Neural Networks and Matrix Product States

David Maier

BACHELOR THESIS

Faculty of Physics
at Ludwig-Maximilians-Universität
Munich

submitted by

David Maier

Munich, 26.07.2017

Supervisor: Prof. Dr. Jan von Delft

Neuronale Netze und Matrix Produkt Zustände

David Maier

BACHELORARBEIT

Fakultät für Physik
an der Ludwig-Maximilians-Universität
München

vorgelegt von

David Maier

München, 26.07.2017

Betreuer: Prof. Dr. Jan von Delft

Contents

1 Introduction

Recent developments such as the ascent of self-driving cars, the introduction of face recognition into our daily lives, and the omnipresence of machine learning algorithms in todays Internet show the enormous potential of neural networks and machine learning techniques. They are also used in a wide range of applications in chemistry, material science and condensed matter physics [1, 2, 3, 4, 5, 6].
Despite being highly successful, the formal understanding of these algorithms is only gradually unfolding and for a surprisingly big part still remains illusive. Most of the methods used in practice to optimize neural networks are largely based on heuristics and lack deeper theoretical understanding and foundation [7, 8].
Meanwhile they exhibit great structural similarities to one of the most successful and important tools in theoretical condensed matter physics, the *renormalization group* and the later developed *tensor networks*. These techniques have been applied to a great variety of physics problems and stand on a very solid theoretical basis.
Both fields can benefit from potential conceptual and technical overlaps and recently a lot of work has been published trying to shed some light on the topic, for example by using the mathematical and physical understanding of tensor networks to optimize neural networks [9] or using neural networks to find the ground state of quantum wavefunctions [10].
In this bachelor thesis we follow the work of Stoudenmire and Schwab [11] who use a tensor network ansatz based on *matrix product states* (MPS), a very popular computational tool in quantum many-body physics, in the context of machine learning. Specifically we focus on the application of MPS to the recognition of handwritten digits from the MNIST dataset and explore the details of the algorithm of Ref. [11].
This thesis is structured as follows. First we will give an introduction into the most basic concepts of machine learning to enable the reader to understand the current research topics and have a basic understanding of the field (Ch. 2). Then some foundations of matrix product states and tensor networks will be covered (Ch. 3) to enable the reader to follow chapter 4 of this work, where the method presented in [11] will be explicitly implemented and explained. The work concludes in a discussion of the presented method and introduces some possibilities for further research into the topic (Ch. 5).

2 Machine learning

In recent years machine learning has been getting an enormous amount of attention, in the media, in science, and in society. In machine learning, a system of connected units, a so called *neural network*, is trained using a specific training algorithm to solve a specific task without being explicitly programmed.
This subfield of computer science is extremely successful in solving complicated classification tasks which are not directly accessible through explicit coding. Therefore, it is used today in a wide range of applications reaching from image classification, recommender systems and language processing to applications in chemistry, material science and condensed matter physics [1, 2, 3, 4, 5, 6]. In the following we will introduce the most basic elements of machine learning before giving a short overview of applications in connection to condensed matter physics. We will start by discussing artificial neurons, the most basic building blocks of neural networks, to gain an understanding on how machine learning uses non-linear elements for their success. Then we will present some basics about the architecture of neural networks and give a detailed explanation of how learning is achieved in order to enable the reader to understand the analogies of our MPS approach to the techniques used in machine learning. In the final part of this chapter we introduce additional concepts of machine learning that are subject of current research and give a short résumé about the applications in connection to condensed mater physics.

2.1 Artificial neurons

The fundamental building blocks of a neural network are artificial neurons, which are named after the neurons in our brain. The neural model applied in machine learning (Fig. 1) contains, just like our brain, connecting links or synapses with respective weights or strengths. In mathematical terms this yields an input signal x_i connected through a synapse to neuron k and multiplied by the weight w_{ki}. After summing the weighted input signals, an activation function is applied to limit the output of the neuron to a signal of finite value. The model also includes a bias b_k which in- or decreases the net input of the activation function. A neuron k is therefore mathematically described by

$$y_k = \varphi(\sum_{i=1}^{m} w_{ki} x_i + b_k), \tag{1}$$

with $x_1, x_2, ..., x_m$ being the input signals, $w_{k1}, w_{k2}, ..., w_{km}$ the respective weights for neuron k, b_k the bias, $\varphi(\cdot)$ the *activation function* and y_k the output signal of the neuron [12].

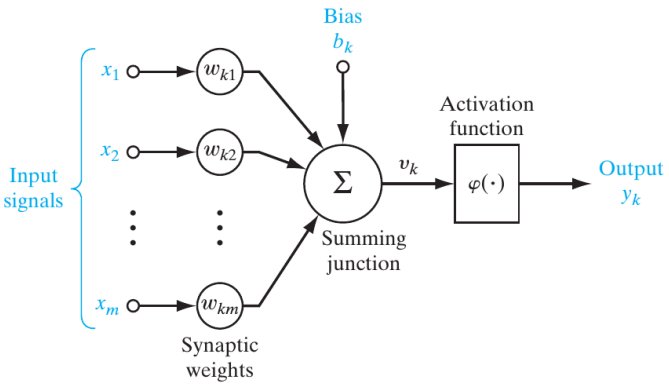


Figure 1: Nonlinear model of a neuron labeled k. [12]

2.1.1 The perceptron

The most basic type of artificial neuron is the so-called *perceptron* developed by Rosenblatt in the 1950s and 60s [13]. While in todays applications other models of artificial neurons, known as *sigmoid neurons*, are used, it is instructive to start with perceptrons in order to understand the rationale behind the definition of the sigmoid neurons.
From several binary inputs $\boldsymbol{x} = x_1, x_2, ...$ a perceptron produces a single binary output. Real numbered weights $\boldsymbol{w} = w_1, w_2, ..$ are introduced as an expression of the importance of the respective inputs to the output. The perceptron's output $f(x)$ is defined as:

$$f(x) = \begin{cases} 1, & \text{if } \boldsymbol{w} \cdot \boldsymbol{x} + b > 0 \\ 0, & \text{otherwise} \end{cases}, \tag{2}$$

with $\boldsymbol{w} \cdot \boldsymbol{x} = \sum_{i=1}^{n} w_i x_i$, n being the number of inputs to the perceptron and b being the bias. The bias shifts the decision boundary of the perceptron and does not depend on any input value. Following the analogy of neural networks to the human brain, the bias is a measure of how easy it is to get the perceptron to fire. McCulloch and Pitts showed in 1943 [14] that every simple logical operator, acting on one or more binary inputs to produce a single binary ouput, e.g. NOT, AND, OR, XOR, NAND, NOR or XNOR, can be approximated with a combination of perceptrons. The perceptron is a linear classifier, a classification algorithm making decisions based on a linear predictor function, which uses the Heaviside step function as the activation function [15].

2.1.2 The sigmoid neuron

For the purpose of making learning possible, a small change in a weight or bias should cause only a small corresponding change in the output. In this way we can gradually make small changes to the weights and biases to gradually improve the behavior of our net. Obviously, a network consisting of perceptrons is not very practical for that purpose since a tiny change in w or b can yield a different output of a perceptron, changing the behavior of the rest of the network completely. Thus another type of artificial neuron called the *sigmoid neuron* is introduced [15] .
The defining feature of the sigmoid neuron is that, instead of using the Heaviside step function as an activation function, it uses the *sigmoid function* $\sigma(w \cdot x + b)$ which is defined as

$$\sigma(z) \equiv \frac{1}{1 + e^{-z}}. \tag{3}$$

As can be seen in Fig. 2, the sigmoid neuron basically represents a smoothed version of a perceptron. The strictly increasing σ exhibits a solid balance between linear and non-linear behavior and the smoothness ensures that small changes δw_i and δb in the weights and bias will lead to a small change in the output.

2.2 Architecture of neural networks

To describe the layout of a neural network, a simplified *architectural graph* is used omitting explicit mentioning of biases and activation functions. Each neuron is then represented by a node, as shown in Figure 3, and the different neurons are connected by synapses.
In this text, we will concentrate on *layered* neural networks where the neurons are structured in layers named after their constituents. The input layer contains all input neurons, the output layer all output neurons and the so-called *hidden neurons* constitute the layers in between. The term "hidden" refers to the fact that this part of the network can neither been seen directly from the input nor output of the network.
The hidden neurons act as feature detectors by performing a nonlinear transformation on the input data into the so-called *feature space*. Through this transformation, classes of interest, that are hardly separable in the original input space, may be more easily separable in feature space. This step is crucial for the extraction of higher-order statistics from the input.

