Gottfried Wilhelm Leibniz Universität Hannover
Fakultät für Mathematik und Physik
Institut für Theoretische Physik

Quantum Machine Learning and Classification

Bachelor Thesis

Author:	Jan Hendrik Pfau
Matriculation Number:	10002345
E-mail:	jan_hendrik.pfau@t-online.de
Degree Programme:	B.Sc. Physics
Date:	September 30, 2020
Supervisor:	Prof. Dr. Tobias J. Osborne

Contents

1 Introduction

It has been an endeavor of human beings for a long time to examine data with the aim of detecting patterns therein. With the emergence of digital computers, it has then become possible to build such techniques with which one can automatically analyze data and enable computer systems to automatically improve. The algorithms learn these improvements from previous observations. Machine learning is the field dealing with these challenges. Examples of the widespread use it has achieved are natural language processing, image recognition, and robot control. Implementing biologically-inspired artificial neural networks is a possibility to realize the computer systems used for machine learning tasks. Building on their concept, deep learning can solve complicated problems by disassembling them into simpler ones [3, 10, 13, 14].

Another field of research with enormous progress in the last decades is given by quantum information science. In particular, the experimental realization of quantum computers has experienced many advances. But also in the area of classically simulating quantum systems, much knowledge has been gained [15].

Quantum machine learning connects the fields of quantum computing and classical machine learning and thereby examines how insights into one can be applied in the other. The use of quantum algorithms, which are performed on quantum computers and can be used for quantum machine learning tasks, is an example for such a transfer of knowledge. It offers the possibility to outperform classical algorithms and hence to provide a speedup compared to them [3, 6].

This bachelor thesis deals with the application of machine learning algorithms to the task of quantum classification. For this purpose, the quantum version of neural networks defined in [2], a publication by the Quantum Information Group at the Leibniz University Hannover, is used.

In Chapter 2, we introduce classical machine learning by presenting neural networks and the backpropagation algorithm. Chapter 3 deals with an approach to quantum machine learning via the quantum neural network. The concept of quantum classification is described in Chapter 4. Building on this, Chapter 5 discusses the application of the quantum neural network to quantum classification. A conclusion can finally be found in the last chapter.

2 Classical Machine Learning

Machine learning tasks can be divided into varied categories, among them supervised learning and unsupervised learning. In supervised learning, a set of training data is provided with labels indicating the category of a sample. It is desired to learn how to label test data not belonging to the training set. In unsupervised learning, the training sets are not equipped with labels. The goal here is to group the data into clusters or to find the categories related to the data [3, 4, 10].

2.1 Neural Networks

First, we want to introduce neural networks as used in [14]. Consider a perceptron, a simple implementation of an artificial neuron. It takes m inputs x_j, which are all either 0 or 1, and computes an output, either 0 or 1 as well. The calculation of the output makes use of real numbers w_j called weights. To each input, one of these weights is assigned, such that its importance can be illustrated. The value of the perceptron's output then depends on the size of $\sum_{j=1}^{m} w_j x_j$. If this sum is greater than a specific threshold value, the output is 1, otherwise it is 0. By denoting the inputs as a vector $\mathbf{x}$ and the weights as $\mathbf{w}$ and by introducing the bias b as the negative of the threshold, the output $f(\mathbf{x})$ can be described as

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \\ 0 & \text{if } \mathbf{w} \cdot \mathbf{x} + b \leq 0 \end{cases}. \tag{1}$$

We can now construct a neural network made out of perceptrons. It contains several layers, of which the first one is the input layer delivering the inputs for the first of the L hidden layers of perceptrons. Each of these layers hands its outputs to the inputs of the next layer, until finally the perceptrons in the output layer describe the output of the network. Learning with such a network corresponds to changing the weights and the biases of the perceptrons with the aim that for all inputs the outputs of the network get as close as possible to the desired form.

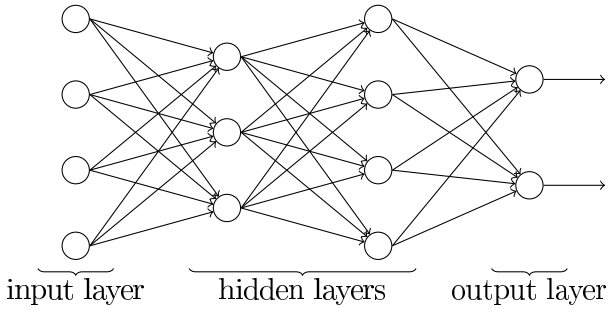


Figure 1: A neural network with two hidden layers.

However, small changes in some weights or a bias might influence the behaviour of the whole network in a large magnitude because of the binary inputs and outputs. Learning

the appropriate weights and biases of the networks hence becomes more difficult. This problem can be avoided if the perceptrons are replaced by so-called sigmoid neurons. A sigmoid neuron is a different version of artificial neurons differing from a perceptron only in the possible inputs and outputs. These can take any value in the interval $[0, 1]$, and the output $f(\mathbf{x})$ is computed according to the sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$ by

$$f(\mathbf{x}) = \frac{1}{1 + \exp(-\mathbf{w} \cdot \mathbf{x} - b)}. \tag{2}$$

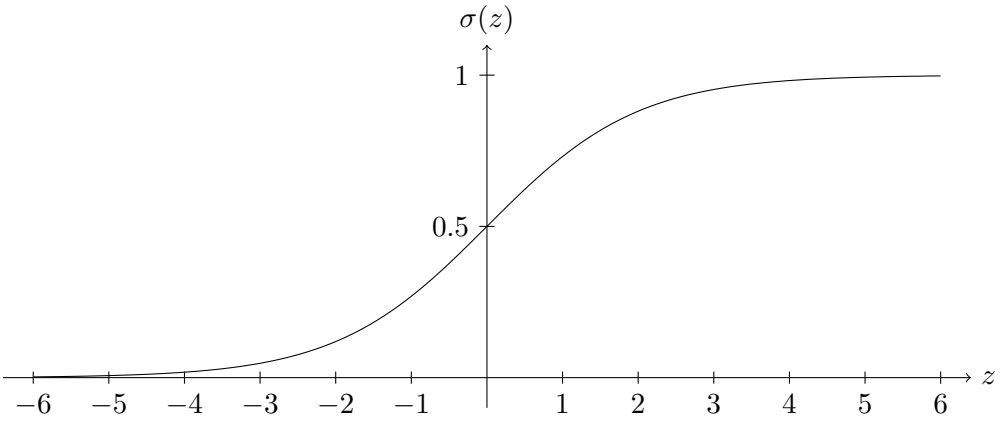


Figure 2: Sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$.

If the argument of the exponential function has a large absolute value, the output of the sigmoid neuron approaches one of the values 0 or 1, depending on the sign of the argument. In this case the sigmoid neuron's behaviour is very similar to the perceptron's one. But there also is an advantageous difference, which is that small changes in the weights or the bias of a sigmoid neuron result in a likewise small change of the output, in contrast to the perceptron's situation. This change is linear in the changes of the w_j and b, as we can also recognize in the output's differential

$$\mathrm{d}f = \sum_j \frac{\partial f}{\partial w_j} \mathrm{d}w_j + \frac{\partial f}{\partial b} \mathrm{d}b. \tag{3}$$

2.2 Gradient Descent

We now want to discuss the question how a neural network of artificial neurons like sigmoid neurons can be made learn some specific task. For this purpose, we examine the case of supervised learning, i.e. applying a set of training data equipped with the desired output for each input. With the aid of the training data, the weights and biases of the neurons are adjusted as described in [14], until the network provides an appropriate approximation of the correct solution for as many inputs as possible.

The quality of the network's performance can be described via a cost function. The cost function indicates how close the output is to the correct output belonging to the input and can be defined as

$$C = \frac{1}{2N} \sum_{\mathbf{x}} \|\mathbf{y}(\mathbf{x}) - \mathbf{a}\|^2, \tag{4}$$

where $\mathbf{x}$ stands for the inputs from the training data, N is the number of elements of the set of used training data and $\mathbf{y}(\mathbf{x})$ is the desired output. The actual output is denoted by $\mathbf{a}$. As we wish the difference $\mathbf{y}(\mathbf{x}) - \mathbf{a}$ to be close to zero for all inputs $\mathbf{x}$, the goal is to reach $C \approx 0$, which means minimizing the cost function by finding appropriate values for the weights and the biases. However, computing the minimum analytically turns out not to be wise, because the cost function depends on a large number of variables. Instead, this minimization can be achieved by gradient descent.

Gradient descent is an algorithm which enables performing optimization and is commonly used for neural networks [18]. Starting with some configuration of the weights and biases, these parameters are repeatedly changed here by a small amount via an update rule. This rule makes use of the gradient of the cost function. Consider for this the function C as depending on the variables v_j, $j = 1, \ldots, m$. A change in the variables v_j induces a change in C

$$\Delta C \approx \sum_{j=1}^{m} \frac{\partial C}{\partial v_j} \Delta v_j. \tag{5}$$

Using the gradient of the cost function

$$\nabla C = \left(\frac{\partial C}{\partial v_1}, \ldots, \frac{\partial C}{\partial v_m} \right)^T, \tag{6}$$

this change can be written as

$$\Delta C \approx \nabla C \cdot \Delta \mathbf{v}, \tag{7}$$

where $\Delta \mathbf{v} = (\Delta v_1, \ldots, \Delta v_m)^T$ denotes the vector of changes in the variables. This equation allows to detect a suitable choice of

$$\Delta \mathbf{v} = -\eta \nabla C \tag{8}$$

for these changes with the learning rate $\eta > 0$. As this yields $\Delta C \approx -\eta \|\nabla C\|^2$, it is thus ensured that $\Delta C \leq 0$, so the function does not increase, but always decreases. Hence, we can determine the update rule for $\mathbf{v} = (v_1, \ldots, v_m)^T$ as

$$\mathbf{v} \mapsto \mathbf{v}' = \mathbf{v} - \eta \nabla C \tag{9}$$

with the effect that by frequently updating the parameters we can hope to find a global minimum of the function. Transferred to the cost function with the weights and the

biases as variables, this yields

$$w_k \mapsto w_k' = w_k - \eta \frac{\partial C}{\partial w_k}, \tag{10}$$

$$b_l \mapsto b_l' = b_l - \eta \frac{\partial C}{\partial b_l}. \tag{11}$$

The training sets used for learning a task and finding an appropriate approximation are often very large. To avoid the resulting long computation time, we can benefit from stochastic gradient descent, which is an extended version of gradient descent. For this, a small set of n randomly chosen training samples $\{X_1, \ldots, X_n\}$ called mini-batch is drawn from the training set and then employed for estimating the gradient ∇C. This estimate can be written as the average over the gradients ∇C_{X_j} computed for each of the training samples,

$$\nabla C \approx \frac{1}{n} \sum_{j=1}^{n} \nabla C_{X_j}. \tag{12}$$

2.3 The Backpropagation Algorithm

In order to really apply gradient descent, it is required to discuss how the gradient of the cost function can be computed. The partial derivatives occuring in that gradient can be calculated by the backpropagation algorithm, see again [14].

The backpropagation algorithm requires two main assumptions about the cost function. The first of these assumptions states that it needs to be expressible as an average, which is performed over the individual cost functions assigned to each of the training samples, i.e. $C = \frac{1}{n} \sum_x C_x$. Hence, the partial derivatives $\partial C / \partial w_{jk}^l$ and $\partial C / \partial b_j^l$ can be computed as the average of those for each sample. The second assumption states that the cost needs to be expressible as a function of the network's outputs.

