Using Quantum Neural Networks to Cool Down Thermal States

Jannik Eggert

20.07.2020
Matrikelnummer: 10002369

Gottfried Wilhelm Leibniz Universität Hannover
Fakultät für Mathematik und Physik

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science

betreut durch Dmytro Bondarenko
unter Anleitung von Prof. Dr. Tobias J. Osborne

Contents

1 Introduction **1**

2 Classic Neural Networks **1**
2.1 Artificial Neurons . . . 1
2.2 Classical Neural Networks . . . 2
2.3 Learning Rules for Classical Neural Networks . . . 3
2.3.1 Cost . . . 4
2.3.2 Optimization . . . 4

3 Quantum-Neural Networks **6**
3.1 CPTP Maps . . . 6
3.2 Architecture of Quantum-Neural Networks . . . 7
3.3 Learning Rules for Quantum-Neural Networks . . . 9
3.3.1 Cost . . . 10
3.3.2 Optimization . . . 10
3.3.3 Learning Algorithm . . . 12

4 One-dimensional Transverse-field Ising Model and Thermal States **13**

5 Generating Data **14**

6 Training **14**
6.1 Mini-Batches . . . 18
6.2 Dynamic Learning Rate, Recursive Cost and Random Batches . . . 22

7 Results and Limits **25**
7.1 Many Batches . . . 26
7.2 Extrapolation . . . 27
7.3 Cooling Factor and Layer Structure . . . 30
7.4 Ratios of the Coupling Constant and the External Field . . . 33
7.5 Temperature Regions . . . 35

8 Conclusion and Outlook **38**

9 References **41**

1 Introduction

The research on algorithms, which we know today as machine learning algorithms, already began in the 1950th. But in the last decades those kinds of algorithms have gained tremendously on attention and success in research and industry, because the computer performance of affordable computers became increasingly capable of applying those learning algorithms on useful problems. Especially problems for helpful human computer interactions like recognition of images, speech or handwritten digits became more and more learnable for machine learning algorithms. [7]
Another field of research that is becoming increasingly interesting over the last few decades is quantum computing. Since the first quantum computation was made in 1990 on a quantum computer with just 7 qubits, the usability and performance of those devices haven been increased enormously [8] and maybe they will be capable of being used for industrial or personal purposes soon in the future.
Also, since quantum computing technology has been massively improved in the last few years and machine learning algorithms are algorithms of interest for a wide spread of problems, of course the first algorithms to implement machine learning on quantum devices have been developed yet. In this thesis, I will use the very general approach for implementing machine learning on quantum systems which is defined in [4].
Implementing this techniques on those devices is coming with a whole lot of advantages. Besides the main advantage, the exponentially growing calculation speed, another advantage is also very interesting. It is not needed to measure quantum input data and feed it in a classical device anymore. With those devices a learning algorithm and network can be directly applied on the given quantum data. [4] This makes quantum machine learning very applicable of solving problems where in- and output is stored in a quantum system.
Another interesting field of research at present are quantum simulators. To examine quantum phenomenons, a classical computer could need up to exponential growing resources to simulate linear growing systems. Not so with a quantum computer. Therewith the need for resources would also grow linear. [14] Those simulations work better when they are performed in low temperature regimes, which is just one reason why developing cooling algorithms is an interesting topic. In the following thesis I will examine and introduce a technique which uses quantum machine learning to cool down thermal states.

2 Classic Neural Networks

2.1 Artificial Neurons

A perceptron or artificial neuron is a composite function that was developed to imitate the functionality of biological neurons with a mathematical model. In particular, it imitates the information propagation of biological neurons through itself and can learn certain patterns when to get activated when not to. It is the building block of neural

networks. Perceptrons receive an input vector and output one or zero depending on if a weighted sum of the inputs is above or beneath 0. In formula:

$$p(x_i) = \begin{cases} 1 & \sum_i^n x_i w_i > 0 \\ 0 & else \end{cases}, \tag{1}$$

with x_i as the inputs, w_i as the weight vector, n as the number of input dimensions and $p(x)$ as the perceptron function. In some definitions the perceptron also adds a bias vector to the weighted inputs and uses an activation function to determine what is given back:

$$p(x_i)' = \sigma\left(\sum_i^n x_i w_i + b_i\right), \tag{2}$$

with x_i as the input vector, w_i as the weight vector, b_i as the bias vector, $\sigma(x)$ as the activation function, n as the number of dimensions and $p(x_i)'$ as the perceptron function with biases and an activation function.[7] To reproduce the first definition of a perceptron, the bias vector would be zero and the activation function would be a binary step function. Since this is the most general approach, I will use this definition for a perceptron in the following.

Usually the weights w_i, biases b_i are defined as real numbers and common activation functions are the sigmoid function, step functions, the hyperbolic tangent function among others.

2.2 Classical Neural Networks

Now that the architecture of a perceptron, the building block of neural networks, is clear, one can define a neural network. But in order to do that, it is helpful to define layers first. A layer is an amount of perceptrons and every perceptron within receives the same input vector - the input vector of the layer. It outputs a vector where every component is the output value of one perceptron within the layer. It can be described as:

$$l(x_i)_j = p_j(x_i), \tag{3}$$

with p_j as the $j-th$ perceptron of the layer and $l(x_i)$ as the function that represents the layer of perceptrons. So, the $j-th$ component of the output vector of the layer is the output of the $j-th$ perceptron of the layer. A neural network is then just an alignment of layers one after another. So, mathematically it can be formulated as:

$$N(x_i)_j = l^k(l^{k-1}(\ldots l^1(x_i)\ldots))_j, \tag{4}$$

with $l^k(x_i)$ as the layer function of the $k-th$ layer of the network and $N(x_i)$ as the network function which represents a network with $k+1$ layers. So, the components of the network function are just the components of the last layer function $l^k(x_i)_j$ which calculates its values by using the outputs of the layer next to itself $l^{k-1}(x_i)$ as input

components. The first layer $l^0(x_i)$ is also called the input layer and the last layer $l^k(x_i)$ is also called the output layer. The layers between these layers are called hidden layers. It is also worth mentioning that this definition makes it possible to vary the sizes of those hidden layers arbitrarily.[7]

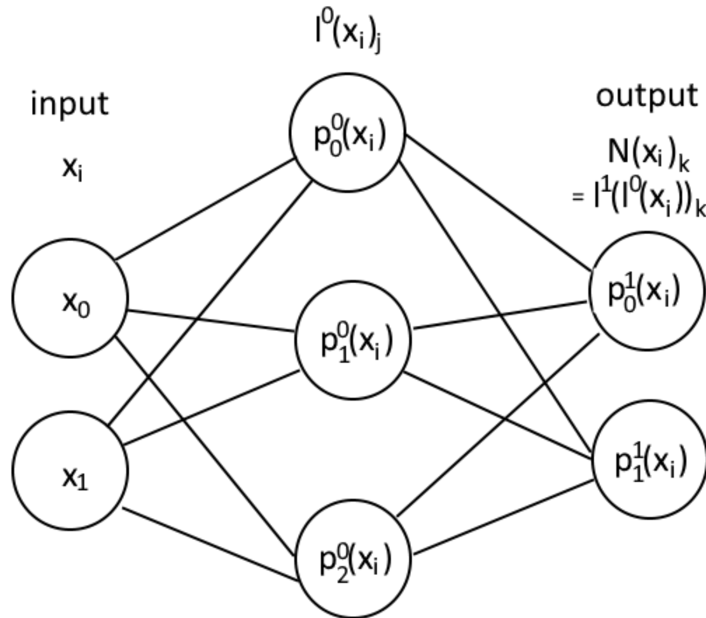

Figure 1: Functionality of a neural network with 3 perceptrons in the first layer in 2 perceptrons in the second layer. Every perceptron of the first layer receives the inputs of the network as inputs. The perceptrons in the second layer receive the output of the first layer as inputs. The outputs of the last layer are the outputs of the whole network.

2.3 Learning Rules for Classical Neural Networks

At first, during initialization, the weights w_i of every perceptron are usually set to random or predefined values. With such a configuration of weights the network will, of course, output values that are more or less meaningless. The aim of the network training is to change the weights of every perceptron in a way, that the predictions, meaning the outputs, of the network are the values the network is asked to predict. In order to do that, a set of inputs $(x_i)_j$ and their corresponding desired outputs $(label_i)_j$ are needed. Here, i stands for the component number and j is the number of that data point in the data set. The desired output of an input vector is often called the label of the input vector. With such a data set, an algorithm that changes the weights so that the network predicts the right labels for the inputs as precise as possible, can be defined. In other

words, the average error between the label of an input vector and the predicted output should be as small as possible. This error is also called cost. A function can be defined to measures the cost. The advantage of a defined cost function is that the learning task is then the task to find a set of weights and biases for every perceptron such that the cost is minimized.[7]

2.3.1 Cost

There are many ways to define such a function and there is no general cost function that works best for all learning tasks yet. It always depends on the problem and the data, which minimization of which cost function leads to the best accuracy. That is why I will shortly introduce an often used cost function.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(N(x)_i - label_i)^2, \tag{5}$$

where n is the dimensionality of the label and the network output, *label* is the label vector, x the input vector and $N(x)$ the input vector.
It is equivalent to the square of the distance between the network output and the label measured by the metric that is induced by the 2-Norm. It is called mean squared error is often used in regression tasks. But since this measures just the error between a label and a prediction, one step more is needed to define a cost function for the whole data set:

$$C = \frac{1}{m}\frac{1}{n}\sum_{j=1}^{m}\sum_{i=1}^{n}(N(x_j)_i - (label_j)_i)^2, \tag{6}$$

where C is the cost function, i is the index for the components of the output vector and the label vector, n is the number of dimensions of the label vector, m is the number of data samples in the data set and N is the network function.
This is the average mean squared error between the labels and the predictions averaged over the whole data set.[7]

2.3.2 Optimization

Now, with a defined cost function that can be minimized, a minimization algorithm can be made up. Over time, many minimization algorithms, which are also called optimizers, have been invented. The most simplest one of them is the gradient descent algorithm. Nearly all of the first order optimizers are based on this algorithm. That is why I will shortly introduce it.

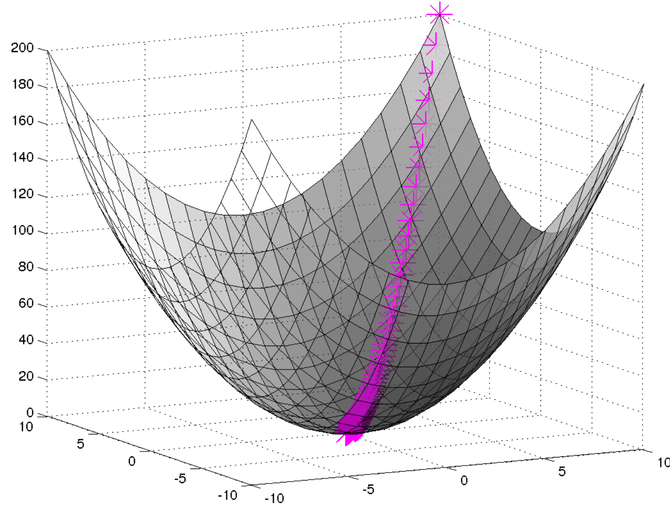

Figure 2: Functionality of a gradient descent algorithm. Starting from point $a_0 = (10, 10)$ it approaches the minimum of $f(x) = x^2 + y^2$ at $a_N = (0, 0)$ more and more with every iteration.[1]

The gradient descent algorithm is based on the assumption that a function decreases fastest in every point if you take a small but finite step in the direction of the negative gradient. Meaning:

$$F(x) > F(x - \lambda \nabla F(x)). \tag{7}$$

Here, $F(x)$ is the function that gets minimized. This can be proven for smooth convex functions as you can see in [11]. Furthermore, the algorithm based on this is iterative, starts with an arbitrary point a and updates this point each iteration by adding the negative gradient from the point to the point multiplied by a certain step size. It can be formulated as:

1. set random a_0
2. $a_{n+1} = a_n - \lambda \nabla F(a_n)$

With a_0 as the arbitrary starting point, $F(x)$ as the function to be minimized, λ as the step size and a_n as the point after the $n - th$ iteration.
Applied on classical neural networks the function F is the cost function C and a is any of the weights in the network. In the following, I will write all the weights of the network as a vector. Then, the gradient descent algorithm for weight updating can be written as:

1. set random w_0

2. $w_{n+1}^i = w_n^i - \lambda \frac{\delta C}{\delta w_n^i}$

Where C is the cost function, λ the step size factor, which is usually called learning rate in a machine learning context, and w_n^i is the $i-th$ weight of the network after the $n-th$ gradient descent step.
If you are working with large data sets and large networks, calculating the cost function tends to become the most time consuming part of this algorithm. This is why there is a technique where the cost function is just calculated on a representative subset of the data set. Those subsets are then called mini-batches. Applied on the above formulated cost function, you could write it as:

$$C_{mini-batch} = \frac{1}{m'} \frac{1}{n} \sum_{j=1}^{m'} \sum_{i=1}^{n} (N(x_j)_i - (label_j)_i)^2, \quad (8)$$

where $C_{mini-batch}$ is the mini-batch cost function, i is the index for the components of the output vector and the label vector, n is the number of dimensions of the out- and label vector, m' is the number of data samples in a representative subset of the data and N is the network function. A gradient descent algorithm applied on this cost function is then called a stochastic gradient descent optimizer and is often used when calculating the cost function becomes too time-consuming.[10]

Since the cost function needs to be calculated at least once each step in the gradient descent algorithm for each weight, this can still be very time consuming in big neural networks, even if a stochastic gradient descent optimizer is used. That is why another technique got invented, the backpropagation. In backpropagation the gradient of the cost function with respect to each weight is calculated for each layer by the chain rule. For one layer after another starting from the output layer. This technique makes it up to N times faster to update all the weights in a network because you can reuse the results of the previous layer while updating a layer. With N being the number of weights in the neural network. You can read more about it in [7].