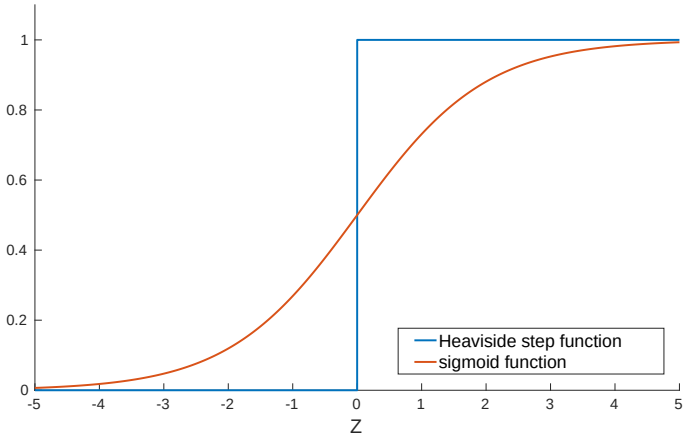


Figure 2: Comparison of the Heaviside step and the sigmoid function, the respective activation functions of the perceptron and sigmoid neuron.

The neural network shown in Fig. 3 is not *fully connected* as not every node in each layer is connected to every node in the next forward layer and is therefore called *partially connected* [12].

The design of in- and output layer is often straightforward and dictated by the task at hand. In the case of identifying handwritten digits (for a detailed description of the MNIST dataset of handwritten digits see Ch. 4.2) each pixel of the input picture will be an input neuron with the grayscale intensities scaled between 0 and 1, while the ten possible different outputs $0, 1, 2, .., 9$ make up the ten output neurons.
This choice seems rather natural at first, but from a programming perspective it would seem much more efficient to use just four output neurons taking on binary values resulting in $2^4 = 16 > 10$ possibilities. The justification for the choice of output neurons here is empirical and using an architecture with ten neurons instead of four just learns to recognize digits better [15]. This is a great example of how much of the optimization of neural networks just depends on heuristics.
The design of the hidden layers is usually more difficult. Neural network researchers have developed many design heuristic for hidden layers, e.g. by determining trade-offs between the number of hidden layers and the required training time [15].
If the signal is not passed in a circle but instead the output of one layer is used as the input for the next layer, the underlying neural network is called a *feedforward* neural network. Models of networks allowing feedback loops are known as *recurrent* neural networks. The loops in the network create an internal state of the network which allows for dynamic temporal behavior. While recurrent neural networks are less popular than feedforward networks, partly because their learning algorithms are less powerful, they are much closer to how the human brain works [15].

2.3 Learning

In a neural network context, there are generally three different types of learning: *unsupervised learning*, *reinforcement learning* and *supervised learning*, which will be the focus of this section and our approach in Ch. 4 also falls under this category.
Training a neural network means gradually adjusting the weights and biases of the network so that the output eventually approximates the desired output $y(x)$ for all training inputs x. Generally speaking, given a specific task and a class of functions F, learning means using a set of observations to find $f^* \in F$ which solves the task in some optimal sense.
Supervised learning requires a *teacher*, whom we may think of as having knowledge about the environment of interest in the form of *input-output examples*. The neural network however does not know about the environment. Therefore a *cost function* $C : F \mapsto \mathbb{R}$ is defined so that for the

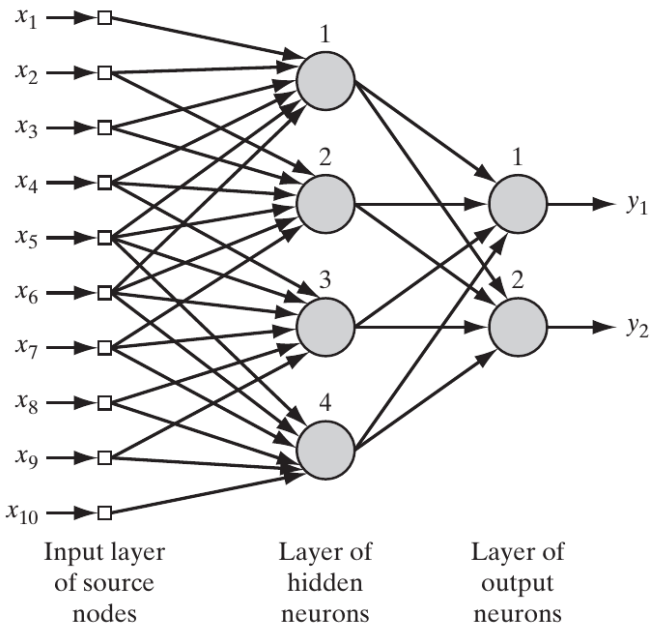


Figure 3: Architectural graph of a *layered feedforward partially-connected* neural network consisting of an input, a hidden and an output layer.

optimal solution f^*

$$C(f^*) \leq C(f) \qquad \forall f \in F. \tag{4}$$

Take for example

$$C(w,b) \equiv \frac{1}{2n} \sum_x ||y(x) - a(x,w,b)||^2, \tag{5}$$

with weights w, biases b, total number of training inputs n, and a the vector of outputs from the network for the input x. C is called the *quadratic cost* function or *mean squared error* (MSE) [15]. The cost function is a measure of how far away from an optimal solution a particular solution is. A learning algorithm then searches through the solution space in the form of a multidimensional error surface to find a function that minimizes C. The network parameters are then adjusted iteratively in a step-by-step fashion with the aim of the network eventually emulating the supervisor. In this way, knowledge about the environment is passed to the neural network through supervised training examples, which is stored in the form of synaptic weights representing the long-term memory. The network can then be separated from the teacher and deal with the environment independently [12].
Unsupervised and reinforcement learning are categorized as learning processes without a teacher. This implies that there are no labeled training examples. In reinforcement learning the network is continuously in contact with the environment. One form of a reinforcement-learning scheme is built around a so-called *critic*, which is defined as converting a primary reinforcement signal from the environment into a *heuristic reinforcement signal*. The learning then occurs through *delayed reinforcement* as the network observes the temporal sequence of reinforcement signals. This can be interpreted as a *cost-to-go function*, the expectation of the cumulative cost of actions taken over a number of steps, being minimized [12].
Unsupervised learning works completely without external teacher or critic. The parameters of the network are adjusted through a *task-independent measure* and some sort of a competitive-learning rule, where neurons in a competitive layer compete for the chance to respond to features in the input data. The simplest form being a "winner takes it all" model where only the neuron with the greatest total input turns on, while the others switch off [12].

2.3.1 Gradient descent

Since we apply a modified gradient scheme in the context of our implementation in Chapter 4 to train our MPS, we introduce the basic concept of *gradient descent*. It is instructive to go into detail here to understand where the ideas for the algorithm in Ch. 4.1.3 come from and how they are justified.
In a supervised learning context, a system is able to reach a global (or local) extremum through the gradient of the error surface. The gradient is the vector that points in the direction of steepest slope [12]. In order to find a local minimum of the surface one takes steps proportional to the *negative* of the gradient therefore always moving in the direction of steepest descent. For C being a function of n variables, $v_1, v_2, ..., v_n$ with $\Delta v = (\Delta v_1, \Delta v_2, ..., \Delta v_n)^T$ we get

$$\Delta v = -\eta \nabla C, \tag{6}$$

where η is a small, positive parameter called the *step size* or *learning rate* and ∇C the gradient vector $\nabla C \equiv (\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}, ..., \frac{\partial C}{\partial v_n})^T$.
One typically uses the approximation,

$$\Delta C \approx \nabla C \cdot \Delta v = -\eta ||\nabla C||^2, \tag{7}$$

which guarantees $\Delta C \leq 0$. In this way, the function C is always decreased in every iteration. This results in a simple update rule for v,

$$v \to v' = v - \eta \nabla C. \tag{8}$$

In order for this method to work, one must choose the learning rate η sufficiently small for (7) to be a good approximation. Otherwise ΔC could become positive. If η exceeds a certain critical value the method becomes unstable and diverges. At the same time the learning rate should not be chosen too small, since this would lead to a small step size (6) and therefore the time needed for the gradient descent algorithm to reach a minimum would become very large. In the context of neural networks, the gradient descent update rule (8) takes the following form

$$w_k \to w_k' = w_k - \eta \frac{\partial C}{\partial w_k} \tag{9}$$

$$b_l \to b_l' = b_l - \eta \frac{\partial C}{\partial b_l}. \tag{10}$$

As can be seen from Eq. (5), the quadratic cost function $\nabla C = \frac{1}{n} \sum_x \nabla C_x$ strongly depends on the number of training inputs n, as the gradient for each training input has to be calculated separately before averaging over all of them. Therefore learning slows down significantly for large n.
To speed up learning in these regimes, *stochastic gradient descent* can be used. The idea of this methods is to estimate the gradient ∇C by computing ∇C_x for a small so-called *mini-batch* of m randomly chosen training inputs $X_1, X_2, ..., X_m$. Averaging over this sample results in a good approximation of the true gradient within a small amount of time, provided m is large enough and

$$\nabla C \approx \frac{1}{m} \sum_{j=1}^{m} \nabla C_{X_j}. \tag{11}$$

The update rule for the weights and biases then becomes

$$w_k \to w_k' = w_k - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial w_k} \tag{12}$$

$$b_l \to b_l' = b_l - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial b_l}, \tag{13}$$

where the sums include all X_j in the current mini-batch. For each training step a new mini-batch is randomly selected until all training inputs are exhausted, concluding a so called *epoch* of training. Training then continues with a new training epoch [15]. While batch learning allows for a parallelization of the learning process, it also comes with a high demand in storage requirements. The extremal case of $m = 1$ is known as *on-line* or *incremental* learning, where the network learns from just one learning example at a time, and avoids this disadvantage. The stochastic nature of the procedure reduces the likelihood of the learning process getting stuck in a local minimum [12]. Stochastic gradient descent can dramatically speed up learning in neural networks and is therefore commonly used today. A very detailed explanation of the algorithm can be found in [12].