Let w_{jk}^l be the weight describing the connection between the k^{th} neuron in the $(l-1)^{\text{th}}$ layer and the j^{th} neuron in the l^{th} layer. Similarly, let b_j^l be the bias of the j^{th} neuron in the l^{th} layer. The activation a_j^l, i.e. the output of the j^{th} neuron in the l^{th} layer, can then be expressed by the activations in the $(l-1)^{\text{th}}$ layer using the sigmoid function:

$$a_j^l = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l \right). \tag{13}$$

With the weighted input z_j^l, the error δ_j^l of the j^{th} neuron in the l^{th} layer can be defined as

$$\delta_j^l = \frac{\partial C}{\partial z_j^l}. \tag{14}$$

There are four fundamental equations on which backpropagation is based. These enable the computation of the gradient. The first one is an equation for the errors in the output

layer and reads

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L). \tag{15}$$

The second equation describes the errors δ_j^l as a function of the errors δ_j^{l+1} in the next layer as

$$\delta_j^l = \sigma'(z_j^l) \cdot \sum_i w_{ij}^{l+1} \delta_i^{l+1}. \tag{16}$$

Note that w_{ij}^{l+1} corresponds to the weight connecting the j^{th} neuron in layer l and the i^{th} neuron in layer $l+1$. The first two equations thus provide us a possibility to calculate the errors of all neurons in the network by starting with the last layer and continuing with the preceding layer. The third equation states that the partial derivative of the cost with respect to any bias in the network is

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l. \tag{17}$$

Finally, the fourth and last equation provides the computation of the derivative of the cost with respect to any weight as

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l. \tag{18}$$

This enables us to name the individual steps of the backpropagation algorithm. First, the activations for the input layer need to be set by using a training sample, followed by the next step, feedforward, where the activations for each layer are calculated from the previous ones successively. After reaching the last layer, the output errors are computed via the first equation behind backpropagation. This is followed by the backpropagation of the errors via the second equation. Ultimately, the components of the gradient of the cost function, $\partial C/\partial w_{jk}^l$ and $\partial C/\partial b_j^l$, are given by equations three (17) and four (18).

2.4 Deep Neural Networks

A neural network featuring a structure with two or more hidden layers is also known as a deep neural network. When training such a deep neural network, a problem that can occur is the unstable gradient problem. This is expressed by the circumstance that the networks' layers are likely to learn at different speeds because the partial derivatives, which are used for updating the parameters, tend to change their size when we propagate to the early layers. Usually, the neurons in the early layers do not learn as fast as the neurons in later layers. Their derivatives then become smaller overall. We call this case the vanishing gradient problem. Alternatively, it is also possible that the gradient takes higher values in earlier layers. This case is called the exploding gradient problem.

The reason for this behaviour can be found in the expressions for $\partial C/\partial w_{jk}^l$ and $\partial C/\partial b_j^l$. As we have already seen in equations 15 to 18, they can be described by the errors of

the subsequent layers $l+1, \ldots, L$. When formulated in terms of the weights, one can obtain expressions containing products whose factors are these weights of the neurons and the derivatives of the sigmoid function at the weighted inputs. This derivative reads $\sigma'(z) = \frac{e^{-z}}{(1+e^{-z})^2}$ and always has an absolute value less than $1/4$ as it reaches its maximum at $z = 0$ with $\sigma'(0) = 1/4$. Moreover, the weights are initialized at random, often according to a Gaussian distribution with mean 0 and standard deviation 1, so their absolute value is less than 1 in most cases. The closer a neuron is to the input layer, the more of such factors are multiplied. The corresponding components of the gradient used for updating therefore tend to be smaller than they are for later layers. This results in the vanishing gradient problem.

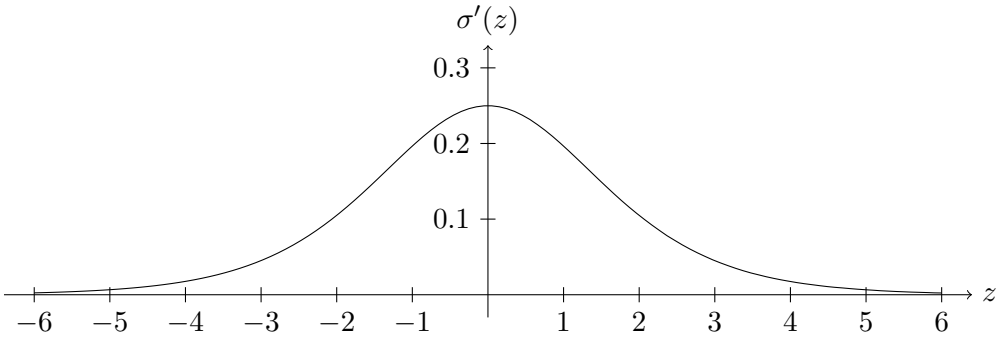


Figure 3: Derivative of the sigmoid function $\sigma'(z) = \frac{e^{-z}}{(1+e^{-z})^2}$.

If, on the contrary, the weights are chosen to be large and the biases are such that the weighted inputs are close to 0, the components of the gradient grow exponentially when reaching the neurons in earlier layers. This results in the exploding gradient problem. Again, the cause is that a product of factors from the later layers is used for computing the gradient [14].

2.5 Classification and Support Vector Machines

Classification is a frequently occuring task in machine learning. It is desired here to predict the class of an input by learning a function that maps this input into the set of classes or finds a corresponding probability distribution. For this purpose, the algorithm observes a training dataset with objects whose classes are known. This training dataset has the form $D_n = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$, where n is the number of samples, $\mathbf{x}_i$ is the vector belonging to input $i \in \{1, \ldots, n\}$ and y_i is the respective class. In case of binary classification, there are two possible classes like for example -1 and $+1$ [9, 10].

The application of a support vector machine (SVM) is a possibility to perform such a classification task. The SVM learns a linear function $\mathbf{w} \cdot \mathbf{x} + b$ with a vector $\mathbf{w}$ and a scalar b. For a vector $\mathbf{x}$ of observations, the output of the SVM will be the class $+1$ if $\mathbf{w} \cdot \mathbf{x} + b > 0$, if on the contrary $\mathbf{w} \cdot \mathbf{x} + b < 0$, the output will be the class -1.

The use of an SVM does also allow learning a function that is nonlinear in $\mathbf{x}$ via the

kernel trick. For this, we note that the linear function can be expressed as

$$\mathbf{w} \cdot \mathbf{x} + b = b + \sum_{i=1}^{n} \alpha_i \mathbf{x} \cdot \mathbf{x}^{(i)}, \tag{19}$$

where each $\mathbf{x}^{(i)}$ is a training sample and each α_i a coefficient. The idea is now to replace $\mathbf{x}$ with a nonlinear feature function $\phi(\mathbf{x})$ and the dot product with the kernel

$$k(\mathbf{x}, \mathbf{x}^{(i)}) = \phi(\mathbf{x}) \cdot \phi(\mathbf{x}^{(i)}). \tag{20}$$

The decision to which class the input $\mathbf{x}$ belongs will then be made via the sign of

$$f(\mathbf{x}) = b + \sum_{i} \alpha_i k(\mathbf{x}, \mathbf{x}^{(i)}). \tag{21}$$

So the procedure consists in transferring all inputs into a high-dimensional feature space via ϕ and finding a hyperplane dividing this space into two parts [21].

An SVM can learn the coefficients α_i in such a way that most of them are zero. Only those training samples with nonzero $\alpha_i = 0$ have an impact on the classification of new inputs then, which is why they are called support vectors [10].

3 Quantum Machine Learning

In the field of quantum machine learning, we can distinguish different types of learning tasks. One of the considered categories is the used computer, which can be either classical or quantum. Moreover, the analyzed data can also be either classical or quantum [3]. Thus, we can identify four different combinations [7], which can be denoted by "CC", "CQ", "QC", and "QQ". The first letter shall refer here to the kind of data, the second one to the algorithm, see [2]. Classical machine learning is therefore described by "CC". The other three combinations are the approaches for quantum machine learning. More precisely, improvements of quantum tasks via classical machine learning belong to the "QC" scenario. An example for this is the many-body problem in quantum physics, where an artificial neural network can be used to learn the wave function [5]. "CQ" offers speedups of classical machine learning by applying quantum computing devices. For instance, it is possible to accelerate algorithms for clustering tasks in unsupervised learning [1]. Finally, "QQ" examines the use of quantum algorithms for tasks with quantum data.

3.1 Quantum Neural Network

We want to focus on the "QQ" approach now. When the neural network used in classical machine learning receives quantum states as inputs and is also supposed to output quantum states, a problem arises. According to the no-cloning theorem, it is impossible to create an identical copy of an unknown quantum state [23]. But the output of a neuron is often used for several other neurons in the next layer and is thus needed more then once. Therefore, a quantum generalization of artificial neurons and neural networks avoiding this problem first needs to be developed.

A version of a quantum perceptron that has been developed is given in [20]. This is a perceptron implemented with a sigmoid activation function. A network of such perceptrons can approximate continuous functions. However, we concentrate on a different approach and consider the quantum neural network as introduced in [2].

First, it is appropriate to discuss the concept of quantum circuits as it is used for the generalization of the neural network. Regarding a classical computer, we find that it realizes an electrical circuit consisting of wires and logic gates. Quantum computers on the other hand can be modeled via a quantum circuit. Its components are quantum gates operating on the quantum information [15].

We continue now with the modification of the concept of artificial neurons. One of the possibilities for this consists in defining the quantum version of the perceptron as an arbitrary unitary operator which acts on m input qubits and, in our case, one output qubit. We write ρ_{in} for the mixed state of the input qubits, while the product state of the output qubit is denoted by $|0\cdots 0\rangle_{\text{out}}$. The operators' parameters include the weights and biases of the perceptrons arising as in the classical analogue.

This definition of the quantum neuron allows specifying the structure of the quantum neural network (QNN) similar to the classical neural network. A QNN is declared as a quantum circuit of quantum neurons. These are arranged into an input layer, L hidden layers and an output layer just as in the classical case. By multiplying the operators U_j^l

of the individual perceptrons j of a layer l, unitaries U^l describing the whole layer can be constructed. These layer unitaries U^l in turn can be merged into an operator describing the whole QNN by $\mathcal{U} = U^{\text{out}} U^L \cdots U^1$. Thus the output of the network reads

$$\rho^{\text{out}} = \text{tr}_{\text{in,hid}} \left(\mathcal{U} \left(\rho^{\text{in}} \otimes |0 \cdots 0\rangle_{\text{hid,out}} \langle 0 \cdots 0| \right) \mathcal{U}^\dagger \right). \tag{22}$$

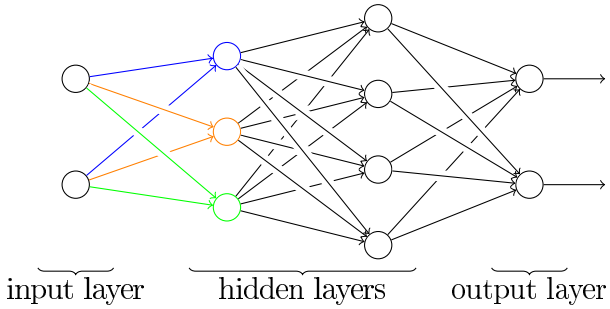


Figure 4: A quantum neural network with two hidden layers. The input layer and the output layer need to have the same number of neurons. As the unitaries of the neurons do not commute, we determine that they are applied from top to bottom. In this example (U_1^1 blue, U_2^1 orange, U_3^1 green), the first layer unitary is $U^1 = U_3^1 U_2^1 U_1^1$.

It is important to note that the network's output can also be described by completely positive maps $\mathcal{E}^l$, each of which characterizes the transition from one layer to another. These transition maps are applied successively, such that the output is

$$\rho^{\text{out}} = \mathcal{E}^{\text{out}}(\mathcal{E}^L(\ldots \mathcal{E}^2(\mathcal{E}^1(\rho^{\text{in}}))\ldots)) \tag{23}$$

with

$$\mathcal{E}^l\left(X^{l-1}\right) = \text{tr}_{l-1}\left(\prod_{j=m_l}^{1} U_j^l \left(X^{l-1} \otimes |0\ldots0\rangle_l \langle 0\ldots0| \right) \prod_{j=1}^{m_l} U_j^{l\,\dagger} \right) \tag{24}$$

and the number m_l of neurons in the corresponding layer. This circumstance makes clear that in a network of this design, information is led from input to output, such that it can be regarded as a quantum feedforward neural network.