3 Quantum-Neural Networks

In the following, I will elaborate a quantum neural network (QNN) architecture that implements the main functionality of classical neural networks on quantum devices. In particular, I will introduce the quantum neural network architecture for quantum inputs, that is described in [4].

3.1 CPTP Maps

A CPTP (complete positive trace preserving) map or quantum channel is a map acting on density matrices. It has the properties to be, first of all, linear. Secondly, it is positive to preserve the positivity of density matrices. It is also trace preserving because the input and the output density matrix needs to have trace 1. And, at last, it also

needs to be completely positive. Meaning that, if Φ is the channel, $I_n \otimes \Phi$ also needs to be positive for all n. In some literature the property of preserving traces is sometimes weakened. In those definitions it is enough that CPTP maps are not trace-increasing. But in the following I will continue with the more strict definition. Due to the Stinespring factorization theorem, in an open-system representation a CPTP map $\Phi : \mathcal{T}(\mathcal{H}_{in}) \rightarrow \mathcal{T}(\mathcal{H}_{out})$, with $\mathcal{T}(\mathcal{H})$ being the trace class operator space on the Hilbert space $\mathcal{H}$, can always be written as:

$$\Phi(\rho) = tr_{rest}(U(\rho \otimes |o\rangle \langle o|)U^{\dagger}), \tag{9}$$

where $\Phi(\rho)$ is the CPTP map applied on ρ, U is the representative isometry for Φ acting on $\mathcal{H}_{in} \otimes V$, $|o\rangle$ is a normalized vector in Hilbert space $V \subseteq \mathcal{H}_{out \otimes out}$ and tr_{rest} the partial trace over $\mathcal{H}_{in}$ and V, so that the outcome lives in the trace class operator space $\mathcal{T}(\mathcal{H}_{out})$.[2]
One can also say, that there is an adjoint of every CPTP map $\Phi^{\dagger}$. The condition for such an adjoint map is:

$$tr\left(\rho\Phi(\sigma)\right) = tr\left(\Phi^{\dagger}(\rho)\sigma\right), \tag{10}$$

with Φ as a CPTP-Map and $\Phi^{\dagger}$ as its adjoint form. This condition will help in the optimization algorithm of quantum-neural networks later. [2]

3.2 Architecture of Quantum-Neural Networks

At first, the quantum analog of the classic perceptron needs to be defined. Considering the perceptron needs to sum and weight up n different inputs and output m values, a general approach to formulate such a perceptron would be to assume it as a quantum channel $\mathcal{T}(\mathbb{C}^n) \rightarrow \mathcal{T}(\mathbb{C}^m)$. So a perceptron function can be written as:

$$\rho_{out} = tr_{in}(U(\rho_{in} \otimes |0...0\rangle_{out} \langle 0...0|)U^{\dagger}), \tag{11}$$

[4] with ρ_{out} as the output state, ρ_{in} as the input state and U as the unitary of the perceptron channel. The quantum analog of a layer would then be a multiplication of all perceptron unitaries within that layer. All in all the $i-th$ layer is then defined as:

$$U^i = U^i_L U^i_{L-1} ... U^i_1 = \prod_{n=1}^{L} U^i_{L-n}, \tag{12}$$

where L is the number of perceptrons the $i-th$ layer has and U^i_k is the $k-th$ perceptron of the $i-th$ layer. The layer function of the $i-th$ layer could thus be written as:

$$\rho_{out} = tr_{in}(U^i(\rho_{in} \otimes |0....0\rangle_{L_i} \langle 0...0|)U^{i\dagger}), \tag{13}$$

with ρ_{in} as the mixed input state, ρ_{out} as the mixed output state and L_i as the number of perceptrons in layer i. Now, the next thing to do is to formulate the equation for the

output of the whole network supposing it has more than one layer. But to make that easier, it is recommended to name the $i-th$ layer channel of the equation above:

$$\rho_{out} = \mathcal{E}^i(\rho_{in}) = tr_{in}(U^i(\rho_{in} \otimes |0....0\rangle_{L_i} \langle 0...0|)U^{i\dagger}), \tag{14}$$

where $\mathcal{E}^i(\rho_{in})$ is now the channel of the $i-th$ layer. With this definition, the whole network channel can be formulated a lot easier as:

$$\rho_{out} = \mathcal{E}(\rho_{in}) = \mathcal{E}^I(\mathcal{E}^{I-1}(\dots \mathcal{E}^1(\rho_{in}) \dots), \tag{15}$$

with I as the number of layers the network has, $\mathcal{E}^i(\rho)$ as the $i-th$ layer function and $\mathcal{E}(\rho)$ as the network channel.[4]

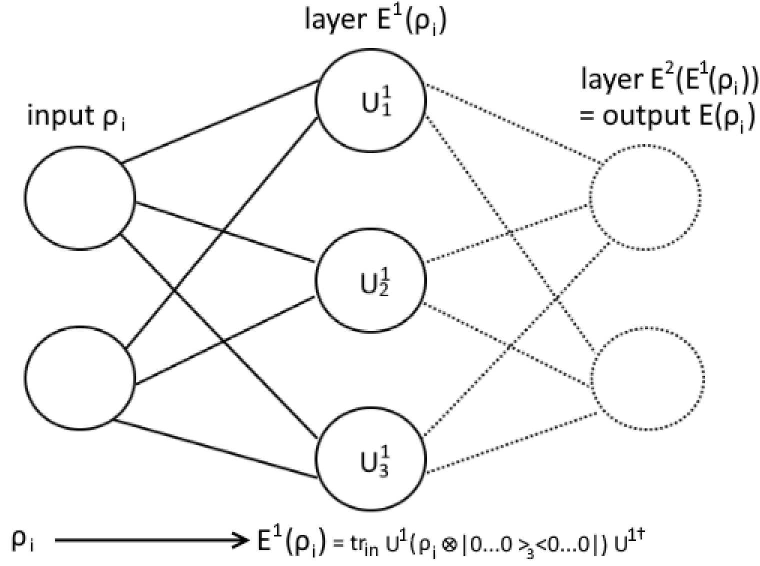

Figure 3: Feedforward functionality in QNNs. Here, the input-information propagates through the first layer channel. Once the state of the first layer ($\mathcal{E}^1$) is calculated by the above formula, the input space is traced out.

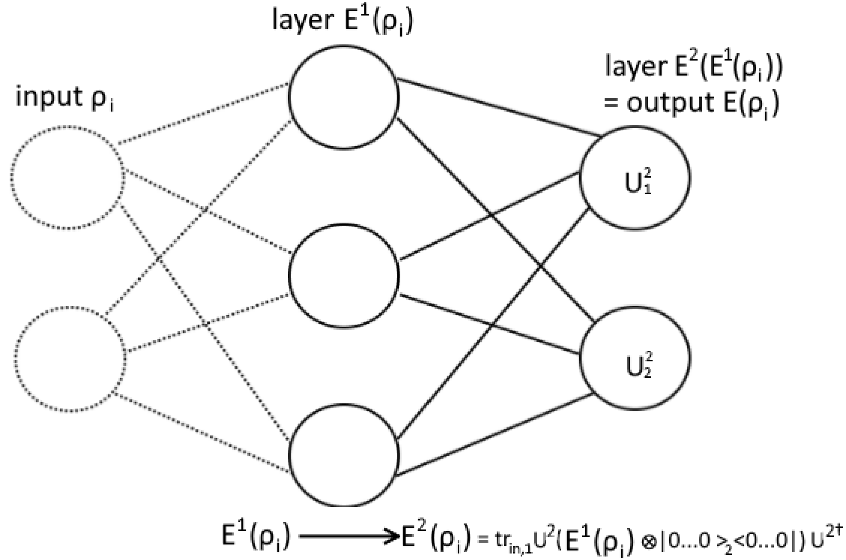

Figure 4: Feedforward functionality in QNNs. Here, the information stored in the second layer propagates through second (output) layer channel. Once the state of the output layer is calculated by the above formula, the space of the hidden layer ($\mathcal{E}^1$) is traced out.

3.3 Learning Rules for Quantum-Neural Networks

Since the model of a quantum neural network is defined, one can formulate an algorithm for training and optimizing the channel unitaries U_l^i. As in classic training algorithms it needs to be assumed, that we have got a big enough number of accessible learning pairs. In classical neural networks those are defined as $(x_i, label_i)$. Where x_i is the $i-th$ training example with the best possible output, the label $label_i$. In this case, inputs and outputs are quantum states. So one can define the training pairs as $(|\phi_{in}^i\rangle, |\phi_{label}^i\rangle)^{\otimes N}$, where $|\phi_{in}^i\rangle$ is the $i-th$ training example with the label $|\phi_{label}^i\rangle$ and N is the number of copies of every pair one needs for an accurate enough training, since one training pair can only be measured once. Therefore, the goal of this learning algorithm is to optimize the channel unitaries U_n^i so that the neural network maps as much as possible input examples to their labels as near as possible such as

$$\mathcal{E}(|\phi_{in}^i\rangle) = |\phi_{label}^i\rangle \,. \tag{16}$$

In view of this application example, where our in- and output states are mixed thermal states, the training pairs need to be defined as $(\rho_{in}^i, \rho_{label}^i)$ and the above written equation would then look like:

$$\mathcal{E}(\rho_{in}^i) = \rho_{label}^i \,. \tag{17}$$

[4] Once the training data is defined, one can develop a cost function.

3.3.1 Cost

As in classical algorithms to train neural networks, there need to be a method to measure how different the desired output of a network ρ_{label} is to the actual output $\rho_{out} = \mathcal{E}(\rho_{in})$ of the network to define a cost function. In quantum algorithms there is a very natural approach to measure the mathematical distance between the states, the fidelity. The fidelity of a pure and a mixed states is:

$$\mathcal{F}(\rho, |\phi\rangle) = \langle\phi| \rho |\phi\rangle . \tag{18}$$

In our case ρ_{label} and ρ_{out} are mixed. The fidelity of two mixed states is

$$\mathcal{F}(\rho, \sigma) = \left(tr \left(\sqrt{\sqrt{\rho}\sigma\sqrt{\rho}} \right) \right)^2 . \tag{19}$$

In general any measurement method, that measures the theoretical distance between two mixed states can be used, so in the following I will define the underlying distance function as $\mathcal{D}(\rho, \sigma)$. A distance function based on the fidelity would then be

$$\mathcal{D}(\rho, \sigma) = 1 - \mathcal{F}(\rho, \sigma), \tag{20}$$

because the only condition to this measurement method is that the measured distance between two equal states is 0 and 1 between two orthogonal states.
On the basis of this distance measurement method a function can be formulated, that measures the overall cost of the mapping of the neural network such as:

$$C = \frac{1}{N} \sum_{i=0}^{N} \mathcal{D}(\mathcal{E}(\rho_{in}^i), \rho_{label}^i). \tag{21}$$

With this definition, a function that needs to be minimized is given and the task of optimizing the channel unitaries U_l^i is now a task to minimize that function in view of the channel unitaries. Now, if the cost goes to zero while training, the training would be completed and every training input example ρ_{in}^i would be mapped to their desired output, so that $\mathcal{E}(\rho_{in}^i) = \rho_{out}^i = \rho_{label}^i$ for every i. [4]

Since the Hilbert-Schmidt norm (HSN) is measurable with a small circuit and I could not find an appropriate circuit that measures the fidelity of two states with as little effort as the HSN-circuit and it is not clear if fidelity has any advantages over HSN for our tasks, I will use the HSN as the underlying distance measurement in the following.[5]

3.3.2 Optimization

In classical backpropagation, the optimization steps can be interpreted as one calculates which error propagates from which weight into the prediction at first and then changes the weights in that direction where they should cause less error. The optimization of a quantum neural network is very similar to that.