2.3.2 Backpropagation

In order to apply the gradient descent algorithm, one has to compute the gradient of the cost function, ∇C. This is typically performed employing a process known as *backpropagation*. In this procedure the error is calculated at the output of the neural net and then propagated backwards through the layers to compute the gradients for the individual weights and biases in a simple and effective way. Even though this algorithm is not implemented in our MPS approach, it is integral to many approaches in the field of machine learning. For completeness it is also instructive to go into a little bit of detail here.
For the algorithm to work, two assumptions about the cost function C are necessary:

1. The cost function can be written as an average $C = \frac{1}{n}\sum_x C_x$ over cost functions C_x for individual training examples x.

2. The cost can be written as a function of the outputs from the neural network.

Assumption 1 ensures that partial derivatives $\frac{\partial C_x}{\partial w}$ and $\frac{\partial C_x}{\partial b}$ can be computed for individual training examples. This allows for averaging over all training examples to compute $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$.
Assumption 2 fixes both the training input x and the corresponding desired output such that the only parameter, that can be influenced by modifying the weights and biases, is the networks actual output [15].
As described earlier, the hidden neurons are not directly accessible but still contribute to the overall error. To enable learning, it is crucial to determine how each internal decision of a hidden neuron contributed to the overall result and how to correct the corresponding weights and biases accordingly. This problem is known as the *credit-assignment problem* and backpropagation offers an elegant way to resolving it in a two-phase process.
In the first phase, the *forward phase*, the input signal is propagated, layer-by-layer, through the network until it reaches the output, while the weights and biases of the network are fixed.
In the second, the *backward phase* an error signal is calculated at the end of the network by comparing the actual output of the network with the desired, correct output. This error is then propagated *backwards* through the network, hence the name *backpropagation*. During that process the weights of the network are successively updated [12].
Next, we will derive the fundamental equations of the backpropagation algorithm following [15]. A more detailed derivation can be found in [12], chapter 4.
We begin by defining the *local error* δ_j^l of the j^{th} neuron in the l^{th} layer as

$$\delta_j^l \equiv \frac{\partial C}{\partial z_j^l}, \tag{14}$$

where z_j^l is the weighted input.
In combination with the output activation a_j^L the *error in the output layer* can simply be computed by applying the chain rule,

$$\delta_j^L = \sum_k \frac{\partial C}{\partial a_k^L}\frac{\partial a_k^L}{\partial z_j^L}. \tag{15}$$

Since the output a_k^L of neuron k only depends on the weighted input z_j^l for neuron j when $k = j$ this further simplifies to

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L), \tag{16}$$

where the second part follows from $a_j^L = \sigma(z_j^L)$. Note that the exact form of $\frac{\partial C}{\partial a_j^L}$ depends on the cost function, yet, it is still easily computable. For the quadratic cost function $C = \frac{1}{2}\sum_j (y_j - a_j^L)^2$ δ^L is given by

$$\delta_j^L = (a_j^L - y_j)\sigma'(z_j^L) \tag{17}$$

In a next step, the error δ^l will be expressed in terms of the error in the next layer δ^{l+1}, which will be crucial to propagate the error from the output through the network,

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} = \sum_k \frac{\partial C}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l}\delta_k^{l+1}, \tag{18}$$

where the chain rule is used to rewrite δ_j^l in terms of $\delta_k^{l+1} = \partial C/\partial z_k^{l+1}$. For further simplification note that

$$z_k^{l+1} = \sum_j w_{kj}^{l+1} a_j^l + b_k^{l+1} = \sum_j w_{kj}^{l+1}\sigma(z_j^l) + b_k^{l+1}, \tag{19}$$

and therefore

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1}\sigma'(z_j^l). \tag{20}$$

Inserting (20) into (18):

$$\delta_j^l = \sum_k w_{kj}^{l+1}\delta_k^{l+1}\sigma'(z_j^l) = ((w^{l+1})^T\delta^{l+1})_j\sigma'(z_j^l) \tag{21}$$

with $(w^{l+1})^T$ the transpose of the weight matrix w^{l+1} for the $(l+1)^{th}$ layer. A compactified matrix notation is used in the last step. This form offers a very intuitive perspective on the algorithm. Suppose the error δ^{l+1} at the $l+1^{th}$ layer is known. To calculate the error of the next layer l, the transpose weight matrix is applied moving the error *backward* through the layers. Componentwise multiplication with $\sigma'(z^l)$ then propagates the error backwards through the activation function in layer l yielding δ^l, the error in the weighted input to layer l.
Simply applying the chain rule analogous to the derivations above, equations for the rate of change of the cost with respect to any bias and with respect to any weight in the neural network are derived as

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{22}$$

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l. \tag{23}$$

Revisiting the sigmoid function (Fig. 2), it is clear that $\sigma'(z_j^L) \to 0$ when $\sigma(z_j^L)$ goes towards 0 or 1. This indicates that learning occurs very slowly in the regimes of low or high activation, as can easily been seen from equations (16) and (21). This phenomenon is known as *saturation*. To avoid this behavior, other activation functions have to be used.
With the equations above, the two-phase backpropagation algorithm can be written as:
Phase 1, forward phase:

1. Setting the corresponding activation a^1 for the input layer
2. Forwarding the signal through the network successively computing $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z^l)$ for every layer.

Phase 2, backward phase:

1. Calculating the output error $\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L)$.
2. Backpropagating the error through the network by successively computing $\delta_j^l = ((w^{l+1})^T\delta^{l+1})_j\sigma'(z_j^l)$.
3. Calculating the gradient of the cost function as $\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l$ and $\frac{\partial C}{\partial b_j^l} = \delta_j^l$.

This provides a simple and storage saving approach to calculating all the gradients needed for gradient descent, and therefore offers a stable and quick algorithm to allow learning in neural networks. By simultaneously computing all partial derivatives $\partial C/\partial w_j$ using just one forward and one backward pass the computational cost of the algorithm is roughly the same as only two forward passes through the network. This offers a very significant speedup compared to earlier methods where gradients had to be computed individually [15].

2.4 Basic neural networks

In the last two years the intersection of machine learning and numerical methods from physics has attracted a lot of attention. Many of these publications rely on the same fundamental elements of machine learning known as *(restricted) Boltzmann machines* and *deep neural networks*. It is therefore instructive to introduce these concepts before moving on to the current research.

2.4.1 Boltzmann machines

A *Boltzmann machine* (BM) is one of the most basic and general neural networks. It simply consists of computing units which are interconnected by bidirectional links. The weights on the links between the units can take on real values of either sign. Through minimizing a cost function one arrives at the configuration that best satisfies the constraints given by the task, e.g. 'weak' constraints for pattern recognition [16].
A *restricted Boltzmann machine* (RBM) is a Boltzmann machine with a bipartite connectivity graph. It is a two-layer network consisting of only one visible and one hidden layer. A pair of units from each of the groups may have a symmetric link between them, but unlike BMs, no connections between units of the same group are allowed for RBMs. An RBM can approximate any distribution and with a sufficiently large number of hidden units can even represent them exactly. This may need a huge number of elements and therefore training examples [17].
The hidden units of a trained RBM may also reveal correlations of the data with physical meaning. For example in an RBM trained with the MNIST dataset of handwritten digits, the connection weight contains the information about pen strokes [9, p.2].

