of Beer et al. [4] are transferable to the more general learning algorithm derived in this thesis. It provides comparable generalisation patterns and the ability to simulate quantum channels for mixed quantum states. Furthermore, this thesis offers findings on tuning the network parameters for various quantum channels.

References

[1] Nahed Abdelgaber and Chris Nikolopoulos. "Overview on Quantum Computing and its Applications in Artificial Intelligence". In: *2020 IEEE Third International Conference on Artificial Intelligence and Knowledge Engineering (AIKE)*. 2020, pp. 198–199. DOI: 10.1109/AIKE48582.2020.00038.

[2] Biondi et. al. *Quantum computing use cases are getting real—what you need to know*. 2021. URL: https://www.mckinsey.com/business-functions/mckinsey-digital/our-insights/quantum-computing-use-cases-are-getting-real-what-you-need-to-know.

[3] Kerstin Beer et al. *Code for Feedforward Quantum Neural Networks*. Jan. 2022. URL: https://github.com/qigitphannover/DeepQuantumNeuralNetworks/blob/d1f8dba59fb7bc8b6f5c770f6ff3fd901d0f4b62/DQNN_basic.ipynb.

[4] Kerstin Beer et al. "Training deep quantum neural networks". In: *Nature Communications* 11.1 (Feb. 2020). DOI: 10.1038/s41467-020-14454-2.

[5] Paul Benioff. "The computer as a physical system: A microscopic quantum mechanical Hamiltonian model of computers as represented by Turing machines". In: *Journal of Statistical Physics* 22.5 (May 1980), pp. 563–591. DOI: 10.1007/bf01011339.

[6] Giuseppe Carleo and Matthias Troyer. "Solving the quantum many-body problem with artificial neural networks". In: *Science* 355.6325 (Feb. 2017), pp. 602–606. DOI: 10.1126/science.aag2302.

[7] Hsun-Hsien Chang. "An Introduction to Error-Correcting Codes: From Classical to Quantum". In: (Feb. 2006). arXiv: quant-ph/0602157 [quant-ph].

[8] Bec Crew. "Google Scholar reveals its most influential papers for 2021". In: *Nature* (2021). URL: https://www.natureindex.com/news-blog/google-scholar-reveals-most-influential-papers-research-citations-twenty-twenty-one.

[9] Mike Demler. "Mythic multiplies in a flash". In: *Microprocesser Report* (2018).

[10] Yuxuan Du et al. "The Expressive Power of Parameterized Quantum Circuits". In: *Phys. Rev. Research 2, 033125 (2020)* (Oct. 2018). DOI: 10.1103/PhysRevResearch.2.033125. arXiv: 1810.11922 [quant-ph].

[11] *Duden - Die deutsche Rechtschreibung*. Bibliograph. Instit. GmbH, Aug. 2020. 1294 pp. ISBN: 978-3-411-04018-6. URL: https://www.ebook.de/de/product/39011031/duden_die_deutsche_rechtschreibung.html.

[12] Richard P. Feynman. "Simulating physics with computers". In: *International Journal of Theoretical Physics* 21.6-7 (June 1982), pp. 467–488. DOI: 10.1007/bf02650179.

[13] Elizabeth Gibney. "Hello quantum world! Google publishes landmark quantum supremacy claim". In: *Nature* 574.7779 (Oct. 23, 2019), pp. 461–462. DOI: 10.1038/d41586-019-03213-z.

[14] Nicolas Gisin et al. “Quantum cryptography”. In: *Reviews of Modern Physics* 74.1 (Mar. 8, 2002), pp. 145–195. DOI: 10.1103/revmodphys.74.145.

[15] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016.

[16] Paul Hiemstra. *Why do humans learn so much faster than machine learning models?* Sept. 2021. URL: https://towardsdatascience.com/why-do-humans-learn-so-much-faster-than-machine-learning-models-cab0f5c111b6.

[17] Hsin-Yuan Huang et al. “Provably efficient machine learning for quantum many-body problems”. In: (June 2021). arXiv: 2106.12627 [quant-ph].

[18] IEA. *Key World EnergyStatistics 2021*. Sept. 2021. URL: shorturl.at/fAU25.

[19] Aishwarya Jhanwar and Manisha J. Nene. “Enhanced Machine Learning using Quantum Computing”. In: *2021 Second International Conference on Electronics and Sustainable Communication Systems (ICESC)*. IEEE, Aug. 2021. DOI: 10.1109/icesc51422.2021.9532638.

[20] Richard Jozsa. “Fidelity for Mixed Quantum States”. In: *Journal of Modern Optics* 41.12 (Dec. 1994), pp. 2315–2323. DOI: https://doi.org/10.1080/09500349414552171.

[21] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. DOI: 10.48550/ARXIV.1412.6980.

[22] Fuad Kittaneh. “Inequalities for the Schatten-p-norm”. In: *Glasgow Mathematical Journal* 29.1 (Jan. 1987), pp. 99–104. DOI: 10.1017/s0017089500006716.