Since there are no weights and just perceptron channel unitaries in quantum neural networks, the updating process needs to be different to the classical one. It needs to be ensured that the CPTP maps can only be changed in a way whereby they remain complete positive and trace preserving. The desired update rule is

$$U_l^{i,n+1} = e^{i\lambda K_l^i} U_l^{i,n}, \tag{22}$$

with K_l^i being arbitrary but self adjoint, i being the imaginary unit, λ being the learning rate and $U_l^{i,n}$ being the perceptron channel unitary of the $i-th$ perceptron in the $l-th$ layer after the $n-th$ step.

As in the classical gradient descent optimizer those K update matrices could now get calculated numerically. Here, this would mean to calculate the difference quotient of the cost function in the direction of every basis element of the K matrix space for every channel. While this algorithm can be executed on a quantum computer, it is slow and the search for more efficient quantum algorithms is an interesting research topic. For the classical simulation, however, it is much faster to evaluate the K update matrices for the gradient descent optimizer exactly. To calculate the cost function, one needs to either apply the quantum neural network to the input state $\mathcal{E}(\rho_{in})$, or apply the channel that is adjoint to the quantum neural network to the label state:

$$\sigma = \mathcal{F}(\rho_{out}), \tag{23}$$

with $\mathcal{F}$ as the adjoint network channel $\mathcal{F} = \mathcal{E}^\dagger$, ρ_{out} as any label and σ as the input state that would be mapped to the label ρ_{out} by the network. But you do not need to apply the whole adjoint network channel at once, you can also apply the adjoint layer channels one after another:

$$\sigma = \mathcal{F}^0(...\mathcal{F}^L(\rho_{out})...), \tag{24}$$

where $\mathcal{F}^l$ is $l-th$ adjoint layer channel $\mathcal{F}^l = \mathcal{E}^{l\dagger}$. Also:

$$\rho^l = \mathcal{E}^{l-1}(...\mathcal{E}^0(\rho^{in})...), \tag{25}$$

with ρ^l being the state of the $l-th$ layer after feeding the data point ρ^{in} forwards into the network, meaning applying all preceding layer functions $\mathcal{E}$ after another to the input state. And you can calculate which state a layer should have by feeding the label of the data point backwards into the adjoint network until the considered layer is reached:

$$\sigma^l = \mathcal{F}^{l+1}(...\mathcal{F}^L(\rho^{out})...), \tag{26}$$

where σ^l is the state, the $l-th$ layer should have so that the network would predict the desired label ρ_{out}. Now that you have the desired state of each layer and the actual state of the layers you can calculate the optimal update matrix K_l^i for every perceptron unitary in that layer by applying the adjoint of every following layer to the desired state, applying every preceding layer to the actual state, then calculating the commutator

of those two unitaries and tracing out everything except of the Hilbert space of the perceptron channel itself:

$$K_l^i = tr_{rest}\left(\left[U^i(\rho^{l-1} \otimes |0...0\rangle_{L_{l-1}} \langle 0...0|)U^{i\dagger}, U^{i\dagger}(\mathbb{1} \otimes \sigma^l)U^i\right]\right), \tag{27}$$

where K_l^i is the update matrix for the $l-th$ perceptron unitary in the $i-th$ layer. But since this is just the update matrix for the perceptron unitary for one data point, you need to calculate the average update matrix of all data points in the data set or mini-batch like in the following:

$$K_l^i = \frac{1}{N}\sum_{k=1}^{N} tr_{rest}\left(\left[U^i(\rho_k^{l-1} \otimes |0...0\rangle_{L_{l-1}} \langle 0...0|)U^{i\dagger}, U^{i\dagger}(\mathbb{1} \otimes \sigma_k^l)U^i\right]\right), \tag{28}$$

with N as the number of data points in the data set or batch.[4]

3.3.3 Learning Algorithm

As in [3], one can summarize the whole learning algorithm in the following steps:

1. Initialize: Start with randomly chosen U_j^l for all j and l
2. Feedforward: For every data point $(\rho_{in}^i, \rho_{out}^i)$, apply the layer channels l with the following steps one after another:
   2.1. Tensor the output of $\mathcal{E}^{l-1}$ to the projector of the normalized state $|0...0\rangle_{L_l} \langle 0...0|$
   2.2. Left-multiply the channel unitary U^l and right-multiply the adjoint channel unitary $U^{l\dagger}$
   2.3. Trace out layer $l-1$ and store the outcome
3. Update: For each perceptron channel unitary U_j^l perform the following steps:
   3.1 Calculate the update matrices K_j^l
   3.2 Update each perceptron channel unitary with $U_j^{l,n+1} = e^{i\lambda K_j^l} U_j^{l,n}$
4. Repeat from step 2 till the cost is small enough

This describes the quantum neural network analog of a simple gradient descent optimizer. In the following I will use an optimizer, that is similar to the one described above but based on the more complex Nadam optimizer instead of a simple gradient descent optimizer. You can find more about the Nadam optimizer in [10].

4 One-dimensional Transverse-field Ising Model and Thermal States

To extract statistical information about a system, one has to take a look at the canonical partition function. Defined as:

$$Z = \sum_{i}^{N} e^{-\dfrac{E_i}{k_B T}}, \tag{29}$$

where E_i is a possible energy of a particle, k_B the Bolzmann constant, T the temperature of the system and Z the canonical partition function. It contains information for the whole system. For example, you can extract the possibility for a particle to have the energy E_i with:

$$p_i = \frac{1}{Z} e^{-\dfrac{E_i}{k_B T}}, \tag{30}$$

with p_i as the probability for a particle to have the energy E_i In quantum mechanics the E_i is replaced with the Hamiltonian and it is traced in the end. So it can be written as

$$Z_{QM} = tr\left(e^{-\dfrac{\hat{H}}{k_B T}} \right). \tag{31}$$

And the quantum mechanical analog of the probability to extract a particle of a certain energy, the density matrix, can now be defined as

$$\hat{\rho} = \frac{1}{Z} e^{-\dfrac{\hat{H}}{k_B T}} = \frac{exp\left(-\dfrac{\hat{H}}{k_B T}\right)}{tr\left(exp\left(-\dfrac{\hat{H}}{k_B T}\right)\right)}. \tag{32}$$

This is the main connection between quantum mechanics and statistical physics. With this equation you can extract statistical information for every subsystem with just examining the Hamiltonian.[6] So, to make usage of this connection, a Hamiltonian needs to be defined. Since it is the goal to simulate the cooling of a system, one should use a Hamiltonian that is representative for most systems that can be cooled. Luckily, a very general Hamiltonian for quantum systems with dipoles arranged with a certain coupling and an external field, that affects the dipoles in their alignment, is already defined. It is the Hamiltonian of the transverse-field Ising model and is mathematically described with:

$$\hat{H} = J\sum_{i}^{N} \sigma_i^z \sigma_{i+1}^z + h\sum_{i}^{N} \sigma_i^x, \tag{33}$$

with the boundary condition $\sigma_{N+1}^z = \sigma_0^z$ and with σ_i^x being the Pauli spin-1/2 matrix in x direction of the particle i, J being the coupling constant in x direction and h being the factor of the outer field. Since this Hamiltonian is very general and a big group of other systems can be represented by this Hamiltonian, it is enough to show that these kind of systems can be cooled by this technique to show that other systems could also be cooled.[9,12] Also worth mentioning is that calculations on D-Wave quantum computers are made by modeling and manipulating problems with the Ising model.[13] The probable usage of this technique in those machines would be reason enough to examine these kinds of systems.

5 Generating Data

To train an artificial neural network, if its quantum or not, a big number of training examples and their labels is needed. As in interpolating functions the model of a function gets better the more data points are provided. So, to provide a big number of data points to train this QNN, it is necessary to develop a function that returns a big number of states with different temperatures and their labels. This is done in a loop over the wished number of data. Within every iteration a random temperature gets generated. For this temperature, a Hamiltonian for a predefined number of particles gets calculated by another function. Out of this Hamiltonian the program calculates a density matrix which is now the data point for this temperature. Once finished, the program then does the same for a bit lower temperature. That density matrix is then the label for that data point. It is predefined how much the label density matrix is cooler then the data density matrix. I got the best training results by using a certain percentage of data temperature as the label temperature. I also tried to use the data temperature minus a fixed number of degrees as label temperature or to approximate the quantum-neural network system as a thermal bath with a certain temperature in order to calculate most natural cooling factor for the label temperature, but nothing worked significantly better, so cooling by a certain percentage was used in the following.

6 Training

The following parameters for training a quantum neural network are to be specified:

Parameter	Description
T_{min}	The minimum temperature of an input data point for training
T_{max}	The maximum temperature of an input data point for training
$N_{particles}$	The number of input and output particles
N_{data}	The number of data points in the training data set
$h_{training}$	The external field factor of the input data
$j_{training}$	The coupling constant of the input data
$\delta T_{training}$	The cooling factor ($T_{input}\delta T_{training} = T_{output}$)
λ	The learning rate of the QNN
$iterations$	How many times every perceptron channel unitary gets updated.
M	Number of perceptrons per layer

Here, for example $M = [232]$ would mean, that there are 3 layers. The first and the last layer would have 2 perceptrons and the middle layer would have 3. In the first attempts of training a quantum neural network, I used the following set of parameters for training:

	Attempt			
Parameter	1	2	3 4	
T_{min}	80	12	4	4
T_{max}	100	20	8	8
$N_{particles}$	2	2	2	2
N_{data}	1000	1000	1000	1000
$h_{training}$	1	1	1	1
$j_{training}$	1	1	1	1
$\delta T_{training}$	0.8	0.8	0.8	0.8
λ	1	1	1	1
$iterations$	20	20	20	1000
M	[2 2 2]	[2 2 2]	[2 2 2]	[2 5 2]

During training, the cost functions in every attempt went nearly steady to nearly 0. As you can the in the following figure (figure 5).

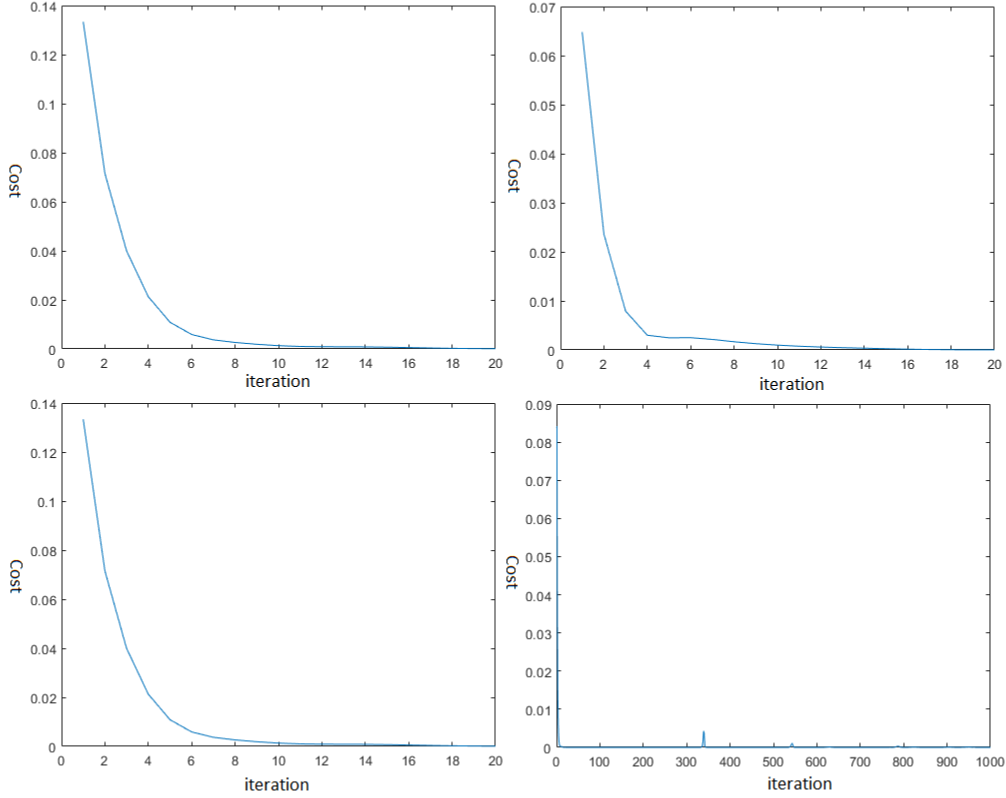

Figure 5: Cost functions for attempt 1-4. Top left: attempt 1 with T $\in [80, 100]$. Top right: attempt 2 with T $\in [12, 20]$. Bottom Left: attempt 3 with T $\in [4, 8]$. Bottom Right: attempt 4 with $T_{training} \in [4, 8]$ and with more training data and a wider network than in attempt 3. All of them are approximately 0 in the end of the training.