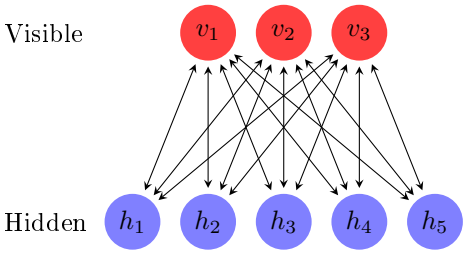


Figure 4: Structure of a Restricted Boltzmann Machine with 3 visible and 5 hidden units.

2.4.2 Deep neural networks

A *deep neural network* or *deep belief network* (DBN) is a probabilistic generative model consisting of multiple layers of stochastic, latent variables. It can be seen as a composition of simple RBMs

where the output of one serves as the input of the next, learning features of increasingly higher complexity. This way a DBN breaks down a very complex question into very simple question answerable at the level of single inputs, e.g. pixels. Early layers answer very simple and specific questions and later layers build up a hierarchy of ever more complex and abstract concepts.
A *deep convolutional arithmetic circuit* (ConvAC) is a deep convolutional network that operates exactly as a regular convolutional network just with linear activations and product pooling layers which introduce the non-linearity instead of the more common non-linear activations and average/-max pooling. Its underlying tensorial structure resembles the quantum many-body wave function [8, p.5].

2.5 Current research

With the knowledge of the fundamentals of neural networks established over the past chapters we now highlight some interesting examples of research connecting the fields of machine learning and condensed matter physics and, in particular, tensor network methods.
Cichocki [18] gives a detailed discussion about the many potential applications of tensor networks in the field of big data.
Mehta and Schwab show the intimate relation between deep learning and the renormalization group, an iterative coarse-graining scheme that allows for the extraction of relevant features from a physical system, e.g. in the form of operators. They then construct an exact mapping between the variational renormalization group and architectures based on RBMs and illustrate this mapping by analytically constructing a deep neural network for the 1D Ising model and numerically examining the 2D Ising model. Their results indicate that deep learning might be employing a generalized renormalization group-like scheme for feature extraction [7].
Carleo and Troyer [10] introduce a representation of quantum states as an RBM and then demonstrate a reinforcement learning scheme to train the network to represent the quantum wave function and determine the ground-state or describe the unitary time evolution of interacting systems. To validate their scheme they consider the problem of finding the ground state of the transverse-field Ising (TFI) model and the antiferromagnetic Heisenberg (AFH) model where they achieve some of the best variational results so-far-reported. They describe it as a 'new powerful tool to solve the quantum many-body problem' [10].
Novikov, Oseledets and Trofimov [19] factorize exponentially large tensors to tensor trains [1]. This format allows them to regularize the model and control the number of underlying parameters. They then develop a stochastic Riemannian optimization procedure to fit large tensors and use the model on synthetic data and the MovieLens 100k dataset [19].
Chen et al. [9] developed an algorithm to translate an RBM into a tensor network state (TNS) and give sufficient and necessary conditions to determine whether a TNS can be transformed into an RBM of given architecture. This connection can then be used to design more powerful deep learning architectures, rigorously quantify their expressive power through the entanglement entropy bound of TNS or represent a quantum many-body state as an RBM with fewer parameter as a TNS [9].
Levine et al. [8] show an equivalence between the function realized by a ConvAC and a quantum many-body wave function. The construction of a ConvAC as a tensor network enables them to carry out a graph-theoretic analysis of a convolutional network providing direct control over the inductive bias of the network [8].
The work by Stoudenmire and Schwab [11] provides a concrete example of applying a tensor network technique (matrix product states) to a machine learning problem, the recognition of handwritten digits, which serves as a benchmark test for neural networks.
In this thesis, we will focus on the algorithm presented in this work, to take a closer look at this interesting intersection between mathematical methods developed in physics and the field of machine learning.

[1] tensor trains = matrix product states

2.6 Problems with machine learning

The process of training a neural network comes with several difficulties, and poses the biggest challenges in machine learning. To finish off the chapter on machine learning, we introduce some of the main problems and their possible solutions. Some of these problems might apply to our learning process, through a tensor network approach, as well.

2.6.1 Overfitting

The ability to generalize is a vital feature of a trained neural network. If the input is slightly different from the examples used to train the network it is still able to produce a correct result. If a neural network is trained with too many training examples it may end up memorizing the training data losing the ability to generalize properly. It ends up modeling random error or noise instead of the underlying function. This problem is known as *overfitting* or *overtraining* and occurs when the model is to complex, e.g. when it has too many parameters relative to the number of observations. The root of this problem is that the criterion for training the model (minimizing the cost function over a set of training data) is not the same as the criterion for judging its effectiveness (its performance on unseen test data).
The simplest way to avoid overtraining is to increase the number of training examples. The size of the training sample N should be of order W/ϵ following *Widrow's rule of thumb*, where W is the number of free parameters in the network and ϵ the fraction of classification errors permitted on test data [12, p.166]. For a small training sample it will be easy for the neural net to just memorize the training data in its entirety, thus minimizing the cost function but failing to correctly classify the test data. Because acquiring a large set of structured data is mostly very difficult in modern applications, other methods have been developed to avoid overfitting. Employing *early stopping*, one devises a set of rules determining when to stop training the network. The model is then trained for a while and then stopped well before it approaches the global minimum.
Weight decay offers a more explicit method for regularization by adding a penalty λJ, with $\lambda \geq 0$ a tuning parameter, to the error function like the *weight elimination* penalty

$$J = \sum_{km} \frac{w_{km}^2}{1 + w_{km}^2} + \sum_{ml} \frac{b_{ml}^2}{1 + b_{ml}^2}, \tag{24}$$

which has the effect of dampening the weights and biases [20].

2.6.2 Starting values

The choice of the initial values for the weights and biases can have a significant effect on the success of the learning process. Usually random values close to zero are chosen where the sigmoid function is roughly linear so that the model becomes nonlinear as the weights increase. Large weights often lead to poor solutions, while zero weights lead to zero derivatives which in turn leads to zero update in the learning algorithm. Choosing initial values close to one, results in the sigmoid function become very flat and therefore slow down the learning speed [20].
There are many texts such as Hinton, 2010 [21] which offer detailed recommendations on how to optimize specific architectures of neural networks but are largely based on heuristics.

3 Tensor networks and matrix product states

Machine learning has recently attracted much attention in condensed matter physics [7, 10]. It presents a possible route to resolving long-standing physical questions like high-T_c superconductivity, which are in the center of active research. *tensor network* (TN) methods, where the wave function of a system is described by a network of interconnected tensors [22], are one family of approaches to resolve many-body problems.
Exploring the combination and connections of tensor networks and machine learning can potentially be very useful for both fields. Machine learning can by improved by TN ideas, e.g. neural network architectures can be optimized through physical considerations such as the entanglement entropy bound TN states after developing an exact mapping between the two [9]. At the same time, condensed matter physics can profit from machine learning ideas, e.g. by representing quantum systems as neural networks to find the ground state of the system [10] or by representing quantum many-body states as an RBM with fewer parameters compared to a TN [9].
There exist different TN representations suitable for the description of different systems. In this text we will only focus on one type of tensor network, so-called *matrix product states* (MPS), which will be applied in the machine learning context in chapter 4. In the following, we will introduce the graphical *tensor network notation*, then explain the basic ideas behind MPS, before going into some important technical details like the *singular value decomposition* needed for our machine learning application as well.

3.1 Tensor network theory

In quantum mechanics states are typically described by a set of coefficients of a wave function in a certain basis. Tensor networks adopt a different representation of a quantum state in terms of a set of interconnected tensors. This allows for a numerically more efficient treatment of a many-body wave function, since one can mediate the otherwise exponential increase in numerical complexity with system size [23]. This formulation also makes information about the structure of entanglement in the system directly available [22]. In the machine learning context we also deal with an exponentially large number of parameters. Therefore, the TN mechanism potentially offers an elegant way to perform the optimization of a neural network.

3.2 Graphical notation for tensor networks

A huge benefit of working with tensor networks is the simple and very transparent notation that has been developed for them. The graphical notation elegantly avoids the explicit treatment of many indices in the standard notation and makes the structure manifest and clean. The *tensor network notation* (TNN) can be considered a generalization of the Einstein summation notation [24]. TNN is essential when dealing with more complex tensor networks such as PEPS [25] and MERA [26], since their structure is so complex that traditional notation becomes unmanageable, but it is already helpful in the context of MPS.