[23] Rodolfo Llinás. “The contribution of Santiago Ramon y Cajal to functional neuroscience”. In: *Nature Reviews Neuroscience* 4.1 (Jan. 2003), pp. 77–80. DOI: 10.1038/nrn1011.

[24] Yu Manin. *Computable and Noncomputable*. 1980. URL: https://web.archive.org/web/20130510173823/http://publ.lib.ru/ARCHIVES/M/MANIN_Yuriy_Ivanovich/Manin_Yu.I._Vychislimoe_i_nevychislimoe.(1980).[djv].zip.

[25] Seymour A. Papert Marvin Minsky. *Perceptrons*. MIT Press Ltd, Sept. 2017. 316 pp. ISBN: 0262534770. URL: https://www.ebook.de/de/product/28875910/marvin_minsky_seymour_a_papert_perceptrons.html.

[26] Warren Mcculloch and Walter Pitts. “A logical calculus of the ideas immanent in nervous activity”. In: *The Bulletin of Mathematical Biophysics* 5.4 (Dec. 1943), pp. 115–133. DOI: 10.1007/bf02478259.

[27] Pranav Santosh Menon and M. Ritwik. “A Comprehensive but not Complicated Survey on Quantum Computing”. In: *IERI Procedia* 10 (2014), pp. 144–152. DOI: 10.1016/j.ieri.2014.09.069.

[28] Isaac L. Chuang Michael A. Nielsen. *Quantum Computation and Quantum Information*. Cambridge University Pr., Dec. 2010. ISBN: 1107002176. URL: https://www.ebook.de/de/product/13055864/michael_a_nielsen_isaac_l_chuang_quantum_computation_and_quantum_information.html.

[29] Azalia Mirhoseini et al. "A graph placement methodology for fast chip design". In: *Nature* 594.7862 (June 9, 2021), pp. 207–212. DOI: 10.1038/s41586-021-03544-w.

[30] John Naughton. *Can the planet really afford the exorbitant power demands of machine learning?* Nov. 2019. URL: https://www.theguardian.com/commentisfree/2019/nov/16/can-planet-afford-exorbitant-power-demands-of-machine-learning.

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

[32] Román Orús, Samuel Mugel, and Enrique Lizaso. "Quantum computing for finance: Overview and prospects". In: *Reviews in Physics* 4 (Nov. 2019), p. 100028. DOI: 10.1016/j.revip.2019.100028.

[33] Carlos Outeiral et al. "The prospects of quantum computing in computational molecular biology". In: *WIREs Computational Molecular Science* 11.1 (May 2020). DOI: 10.1002/wcms.1481.

[34] Aditya Ramesh et al. "Zero-Shot Text-to-Image Generation". In: *CoRR* abs/2102.12092 (2021). arXiv: 2102.12092. URL: https://arxiv.org/abs/2102.12092.

[35] Nils Renziehausen. *Code Bachelor Thesis*. May 2022. URL: https://github.com/renziehausen/Bachelor_Thesis.

[36] Lorenzo Rosasco et al. "Are Loss Functions All the Same?" In: *Neural Computation* 16.5 (May 1, 2004), pp. 1063–1076. DOI: 10.1162/089976604773135104.

[37] F. Rosenblatt. "The perceptron: A probabilistic model for information storage and organization in the brain." In: *Psychological Review* 65.6 (1958), pp. 386–408. DOI: 10.1037/h0042519.

[38] David Rumelhart, Geoffrey Hinton, and Ronald Williams. "Learning representations by back-propagating errors". In: *Nature* 323.6088 (Oct. 1986), pp. 533–536. DOI: 10.1038/323533a0.

[39] Saptarshi Sahoo et al. "A critical overview on Quantum Computing". In: *Journal of Quantum Computing* 2.4 (2020), pp. 181–192. DOI: 10.32604/jqc.2020.015688.

[40] Iqbal H. Sarker. "Machine Learning: Algorithms, Real-World Applications and Research Directions". In: *SN Computer Science* 2.3 (Mar. 2021). DOI: 10.1007/s42979-021-00592-x.

[41] Frank Schwierz and Juin J. Liou. "Status and Future Prospects of CMOS Scaling and Moore's Law - A Personal Perspective". In: *2020 IEEE Latin America Electron Devices Conference (LAEDC)*. IEEE, Feb. 2020. DOI: 10.1109/laedc49063.2020.9073539.

[42] Neha Sharma, Reecha Sharma, and Neeru Jindal. "Machine Learning and Deep Learning Applications-A Vision". In: *Global Transitions Proceedings* 2.1 (June 2021), pp. 24–28. DOI: 10.1016/j.gltp.2021.01.004.

[43] W. K. Wootters and W. H. Zurek. "A single quantum cannot be cloned". In: *Nature* 299.5886 (Oct. 1982), pp. 802–803. DOI: 10.1038/299802a0.