But since cost functions are just a measurement for how precise the training data is mapped to their labels by the network, it is mandatory to determine how precise new data will be mapped to their labels. To do so, I will first define a function that displays the difference of a given density matrix to another thermal state with a given temperature. From now on, I will call it a comparison function. A comparison function is in formula:

$$F_{\rho}^{comp}(T) = \mathcal{D}(\rho, \rho_{thermal}(T)), \tag{34}$$

with F_{ρ}^{comp} as the comparison function of the density matrix ρ, T as an arbitrary temperature, $\mathcal{D}$ as an arbitrary measure and $\rho_{thermal}$ as the thermal state related to the temperature T. So, the comparison function of a given ρ is the distance of that ρ to the thermal state of the temperature T. Where the distance is, of course, defined by the underlying measure.

For each attempt a set of 6 thermal states was calculated on temperature grid of 6 equidistant temperatures. Also, a label set of 6 thermal states was calculated on the same temperature grid multiplied by δT. In the end, the network was applied to the first 6 thermal states to calculate a prediction set of 6 thermal states. In figure 6, you can see the comparison functions for the label set (purple to red) and the prediction set (black to green) for each attempt.

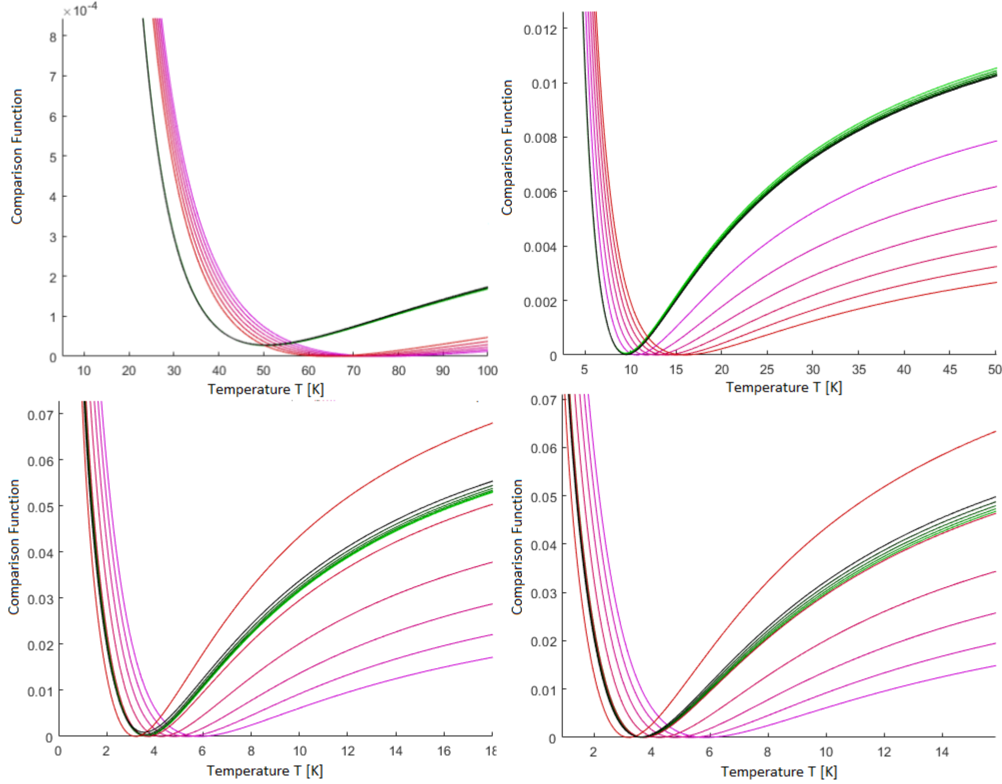

Figure 6: Comparison functions of the predicted states and the label states of attempt 1-4, with the Hilbert-Schmidt norm as underlying norm. Top left: attempt 1 with $T_{training} \in [80, 100]$. Top right: attempt 2 with $T_{training} \in [12, 20]$. Bottom: attempt 3 with $T_{training} \in [4, 8]$. Bottom Right: attempt 4 with $T_{training} \in [4, 8]$ and with more training data and a wider network than in attempt 3.

As you can see, the predicted thermal states are kind of similar to the label thermal states. But if you take a closer look you can also see, that all the predicted thermal states are very close to each other. That means, that the network does cool the input states to a certain degree, but does not differentiate enough between the inputs. What also can be seen is, that every predicted states is more similar to a thermal state of a lower temperature than the actual label. All in all it can be concluded, that the network does cool some states too much and some states too little. Attempt 1 to 3 just differ in

the training temperature interval, but even with changing all the other parameters, the results do have the same kind of defect every time. The only tendency one can observe is that the results are getting a bit better with lower temperatures and with smaller input temperature intervals. Attempt 4 was made with the same parameters as the best Attempt of 1-3, but with the usage of much more computational resources, to examine how the algorithms accuracy increases with more computational resources. Since it is easy to see that it does increase with more resources I will implement one attempt with a maximum of resources within every series of tests to see which accuracy can be reached with that algorithm. So, in order to get better results, it should be examined what can be done better within the algorithm for training.

6.1 Mini-Batches

As the predictions of the neural network are too close to each other, it seems like the cost function has a local minimum. A local minimum where the network - to exaggerate - predicts a constant state independently of the input. So, the algorithm for training should be modified in a way so that it is not that susceptible for being stuck in a local minimum anymore. In order to do that, lets take a look at the stochastic gradient descent algorithm again. The point of that kind of algorithm is to speed up the calculation of the cost function to make the training algorithm faster. But it also has the risk of adding too much randomness to the cost function in case of too small mini-batches, where too small means that the mini-batches are not representative for the data set anymore. Now, since a way to get out of a local minimum is to add noise to the optimizer, a possible approach to make the training algorithm less susceptible for getting stuck in a local minimum is to calculate the cost function on non representative subsets of the data. Meaning, to train the network on too small mini-batches which are getting exchanged after a certain number of iterations. To make sure that these mini-batches are not representative for the data, I will not only let them be really small but also let them get calculated on a random temperature subinterval of the actual training temperature interval. So, a few redefinitions in the training parameters are needed:

Parameter	(new) Description
$N_{training}$	The number of mini-batches the network will be trained on.
N_{data}	The number data points in a mini-batch
$\lvert T_{mini-batch} \rvert$	The temperature subinterval size, the mini-batches are calculated on
$iterations$	How many times every perceptron channel unitary gets updated per mini-batch

To examine if the new training algorithm leads to better results, I will now carry out training sessions on nearly the same parameters as in attempt 1-4 but now with the new feature of mini-batches. I will also use different sizes of temperature subintervals

to evaluate if this new technique makes a difference. So, the parameters for attempt 5-8 are:

	Attempt			
Parameter	5	6	7	8
T_{min}	4	4	4	4
T_{max}	8	8	8	8
$\|T_{mini-batch}\|$	1	2.5	4	4
$N_{particles}$	2	2	2	2
N_{data}	6	6	6	60
$N_{training}$	20	20	20	20
$h_{training}$	1	1	1	1
$j_{training}$	1	1	1	1
$\delta T_{training}$	0.8	0.8	0.8	0.8
λ	1	1	1	0.1
$iterations$	50	50	50	500
M	[2 2 2]	[2 2 2]	[2 2 2]	[2 5 2]

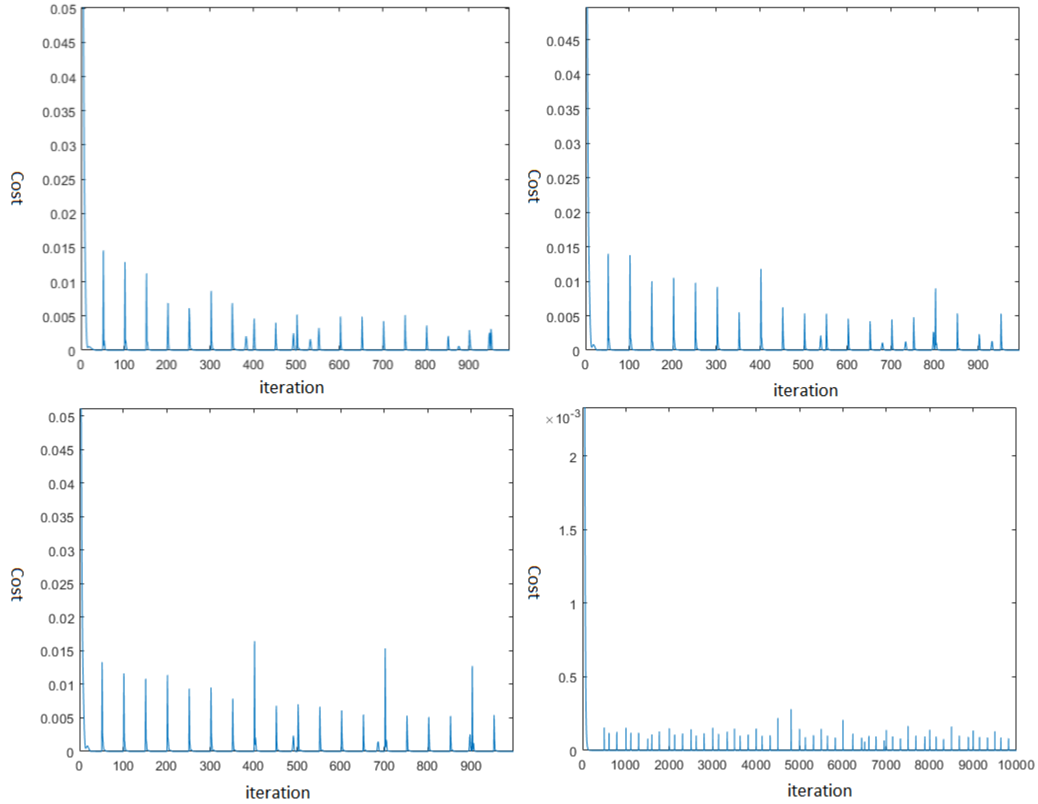

Figure 7: Cost functions for attempt 5-8. Top left: attempt 5 with $|T_{mini-batch}| = 1$. Top right: attempt 6 with $|T_{mini-batch}| = 2.5$. Bottom left: attempt 7 with T $|T_{mini-batch}| = 4$. Bottom Right: attempt 8 with $|T_{mini-batch}| = 4$ and with more training data and a wider network than in attempt 7. All of them are approximately 0 in the end of the training of every mini-batch. The peaks when starting a new batch are decaying to a certain value.

What is noticeable in the cost functions of the attempts with the new mini-batch technique is that there are small peaks. Those peaks are the effects of starting to train on a new mini-batch.

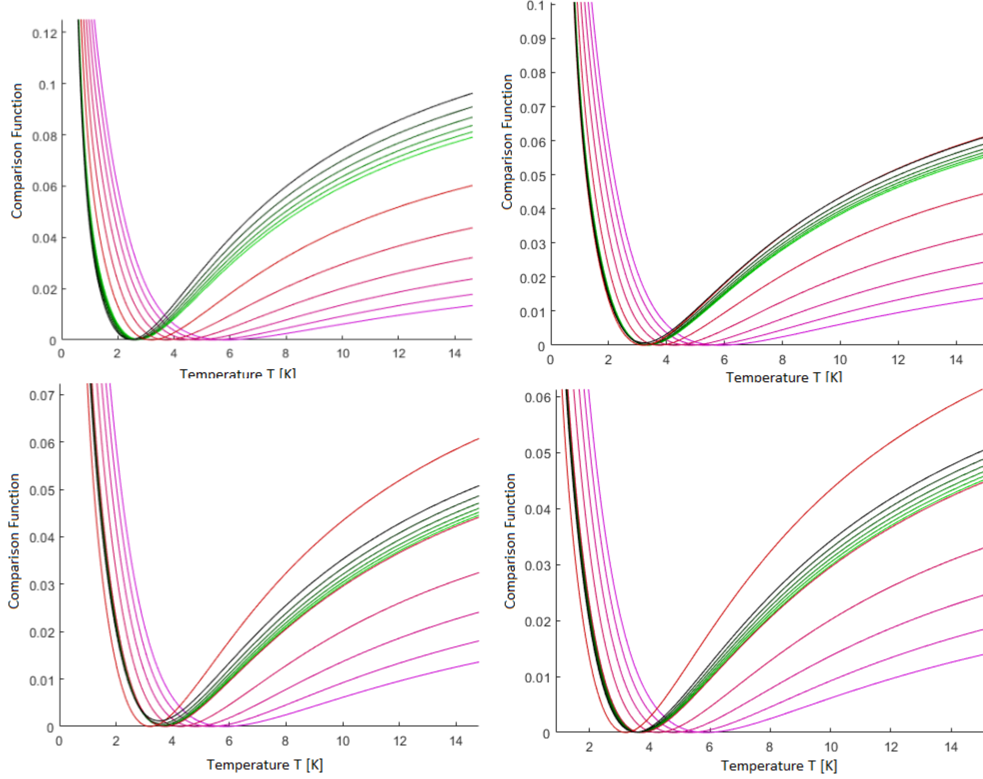

Figure 8: Comparison functions of the predicted states and the label states of attempt 5-8, with the Hilber-Schmidt norm as underlying norm. Top left: attempt 5 with $|T_{mini-batch}| = 1$. Top right: attempt 6 with $|T_{mini-batch}| = 2.5$. Bottom: attempt 7 with $|T_{mini-batch}| = 4$. Bottom Right: attempt 8 with $|T_{mini-batch}| = 4$ and with more training data and a wider network than in attempt 7.