3.2.1 Tensors

Tensors represent the generalization of scalars (rank-0 tensors), vectors (rank-1 tensors) and matrices (rank-2 tensors). While a d-dimensional vector lives in $\mathbb{C}^d$ and a $m \times n$ matrix is element of $\mathbb{C}^{m \times n}$, a rank-r tensor of dimensions $d_1 \times ... \times d_r$ is an element of $\mathbb{C}^{d_1 \times ... \times d_r}$. For our purposes, a tensor is a multidimensional array of complex numbers, with the rank equal to the number of indices.
The basic graphical notation for a tensor is a closed geometrical shape, typically a circle, though other shapes can be used to distinguish different kinds of tensors. Each index of the tensor is represented by a line or "leg" coming from it. The direction of the legs can be used to indicate certain properties, e.g. whether a quantum state lives in Hilbert space ("ket") or is dual ("bra").

This is analogous to denoting upper and lower indices in Einstein notation [24].

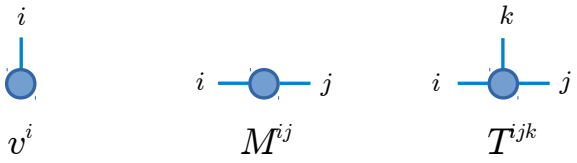


Figure 5: Graphical notation for a vector v^i, a matrix M^{ij} and a rank-3 tensor T^{ijk}.

3.2.2 Tensor operation

Tensor operations also have a very simple diagrammatic representation. To indicate that a certain pair of indices are contracted, the corresponding legs are simply connected through a line.

Figure 6: Graphical (top) and explicit index (bottom) notation for a matrix-vector multiplication (left) and a more general tensor contraction of a rank-4 and a rank-3 tensor.

Other operations like the tensor product and the trace have equally simple and instructive representations:

Figure 7: Graphical notation for the tensor product (left) and the trace operation (right).

Instead of explicitly writing out the full expressions, the TNN is compact and avoids the need to explicitly write every index sum performed in an operation. The rank of the final result can easily be determined by counting the number of open lines after all operations have been performed. In particular a complicated set of tensor operations can be recognized as a scalar result if no indices remain open in a particular diagram [11].

3.3 Matrix product states

An arbitrary quantum state can be represented by a coefficient tensor in Fock space. Consider a one-dimensional lattice with L sites and d-dimensional local state spaces $|\sigma_i\rangle$ on the sites $i = 1, ..., L$. A general pure quantum state on the lattice is given by

$$|\Psi\rangle = \sum_{\sigma_1 ... \sigma_L} c_{\sigma_1 ... \sigma_L} |\sigma_1 ... \sigma_L\rangle, \tag{25}$$

with a coefficient tensor containing d^L elements, that clearly scales exponentially with system size L.
How can we represent a quantum state of such a system as a tensor network and avoid the exponential scaling of the number of coefficients with system size? To this end, the coefficient tensor is approximated by an array of N lower-rank tensors, which are then contracted over a

sequence of virtual bond indices,

$$|\Psi\rangle = \sum_{\sigma_1,...,\sigma_L} \sum_{\alpha_1,...,\alpha_{L+1}} A^{\sigma_1}_{\alpha_1\alpha_2} A^{\sigma_2}_{\alpha_2\alpha_3} ... A^{\sigma_{L-1}}_{\alpha_{L-1}\alpha_L} A^{\sigma_L}_{\alpha_L\alpha_{L+1}} |\sigma_1, ..., \sigma_L\rangle, \tag{26}$$

where we have used a compact notation of the rank-3 tensors $A^{[\sigma_j]}_{\alpha_j\alpha_{j+1}}$, with virtual bond indices $\alpha_j = 1, ..., m$. For open boundary conditions, as realized in Ch. 4, α_1 and α_{L+1} are 1. This one-dimensional decomposition of a tensor (see Fig. 8) is called a *matrix product state* (MPS) [11], and will be discussed in more detail later.
To avoid the exponential scaling of the number of coefficients we now limit the coefficient space by a bound on the bond indices, the so-called *bond dimension* controlling the accuracy of the MPS approximation. Despite neglecting large parts of the Hilbert space in this way, the MPS is still an excellent approximation for many physical states of 1D quantum systems, e.g. low energy or thermal states [27]. Many numerical techniques in condensed matter physics rely on the MPS framework. The most famous example is the *density matrix renormalization group* (DMRG) [28, 29] and its generalizations. In Ch. 3.3.1 we will explain the rationale for the truncation and the resulting approximation.
Matrix product states present a formalism that can not only be used in physics but also has great potential for applications in machine learning. In this thesis we will use a set of techniques derived from DMRG to optimize our neural network in chapter 4. Next, we introduce a key mathematical method used for most MPS algorithms, the *singular value decomposition* (SVD), which also represents an important tool for our machine learning algorithm.

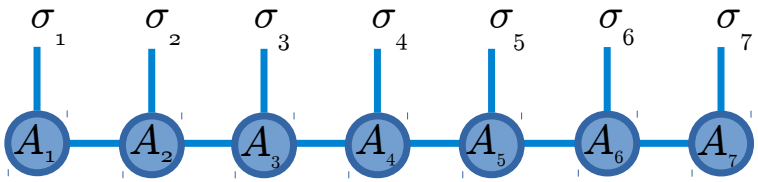


Figure 8: Graphical notation of a matrix product state for a length-7 system consisting of 7 tensors A_i

3.3.1 Singular value decomposition

A *singular value decomposition* (SVD) is a linear algebra tool which lies at the core of most MPS algorithms. A SVD decomposes an arbitrary (rectangular) matrix M of dimension $(N_A \times N_B)$ into

$$M = USV^\dagger, \tag{27}$$

with the following properties:

- U is a $(N_A \times min(N_A, N_B))$ matrix with orthonormal columns the so-called *left singular vectors*. $U^\dagger U = I$ especially if $N_A \leq N_B$ also $UU^\dagger = I$ (in that case U is unitary).
- S is a $(min(N_A, N_B) \times min(N_A, N_B)$ diagonal matrix with non-negative entries called the *singular values* $S_{aa} \equiv s_a$. The number of non-zero singular values is the *(Schmidt) rank* of M and descending order of the singular values is assumed in this text: $s_1 \geq ... \geq s_r \geq 0$.
- $V^\dagger$ is $(min(N_A, N_B) \times N_B)$ matrix with orthonormal rows the so-called *right singular vectors*. $V^\dagger V = I$ especially if $N_A \geq N_B$ also $VV^\dagger = I$ (in that case V is unitary).

An important consequence is the optimal approximation of M of rank r by a matrix M' of rank $m < r$ in the Frobenius norm $||M||_F^2 = \sum_{ij} |M_{ij}|^2$ induced by the inner product $\langle M|N\rangle = Tr(M^\dagger N)$ given by

$$M' = US'V^\dagger, \qquad \text{with} \qquad S' = diag(s_1, s_2, ..., s_m, 0, ...). \tag{28}$$

Figure 9: Graphical representation of the matrix shapes resulting from a SVD $M = USV^\dagger$. The diagonal line indicates that S is a strictly non-negative diagonal matrix.

This is achieved by setting all but the largest m singular values to zero. In numerical practice the column dimension of U and the row dimension of $V^\dagger$ are also reduced accordingly [27].
This offers a very simple method to reduce or limit the dimension of a tensor. When the dimension of the tensor M becomes bigger than a desired value during an application it can simply be truncated by using its optimal approximation M' instead, thereby keeping the dimension fixed to the desired m. The value of m for truncation can also be set adaptively by keeping only the those singular values bigger than a certain threshold. Truncation is also very important in our machine learning algorithm. There the dimension of the tensor being optimized grows very quickly and therefore has to be truncated to a fixed value to keep the time needed for computation low.

3.3.2 Decomposing arbitrary states into a MPS

An arbitrary quantum state can easily be decomposed into a matrix product state. This is important since an analogous construction will enable us to describe the weights of our neural network as a MPS.
The decomposition of the state (25) into (26) is achieved through a series of SVDs (a very detailed derivation can be found in [27, Chapter 4.1.3]), where $A^{\sigma_j}_{a_i,a_j} = U_{(a_i\sigma_j),a_j}$ and the decomposition was started from the left.
These A-tensors exhibit the following properties:

- For an exact decomposition of the state the bond indices (a_i, a_j) of the first A-tensor start of as $(1, d)$, then scale exponentially until they reach $(d^{L/2-1}, d^{L/2})$ and $(d^{L/2}, d^{L/2-1})$ for the A-tensors in the middle of the chain, assuming even L, and then decrease exponentially to reach $(d, 1)$ at the last site. In practical calculations, it is typically impossible to carry out this exact decomposition as the tensor dimensions blow up exponentially. To allow for numerical feasibility a upper cutoff dimension m is required.

- Each SVD guarantees $U^\dagger U = I$ and therefore

$$\sum_{\sigma_l} A^{\sigma_l\dagger} A^{\sigma_l} = I. \tag{29}$$

  Tensors fulfilling this condition are called *left-normalized* and matrix product states consisting of only left-normalized tensors are *left canonical*.