3.2 The Learning Task

In the supervised learning task considered in [2], the training data consists of pairs of pure quantum states, $\left(|\phi_x^{\text{in}}\rangle, |\phi_x^{\text{out}}\rangle\right)$, $x = 1, \ldots, n$, where numerous copies of each pair can be received. The first state of these pairs is used for the input state of the network. As the input qubits are initialized in a mixed state, we write for this input $\rho_x^{\text{in}} = |\phi_x^{\text{in}}\rangle \langle \phi_x^{\text{in}}|$. The second state is the one that we wish the output ρ_x^{out} of the network to be as close as possible to. This desired output state $|\phi_x^{\text{out}}\rangle$ is specified by choosing a unitary operator

V. We determine that the state is obtained by applying the unitary V to the input state, so it can be written in the form $|\phi_x^{\text{out}}\rangle = V|\phi_x^{\text{in}}\rangle$. However, V is unknown to us. By training the QNN, we try to achieve the goal of learning this unitary.

Similar to the classical case, a cost function is needed to make the network learn. For the QNN, the cost function can be chosen to be the average of the fidelity between the output state of the QNN and the desired output state given by the training data and reads

$$C = \frac{1}{n}\sum_{x=1}^{n} \langle\phi_x^{\text{out}}|\rho_x^{\text{out}}|\phi_x^{\text{out}}\rangle\,. \tag{25}$$

Hereby, it can be measured how close the desired and the actual outputs are. The value of the cost function lies in the interval $[0, 1]$. Unlike in the classical case, where the result is better if the chosen cost function is closer to 0, a value of 0 is worst here. The best possible value in the quantum case is 1.

3.3 Classical Simulation of Learning Tasks

As it is desirable to simulate the training of the QNN on a classical computer, the algorithm from [2] for this concern shall be discussed now. The simulation consists in the following procedure:

1. It is set $s = 0$ and the unitaries $U_j^l(s = 0)$ are randomly chosen.
2. The following steps are repeated numerous times:
   a) The input is fed forward via the transition maps $\mathcal{E}^l$ according to equation 24. That is, for every training pair and with $\rho_x^{\text{in}} = |\phi_x^{\text{in}}\rangle\langle\phi_x^{\text{in}}|$, it is performed for every layer successively:
      i. Tensor:
      $$\rho_x^{l-1,l} = \rho_x^{l-1}(s) \otimes |0\ldots0\rangle_l\langle 0\ldots0| \tag{26}$$
      ii. Apply the unitaries:
      $$U_{m_l}^l(s)\ldots U_1^l(s)\left(\rho_x^{l-1,l}\right) {U_1^l}^\dagger(s)\ldots {U_{m_l}^l}^\dagger(s) \tag{27}$$
      iii. Trace out:
      $$\text{tr}_{l-1}\left(U_{m_l}^l(s)\ldots U_1^l(s)\left(\rho_x^{l-1,l}\right) {U_1^l}^\dagger(s)\ldots {U_{m_l}^l}^\dagger(s)\right) \tag{28}$$
      iv. Store this as $\rho_x^l(s)$.
   b) The unitaries are updated.
      i. Cost function:
      $$\frac{1}{n}\sum_{x=1}^{n} \langle\phi_x^{\text{out}}|\rho_x^{\text{out}}(s)|\phi_x^{\text{out}}\rangle \tag{29}$$

ii. Parameter matrices $K_j^l(s)$ describing the j^{th} perceptron in layer l:

$$K_j^l(s) = \eta \frac{2^{m_{l-1}} i}{n} \sum_{x=1}^{n} \mathrm{tr}_{\mathrm{rest}} \left(M_j^l(s) \right) \tag{30}$$

iii. Update unitaries:

$$U_j^l(s+\epsilon) = e^{i\epsilon K_j^l(s)} U_j^l(s) \tag{31}$$

iv. Set $s \mapsto s + \epsilon$.

For the calculation of $K_j^l(s)$, it is necessary to compute the following commutator:

$$M_j^l(s) = \left[\prod_{\alpha=j}^{1} U_\alpha^l \left(\rho_x^{l-1,l} \right) \prod_{\alpha=1}^{j} U_\alpha^{l\,\dagger}, \prod_{\alpha=j+1}^{m_l} U_\alpha^{l\,\dagger} \left(\mathbb{I}_{l-1} \otimes \sigma_x^l \right) \prod_{\alpha=m_l}^{j+1} U_\alpha^l \right]. \tag{32}$$

We have $\sigma_x^l = \mathcal{F}^{l+1}(\dots \mathcal{F}^{\mathrm{out}}(|\phi_x^{\mathrm{out}}\rangle \langle \phi_x^{\mathrm{out}}|) \dots)$, where $\mathcal{F}^l$ is the adjoint channel to $\mathcal{E}^l$ given by

$$\mathcal{F}_s^l(X^l) = \mathrm{tr}_l \left(\mathbb{I}_{l-1} \otimes |0 \dots 0\rangle_l \langle 0 \dots 0| \, U^{l\,\dagger}(s) (\mathbb{I}_{l-1} \otimes X^l) U^l(s) \right). \tag{33}$$

This means that the desired output state $|\phi_x^{\mathrm{out}}\rangle \langle \phi_x^{\mathrm{out}}|$ is fed backward through the network to layer l.

4 Quantum Classification

It has been shown in [17] that quantum computers can be exploited for an implementation of support vector machines. There, the training set consists of pairs of a vector and a class and thus has the form $\{(\mathbf{x}_i, y_i) : \mathbf{x}_i \in \mathbb{R}^N, y_i = \pm 1, i = 1 \dots n\}$. The machine can learn to classify the class of a vector. However, this is not the scenario we want to examine here.

In the scenario considered in [9] and in this work, we are given a set containing pure quantum states, and each of these states is provided with an associated label according to the class it belongs to. Moreover, we receive an unknown state from this set. It is our goal to identify its class. This task of quantum state discrimination shall be explored by taking a machine learning approach now.

4.1 Basic Concepts

The quantum version of the training dataset is composed of n pairs of a pure quantum state and a class, $D_n = \{(|\psi_1\rangle, y_1), \dots, (|\psi_n\rangle, y_n)\}$, where the $|\psi_i\rangle$ are the quantum states and y_i the corresponding classes, which we assume to be classical. We have several ways in which the states of the dataset can be described, such as by a finite number of copies or by a classical description. Therefore, it is appropriate to provide the following definition of learning classes as in [9].

Such a learning class is indicated by $L_{\text{goal}}^{\text{context}}$, where the superscript provides information about the training dataset or the computer. The subscript stands for the goal we pursue, which can be a classical or a quantum task. In our case, the goal is quantum. We define that $L_{\text{qu}}^{\text{cl}}$ is the learning class providing the classical descriptions of the quantum states. This means that we also have access to as many copies of the state as we wish. In the learning class $L_{\text{qu}}^{\otimes s}$, at least s copies of each state are available.

If we have access to more copies of a quantum state, we might obtain more information on this state. A classical description does even offer a complete knowledge about it. This circumstance can also be described by an order relation for the quantum learning classes which illustrates how much information can be achieved. We write $\equiv_l$ and $\leq_l$ for the relationships "equivalence" and "weaker or equal" showing from which learning class we can get more information. Thus, the following relations can be found:

$$L_{\text{qu}}^{\otimes s} \equiv_l L_{\text{qu}}^{\text{cl}} \text{ as } s \to \infty, \tag{34}$$

$$L_{\text{qu}}^{\otimes 1} \leq_l \cdots \leq_l L_{\text{qu}}^{\otimes s} \leq_l L_{\text{qu}}^{\otimes s+1} \leq_l \cdots \leq_l L_{\text{qu}}^{\text{cl}}, \tag{35}$$

$$L_{\text{qu}}^{\otimes s} + L_{\text{qu}}^{\otimes 1} \leq_l L_{\text{qu}}^{\otimes s+1}. \tag{36}$$

When training a classifier, it can of course happen that it does not deliver the correct class. Therefore, we have a look at the training error now. The probability that a

classifier f does not predict the correct class y_i of a randomly chosen state $|\psi_i\rangle$ from the training set is called the training error ϵ_f. It can be written as

$$\epsilon_f = \frac{1}{n} \sum_{i=1}^{n} \text{Prob}(f(|\psi_i\rangle) \neq y_i). \tag{37}$$

In contrast to this, the generalization error indicates how well the classes of states can be predicted that do not belong to the training set.

It is not guaranteed that we can always achieve $\epsilon_f = 0$. For instance, dealing with quantum states can make it harder to determine the classes because of quantum effects. For such difficult problems, we introduce the regret of f as the difference

$$r_f = \epsilon_f - \epsilon_{\text{opt}}, \tag{38}$$

where ϵ_{opt} is the smallest possible error. A classifier is optimal if $r_f = 0$.

When two learning tasks are considered, it is desired to observe how an advance in one task can be applied to improve the other one. For that, we regard the concept of learning reductions. Consider two learning tasks A and B. We say that the task A reduces to the task B if the ability to solve B via a black box implies the ability to solve A. In this case, we speak of a learning reduction. With such a reduction, any improvement on learning task B can also be applied to solve A. As an example, such a reduction can be used for the solution of the weighted binary tasks via standard binary classification, which we will look at briefly later.

Every time the black box is called, some copies of a state might become unusable as the measurements are made. Therefore, a cost indicating the number of needed copies of a quantum state can be related to a reduction. We define the training cost of a reduction as the product of the number of calls to the black box by the reduction and the number of copies of each quantum state per call. The classification cost is the number of a state's copies used to determine its class after the training.

4.2 Binary Classification

We will now examine the case that each state has one out of two possible classes, which we consider to be -1 and $+1$. This binary classification problem is also discussed in [9]. It is our goal to find a classifier f minimizing the training error ϵ_f. Here, this classifier is desired to be a Positive Operator Valued Measure (POVM).

According to [15], a POVM is a set of operators $\{E_m\}$ for which each operator E_m is positive, and the completeness relation $\sum_m E_m = I$ holds. For such a POVM, there exist measurement operators $M_m = \sqrt{E_m}$ which define a measurement. The probability of obtaining m in a system in the state $|\psi\rangle$ reads $p(m) = \langle\psi|E_m|\psi\rangle$.

For an analysis of the minimal error rate, let m_+ be the number of states in the training set with $y_i = +1$ and m_- the number of states with $y_i = -1$, with $m_+ + m_- = n$. Furthermore, let $p_+ = m_+/n$ and $p_- = m_-/n$. The statistical mixture representing the

class with label ± 1 is

$$\rho_{\pm} = \frac{1}{m_{\pm}} \sum_{i=1}^{n} I\{y_i = \pm 1\} \left|\psi_i\right\rangle \left\langle\psi_i\right| \tag{39}$$

with

$$I\{\text{premise}\} = \begin{cases} 1 & \text{if premise true} \\ 0 & \text{otherwise} \end{cases}. \tag{40}$$

These mixtures allow a statement on the minimal training error depending on their statistical overlap. According to [9, 12], the following theorem holds.

Theorem 1. *A lower bound for the error rate of discriminating between ρ_+ and ρ_- is given by*

$$\epsilon_{\text{hel}} = \frac{1}{2} - \frac{D(\rho_-, \rho_+)}{2} \tag{41}$$

with the trace distance

$$D(\rho_-, \rho_+) = \text{Tr}|p_-\rho_- - p_+\rho_+|. \tag{42}$$

It is possible to reach this bound by the optimal classifier, which is called the Helstrom measurement.

Obviously, the regret of the Helstrom measurement is $r_{\text{hel}} = 0$. We want to illustrate the Helstrom bound with an example. Let $p_+ = p_- = \frac{1}{2}$. If ρ_+ and ρ_- stand for the same state, this yields $D(\rho_-, \rho_+) = 0$ and $\epsilon_{\text{hel}} = \frac{1}{2}$. If ρ_+ and ρ_- are orthogonal, we have $D(\rho_-, \rho_+) = 1$ and $\epsilon_{\text{hel}} = 0$.

In general, if only one copy of an unknown quantum state from the training dataset is available, it is only then possible to get a training error $\epsilon_f = 0$ if all states in the training set are mutually orthogonal. However, for a classifier f with an error $\epsilon_f < \frac{1}{2}$, the probability for a wrong classification can be scaled down by applying the classifier multiple times. The output is then the prediction which is in the majority.