In figure 8 you can see the comparison functions for the latest attempts. You can see very well, that the predicted states went apart. Meaning, that the network cares more about which input is given now, as it was desired. Also, the functions are now in a less wrong region, which means, that the network became more precise in general. Surprisingly, the network that was trained on mini-batches on the whole temperature sub interval, had the best accuracy. This means, that the technique of using smaller temperature sub intervals for the mini-batches does not seem to be useful. Also surprising is the quality of the results of attempt 8, since those results are not significantly better than the results of attempt 7, where only a fractional of computational resources was used. This can either be attributed to the fact that λ is not as well chosen for the bigger layer structure as it is for the smaller once since it is a slow progress to find a good λ for a certain layer structure. Or it could be attributed to the fact that the ability to learn this kind of problem does not scale that much with the number of perceptrons. But all in all the mini-batches are a good new extension to the training algorithm and they seem to make the trained network much more precise. But the results could be

better. That is why one should examine what else could be done better in the training algorithm.

6.2 Dynamic Learning Rate, Recursive Cost and Random Batches

As you can see in the cost functions of the latest series of tests in figure 7, the cost function peaks at the beginning of every new mini-batch are decaying to a certain value. Testing the network on a test set is to apply it to unknown data, which is exactly like applying it to a new mini-batch. So, you can say that if the network should predict unknown data more precisely, those peaks need to decay to a lower value and in the best case to zero. In order to do that, one should examine how to lower this decay limit. A possible factor that could cause such a limit is that the updating steps for updating the perceptron maps are too big. Meaning, that the learning rate of the optimizer is too big. But since one can not just reduce the learning rate, because that would lead the cost function to get stuck in a local minimum, it would may be a good approach to lower the learning rate polynomially with every new mini-batch. So, the learning rate of the optimizer λ could get calculated with the following formula:

$$\lambda_n = \frac{1}{\frac{1}{\lambda_{max}} + n^{p_\lambda}\left(\frac{\frac{1}{\lambda_{min}} - \frac{1}{\lambda_{max}}}{N_{training}^{p_\lambda}}\right)}, \tag{35}$$

where n is the batch number and λ_n the learning rate λ for the $n-th$ batch.
Another good technique to train to a more precise network is to calculate the cost function not only out of the difference of the label and the predicted state but also the difference of the second label and the second application of the network to the input state. The second label is meant to be the actual label state but cooled by the cooling factor once more ($T_{input}\delta T_{training}\delta T_{training} = T_{2nd\,label}$). In the following I will call this technique "recursive cost". The Idea of this technique is to increase the impact of the error of the first prediction to the cost function. Due to error propagation, the measurable error after the second apply of the network should be much bigger than the error when the network is just applied once. In formula, one could the write the new cost function as:

$$C_{mini-batch} = \frac{1}{m'}\sum_{i=1}^{m'}\left(\mathcal{D}(\mathcal{E}(\rho_{in}^i), \rho_{label}^i) + \mathcal{D}(\mathcal{E}(\mathcal{E}(\rho_{in}^i)), \rho_{label}^{2nd,i})\right), \tag{36}$$

where m' is the number of data points in the mini-batch, ρ_{in}^i the input state of the $i-th$ data point, ρ_{label}^i the label state of the $i-th$ data point, $\rho_{label}^{2nd,i}$ the second label of the $i-th$ data point, $\mathcal{E}$ the network map, $\mathcal{D}$ the underlying measurement function and $C_{mini-batch}$ the overall cost function for that mini-batch. To implement those two new features in the cost function, a few new definitions in the training parameters are needed:

Parameter	(new) Description
λ_{min}	The minimum learning rate λ
λ_{max}	The maximum learning rate λ
p_λ	The polynomial degree of the function that calculates λ for every new batch

Also, since the smaller batch temperature intervals did not work out so well, I am going to try a slightly different technique. To add even more randomness to the batch temperature intervals I am not only going to chose their borders randomly, but also their temperature interval sizes.
To evaluate all three new extensions I will run 4 tests. In the first test, I will just apply the recursive cost technique, to evaluate the worth of this technique. In the second test, I will also apply the random sized batch temperature intervals, to evaluate this extension. In the third and fourth, I will try out two different degrees for the polynomial decaying λ, to learn how much difference this technique makes. For every single test, I will now use a maximum number of batches and perceptrons for training to make the difference more visible and to ensure, that the lack of batches or perceptrons does not effect the quality of the results.

	Attempt			
Parameter	9	10	11	12
T_{min}	4	4	4	4
T_{max}	8	8	8	8
$\lvert T_{mini-batch}\rvert$	4	rnd $\in [0,1]$	rnd $\in [0,1]$	rnd $\in [0,1]$
$N_{particles}$	2	2	2	2
N_{data}	6	6	6	6
$N_{training}$	100	100	100	100
$h_{training}$	1	1	1	1
$j_{training}$	1	1	1	1
$\delta T_{training}$	0.8	0.8	0.8	0.8
λ_{min}	1	1	0.00025	0.00025
λ_{max}	1	1	1	1
p_λ	0	0	1	3
$iterations$	50	50	50	50
M	[2 5 2]	[2 5 2]	[2 5 2]	[2 5 2]

In figure 10 you can see the cost functions of the test introduced above. One can see, that the peaks decay really slow and converge to a certain value, if no random batch temperature interval sizes were used. In the cost function for attempt 10 one can see very clearly that the effort of learning a new batch is now very randomly distributed

due to the random distributed temperature interval sizes. It should also be mentioned that every batch is fully learned after the 50 iterations, because the costs decays to nearly zero for every batch. In attempt 11 and 12 one can see, that the peak heights are still randomly distributed, but with a decaying λ the new batches are not getting fully learned. Maybe this could be attributed to too less batches and consequently to a too fast decaying λ. This is why I will also run a test with even more batches in the next section (7.1).

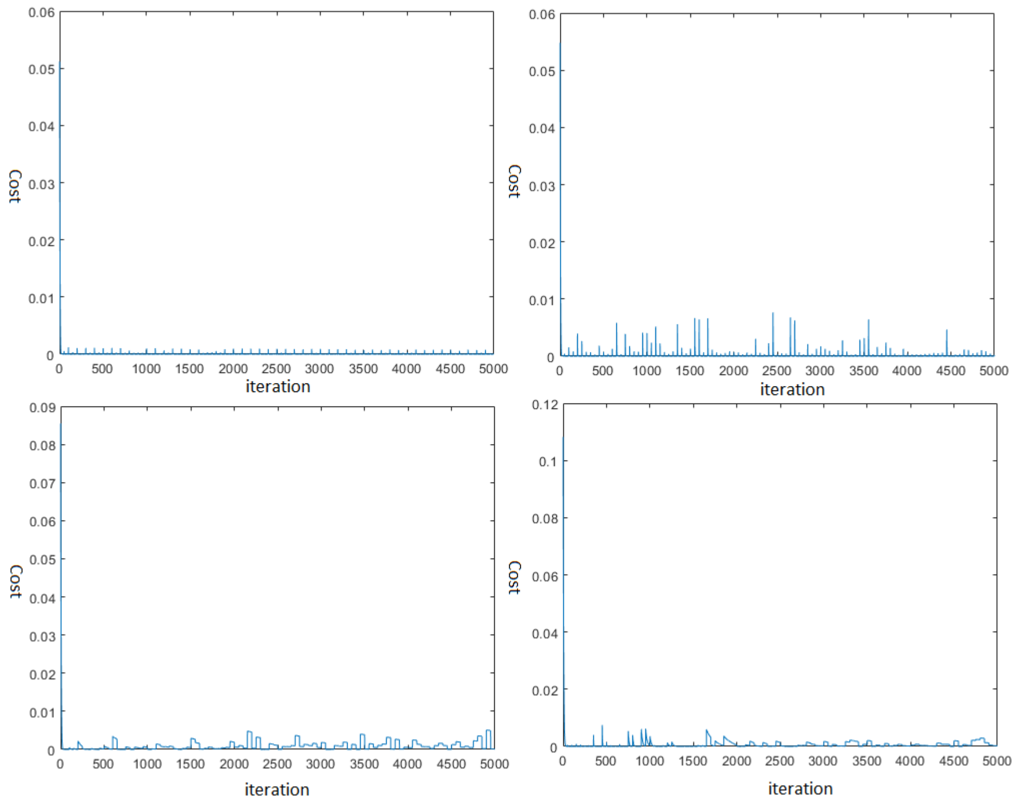

Figure 9: Cost functions for attempt 9-12. Top left: attempt 9 with just the recurrent cost function. Top right: attempt 10 with the recurrent cost function and random batch temperature interval sizes. Bottom left: attempt 11 with every new extension and a linearly decaying λ. Bottom Right: attempt 12 with every new extension and the λ-decay degree of 3.

In figure 10 you can see the comparison function of the above introduced tests. One can see that the results are slightly better distributed with the recurrent cost function, but they are still pretty close to each other. It is also noticeable that the recurrent cost somehow made the network to output states on the upper band of desired labels. The most difference made the random distributed batch temperature size which is first applied in test 10 on the top left of the picture. The predictions are now pretty far apart

from each other and nearly equal to their actual labels. In the comparison functions for the attempts 11 and 12 one can see, that the decaying λ did not made such a difference. One could even assume that this technique makes the training algorithm even worse, but this could also be attributed to the problem described above in the discussion of the cost functions of this tests.

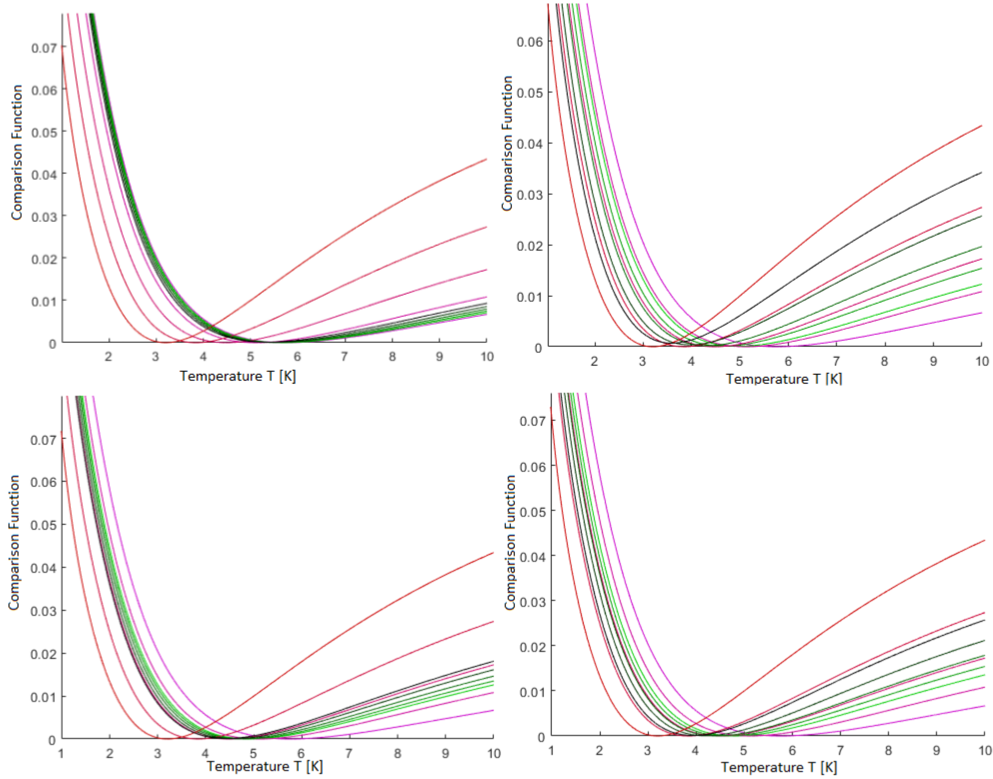

Figure 10: Comparison functions of the predicted states and the label states of attempt 9-12, with the Hilbert-Schmidt norm as underlying norm. Top left: attempt 9 with just the recurrent cost function. Top right: attempt 10 with the recurrent cost function and random batch temperature interval sizes. Bottom left: attempt 11 with every new extension and a linear decaying λ. Bottom Right: attempt 12 with every new extension and the λ-decay degree of 3.