Analogously a similar decomposition can be obtained starting from the right to obtain

$$|\Psi\rangle = \sum_{\sigma_1,\ldots,\sigma_L} B^{\sigma_1} B^{\sigma_2} \ldots B^{\sigma_{L-1}} B^{\sigma_L} |\sigma_1, \ldots, \sigma_L\rangle. \tag{30}$$

The B-tensors can be shown to have the same tensor dimension bound as the A-tensors and from $V^\dagger V = I$ follows, that

$$\sum_{\sigma_l} B^{\sigma_l} B^{\sigma_l\dagger} = I. \tag{31}$$

These tensors are therefore called *right-normalized* and an MPS built entirely from such tensors is *right-canonical*.
Introducing the vectors

$$|a_l\rangle_A = \sum_{\sigma_1,\ldots,\sigma_l} (A^{\sigma_1} \ldots A^{\sigma_l})_{1,a_l} |\sigma_1, \ldots, \sigma_l\rangle \tag{32}$$

$$|a_l\rangle_B = \sum_{\sigma_{l+1},...,\sigma_L} (B^{\sigma_{l+1}}...B^{\sigma_L})_{a_l,1}|\sigma_{l+1},...,\sigma_L\rangle, \tag{33}$$

the state can be written as

$$|\Psi\rangle = \sum_{a_l} s_a|a_l\rangle_A|a_l\rangle_B, \tag{34}$$

where $s_a = S_{aa}$, known as a *mixed-canonical basis* [27]. This representation allows for local updates of the wave function in an optimal way, which is key to approaches such as DMRG (see [27]) as well as for our machine learning algorithm.
In physical applications the truncation is very successful because of the so called *entanglement area laws* which guarantee an exponentially decreasing singular value spectrum. Therefore the optimal approximation truncated through a threshold is typically a good representation of the actual state. Again [27] offers a very detailed discussion about the physical background but since these area laws do not apply in the machine learning context we will not go into further detail here. Also Cichocki offers further background about these more technical aspects of MPS in [18].

4 MPS framework for machine learning

Recently, several papers suggested that tensor networks can not only be applied in quantum many-body systems but can also be effective in a machine learning context. Two papers in particular pointed out that tensor networks can represent powerful tools in the setting of non-linear kernel learning [11, 19]. This means optimizing a decision function of the form

$$f(\boldsymbol{x}) = W \cdot \Phi(\boldsymbol{x}), \tag{35}$$

where input vectors $\boldsymbol{x}$ are mapped into a higher dimensional space via a *feature map* $\Phi(\boldsymbol{x})$ and the feature vector $\Phi(\boldsymbol{x})$ and the weight tensor W can be exponentially large.
Here, we adapt the approach taken in Ref. [11] following a different direction than the typical machine learning approaches described in Chapter 2. To this end, we approximate the optimal weight tensor W as a matrix product state to optimize the weights directly and adaptively change their number by locally varying W two tensors at a time. The rest of the network is stored in two blocks, which are not altered during the local update. The details of this procedure closely follow the DMRG algorithm. During training, dimensions of tensor indices grow and shrink to concentrate resources on the most relevant correlations within the data. This training procedure scales only linearly with the training set size. Furthermore, the MPS representation of W offers the possibility to extract information from the trained model that would otherwise be hidden. Additionally, the form of the tensor network adds another type of regularization beyond the choice of $\Phi(\boldsymbol{x})$, which could have interesting consequences for generalization [11].
In this chapter an MPS approach for solving a machine learning task, specifically the recognition of handwritten digits from the MNIST dataset, will be implemented in MatLab following Stoudenmire and Schwab [11]. We will explain the feature map and the MPS approximation of the weight tensor W and then go through the details of the optimization algorithm. Finally we will discuss the results we obtained.

4.1 Algorithm

4.1.1 Encoding input data

To account for the 1D structure of the MPS representation, we first have to classify the input data by mapping each component x_j of the input data vector $\boldsymbol{x}$ to a d-dimensional vector. Analogous to many-body Fock states, we choose a feature map of similar form for our machine learning approach

$$\Phi^{s_1 s_2 \cdots s_N}(\boldsymbol{x}) = \Phi^{s_1}(x_1) \otimes \Phi^{s_2}(x_2) \otimes \ldots \Phi^{s_N}(x_N). \tag{36}$$

The tensor $\Phi^{s_1 s_2 \cdots s_N}$ is the tensor product of the same *local feature map* $\Phi^{s_j}(x_j)$ applied to each input x_j and the indices $s_j \in [1, d]$, where d is known as the *local dimension*. Analogous to working with normalized wave functions in physics, local feature map is required to have unit norm, which in turn implies that $\Phi(\boldsymbol{x})$ also has unit norm [11]. In physical terms the feature map has the structure of a product state or unentangled wave function. The graphical notation is shown in Fig. 10.

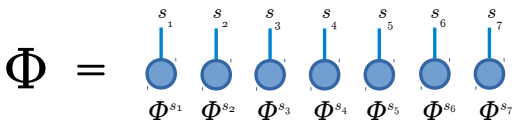


Figure 10: Graphical notation of the feature map Φ, a normalized order N tensor of dimension d^N and rank-1 product structure.

In our example, the input data are grayscale pictures with N pixels. Every pixel has a value

between 0 for white and 1 for black [2]. We then choose the simple local feature map

$$\Phi^{s_j}(x_j) = [\cos(\frac{\pi}{2}x_j), \sin(\frac{\pi}{2}x_j)] \tag{37}$$

for the input x_j from pixel j. Thus the full picture is represented by a tensor product of this local vector for each pixel according to equation (36). Even though in our implementation only this feature map was tested, it would be interesting to try other higher dimensional ($d > 2$) local feature maps, corresponding to higher spin models, to investigate what role they play in performance and optimization cost of the model. For a more detailed discussion see Ref. [11, Appendix B].

4.1.2 MPS approximation

For classification we generalize the decision function in Eq. (35) to a set of functions indexed by a label l,

$$f^l(\boldsymbol{x}) = W^l \cdot \Phi(\boldsymbol{x}). \tag{38}$$

An input $\boldsymbol{x}$ is then classified by choosing the label l for which $|f^l(\boldsymbol{x})|$ becomes maximal. The quantity that depends on the label l is the weight vector W^l which will be viewed as a tensor of order $N+1$ with $N_L \cdot d^N$ components, where N_L is the number of labels. To regularize and optimize this tensor efficiently it is decomposed into a matrix product state of the form

$$W^l_{s_1 s_2 \ldots s_N} = \sum_{\{\alpha\}} A^{\alpha_1}_{s_1} A^{\alpha_1\alpha_2}_{s_2} \cdots A^{l;\alpha_{j-1}\alpha_j}_{s_j} \cdots A^{\alpha_{N-1}}_{s_N}. \tag{39}$$

The exponentially large set of components is approximated by a much smaller set of parameters whose number only grows polynomially with the size of the input space [3]. The compact graphical notation of (39) is shown in Fig. 11.

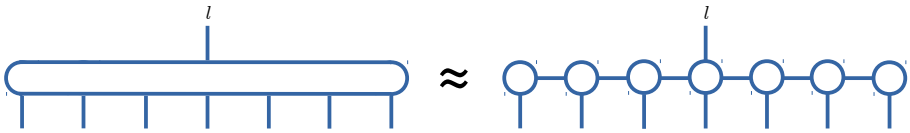


Figure 11: Graphical notation of the approximation of the weight tensor W^l by a matrix product state (see Eq. (39)). The label index l is placed on an arbitrary tensor of the MPS and can be moved to another location.

The parameter controlling the MPS approximation is the *bond dimension* m of the virtual indices α_j. Just like an RBM can approximate any distribution with a sufficiently large number of hidden units (see Ch. 2.4.1), an MPS can represent any tensor for a sufficiently large m [30]. In physics application m is typically set between 10 and 10,000 and it is desirable to set it as large as possible since a larger bond dimension means more accuracy. In Eq. (39) the label index l is put on the j^{th} tensor but this choice is arbitrary. In fact, the index can be moved to any other tensor of the MPS through a singular value decomposition similar to the procedure explained in 4.1.3 without changing the overall W^l.
In our example, the MPS is initialized as a chain of N tensors filled with random numbers between 0 and 1, where N equals the number of pixels. The label index l was put on the N^{th} tensor of the chain and the MPS is brought into left-canonical form through a series of SVDs. The bond dimension m is chosen between 10 and 120.

[2] In the actual data the values range from 0 for white to 255 for black. They are divided by 255 for the feature map.

[3] Only if we impose a cut-off dimension m.