The standard binary classification task we have considered so far can be modified. For instance, let us regard the weighted binary classification task. In this scenario, there are again two possible classes, but now each quantum state is equipped with a weight w_i such that the training dataset has the form $D_n = \{(\left|\psi_1\right\rangle, y_1, w_1), \ldots, (\left|\psi_n\right\rangle, y_n, w_n)\}$, where $y_i \in \{-1, +1\}$ and $w_i \in [0, +\infty)$. These weights show how important it is to classify the respective state correctly. For this problem, a learning reduction to standard binary classification can be made. Consider the case that we are given the classical descriptions of the states, $L_{\text{qu}}^{\text{cl}}$. Convert the weights:

$$p_i = \frac{w_i}{\sum_{j=1}^{n} w_j}. \tag{43}$$

With this, the standard binary classifier can be used to minimize the error between the new densitiy matrices

$$\hat{\rho}_{\pm} = \sum_{i=1}^{n} p_i I\{y_i = \pm 1\} \left|\psi_i\right\rangle \left\langle\psi_i\right| . \tag{44}$$

This also enables minimizing the error on the weighted training set [9].

Finally, we also want to briefly examine an example of recent work on binary classification. It is given in [19], where the goal is to perform a binary classification task in an unsupervised way, that is, without access to labeled training data. It is assumed that a source prepares N quantum systems, each described by a pure state, and there are only two possibilities for the states. Moreover, no further information about these states is available. It is the aim to arrange the systems in two clusters, such that in each of the clusters all states are the same. For this purpose, notice that performing separated measurements on the quantum systems yields N data points. As the results of measuring sytems that are all in the same state obey some probability distribution, these N points are actually drawn from two probability distributions. This means that in this unsupervised classification task it is aimed to determine which points belong to which distribution.

4.3 Quantum Variational Classification

We want to show an example for a classification algorithm in more detail now. This kernel method is presented in [11] as "quantum variational classification" and trains a variational quantum circuit $W(\boldsymbol{\theta})$ parametrized by $\boldsymbol{\theta}$, which is to optimize during the training. The space of quantum states is used as a feature space, and classical data is mapped into this space. After applying the circuit to the states, binary measurements are performed. This procedure yields a separating hyperplane in the state space and thus a linear decision function. Even though the algorithm is used for the classification of classical data, it will turn out to be useful for later examinations.

During the training, the cost function makes use of the probability of misclassifying a sample, which is estimated to be

$$\Pr(\tilde{m}(\mathbf{x}) \neq y | m(\mathbf{x}) = y) \approx \sigma \left(\sqrt{R} \frac{\frac{1-yb}{2} - \hat{p}_y}{\sqrt{2(1-\hat{p}_y)\hat{p}_y}} \right), \tag{45}$$

where $m(\mathbf{x})$ is the correct label and $\tilde{m}(\mathbf{x})$ the assigned label. For the cost function, average over all samples. By minimizing the resulting expression, the optimal $\boldsymbol{\theta}^*$ and an additional bias $b \in [-1, 1]$ for the decision rule are found.

We are especially interested in the classification protocol after the training. The classifier assigns labels to data via the following procedure:

1. The feature map circuit $U_{\phi(\mathbf{x})}$ mapping the data to quantum states is used:

$$\phi : \mathbf{x} \in \Omega \subset \mathbb{R}^d \mapsto \left|\phi(\mathbf{x})\right\rangle \left\langle\phi(\mathbf{x})\right| \tag{46}$$

2. The optimized short-depth quantum circuit $W(\boldsymbol{\theta}^*)$ is applied to the state.

3. With $y \in \{+1, -1\}$, a binary measurement $\{E_y\}$ diagonal in the z-basis is used for the state $W(\boldsymbol{\theta}^*)U_{\phi(\mathbf{x})}\left|0\right\rangle^n$, where the probability for outcome y is

$$p_y = \langle\phi|W^\dagger(\boldsymbol{\theta}^*)E_y W(\boldsymbol{\theta}^*)|\phi\rangle\,. \tag{47}$$

   R measurement shots are performed for the distribution of empirical probabilities $\hat{p}_y(\mathbf{x})$.

4. The label $\tilde{m}(\mathbf{x}) = y$ is attached if

$$\hat{p}_y > \hat{p}_{-y} - yb \tag{48}$$

   with the optimal bias $b \in [-1, 1]$.

5 Supervised Binary Classification with the QNN

We will now try to find a possibility for performing quantum classification, and it is our goal to use the already presented QNN for this. More precisely, we will examine the task of supervised binary classification. This means that the training samples are of the form $(|\phi_j\rangle, l_j)$ with the quantum states $|\phi_j\rangle$ and the labels $l_j \in \{0, 1\}$. For reasons which will become clear later, we will use 0 and 1 instead of 1 and -1 to indicate the classes.

5.1 Example for the Training Data

We want to make our considerations more concrete and examine an example for the creation of training data, which will be used for the construction and the test of an algorithm. In the example, the Bloch sphere finds application, so it is necessary to discuss its concept first. This can be found in [15] and [22].

In analogy to the classical bit used for classical computation, whose value can either take 0 or 1, a quantum bit (qubit) can be viewed as the fundament of quantum computation. A qubit has two basis states, which are denoted by $|0\rangle$ and $|1\rangle$. The state of a qubit can be written as a linear combination, i.e. a superposition, of those,

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle\,, \tag{49}$$

where α and β are complex numbers, as it can be seen as an element of a two-dimensional complex vector space. If a measurement is performed on a qubit, the outcome will be either 0 or 1, and $|\alpha|^2$ and $|\beta|^2$ are the probabilities respectively. Therefore, it must be satisfied $|\alpha|^2 + |\beta|^2 = 1$, which means that $|\psi\rangle$ is a unit vector. Hence, the state can also be expressed as

$$|\psi\rangle = e^{i\gamma}\left(\cos\frac{\theta}{2}\,|0\rangle + e^{i\varphi}\sin\frac{\theta}{2}\,|1\rangle\right) \tag{50}$$

with the real numbers θ, φ and γ. The factor $e^{i\gamma}$ has no observable effects, which is why the state can also be described as

$$|\psi\rangle = \cos\frac{\theta}{2}\,|0\rangle + e^{i\varphi}\sin\frac{\theta}{2}\,|1\rangle\,. \tag{51}$$

This formulation enables us to regard and visualize the state of the qubit as a point on the unit sphere S^2 determined by the angles θ and φ. Figure 5 shows such a Bloch sphere with a state $|\psi\rangle$.

By choosing $|0\rangle = \binom{1}{0}$ and $|1\rangle = \binom{0}{1}$, a qubit state can be mapped to the Bloch sphere via the density matrix

$$\rho = |\psi\rangle\langle\psi| = \begin{pmatrix} |\alpha|^2 & \alpha\beta^* \\ \alpha^*\beta & |\beta|^2 \end{pmatrix} \tag{52}$$

by

$$\rho = \begin{pmatrix} \rho_{00} & \rho_{01} \\ \rho_{10} & \rho_{11} \end{pmatrix} \mapsto \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 2\,\mathrm{Re}\,\rho_{01} \\ 2\,\mathrm{Im}\,\rho_{10} \\ 2\rho_{00} - 1 \end{pmatrix} \in S^2. \tag{53}$$

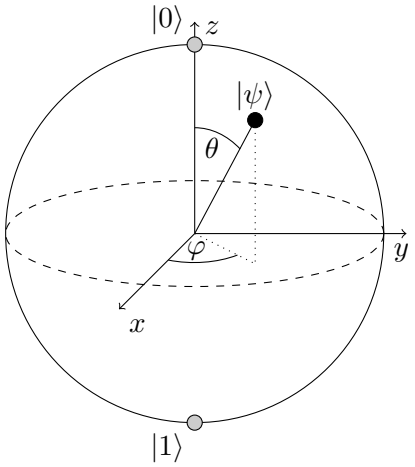


Figure 5: Bloch sphere.

Now that we have introduced this map, the training data for our learning algorithm can be specified. For this, we construct a plane given by a normal vector $\mathbf{n}$ with length $|\mathbf{n}| = 1$ and a shift b with values from -1 to 1. Let us describe the plane by the set $\{\mathbf{x} \in \mathbb{R}^3 : \mathbf{n} \cdot \mathbf{x} + b = 0\}$, motivated for example by [8]. By letting the Bloch sphere and the plane intersect, the sphere is divided into two parts. For $b \neq 0$, these parts do not have the same size, as we can also see in Figure 6a. The label a quantum state receives, is decided according to which part of the Bloch sphere it is located on. Let $\mathbf{x}$ be the position of the state on the Bloch sphere now. If $\mathbf{n} \cdot \mathbf{x} + b > 0$, we assign the label 0, otherwise 1.

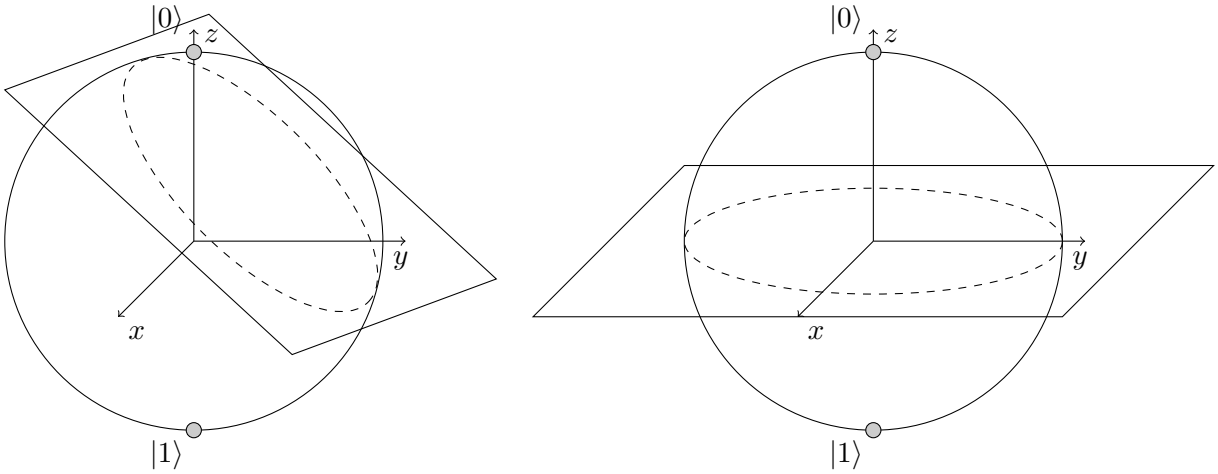
Consider the case $\mathbf{n} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$, $b = 0$. Here, we attach the label 0 to states on the Bloch sphere with $z > 0$, otherwise the assigned label is 1. So states on the upper half of the Bloch sphere receive the label 0 and states on the lower half the label 1. Since $z = 2\rho_{00} - 1 = 2|\alpha|^2 - 1$ is 0 if and only if $|\alpha|^2 = \frac{1}{2}$, this is equivalent to attaching the label 0 to states with $|\alpha|^2 > 0$, i.e. to states for which the probability of obtaining the state $|0\rangle$ in a measurement is higher than the probability of obtaining $|1\rangle$. This is also the reason why we use 0 and 1 instead of 1 and -1 for the labels. Figure 6b illustrates such a decision according to whether a state is on the upper or the lower half.

A qubit can be measured by applying a binary measurement diagonal in the z-basis. Such a measurement can be written here as $E_0 = |0\rangle\langle 0|$, $E_1 = |1\rangle\langle 1|$ according to [11]. With our choice for the basis, this yields

$$E_0 = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, \tag{54}$$

$$E_1 = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}. \tag{55}$$

The probability for outcome $|0\rangle$ is then $\mathrm{tr}\,(E_0\rho) = |\alpha|^2$, and the one for outcome $|1\rangle$ is $\mathrm{tr}\,(E_1\rho) = |\beta|^2$.



(a) A possibility for a choice of a separating plane with $b \neq 0$.

(b) The special case that the separating plane is the xy-plane.

Figure 6: Two examples for the creation of the training data. The dashed line is the intersection of the Bloch sphere and the plane.