7 Results and Limits

All the test series were performed on nearly the same set of training parameters to evaluate the quality of the advanced training algorithm features. But, as mentioned, I also made sure that those added training algorithm features are as good for other parameter sets as they are for the above chosen parameter set. Nevertheless, it is interesting to see how the network architecture and its training algorithm performs

on different parameter sets like an completely different ratio of the coupling constant $j_{training}$ to the external field $h_{training}$ or complete different temperature regions. Also a different cooling factor or network architecture would sure be interesting. What also still needs to be done is testing a decaying λ with very many batches. Therefore, I will try out those parameter set regions and evaluate the performance of the architecture and its training algorithm in order to estimate the worth of this technique to cool down thermal states.
As you can see in the following table, to make sure the results are affected as little as possible by a not sufficient network size or the amount of training, I will perform this test series exclusively on the parameter configurations of nearly the maximum computing resources. At least the amount of iterations and batches can then be ruled out as an error source. The network layer structure M and especially the amount of perceptrons in the hidden layer could still lead to horrendous performance losses. But since the time needed to train the network can scale exponentially with the layer width and the above mentioned parameter configurations are already challenging for my processing unit, it is difficult to set up a test series with a bigger layer width. Therefore, one needs to assume that the performance of the network does scale with the layer width as moderately as it does in the first three test series.

7.1 Many Batches

As mentioned in the section above, the decaying λ method still needs to be applied on a very big amount of batches. Therefore I will repeat attempt 12 as test 1, but with 2000 batches this time.

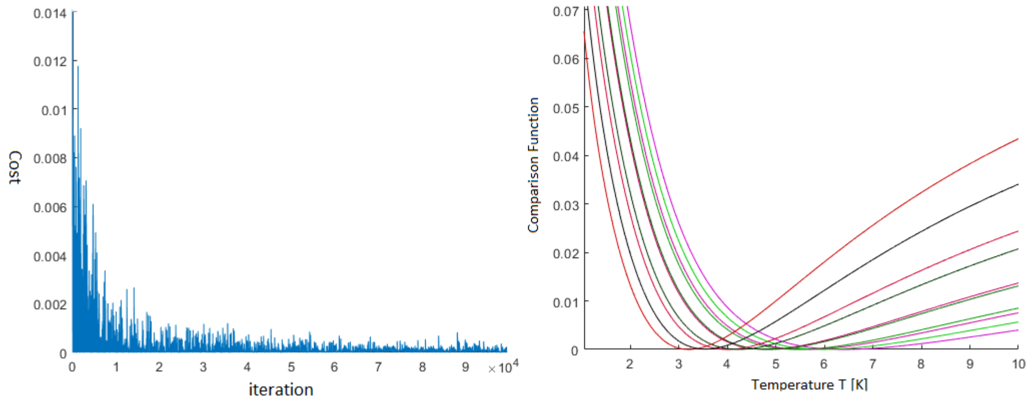

Figure 11: Cost and comparison functions of the predicted states and the label states of test 1, with the Hilbert-Schmidt norm as underlying norm. Left: cost function, Right: comparison functions

On the left in figure 11 one can see the cost function for test 1, which is the same as attempt 12 but trained with 2000 batches. One can see well, that the peaks of the

cost function when starting to learn a new batch, are decaying to nearly zero. With the comparison to the cost of attempt 12 it can be interpreted that the decaying learning rate method needs very many batches to finish training, while the same training method without a decaying learning rate have already nearly finished training after 100 batches. If you take a look at the comparison function of the label states and the predicted states in attempt 13, you can also see, that the predicted states nearly match the label states, or at least better than the predicted states of the network that was training without a decaying λ in attempt 12. Therefore one can say that if the computational resources are given to perform training on a big enough amount of batches, the training method with a decaying λ is the better choice.

7.2 Extrapolation

Every network was only tested on data out of a temperature region it was trained on. Meaning, it was only applied to states, which it was also trained on. Now, I want to examine how the best network (the network of test 1 in 7.1, which was trained with the parameters of attempt 12), performs when it needs to "cool down" states out of a region it has never "seen" while training not even as second labels. Does the network still cool down those states and if so, what are the limits?

In the following figure [12] you can see the comparison functions of the input states (red to pink) and the outputs (black to green) of the network. Note that the red to pink states are now the input states and not the labels of the input states anymore. Meaning that the network maps the red state to the black state, the pink state to the green state and so on.

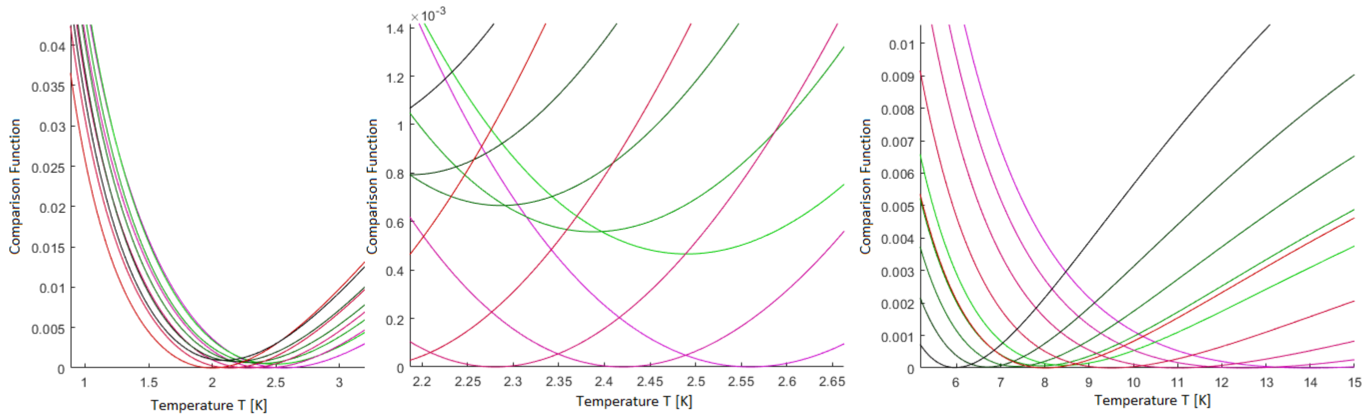

Figure 12: Extrapolation with the network of test 1. On the left and in the middle: the input states (red to pink) and the output states (black to green) of the network in a temperature region below the temperature region the network was trained on. On the right: the input states (red to pink) and the output states (black to green) of the network in a temperature region above the temperature region the network was trained on.

On the left of figure 12 one can see the performance of the network in a temperature region below the temperature region the network was trained on. One can see that the predictions are getting worse, the more you move away from the training temperature region. But it is also noticeable, that the network still cools the input states down to an input temperature of approximately 2.3 K. On the right of the figure one can see the in- and output states of the network in a temperature region above the temperature region it was trained on. One can see that the network cools down those states nearly perfectly, besides the fact, that it cools more then it should, the more one moves away from the training temperature interval.

If one neglects that the outputs similarity to an actual thermal state fades away with more distance to the training temperature interval, one can clearer plot the in- and output temperatures as you can see in the following figure [13].

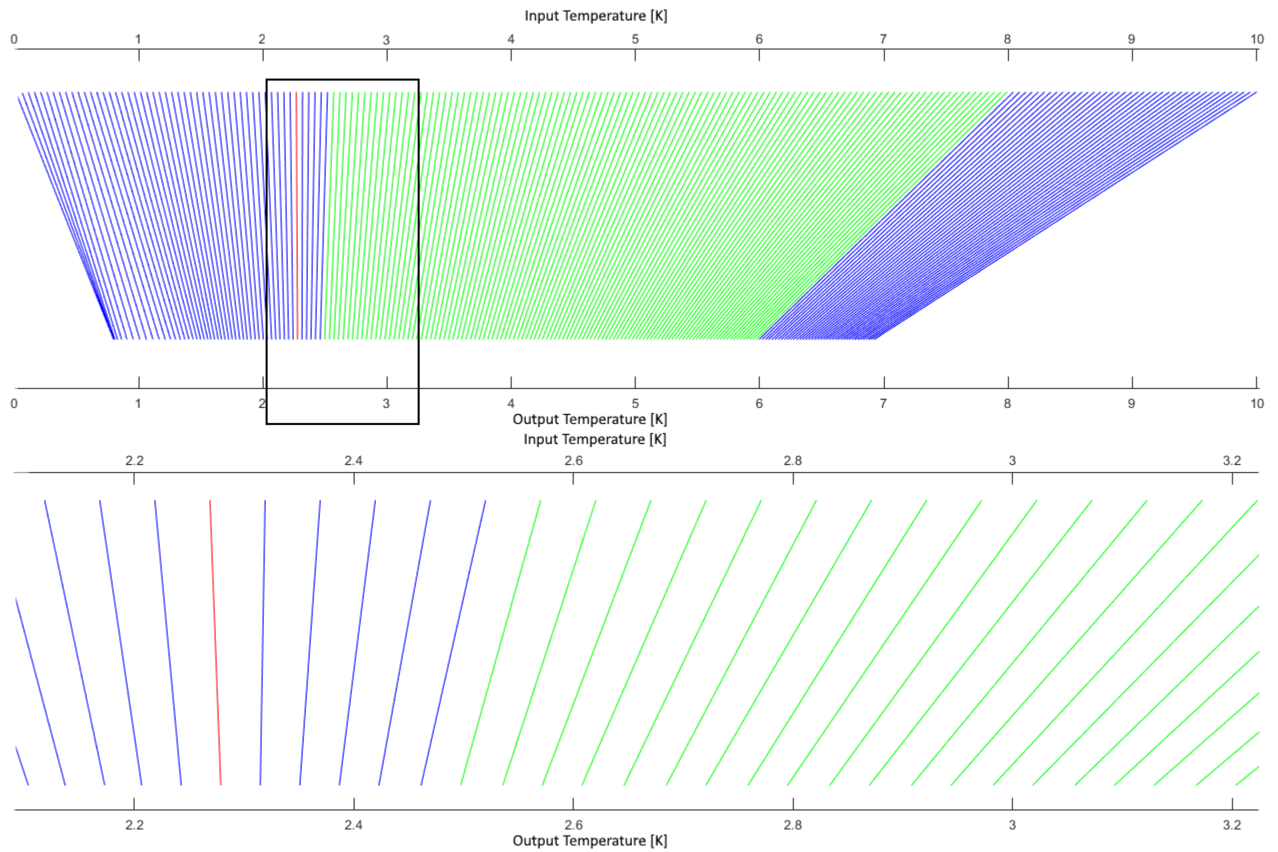

Figure 13: Extrapolation with the network of test 1. The lines are visualizing the effect the network has on the input thermal states with the temperature of the x-coordinate of the upper end of each line. It effects them in the way, that each output thermal state has the temperature of the x-coordinate of the lower end of each line. The blue connections are attempts to cool states the network was not trained to cool, the green connections are attempts to cool states the network was trained on and the red connection is the highest temperature state, that was not cooled by the network. Every state above that temperature gets cooled by the network. First plot: an overview of the temperature mapping of the network. Second plot: the marked area in detail.

As you can see, the network cools the input states if the input temperature is above 2.3K as expected. It also cools the input states with a temperature above the training temperature interval.
All in all one can abstract, that the network is able to operate on temperature regions below and above the region in was trained on. Indeed the outputs in the temperature region below the training temperature stray further away from being a thermal state the more one moves away from the training temperature and it only cools to a certain lower limit of temperature, but this may depend on the network width or length and

with more computational resources one could maybe produce a network that could still be used in a temperature region further away from its training temperature region.