4.1.3 Sweeping algorithm for optimizing weights

The core of our algorithm is a mechanism inspired by the DMRG algorithm used in physics. The algorithm "sweeps" back and forth along the MPS iteratively minimizing the cost function C by updating the tensors locally in a two-site update. The cost function C in our classification task is the quadratic cost

$$C = \frac{1}{2}\sum_{n=1}^{N_T}\sum_{l}(f^l(\boldsymbol{x}_n) - \delta^l_{L_n})^2, \qquad \text{with} \quad \delta^l_{L_n} = \begin{cases} 1, & \text{if } l = L_n \\ 0, & \text{otherwise} \end{cases}, \tag{40}$$

where N_T is the number of training inputs and L_n is the known correct label for training input n. For our optimization we could also choose a one-site update where we only vary one tensor at a time, however, the procedure where two adjacent tensors are varied at the same time turns out to be much more convenient here. This does not only enable us to adaptively change the MPS bond dimension but also offers a convenient way to permute the label index l through the network.
Let us now consider the details of this two-site update procedure. Assume we have moved the label index l to the j^{th} tensor $A^l_{s_j}$. This particular tensor shares the j^{th} bond with the $(j+1)^{th}$ tensor $A_{s_{j+1}}$. These two are then contracted to form a single *bond tensor* by contracting over α_j,

$$A^{l;\alpha_{j-1}\alpha_j}_{s_j} A^{\alpha_j\alpha_{j+1}}_{s_{j+1}} = B^{\alpha_{j-1};l;\alpha_{j+1}}_{s_j s_{j+1}}. \tag{41}$$

The much simpler graphical notation can be seen in Fig. 12. In analogy to the gradient descent

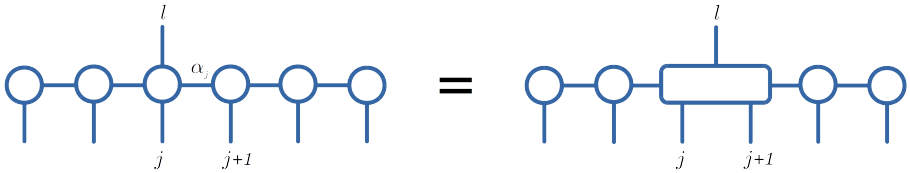


Figure 12: Forming the bond tensor B by contracting the two tensors on sites j and $j+1$.

step used in machine learning, we now compute the derivative of the cost function C with respect to the bond tensor B^l to iteratively update the components of the MPS. Since we only update two sites at a time, we can use a localized approach. Therefore each training input $\boldsymbol{x}_n$ is projected through the fixed *local projection* of the MPS shown in Fig. 13. This results in a $\tilde{\Phi}_n$ with four indices as shown on the right-hand side of the same figure.

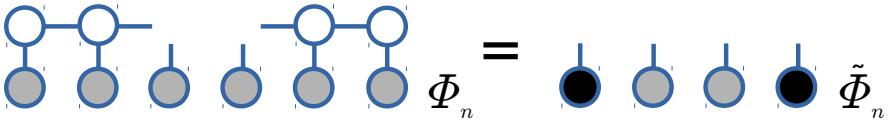


Figure 13: Projecting a training input into the MPS basis at bond j.

A detailed discussion about how these projected inputs are efficiently calculated and stored in our code can be found later in Ch. 4.1.4. Here we will concentrate on the update of the bond tensor B^l. Given the projected input $\tilde{\Phi}_n$, the local decision function can be efficiently computed combining $\tilde{\Phi}_n$ and the current bond tensor B^l (see Fig. 14) as

$$f^l(\boldsymbol{x}_n) = \sum_{\alpha_{j-1}\alpha_{j+1}}\sum_{s_j s_{j+1}} B^{\alpha_{j-1};l;\alpha_{j+1}}_{s_j s_{j+1}} (\tilde{\Phi}_n)^{s_j s_{j+1}}_{\alpha_{j-1}\alpha_{j+1}}. \tag{42}$$

Analogous to the gradient descent algorithm discussed in detail in 2.3.1, the leading order

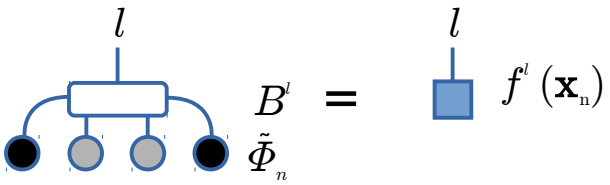


Figure 14: Computing the local decision function in terms of the projected input $\tilde{\Phi}_n$.

update to the bond tensor is then computed as

$$\begin{aligned}\Delta B^l &\equiv -\frac{\partial C}{\partial B^l} \\ &= \sum_{n=1}^{N_T}\sum_{l'}(\delta^{l'}_{L_n} - f^{l'}(\boldsymbol{x}_n))\frac{\partial f^{l'}(\boldsymbol{x}_n)}{\partial B^l} \\ &= \sum_{n=1}^{N_T}(\delta^{l}_{L_n} - f^{l}(\boldsymbol{x}_n))\tilde{\Phi}_n.\end{aligned} \tag{43}$$

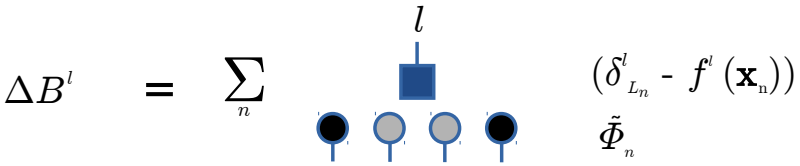


Figure 15: Gradient of the bond tensor B^l.

The resulting object ΔB^l (see Fig. 15) has the same index structure as the bond tensor B^l. In analogy to the gradient descent update, we now add this small update to B^l (see Fig. 16)

$$B^l \quad \rightarrow \quad B'^l = B^l - \alpha\frac{\partial C}{\partial B^l} = B^l + \alpha\Delta B^l, \tag{44}$$

with α being the analogon to the step size used to control convergence. As in the case of machine learning, the step size in our application is set following empirical observations and its effect on the algorithm is discussed in detail in Ch. 5.

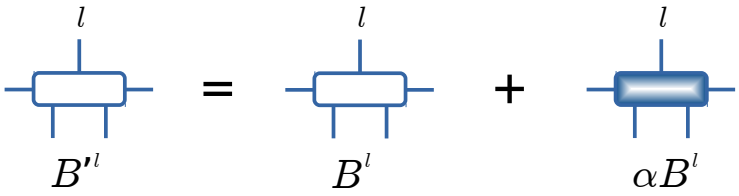


Figure 16: Gradient descent-like update of the bond tensor B^l with step size α.

After the update, the bond tensor B^l has to be decomposed into separate tensors to restore the original form of the MPS and to be able to apply the algorithm again to the next bond. Assume we are moving from left to right in our optimization process and the next bond to be optimized is the $(j+1)^{th}$. We then compute a singular value decomposition of B^l as shown in figure 17, or explicitly given by

$$B^{\alpha_{j-1} l \alpha_{j+1}}_{s_j s_{j+1}} = \sum_{\alpha'_j \alpha_j} U^{\alpha_{j-1}}_{s_j \alpha'_j} S^{\alpha'_j}_{\alpha_j} V^{\alpha_j l \alpha_{j+1}}_{s_{j+1}}. \tag{45}$$

In this way, we restore the MPS form and at the same time move the label index l one site to the right onto the $(j+1)^{th}$ tensor. In order to fully restore the MPS, we define $U_{s_j} = A'_{s_j}$ as the new tensor at site j and $SV^l_{s_{j+1}} = A'^l_{s_{j+1}}$ to be the new tensor on site $j+1$.

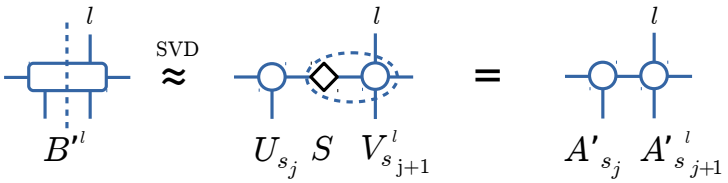


Figure 17: Restoration of MPS form and translation of label index l through a SVD.