5.2 Algorithm

Having introduced the problem, we can now develop an algorithm for its solution with our QNN. The code can be found on GitHub [16] and in the Appendix. This algorithm is inspired by the procedure in [11]. For our problem, it is important to note that the training data applied for the QNN consists of pairs of quantum states, while the training data for the classification task is comprised of pairs of a quantum state and a label. In order to have the possibility to use the QNN for the latter task without changing its whole structure, the idea now is to change the form of the training dataset used for classification. Therefore, we replace each of the labels by a quantum state. The preceding considerations suggest that the label 0 corresponds to the state $|0\rangle$ and the label 1 corresponds to the state $|1\rangle$. Hence, every label $l_j \in \{0, 1\}$ is replaced by the state $|l_j\rangle \in \{|0\rangle, |1\rangle\}$, and the training set takes the form $\{(|\phi_1\rangle, |l_1\rangle), \ldots, (|\phi_n\rangle, |l_n\rangle)\}$.

Thus, during the training with the QNN with such an adapted training set, the inputs to the network are $\rho_j^{\text{in}} = |\phi_j\rangle\langle\phi_j|$ just as in its original version. The output states of the network can be expressed as $\rho_j^{\text{out}} = \mathcal{E}(|\phi_j\rangle\langle\phi_j|)$, where we denote the impact of the network on the input states, i.e. the application of all layer-to-layer channels, by $\mathcal{E}$.

Another point to be considered is the architecture of the network. As only one qubit is considered in our example, the input and the output layer must also have one and only one neuron. Possible architectures for the training with the QNN thus are of the form $1 - \cdots - 1$, where the first number stands for the input layer and the last number for the output layer.

A cost function is also necessary for the training. If we define the cost function as the mean of the probabilities of measuring the correct label l_j belonging to the state $|\phi_j\rangle$, we

get

$$C = \frac{1}{n}\sum_{j=1}^{n} \mathrm{tr}\left((E_{l_j}\mathcal{E}(|\phi_j\rangle\langle\phi_j|)\right) \tag{56}$$

$$= \frac{1}{n}\sum_{j=1}^{n} \mathrm{tr}\left(E_{l_j}\rho_j^{\mathrm{out}}\right) \tag{57}$$

$$= \frac{1}{n}\sum_{j=1}^{n} \mathrm{tr}\left(|l_j\rangle\langle l_j|\,\rho_j^{\mathrm{out}}\right) \tag{58}$$

$$= \frac{1}{n}\sum_{j=1}^{n} \langle l_j|\rho_j^{\mathrm{out}}|l_j\rangle\,. \tag{59}$$

This expression coincides with the fidelity used as the cost function in the QNN. Therefore, there is no need to change the network's cost function here.

So now it is possible to carry out some training regarding classification with the network. This consists in trying to find a unitary which maps quantum states on average as close as possible to one of the states $|0\rangle$ and $|1\rangle$, depending on what the label is. However, the classification problem cannot be solved by training with the QNN with the adapted training set alone since the outputs are quantum states, not labels. Another step is necessary for classifying the states after the training with the network.

To accomplish this, we remember that if a state is closer to $|0\rangle$ than to $|1\rangle$, it is more probable to obtain the outcome 0 in a measurement as discussed above, and vice versa. Therefore, it stands to reason to measure the output states of the QNN to distribute the labels. So after the training, the quantum states can be classified by performing R measurement shots for each output state with the binary measurement $\{E_0 = |0\rangle\langle 0|, E_1 = |1\rangle\langle 1|\}$. R copies of the input states are needed for this.

If we know that we use $b = 0$ for the creation of the training data, the assigned label can then simply be chosen to be the one which is obtained more often in the measurements, which is done in the first version of the code. In general, we calculate a threshold $c \in [0, 1]$ for the empirical probability p_0 of label 0, see the second version. For this, we regard all states that should get label 0 and find the minimal value for p_0 among them. Moreover, we regard all states with label 1 and find the maximal value. The threshold c is then the average of these two values, and we assign the label 0 if $p_0 > c$, label 1 if $p_0 < c$. The following connection can be found between c and the shift b of the plane:

$$c = |\alpha|^2_{\mathrm{thr}} \tag{60}$$

$$= \cos^2\left(\frac{\arccos(-b)}{2}\right) \tag{61}$$

$$= \frac{1}{2}\left(1 + \cos\left(\arccos(-b)\right)\right) \tag{62}$$

$$= \frac{1}{2}(1 - b) \tag{63}$$

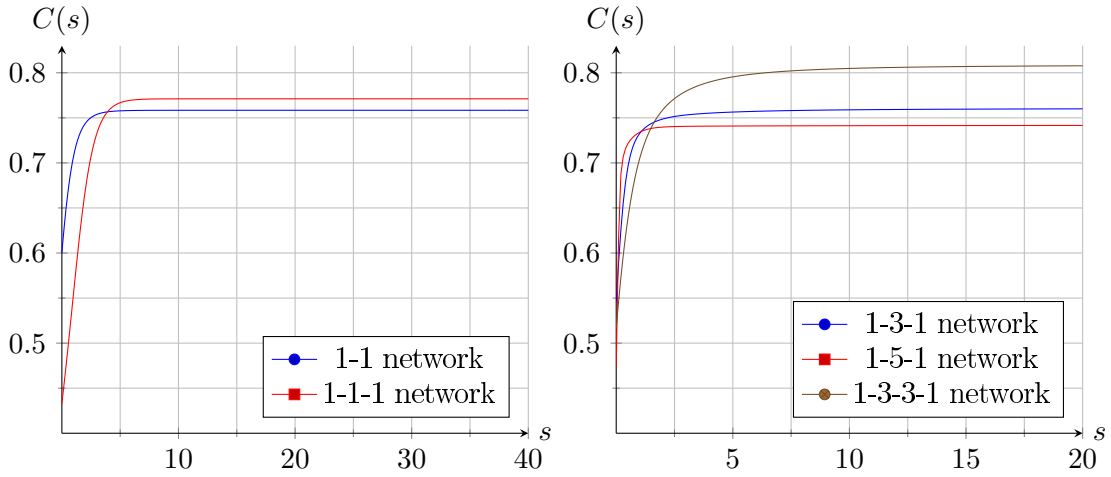
All in all, the algorithm is as follows:

1. Training with the QNN with the data $\{(|\phi_1\rangle, |l_1\rangle), \ldots, (|\phi_n\rangle, |l_n\rangle)\}$.

2. Measurements on the output states:
   a) Feedforward: apply the network $\mathcal{E}$ to the input states.
   b) Choose R times a label according to the probabilities $\mathrm{tr}\left(E_l \rho_j^{\mathrm{out}}\right)$ for measuring the label l.
   c) Compute the empirical probability distributions.
   d) Calculate the threshold c as the average of the minimal p_0 for states with label 0 and the maximal p_0 for states with label 1.
   e) Choose final labels according to the empirical distributions and c.

5.3 Test of the Algorithm

We now wish to check whether our algorithm works and how well it solves the classification task. First, we try the training with various network architectures and regard the development of the cost function of the QNN during the training. The corresponding plots are shown in Figures 7a and 7b. Moreover, we use the network to classify the training data and examine the ratio of wrong classifications to the whole number of classified states for each of the training examples.



(a) 0 % of the training data were falsely classified with the 1-1 network, and 10 % with the 1-1-1 network. The training was held with 400 training rounds.

(b) Falsely classified: 0 % with the 1-3-1 network, 0 % with the 1-5-1 network and 2 % with the 1-3-3-1 network. 200 training rounds were held.

Figure 7: Examples for the development of the cost function of the QNN for several architectures. The training data consists of 50 states, each with the corresponding label. We used $\mathbf{n} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$, $b = 0$. The parameters used in the training are $\lambda = 1.0$, $\epsilon = 0.1$. The labels were assigned via $R = 1000$ measurement shots.

For this training, we use the first version of the code that assumes that the two halfs of the Bloch sphere have the same size, which means that $b = 0$ and $c = 1/2$. The cost function starts with values ranging approximately from 0.4 to 0.6 and initially increases rapidly, until it stagnates at a value greater than 0.7 or in an exceptional case greater than 0.8. With some of the trained networks, none of the states is classified falsely. The greatest error in the examples shown here is 10 %.

It is important to note that the training data is not generated via a unitary but there are only two possible desired output states in the training data. Since the number of training pairs is by far greater than one or two, it is not possible to find a unitary fulfilling that all output states are truly either $|0\rangle$ or $|1\rangle$. This is why the cost function of the QNN does not reach values very close to 1 during the training. However, most of the training data can be classified correctly. There are even cases where no state receives a wrong label. The reason for this lies in the circumstance that the labels are distributed according to which label occurs more often in the empirical distributions obtained by measurements. After the training, the fidelity $\langle l_j|\rho_j^{\text{out}}|l_j\rangle$ for training sample j with the correct label l_j is probably higher than the fidelity with the wrong label even though the cost funtion differs quite strongly from 1. In this case, it is also more likely that a measurement has the outcome l_j, and this will be the more frequently obtained result. It will therefore be the conclusive label, such that the error rate finally is low.

Furthermore, we notice that the exact value of the cost function after the training does not necessarily determine the quality of the classifications. For instance, the cost function for the $1 - 5 - 1$ network has a value less than 0.75 with no falsely classified states, while it reaches a value greater than 0.8 for the $1 - 3 - 3 - 1$ network where not all states were correctly classified.

An important point is that the network does not receive the information how the labels of the states from the training dataset are distributed. If it knew that we have the special case $\mathbf{n} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$, $b = 0$, we could of course skip the training and perform the measurements only. However, we assume that the data we create is given to us without this information, and the network does not know how we create the training data.

To test the quality of the algorithm more extensively, training and classification are performed several times for each of the previously applied architectures. The number of training runs is restricted to twelve for each architecture because of the long computation times. Classification is performed after each of the trainings. Again, the version of the code assuming that $c = 1/2$ is used. The results are shown in Table 1.

When regarding the rate of falsely classified states from the training set, we do not see any striking differences between the used network architectures. It is possible to classify all states correctly, and only up to 12 % of the training states are classified incorrectly in this example.

We also want to test the more general version of the code, which is also suitable for training data created with $b \neq 0$. In this example, we determine that training states with $y > -0.2$ carry the label 0, where y is the second coordinate of the state on the Bloch sphere. As before, several trainings are performed for each architecture, but this time, we also examine the impact of the R measurement shots. Table 2 shows the outcomes.

	1-1	1-1-1	1-3-1	1-5-1	1-3-3-1
1	0.08	0.04	0.06	0.04	0.02
2	0.02	0.04	0.04	0.06	0.00
3	0.04	0.02	0.02	0.06	0.04
4	0.02	0.12	0.00	0.06	0.04
5	0.04	0.04	0.08	0.06	0.06
6	0.04	0.02	0.10	0.02	0.08
7	0.10	0.00	0.04	0.00	0.06
8	0.00	0.06	0.06	0.02	0.08
9	0.04	0.04	0.06	0.00	0.02
10	0.00	0.04	0.08	0.04	0.02
11	0.02	0.10	0.02	0.06	0.10
12	0.10	0.02	0.02	0.02	0.02

Table 1: The fraction of wrong classifications for several architectures, a total of twelve trainings with 50 training states for each. We used $\mathbf{n} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$, $b = 0$. The parameters used in the training are $\lambda = 1.0$, $\epsilon = 0.1$. 100 training rounds were held for the 1-5-1 network, 200 rounds for the 1-3-1 and the 1-3-3-1 network, 400 rounds for the 1-1 and the 1-1-1 network. The labels were assigned via $R = 1000$ measurement shots.

	1-1		1-1-1		1-3-1		1-5-1		1-3-3-1	
	$R = 1$	$R = 1000$	1	1000	1	1000	1	1000	1	1000
1	0.26	0.04	0.38	0.08	0.28	0.04	0.20	0.04	0.26	0.06
2	0.30	0.06	0.32	0.04	0.26	0.04	0.34	0.00	0.32	0.04
3	0.16	0.06	0.34	0.00	0.20	0.00	0.20	0.04	0.22	0.00
4	0.26	0.10	0.16	0.00	0.22	0.06	0.16	0.04	0.34	0.00
5	0.28	0.06	0.26	0.10	0.30	0.04	0.16	0.04	0.14	0.08
6	0.32	0.04	0.24	0.00	0.28	0.00	0.20	0.06	0.16	0.00
7	0.38	0.04	0.16	0.10	0.22	0.10	0.30	0.10	0.28	0.04
8	0.28	0.06	0.26	0.00	0.30	0.04	0.32	0.04	0.14	0.14
9	0.32	0.08	0.40	0.06	0.30	0.04	0.18	0.08	0.24	0.06
10	0.22	0.06	0.40	0.10	0.14	0.10	0.24	0.04	0.20	0.04
11	0.18	0.06	0.18	0.06	0.24	0.12	0.42	0.16	0.32	0.04
12	0.26	0.00	0.24	0.06	0.20	0.00	0.18	0.00	0.26	0.16

Table 2: The fraction of wrong classifications. We have $\mathbf{n} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$, $b = 0.2$ here. The parameters used in the training are $\lambda = 1.0$, $\epsilon = 0.1$. 100 training rounds were held for the 1-5-1 network, 200 rounds for the 1-3-1 and the 1-3-3-1 network, 400 rounds for the 1-1 and the 1-1-1 network.