7.3 Cooling Factor and Layer Structure

What should also be examined is how this kind of learning algorithm and network architecture performs if it is asked to cool with a very lower cooling factor like 0.2 or 0.5. Furthermore it is also interesting to see how other layer structures would perform. Therefore, the parameters of testing for the next test are:

	Test			
Parameter	2	3	4	5
T_{min}	4	4	4	4
T_{max}	8	8	8	8
$\|T_{mini-batch}\|$	rnd ∈ [0,1]	rnd ∈ [0,1]	rnd ∈ [0,1]	rnd ∈ [0,1]
$N_{particles}$	2	2	2	2
N_{data}	6	6	6	6
$N_{training}$	1000	1000	1000	1000
$h_{training}$	1	1	1	1
$j_{training}$	1	1	1	1
$\delta T_{training}$	0.4	1.2	0.8	0.8
λ_{min}	0.00025	0.00025	0.00025	0.00025
λ_{max}	1	1	1	1
p_λ	3	3	3	3
$iterations$	50	50	50	50
M	[2 5 2]	[2 5 2]	[2 3 3 2]	[2 2 2 2 2 2 2]

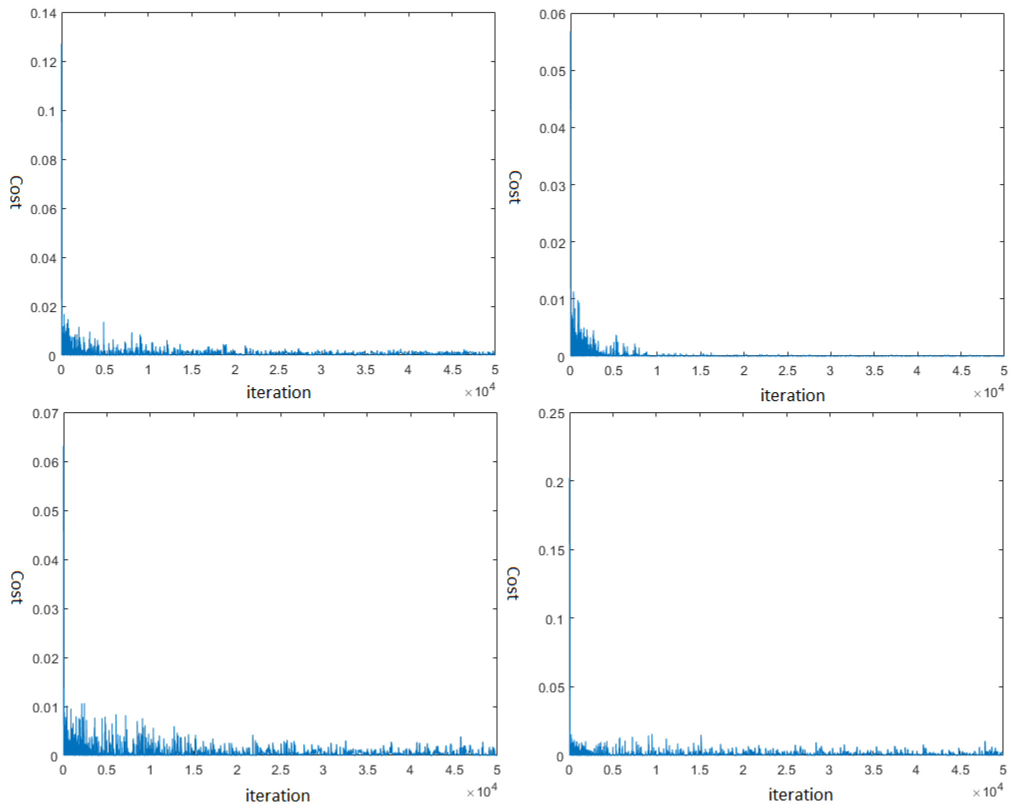

Figure 14: Cost functions for test 2-5. Top left: test 2 with a cooling factor of 0.4. Top right: test 3 with a cooling factor of 1.2. Bottom left: test 4 with $M = [2332]$. Bottom Right: test 5 with $M = [2\,2\,2\,2\,2\,2\,2]$.

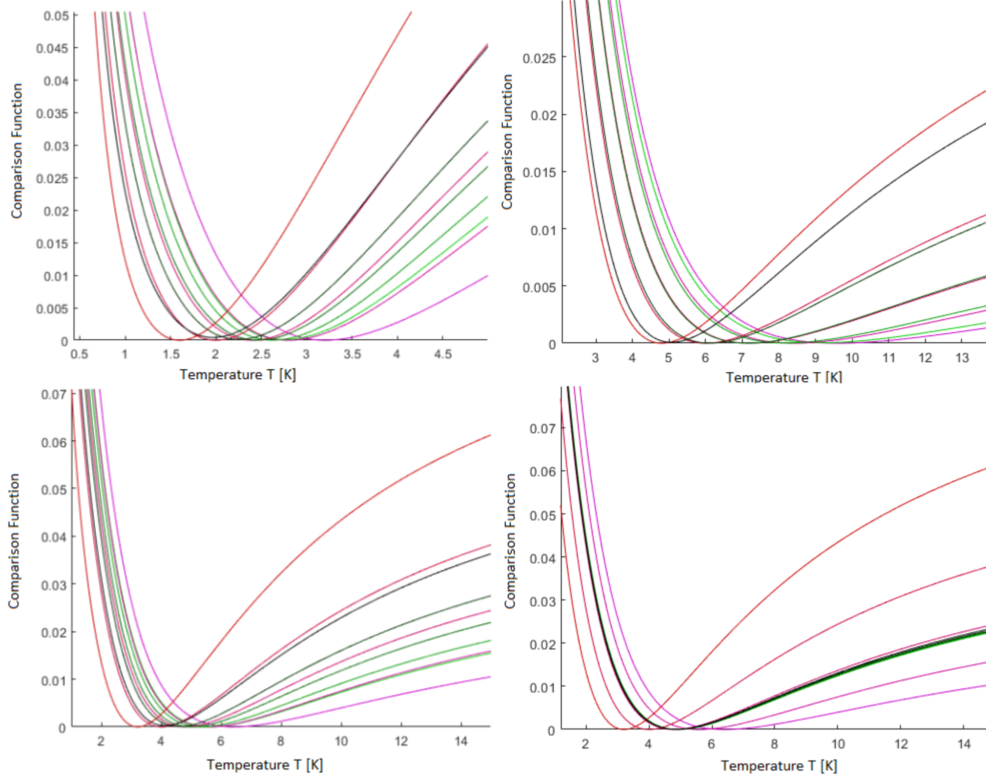

Figure 15: Comparison functions of the predicted states and the label states of test 2-5, with the Hilbert-Schmidt norm as underlying norm. Top left: test 2 with a cooling factor of 0.4. Top right: test 3 with a cooling factor of 1.2. Bottom left: test 4 with $M = [2\,3\,3\,2]$. Bottom Right: test 5 with $M = [2\,2\,2\,2\,2\,2\,2]$.

If one compares the progress of the cost functions during training, one can see which problems are more complicated and which are less complicated to learn for the network. In figure 14 one can see on the top left, that cooling a state with such a low cooling factor, as in test 2, is harder to learn for the network because the cost function decays slower and a certain error remains even when the network finished its training. The same effect is visible in the comparison functions of this test. The predicted states are not as good as in test 1. In the cost and comparison functions of test 6, one can see that heating (cooling with a cooling factor $\delta T_{training} \geq 1$) is easier for the network then cooling down. The cost function decays faster and the comparison functions of the labels are nearly the same as the comparison functions of the predicted states. Using a layer structure with more but smaller layers is decreasing the ability to learn this problem as one can see in the cost and comparison functions of test 4 and test 5. The costs are decaying very slow and the comparison functions of the predicted states are far away from matching the comparison functions of the labels.

7.4 Ratios of the Coupling Constant and the External Field

Moreover, to see if this network architecture and learning algorithm could also be applied to cool down thermal states with input structures with different coupling constants or external fields, I will carry out another test series with the following parameters of training.

	Test			
Parameter	6	7	8	9
T_{min}	4	4	4	4
T_{max}	8	8	8	8
$\|T_{mini-batch}\|$	rnd $\in$ [0,1]	rnd $\in$ [0,1]	rnd $\in$ [0,1]	rnd $\in$ [0,1]
$N_{particles}$	2	2	2	2
N_{data}	6	6	6	6
$N_{training}$	1000	1000	1000	1000
$h_{training}$	1	2	10	1
$j_{training}$	2	1	1	10
$\delta T_{training}$	0.8	0.8	0.8	0.8
λ_{min}	0.00025	0.00025	0.00025	0.00025
λ_{max}	1	1	1	1
p_{λ}	3	3	3	3
$iterations$	50	50	50	50
M	[2 5 2]	[2 5 2]	[2 5 2]	[2 5 2]

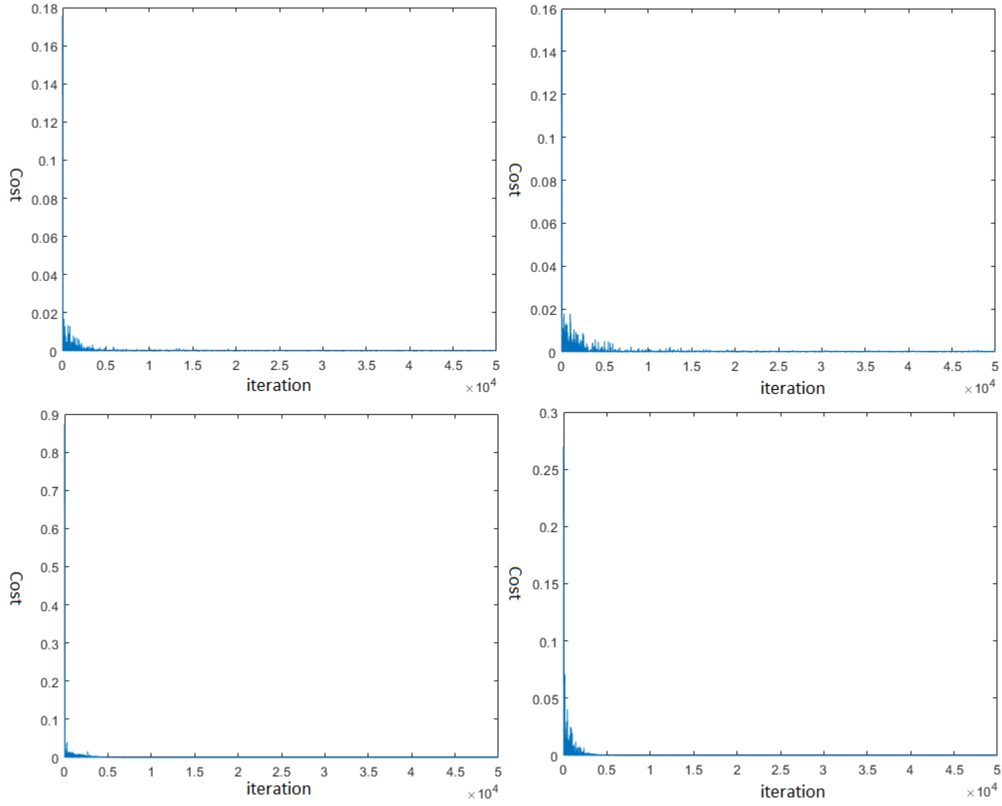

Figure 16: Cost functions for test 6-9. Top left: test 6 with $h_{training} = 1$ and $j_{training} = 2$. Top right: test 7 with $h_{training} = 2$ and $j_{training} = 1$. Bottom left: test 8 with $h_{training} = 10$ and $j_{training} = 1$. Bottom Right: test 9 with $h_{training} = 1$ and $j_{training} = 10$.

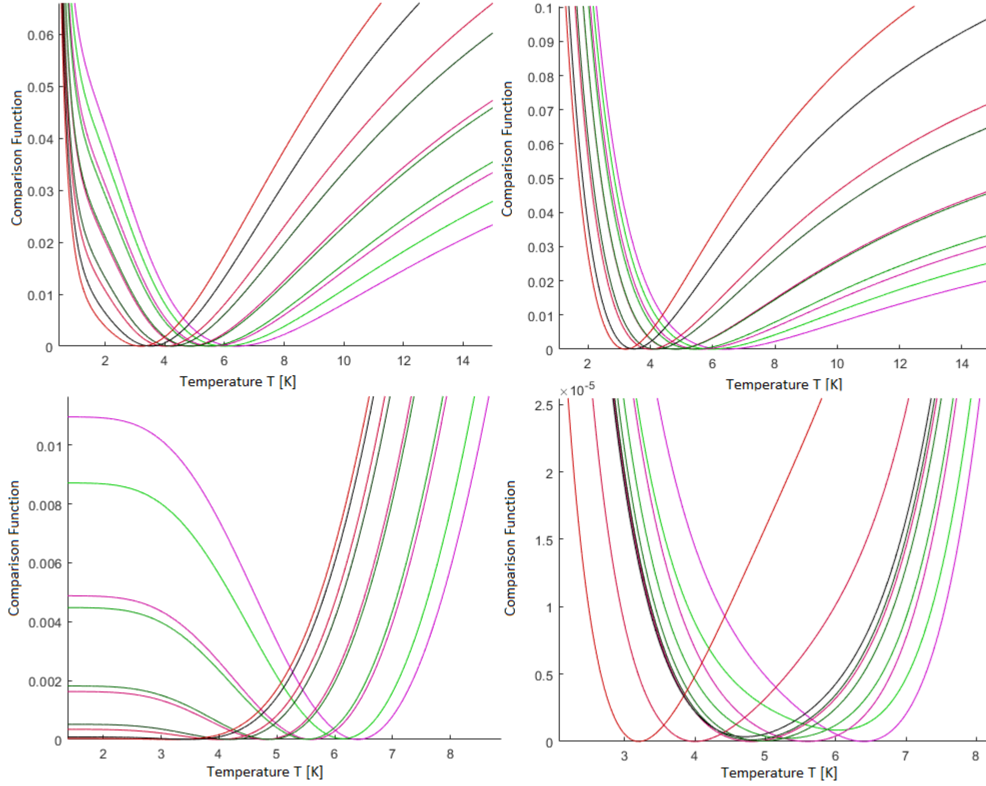

Figure 17: Comparison functions of the predicted states and the label states of test 6-9, with the Hilbert-Schmidt norm as underlying norm. Top left: test 6 with $h_{training} = 1$ and $j_{training} = 2$. Top right: test 7 with $h_{training} = 2$ and $j_{training} = 1$. Bottom left: test 8 with $h_{training} = 10$ and $j_{training} = 1$. Bottom Right: test 9 with $h_{training} = 1$ and $j_{training} = 10$.

In view of the comparison and cost functions of this series of tests one can assume, that it is more difficult for the network to learn this problem the greater the coupling constant $j_{training}$ is in proportion to the external field. The comparison functions of the labels and predictions are differing increasingly with a greater coupling constant (figure 17 on the top left and on the bottom right). But it is also noticeable that the comparison functions of the predictions and labels resembling each other even more, when the coupling constant is small in proportion to the external field. This could lead to the assumption that it is easier for the network to learn the structure of the problem when the external field is greater.

7.5 Temperature Regions

To evaluate the worth of this network architecture and learning algorithm, it is also necessary to examine its performance in completely different temperature regions. Therefore, I will carry another test series with the parameters of training which you can see in the next table.

	Test		
Parameter	10	11	12
T_{min}	1	18	50
T_{max}	3	22	54
$\|T_{mini-batch}\|$	rnd $\in$ [0,1]	rnd $\in$ [0,1]	rnd $\in$ [0,1]
$N_{particles}$	2	2	2
N_{data}	6	6	6
$N_{training}$	1000	1000	1000
$h_{training}$	1	1	1
$j_{training}$	1	1	1
$\delta T_{training}$	0.8	0.8	0.8
λ_{min}	0.00025	0.00025	0.00025
λ_{max}	1	1	1
p_{λ}	3	3	3
$iterations$	50	50	50
M	[2 5 2]	[2 5 2]	[2 5 2]

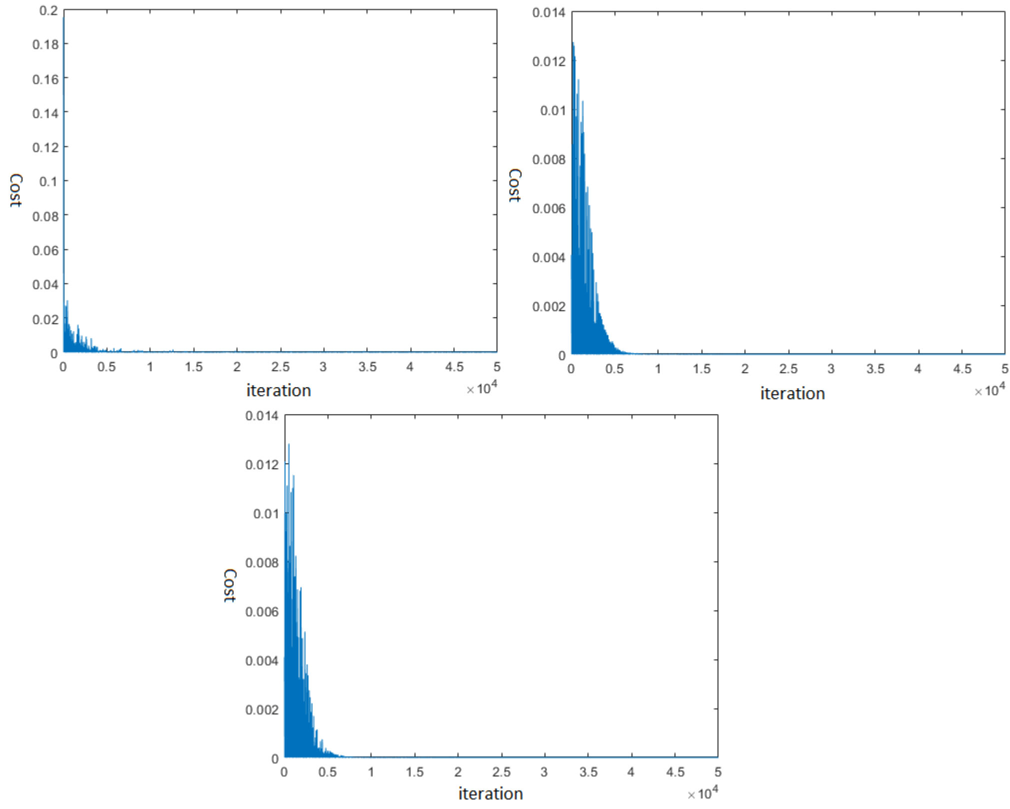

Figure 18: Cost functions for test 10-12. Top left: test 10 with $[T_{min}, T_{max}] = [1, 3]$. Top right: test 11 with $[T_{min}, T_{max}] = [18, 22]$. Bottom left: test 12 with $[T_{min}, T_{max}] = [50, 54]$.

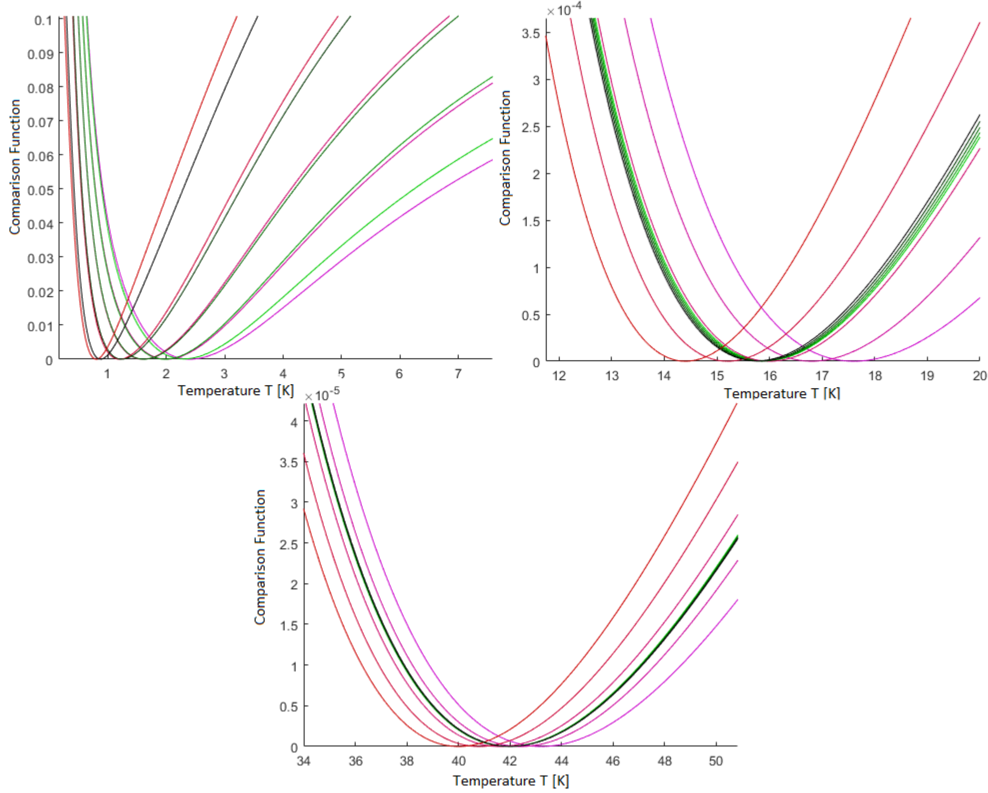

Figure 19: Comparison functions of the predicted states and the label states of test 10-13, with the Hilbert-Schmidt norm as underlying norm. Top left: test 10 with $[T_{min}, T_{max}] = [1, 3]$. Top right: test 11 with $[T_{min}, T_{max}] = [18, 22]$. Bottom left: test 12 with $[T_{min}, T_{max}] = [50, 54]$.

In view of figure 19 one can see that the learnability of this problem for the network probably decreases with temperature. The comparison functions for the labels and output states of test 10 are more similar than those out test 1 and those of test 12 are more similar than those of test 11. So, the outputs differing from their labels more and more with temperature.

8 Conclusion and Outlook

In the results and limits section, it becomes clear that this technique to cool down thermal states is working under certain circumstances. One has to keep in mind, that the quality of results decreases with a higher temperature or a greater coupling constant but also increases with less temperature and a greater external field. But even if it is not very precise in cooling thermal states to a given temperature yet, it was proven that it can cool states and that it gets more precise with bigger network layers. So, with greater computing power this technique might even work under worse circumstances like higher temperature or greater coupling constants with a good enough precision.

Furthermore, one could just obtain the error of a given network in dependence of the input temperature and take it into account since it is a reproducible kind of error. Also necessary to say is that if a certain precision is reached while training, the network also is able to extrapolate. Meaning that it can operate on temperature regions it was not trained on. Whereby the network would be able to produce colder states, that it has "seen" while training, which provides a great spectrum of other possibilities. Also, in consideration of the fact, that the quality of the results also grows with the layer size, one could assume that the extrapolation capacities are also growing with the layer size and it would then be interesting to see what the limits of the extrapolation capacities are. Also, it would be interesting to see how this architecture and training algorithm would perform on tasks like manipulating the coupling constant, the external field or other parameters of related systems. If that would be comparable successful, one could maybe provide a set of networks for quantum simulators with which the parameters of the simulated system are nearly freely selectable. Provided that a great enough computation power on quantum devices is given.

List of Figures

1 Functionality of a neural network . . . 3
2 Functionality of a gradient descent algorithm . . . 5
3 Feedforward functionality in QNNs 1 . . . 8
4 Feedforward functionality in QNNs 2 . . . 9
5 Cost functions for attempt 1-4 . . . 16
6 Comparison functions for attempt 1-4 . . . 17
7 Cost functions for attempt 5-8 . . . 20
8 Comparison functions for attempt 5-8 . . . 21
9 Cost functions for attempt 9-12 . . . 24
10 Comparison functions for attempt 9-12 . . . 25
11 Cost and comparison functions for test 1, trained with very many batches 26
12 Extrapolation with the network of test 1 . . . 28
13 Extrapolation with the network of test 1 . . . 29
14 Cost functions for test 2-5 . . . 31
15 Comparison functions for test 2-5 . . . 32
16 Cost functions for test 6-9 . . . 34
17 Comparison functions for test 6-9 . . . 35
18 Cost functions for test 10-12 . . . 37
19 Comparison functions for test 10-13 . . . 38

9 References

[1] SIMAR (2020). Gradient Descent Visualization (https://www.mathworks.com/matlabcentral/fileexchange/35389-gradient-descent-visualization), MATLAB Central File Exchange. Retrieved May 14, 2020.

[2] Michal M. Wolf (2012). Quantum Channels & Operations (https://www-m5.ma.tum.de/foswiki/pub/M5/Allgemeines/MichaelWolf/QChannelLecture.pdf).

[3] Daniel Scheiermann (2020). Noise robustness of quantum neural networks

[4] Kerstin Beer, Dmytro Bondarenko, Terry Farelly, Tobias J. Osborne, Rober Salzmann, Ramona Wolf (2019). Efficient Learning for Deep Quantum Neural Networks, https://arxiv.org/pdf/1902.10445.pdf

[5] F. Barrat, James Dborin, Matthias Bal, Vid Stojevic, Frank Pollmann and A. G. Green (2020) Parallel Quantum Simulation of Large Systems on Small NISQ Computers, https://arxiv.org/abs/2003.12087, appendix B1

[6] L. D. Landau and E. M. Lifshitz, Statistical Physics (third edition 1980), http://people.physics.tamu.edu/kcolletti1/classes/fall15/stat_mech/84180007-Vol-5-Landau-Lifshitz-Statistical-Physics-Part-1.pdf

[7] Ian Goodfellow and Yoshua Bengio and Aaron Courville (2016, MIT Press). Deep Learning, http://www.deeplearningbook.org

[8] Wikipedia (2020). Forschungsgeschichte - Quantencomputer, (https://de.wikipedia.org/wiki/Quantencomputer#Forschungsgeschichte)

[9] M. Van den Nest, W. Dür, H. J. Briegel (2007), Completeness of the classical 2D Ising model and universal quantum computation, https://arxiv.org/abs/0708.2275

[10] Sebastian Ruder (2017), An overview of gradient descent optimization algorithms, https://arxiv.org/pdf/1609.04747.pdf

[11] Ryan Tibshirani (2013), Convex Optimization Lecture, https://www.stat.cmu.edu/~ryantibs/convexopt-F13/scribes/lec6.pdf

[12] Gemma De las Cuevas and Toby S. Cubitt (2016), Simple universal models capture all classical spin physics, https://arxiv.org/pdf/1406.5955

[13] Jesse J. Berwald (2018), The Mathematics of Quantum-Enabled Applications on the D-Wave Quantum Computer, https://arxiv.org/pdf/1812.00062

[14] Wikipedia (2020). Quantum simulator, https://en.wikipedia.org/wiki/Quantum_simulator