It is striking that the classification error is much lower with $R = 1000$ measurement shots than with only $R = 1$ shot. This supports the assumption that by increasing R, the error can be reduced as the empirical probabilities approximate the actual probabilities.

Furthermore, the algorithm is tested for several values of b. This time, we do not only examine the error rate, which is called ϵ here, but also the value b_{calc} for the shift of the separating plane that is calculated according to the ascertained value for the threshold c via $b = 1 - 2c$. See Table 3.

	$b = 0.0$		$b = 0.1$		$b = 0.2$		$b = 0.3$		$b = 0.4$		$b = 0.5$	
	ϵ	b_{calc}	ϵ	b_{calc}	ϵ	b_{calc}	ϵ	b_{calc}	ϵ	b_{calc}	ϵ	b_{calc}
1	0.00	-0.045	0.04	0.039	0.04	0.217	0.00	0.311	0.04	0.391	0.04	0.498
2	0.10	-0.022	0.06	-0.017	0.00	0.170	0.04	0.500	0.00	0.343	0.00	0.505
3	0.08	0.015	0.00	0.149	0.04	0.252	0.04	0.244	0.00	0.339	0.00	-1.0
4	0.00	0.076	0.04	0.052	0.04	0.214	0.06	0.294	0.06	0.408	0.00	-1.0
5	0.08	0.095	0.04	0.125	0.00	0.156	0.10	0.274	0.00	-1.0	0.04	0.556
6	0.08	0.061	0.08	0.202	0.08	0.259	0.00	-0.451	0.00	0.411	0.00	-1.0
7	0.00	0.049	0.06	0.178	0.04	0.255	0.00	0.353	0.08	0.33	0.08	0.504
8	0.04	-0.017	0.00	0.034	0.04	0.119	0.00	0.173	0.08	-0.146	0.04	0.446
9	0.04	0.086	0.08	0.032	0.04	0.189	0.04	0.269	0.04	0.404	0.00	-1.0
10	0.12	0.048	0.06	0.093	0.00	0.082	0.30	-0.999	0.24	-0.996	0.08	-0.13
11	0.06	-0.042	0.04	0.159	0.06	0.173	0.06	0.284	0.12	-0.889	0.00	-1.0
12	0.04	0.128	0.00	0.254	0.00	0.208	0.00	-1.0	0.04	0.433	0.00	-1.0

Table 3: The effect of b on classification. A 1-1 network is trained with 50 training states, $\lambda = 1.0$, $\epsilon = 0.1$ and 400 training rounds. For each training, a random normal vector is used for the creation of the data. The labels are assigned via $R = 1000$ measurement shots.

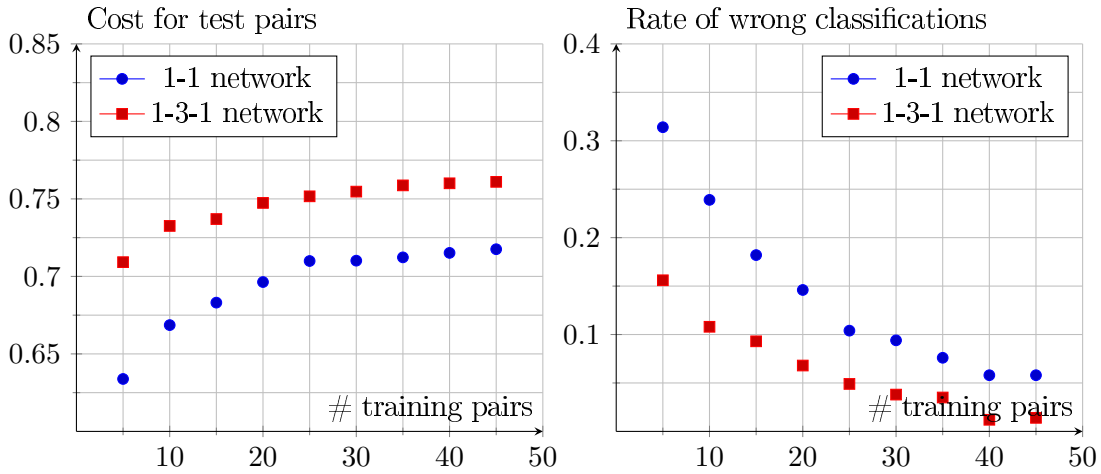
For small values of b, classification works very well. The results for b_{calc} can also be regarded as appropriate. However, it can be noticed that if b increases, it happens more often that the code computes $b_{\text{calc}} = -1$, which cannot be correct because all training states need to have the same label then, and this label is the one only a minority of the states should possess. At the same time, the error rate indicates that no state is classified incorrectly.

Therefore, we finally examine the case of a large shift b. After a training with $b = 0.7$, it can be seen that the QNN maps all input states to states with only one possibility for the diagonal elements. The first one is always 1, while the second one is always 0. Hence, the empirical probability p_0 for the label 0 is 1 for all states, and the one for 1 is 0. Remember now that for the determination of c, we regard all states that should get label 0 and find the minimal value for p_0 among them. And we regard all states with label 1 and find its maximal value there. The threshold c is then the average of these two values. This has the impact that the further computation yields $c = 1$ and thus $b_{\text{calc}} = -1$. Moreover, we cannot tell anymore which label a state should receive

according to the empirical probabilities. Similar to this, $b = -0.7$ causes the network to output states whose second diagonal element is 1. Consequently, every measurement has the outcome 1. The threshold is then determined to be $c = 0$, so $b_{\text{calc}} = 1$. Again, we do not know from the empirical distributions which states are falsely classified and cannot find out what the label is by regarding them. This means that the algorithm is no longer appropriate if the training data is created by a separating plane which is to far away from the point of origin.

5.4 Subset Training

After these tests of the algorithm, our next step is to test its ability to generalize as in [2]. We create a set of N data pairs $(|\phi_j\rangle, |l_j\rangle)$ for this purpose and use only a subset thereof with n samples as the training data. The network trained with this subset is then used for the classification of all N states. When the size n of the subset is varied, the test set remains the same. Figures 8a and 8b show the cost function for all test states after the training and the error rate when classifying all these samples. We use $\mathbf{n} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ and $b = 0$ for the data, as well as the version of the code with $c = 0.5$.



(a) Averaged cost function after the training as a function of the number of training pairs.

(b) Averaged ratio of wrong classifications to all classifications.

Figure 8: For each of the two networks, a set of $N = 50$ test pairs was created. The networks were trained with a subset of $n = 5, 10, \ldots, 45$ training pairs belonging to the corresponding set. These trainings were used to classify all test pairs. For every number of training pairs, we averaged over 20 runs. Moreover, we used $\lambda = 1.5$, $\epsilon = 0.1$, 200 training rounds for the 1-1 network, 150 rounds for the 1-3-1 network and $R = 1000$ measurement shots for the final classifications.

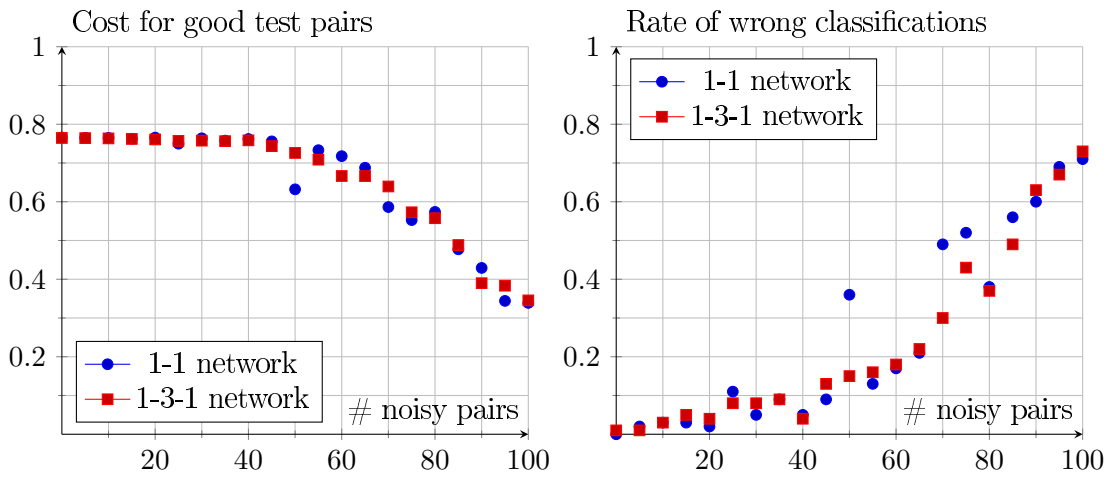
It can be noticed that the example for this generalization with the 1-3-1 network worked better than the one with the 1-1 network. However, we have already seen that the quality of the classification does not really seem to depend on the architecture of the

network. Another point which might have an impact on it is the configuration of the training data. Therefore, it is more likely that the difference occuring here is due to the states contained in the training set.

5.5 Robustness to Noisy Data

The final test that we carry out is about the robustness of the algorithm to noise. More precisely, we examine the behavior when parts of the training data are wrong. We proceed in two steps, both with the first version of the code.

In the first step, parts of the training data are replaced by combinations of a random state and a random label, which can be correct or wrong. We create for this purpose a correct training set of N training pairs in addition to one with N pairs with random labels. The data employed for the training then consists of $N - i$ training pairs of the correct training data and i training pairs of the data with the random labels. After the training, a plot of the cost function for the whole correct training data is created, with the cost as a function of the number of noisy training pairs. Moreover, another plot with the proportion of falsely classified data is made. See Figures 9a and 9b. To still obtain a good classification, up to the half of the training data can be noisy.



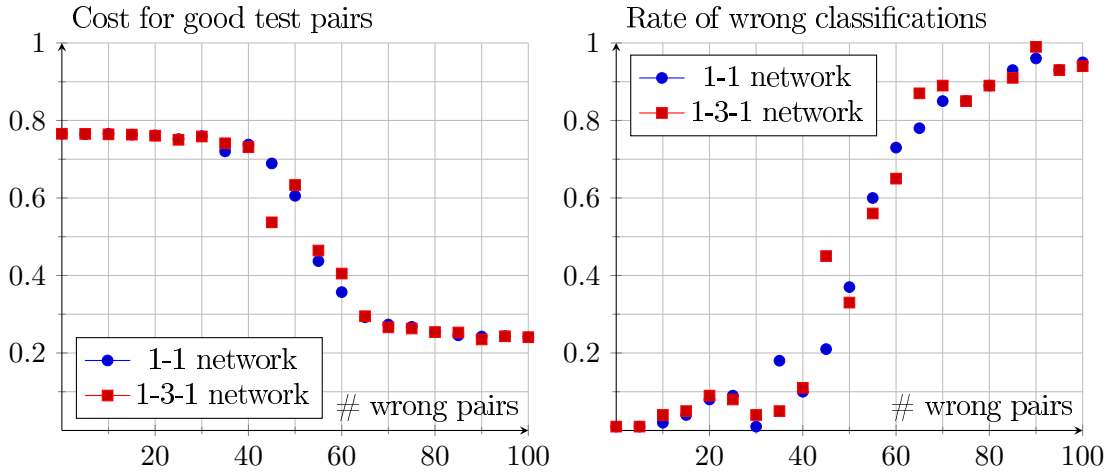
(a) Cost function after the training.

(b) Ratio of wrong to all classifications.

Figure 9: Here, a particular number i of the $N = 100$ training pairs is replaced by states with random labels, i.e. the label of those states can be correct or wrong. The step size of i is 5. The further parameters are $\lambda = 1.0$, $\epsilon = 0.1$, 300 training rounds for the 1-1 network, 100 rounds for the 1-3-1 network and $R = 1000$ measurement shots for the final classifications.

In the second step, parts of the training data are replaced by combinations of a random state and the corresponding wrong label, so the label of such a noisy state is always flipped. The procedure for the analysis of this scenario is the same as in the first step. Figures 10a and 10b show the results. If almost all labels are wrong, then of course

almost no state is classified correctly.



(a) Cost function after the training.

(b) Ratio of wrong to all classifications.

Figure 10: Here, a particular number i of the $N = 100$ training pairs is replaced by states with wrong labels. The step size of i is 5. The further parameters are $\lambda = 1.0$, $\epsilon = 0.1$, 300 training rounds for the 1-1 network, 100 rounds for the 1-3-1 network and $R = 1000$ measurement shots for the final classifications.

6 Conclusion

In this bachelor thesis, we explored how qubits can be classified into one of two possible classes by training with a dataset in which all quantum states are equipped with a label. We developed an algorithm for this classification task using the quantum neural network. Measurements are performed on the output states of the network to identify the class of the qubits.

We could show that the algorithm works very well for the classification of quantum states if the training dataset has been created by dividing the Bloch sphere into two parts via a separating plane. This plane can pass through the point of origin as well as be shifted somewhat.

Moreover, we examined both the ability for generalization to states outside the training set and the robustness to noisy training data. Promising results could be achieved for these approaches.

However, one must pay attention when creating training data as the plane should not be shifted too far. If only few states belong to one of the classes, the processing is not successfull. As the output states of the trained quantum neural network are of such a form then that all measurements yield the same outcome, the procedure is unsuitable here.

Another open question is how quantum states can be classified if it is not possible to find a plane through the Bloch sphere dividing the states correctly. Further investigations are needed for such data which cannot be fully separated thereby.

Appendix

Here are shown the parts of the Classification code [16] that are added to the original QNN code and the parts that originate from the QNN code and are changed for an integration into the Classification code.

qnn_classification_bloch_1.ipynb

This is the code for the case that states on the upper half of the Bloch sphere receive the label 0 and states on the lower half the label 1. First, the creation of the training data and networks:

```
#math related packages
import numpy as np
import scipy as sc
import qutip as qt
#further packages
from time import time
from random import sample
from random import choice
import matplotlib.pyplot as plt
import csv

def blochvec(inputState):
    mat = 2 * sc.array(inputState)
    x = mat[0][1].real
    y = mat[1][0].imag
    z = mat[0][0].real - 1
    return [x,y,z]

def randomTrainingData(N):
    #numQubits = len(unitary.dims[0])
    trainingData=[]
    #Create training data pairs
    for i in range(N):
        a = randomQubitState(1)
        b = a * a.dag()
        c = blochvec(b)
        d = qubit1
        if c[2]>0:
            d = qubit0
        trainingData.append([a,d])
    #Return
    return trainingData

def randomNetwork(qnnArch, numTrainingPairs):
    assert qnnArch[0]==qnnArch[-1], "Not a valid QNN-Architecture."

    #Create the targeted network unitary and corresponding training data
    networkTrainingData = randomTrainingData(numTrainingPairs)

    #Create the initial random perceptron unitaries for the network
    networkUnitaries = [[]]
```

```
    for l in range(1, len(qnnArch)):
        numInputQubits = qnnArch[l-1]
        numOutputQubits = qnnArch[l]

        networkUnitaries.append([])
        for j in range(numOutputQubits):
            unitary = qt.tensor(randomQubitUnitary(numInputQubits+1),\
            tensoredId(numOutputQubits-1))
            unitary = swappedOp(unitary, numInputQubits, numInputQubits + j)
            networkUnitaries[l].append(unitary)

    #Return
    return (qnnArch, networkUnitaries, networkTrainingData)#, networkUnitary)
```

Classification after the training. The function "measurements" simulates R measurements on each state. The function "howManyWrong" returns the fraction of wrong classifications. It is assumed that the Bloch sphere is divided into two halfs of the same size. The function "costFunc" is for control purposes only and defined as in [11].

```
def measurements(qnnArch, unitaries, trainingData, R):
    feed = feedforward(qnnArch, unitaries, trainingData)
    prob = []
    for i in range(len(trainingData)):
        state = feed[i][-1]
        a = [] # probabilities with tr(E_i*rho)
        a.append(np.trace(np.array(qubit0mat*state)).real)
        a.append(np.trace(np.array(qubit1mat*state)).real)
        results = np.random.choice([1,-1], R, p=a)
        #number of measurements with outcome +1:
        k = 0
        for i in range(R):
        if results[i] == 1:
        k = k+1
        #empirical probability distribution:
        plus = k/R
        minus = (R - k)/R
        prob.append([plus, minus])
    return prob

def howManyWrong(qnnArch, unitaries, trainingData, R):
    prob = measurements(qnnArch, unitaries, trainingData, R)
    m = len(trainingData)
    x = 0
    for i in range(m):
        if trainingData[i][1] == qubit0:
            if prob[i][0] < 0.5:
                x = x + 1
        else:
            if prob[i][1] < 0.5:
                x = x + 1
    x = x/m
    return x

def probWrongLabel(R, pr):
```

```
    pWrong = 1/(1+np.exp(-np.sqrt(R)*(1/2-pr)/np.sqrt(2*(1-pr)*pr)))
    return pWrong

def costFunc(qnnArch, unitaries, trainingData, R): #R number of measurements
    prob = measurements(qnnArch, unitaries, trainingData, R)
    m = len(trainingData)
    c = 0
    for i in range(m):
        if trainingData[i][1] == qubit0:
            c = c + probWrongLabel(R, prob[i][0])
        else:
            c = c + probWrongLabel(R, prob[i][1])
    c = c/m
    return c
```

The definition for subset training:

```
def subsetTrainingAvg(qnnArch, initialUnitaries, trainingData, lda, ep,\
trainingRounds, iterations, n, R, alertIt=0):
    costpoints = []
    errorpoints = []

    for i in range(iterations):
        if alertIt>0 and i%alertIt==0: print("n="+str(n)+",␣i="+str(i))

        #Prepare subset for training
        trainingSubset = sample(trainingData, n)

        #Train with the subset
        learnedUnitaries = qnnTraining(qnnArch, initialUnitaries, trainingSubset,\
        lda, ep, trainingRounds)[1]
        storedStates = feedforward(qnnArch, learnedUnitaries, trainingData)
        outputStates = []
        for k in range(len(storedStates)):
            outputStates.append(storedStates[k][-1])

        #Calculate cost with all training data
        costpoints.append(costFunction(trainingData, outputStates))
        errorpoints.append(howManyWrong(qnnArch, learnedUnitaries, trainingData, R))

    newcost = sum(costpoints)/len(costpoints)
    newerror = sum(errorpoints)/len(errorpoints)
    return [newcost, newerror]
```

The definitions for training with noisy data:

```
def noisyDataTraining(qnnArch, initialUnitaries, trainingData, noisyData, lda, ep,\
trainingRounds, numData, stepSize, alertP=0):
    noisyDataPlot = [[], []]
    noisyNewPlot = [[], []]

    i = 0
    while i <= numData:
        if alertP>0: print("Currently␣at␣"+str(i/numData)+"%␣noisy␣data.")
```

```
        #Prepare mixed data for traing
        testData1 = sample(trainingData, numData - i)
        testData2 = sample(noisyData, i)
        if i==0: testData = testData1
        elif i==numData: testData = testData2
        else: testData = testData1 + testData2

        #Train with the mixed data
        learnedUnitaries = qnnTraining(qnnArch, initialUnitaries, testData,\
        lda, ep, trainingRounds)[1]
        storedStates = feedforward(qnnArch, learnedUnitaries, trainingData)
        outputStates = []
        for k in range(len(storedStates)):
            outputStates.append(storedStates[k][-1])

        #Calculate cost with the real training data
        noisyDataPlot[0].append(i)
        noisyDataPlot[1].append(costFunction(trainingData, outputStates))

        #rate of wrong classifications:
        noisyNewPlot[0].append(i)
        noisyNewPlot[1].append(howManyWrong(qnnArch, learnedUnitaries,\
        trainingData, 1000))

        i += stepSize

    return noisyDataPlot, noisyNewPlot

def wrongLabelData(N):
    correctData = randomTrainingData(N)
    wrongData = []
    for i in range(N):
        if correctData[i][1] == qubit0:
            wrongData.append([correctData[i][0], qubit1])
        else:
            wrongData.append([correctData[i][0], qubit0])
    return wrongData
```

Create csv files with the data for the plots:

```
def exporttocsv(name, data):
    with open(name, 'w', newline='') as csvfile:
        filewriter = csv.writer(csvfile, delimiter=',',\
        quotechar='"', quoting=csv.QUOTE_MINIMAL)
        filewriter.writerow(['a','b'])
        filewriter.writerows(list(map(list, zip(*data))))
```

Now the code for tests of the algorithm. Elementary tests:

```
network131 = randomNetwork([1,3,1], 50)
plotlist131, unitaries131 = qnnTraining(network131[0], network131[1],\
network131[2], 1, 0.1, 200)

for i in range(len(plotlist131[1])):
    if plotlist131[1][i] >= 0.75:
```

```
        print("Exceeds cost of 0.75 at training step "+str(i))
        break

plt.plot(plotlist131[0], plotlist131[1])
plt.xlabel("s")
plt.ylabel("Cost[s]")
plt.show()

exporttocsv('classif131.csv',plotlist131)

costFunc(network131[0], unitaries131, network131[2], 1000)

howManyWrong(network131[0], unitaries131, network131[2], 1000)
```

Create a list with the fractions of wrong classifications for several network architectures and a corresponding csv file:

```
errordata = [[], [], [], [], []]
for i in range(12):
    network11 = randomNetwork([1,1], 50)
    plotlist11, unitaries11 = qnnTraining(network11[0], network11[1],\
    network11[2], 1, 0.1, 400)
    error11 = howManyWrong(network11[0], unitaries11, network11[2], 1000)
    errordata[0].append(error11)
    network111 = randomNetwork([1,1,1], 50)
    plotlist111, unitaries111 = qnnTraining(network111[0], network111[1],\
    network111[2], 1, 0.1, 400)
    error111 = howManyWrong(network111[0], unitaries111, network111[2], 1000)
    errordata[1].append(error111)
    network131 = randomNetwork([1,3,1], 50)
    plotlist131, unitaries131 = qnnTraining(network131[0], network131[1],\
    network131[2], 1, 0.1, 200)
    error131 = howManyWrong(network131[0], unitaries131, network131[2], 1000)
    errordata[2].append(error131)
    network151 = randomNetwork([1,5,1], 50)
    plotlist151, unitaries151 = qnnTraining(network151[0], network151[1],\
    network151[2], 1, 0.1, 100)
    error151 = howManyWrong(network151[0], unitaries151, network151[2], 1000)
    errordata[3].append(error151)
    network1331 = randomNetwork([1,3,3,1], 50)
    plotlist1331, unitaries1331 = qnnTraining(network1331[0], network1331[1],\
    network1331[2], 1, 0.1, 200)
    error1331 = howManyWrong(network1331[0], unitaries1331, network1331[2], 1000)
    errordata[4].append(error1331)

print(errordata)

with open('errordata.csv', 'w', newline='') as csvfile:
    filewriter = csv.writer(csvfile, delimiter=',', quotechar='"',\
    quoting=csv.QUOTE_MINIMAL)
    filewriter.writerow(['1-1', '1-1-1', '1-3-1', '1-5-1', '1-3-3-1'])
    filewriter.writerows(list(map(list, zip(*errordata))))
```

Test of subset training:

```
subsetNetwork11 = randomNetwork([1,1], 50)

start = time() #Optional

pointsX = [5,10,15,20,25,30,35,40,45]
pointsAverage = [subsetTrainingAvg(subsetNetwork11[0], subsetNetwork11[1],\
subsetNetwork11[2], 1.5, 0.1, 200, 20, n, 1000, alertIt=20) for n in pointsX]

print(time() - start) #Optional

pointsAverageCost = []
pointsAverageError = []
for k in range(len(pointsAverage)):
    pointsAverageCost.append(pointsAverage[k][0])
    pointsAverageError.append(pointsAverage[k][1])
plt.plot(pointsX, pointsAverageError, 'bo')
plt.show()

exporttocsv('averagecost.csv', [pointsX,pointsAverageCost])
exporttocsv('averageerror.csv', [pointsX,pointsAverageError])
```

Test of the robustness to noisy data:

```
noiseTrainingData = randomTrainingData(100)
noiseNoisyData = [[randomQubitState(1), choice([qubit0,qubit1])] for i in range(100)]
noiseWrongData = wrongLabelData(100)

start = time() #Optional

noiseNetwork11 = randomNetwork([1,1], 0)
noisePlotList11, noiseNewList11 = noisyDataTraining(noiseNetwork11[0], noiseNetwork11[1],\
noiseTrainingData.copy(), noiseNoisyData.copy(), 1, 0.1, 300, 100, 5)

print(time() - start) #Optional

plt.plot(noisePlotList11[0], noisePlotList11[1], 'go')
plt.show()

plt.plot(noiseNewList11[0], noiseNewList11[1], 'go')
plt.show()

exporttocsv('noisecost11.csv',noisePlotList11)
exporttocsv('noiseerror11.csv',noiseNewList11)

start = time() #Optional

wrongLabelNetwork11 = randomNetwork([1,1], 0)
wrongLabelPlotList11, wrongLabelNewList11 = noisyDataTraining(\
wrongLabelNetwork11[0], wrongLabelNetwork11[1], noiseTrainingData.copy(),\
noiseWrongData.copy(), 1, 0.1, 300, 100, 5)

print(time() - start) #Optional

plt.plot(wrongLabelPlotList11[0], wrongLabelPlotList11[1], 'go')
plt.show()
```

```
plt.plot(wrongLabelNewList11[0], wrongLabelNewList11[1], 'go')
plt.show()

exporttocsv('wronglabelcost11.csv',wrongLabelPlotList11)
exporttocsv('wronglabelerror11.csv',wrongLabelNewList11)
```

qnn_classification_bloch_2.ipynb

Creation of the training data. The Bloch sphere is divided by a plane given by a normal vector "nVector" and a shift b.

```
def randomTrainingData(N, nVector, b):
    #numQubits = len(unitary.dims[0])
    trainingData=[]
    #Create training data pairs
    for i in range(N):
        stateIn = randomQubitState(1)
        rho = stateIn * stateIn.dag()
        bloch = blochvec(rho)
        stateOut = qubit1
        if np.dot(nVector, bloch)+b>0:
            stateOut = qubit0
        trainingData.append([stateIn,stateOut])
    #Return
    return trainingData

#for random plane:
#nVector = np.random.rand(3)
#nVector = nVector / np.linalg.norm(nVector)
#b = np.random.uniform(-0.8,0.8)

def randomNetwork(qnnArch, numTrainingPairs, nVector, b):
    assert qnnArch[0]==qnnArch[-1], "Not a valid QNN-Architecture."

    #Create the targeted network unitary and corresponding training data
    networkTrainingData = randomTrainingData(numTrainingPairs, nVector, b)

    #Create the initial random perceptron unitaries for the network
    networkUnitaries = [[]]
    for l in range(1, len(qnnArch)):
        numInputQubits = qnnArch[l-1]
        numOutputQubits = qnnArch[l]

        networkUnitaries.append([])
        for j in range(numOutputQubits):
            unitary = qt.tensor(randomQubitUnitary(numInputQubits+1),\
            tensoredId(numOutputQubits-1))
            unitary = swappedOp(unitary, numInputQubits, numInputQubits + j)
            networkUnitaries[l].append(unitary)

    #Return
    return (qnnArch, networkUnitaries, networkTrainingData)
```

The function "howManyWrong2" is the adapted version of "howManyWrong". A threshold c for the empirical probabilities is calculated, from which the value of b is computed:

```
def howManyWrong2(qnnArch, unitaries, trainingData, R):
    prob = measurements(qnnArch, unitaries, trainingData, R)
    m = len(trainingData)
    high = 1
    low = 0
    x = 0
    for i in range(m):
        if trainingData[i][1] == qubit0:
            if prob[i][0]<high:
                high = prob[i][0]
        else:
            if prob[i][0]>low:
                low = prob[i][0]
    c = (high + low) / 2
    bCalculated = 1 - 2*c
    for i in range(m):
        if trainingData[i][1] == qubit0:
            if prob[i][0] < c:
                x = x + 1
        else:
            if prob[i][1] < 1 - c:
                x = x + 1
    x = x/m
    return [x, c, bCalculated]
```

Tests of the algorithm. First, a list with the fractions of wrong classifications for several network architectures is created. For this, we use $R = 1$ and $R = 1000$.

```
errordata = [[], [], [], [], [], [], [], [], [], []]
for i in range(12):
    network11 = randomNetwork([1,1], 50, [0,1,0], 0.2)
    plotlist11, unitaries11 = qnnTraining(network11[0], network11[1],\
    network11[2], 1, 0.1, 400)
    error1 = howManyWrong2(network11[0], unitaries11, network11[2], 1)[0]
    errordata[0].append(error1)
    error1000 = howManyWrong2(network11[0], unitaries11, network11[2], 1000)[0]
    errordata[1].append(error1000)

    network111 = randomNetwork([1,1,1], 50, [0,1,0], 0.2)
    plotlist111, unitaries111 = qnnTraining(network111[0], network111[1],\
    network111[2], 1, 0.1, 400)
    error1 = howManyWrong2(network111[0], unitaries111, network111[2], 1)[0]
    errordata[2].append(error1)
    error1000 = howManyWrong2(network111[0], unitaries111, network111[2], 1000)[0]
    errordata[3].append(error1000)

    network131 = randomNetwork([1,3,1], 50, [0,1,0], 0.2)
    plotlist131, unitaries131 = qnnTraining(network131[0], network131[1],\
    network131[2], 1, 0.1, 200)
    error1 = howManyWrong2(network131[0], unitaries131, network131[2], 1)[0]
    errordata[4].append(error1)
    error1000 = howManyWrong2(network131[0], unitaries131, network131[2], 1000)[0]
```

```
    errordata[5].append(error1000)

    network151 = randomNetwork([1,5,1], 50, [0,1,0], 0.2)
    plotlist151, unitaries151 = qnnTraining(network151[0], network151[1],\
    network151[2], 1, 0.1, 100)
    error1 = howManyWrong2(network151[0], unitaries151, network151[2], 1)[0]
    errordata[6].append(error1)
    error1000 = howManyWrong2(network151[0], unitaries151, network151[2], 1000)[0]
    errordata[7].append(error1000)

    network1331 = randomNetwork([1,3,3,1], 50, [0,1,0], 0.2)
    plotlist1331, unitaries1331 = qnnTraining(network1331[0], network1331[1],\
    network1331[2], 1, 0.1, 200)
    error1 = howManyWrong2(network1331[0], unitaries1331, network1331[2], 1)[0]
    errordata[8].append(error1)
    error1000 = howManyWrong2(network1331[0], unitaries1331, network1331[2], 1000)[0]
    errordata[9].append(error1000)

print(errordata)
```

The algorithm is tested for several values of b:

```
datalist = []
for b in [0, 0.1, 0.2, 0.3, 0.4, 0.5]:
    errors = []
    bValues = []
    for i in range(12):
        nVector11 = np.random.rand(3)
        nVector11 = nVector11 / np.linalg.norm(nVector11)
        network11 = randomNetwork([1,1], 50, nVector11, b)
        plotlist11, unitaries11 = qnnTraining(network11[0], network11[1],\
        network11[2], 1, 0.1, 400)
        error11, c11, b11 = howManyWrong2(network11[0], unitaries11, network11[2], 1000)
        errors.append(error11)
        bValues.append(round(b11,3))
    datalist.append(errors)
    datalist.append(bValues)

print(datalist)
```

Finally, tests with $b = 0.7$ and $b = -0.7$:

```
network11 = randomNetwork([1,1], 50, [0,0,1], 0.7)
plotlist11, unitaries11 = qnnTraining(network11[0], network11[1], network11[2],\
1, 0.1, 400)
print(howManyWrong2(network11[0], unitaries11, network11[2], 1000))
print(measurements(network11[0], unitaries11, network11[2], 1000))
print(feedforward(network11[0], unitaries11, network11[2]))

network11 = randomNetwork([1,1], 50, [0,0,1], -0.7)
plotlist11, unitaries11 = qnnTraining(network11[0], network11[1], network11[2],\
1, 0.1, 400)
print(howManyWrong2(network11[0], unitaries11, network11[2], 1000))
print(measurements(network11[0], unitaries11, network11[2], 1000))
print(feedforward(network11[0], unitaries11, network11[2]))
```

References

[1] Esma Aïmeur, Gilles Brassard, and Sébastien Gambs. "Quantum speed-up for unsupervised learning". In: *Machine Learning* 90 (2013), pp. 261–287.

[2] Kerstin Beer et al. "Training deep quantum neural networks". In: *Nature Communications* 11.1 (2020), p. 808.

[3] Jacob Biamonte et al. "Quantum machine learning". In: *Nature* 549.7671 (2017), pp. 195–202.

[4] X.-D. Cai et al. "Entanglement-Based Machine Learning on a Quantum Computer". In: *Physical Review Letters* 114.11 (2015), p. 110504.

[5] Giuseppe Carleo and Matthias Troyer. "Solving the quantum many-body problem with artificial neural networks". In: *Science* 355 (2017), pp. 602–606.

[6] Vedran Dunjko and Hans J. Briegel. "Machine learning & artificial intelligence in the quantum domain: a review of recent progress". In: *Reports on Progress in Physics* 81.7 (2018), p. 074001.

[7] Vedran Dunjko, Jacob M. Taylor, and Hans J. Briegel. "Quantum-Enhanced Machine Learning". In: *Physical Review Letters* 117 (2016), p. 130501.

[8] Gerd Fischer. *Lineare Algebra - Eine Einführung für Studienanfänger*. 18th ed. Wiesbaden: Springer Spektrum, 2014.

[9] Sébastien Gambs. "Quantum classification". In: *arXiv:0809.0444 [quant-ph]* (2008). URL: http://arxiv.org/abs/0809.0444.

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

[11] Vojtěch Havlíček et al. "Supervised learning with quantum-enhanced feature spaces". In: *Nature* 567.7747 (2019), pp. 209–212.

[12] Carl W. Helstrom. "Quantum Detection and Estimation Theory". In: *Journal of Statistical Physics* 1 (1969), pp. 231–252.

[13] M. I. Jordan and T. M. Mitchell. "Machine learning: Trends, perspectives, and prospects". In: *Science* 349.6245 (2015), pp. 255–260.

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

[15] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2010.

[16] Jan Hendrik Pfau. *Classification-Python*. 2020. URL: https://github.com/qigitphannover/DeepQuantumNeuralNetworks/tree/master/Classification-Python.

[17] Patrick Rebentrost, Masoud Mohseni, and Seth Lloyd. "Quantum Support Vector Machine for Big Data Classification". In: *Physical Review Letters* 113.13 (2014), p. 130503.

[18] Sebastian Ruder. "An overview of gradient descent optimization algorithms". In: *arXiv:1609.04747 [cs]* (2017). URL: http://arxiv.org/abs/1609.04747.

[19] Gael Sentís et al. "Unsupervised Classification of Quantum Data". In: *Physical Review X* 9.4 (2019), p. 041029.

[20] E. Torrontegui and J. J. García-Ripoll. "Unitary quantum perceptron as efficient universal approximator". In: *EPL* 125 (2019), p. 30004.

[21] Vladimir N. Vapnik. *The Nature of Statistical Learning Theory*. Springer Science+Business Media, 2000.

[22] Peter Wittek. *Quantum Machine Learning: What Quantum Computing Means to Data Mining*. Academic Press, 2014. 176 pp.

[23] W. K. Wootters and W. H. Zurek. "A single quantum cannot be cloned". In: *Nature* 299 (1982), pp. 802–803.