The collective dimension of B^l can increase through the contraction of the two initial MPS tensors to one bond tensor and the subsequent SVD. Starting from two initial tensors of size $d_{j-1} \times d_j \times d_{s_j} \times d_l$ and $d_j \times d_{j+1} \times d_{s_{j+1}}$, the matrix dimension of B^l is $d_{j-1} \cdot d_j \cdot d_{s_j} \times d_j \cdot d_{j+1} \cdot d_{s_{j+1}} \cdot d_l$. Carrying out an exact SVD the two resulting new MPS tensors therefore have a larger bond dimension. Keeping track of the increasing bond dimension throughout the sweep quickly becomes unfeasible.
Therefore it is crucial to control the dimension of the resulting tensors. We use the optimal approximation (see Ch. 3.3.1) of B^l keeping only the m largest singular values in S and discarding the rest along with the corresponding columns of U and $V^\dagger$. If all of the MPS tensors to the left and the right of the bond tensor are in canonical form, then the truncation of B^l is globally optimal for the entire MPS [11]. In our application, this is the case since our initial state is left-canonical per construction.

4.1.4 Data block initialization

Proceeding to the next bond, it would be highly inefficient to perform the full projection of each input vector from scratch, as shown in Fig. 13. Therefore, we have to find a way to store and compute the local projections of the input data efficiently along the way, since they have to be constantly updated to the current local basis.
We start by initializing the data storage. The first entry is formed by simply contracting the first A-tensor with the corresponding input mapped trough the feature map. The next entry is then obtained by first performing the same contraction between the A-tensor and the mapped input on that site and then contracting the result with the tensor formed on the first site (see Fig. 18, step 1). This procedure is iterated site-by-site and the resulting blocks for every site are saved in a cell array until only two sites are left (see Fig. 18, step 2).

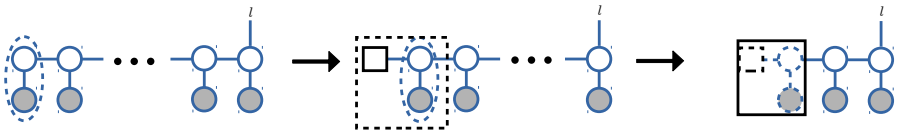


Figure 18: Initialization of the local projections of the input data.

The two remaining sites $N-1$ and N are updated and decomposed into two separate tensors through the SVD procedure described above. The N^{th} tensor is then contracted with its corresponding input Φ^{s_N} to form the first element of a block on the right. The relevant block on the left side has already been calculated before and can be restored from the data storage (see Fig. 19 step (1)). During optimization, the block tensor is then iteratively updated and moved through the MPS chain. The block on the right is updated iteratively using the procedure shown in figure 18, while the block on the left is recycling the blocks saved in the data storage (see Fig. 19 step

(2)). This procedure is iterated until only two sites are on the left (see step (3)). Now the sweeping direction is reversed and the procedure is performed analogously in the opposite direction.

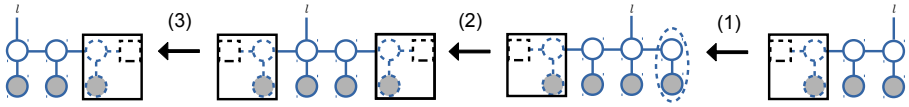


Figure 19: Sweeping algorithm from right to left closely resembling DMRG with one block growing at the expense of the other block with two sites in between.

This sweeping algorithm exhibits clear analogies to DMRG. The projected inputs are obtained and updated in a similar form, as the local Hamiltonian projections occurring in the DMRG algorithm. Also sweeping occurs in the same fashion and dimensionality can similarly be controlled through SVD.
This way of computing the projected input allows the cost of each local step of the algorithm to remain independent of the size of input space, which allows the total algorithm to scale only linearly with input space size.

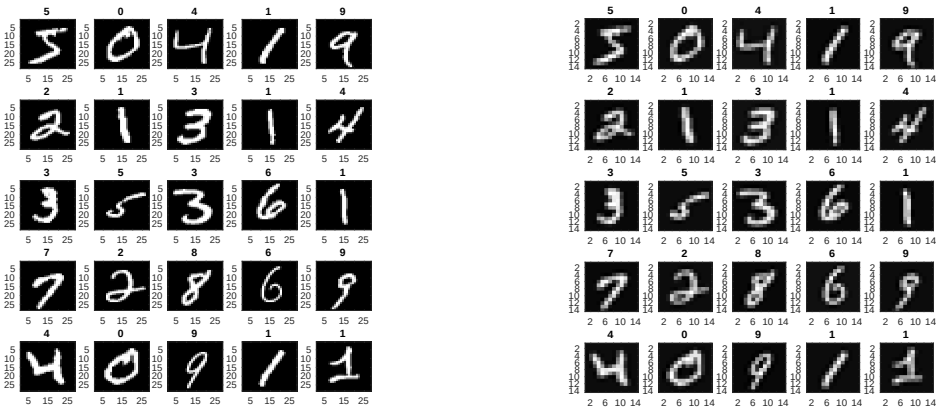
4.1.5 Normalization

To avoid the occurrence of very high or small coefficients in the tensors that could cause numerical instabilities in the algorithm, normalization is necessary. After performing the update of the bond tensor B^l the singular value spectra in the SVD are divided by the trace, $tr(S^\dagger S)$. This normalization is inspired by the physical application of MPS where a normalized SVD spectrum typically yields $\langle\Psi|\Psi\rangle = 1$. Additionally, after every step of the data block update, the blocks are normalized by dividing each component by the largest entry.

4.2 The MNIST dataset

We test the algorithm of Ref. [11] using a standard benchmark test for machine learning: the recognition of handwritten digits from the *MNIST dataset* [4]. The *Modified National Institute of Standards and Technology database* consists of 60,000 training and 10,000 testing images. It was created from two databases from the US National Institute of Standards and Technology containing handwritten digits collected from Census Bureau employees and from high-school students. To get results independent of the choice of training set the two databases where then mixed 50/50 to form the new modified dataset. The training and test sets contain examples from roughly 250 different writers each. The two sets of writers are disjoint to ensure the system being tested can recognize digits from people whose writing it did not see during training.

[4] see http://yann.lecun.com/exdb/mnist/ , the official home of the database and [31]



(a) Original Data (b) Downscaled Data

Figure 20: First 25 digits of the MNIST Dataset with correct labels over every image. Original (left) and downscaled (right) as used in our application.

The original grayscale pictures were size normalized to fit a 20x20 pixel box while keeping their aspect ratio constant and then centered in a 28x28 pixel image by computing the center of mass of the pixels and translating that point to the center (see Fig. 20a).
To reduce computation time, we downscaled the images to 14x14 pixels (see Fig. 20b) by averaging over clusters of four pixels. The pixels are labeled in a "zig-zag" ordering which on average keeps spatially neighboring pixels as close to each other as possible in the MPS representation [11].

5 Discussion

Using the sweeping algorithm described in the last chapter to optimize the weights, we now explore the performance and the convergence properties with respect to changes in different numerical parameters. In particular, we tested the sensitivity of the algorithm with respect to the number of sweeps, the step size α and the bond dimension m. To measure convergence we use test and training error rates, which are defined as the fraction of misclassified images, when checking with the test image set and the set of images the MPS was trained with, respectively.
The algorithm converges quickly in the number of sweeps over the MPS and typically only requires two or three sweeps after which test error rates only vary slightly. This is to some extend similar to the original paper. However, while Stoudenmire achieves "test error rates changing only hundredths of a percent"[11, p.5] after two or three sweeps, our results sometimes still show changes of tenths of a percent, for a MPS trained with 60,000 images. At a bond dimension of $m = 120$, with suboptimal $\alpha = 10^{-6}$ (see minimum in Fig. 23), test error rates as low as 3.5% were achieved. This is not far from the reported 0.97% in the original paper [11], and with the possible improvements of the algorithm discussed in the following, similar error rates might well be achieved. We will also discuss various uncertainties and possible sources of error in our algorithm.

5.1 Normalization

During early testing of the algorithm it became necessary to introduce some kind of normalization as the entries of the MPS tensors would grow rapidly which eventually let the algorithm fail. Therefore all components of the data blocks were divided by the biggest entry after every update of the data blocks. That way the convergence of the algorithm was restored but the normalization remains somewhat arbitrary and it remains unclear how it influences the performance of the algorithm and needs to be investigated further.
Additionally the SVD spectrum was normalized during each update for the same reason. Here we took the inspiration for the normalization from the physical application of MPS where a normalized SVD spectrum typically yields $\langle\Psi|\Psi\rangle = 1$. From this thought we decided to normalize the spectrum by the trace, which lead to good convergence and seemed natural due to the reasoning from physics.

5.2 Bond dimension

As in the original work, test error rates decrease rapidly with the maximum bond dimension m. Fig. 21 illustrates this dependency of the error rates for an MPS trained with 20,000 training images and tested on 10,000 test images with step size $\alpha = 0.001$. The two lines are errors from testing with training and with test data. Fig. 21, 22, 23 suggest that the error recognizing the training data is higher than the error classifying unknown test data. This counterintuitive result could be caused by the fact that the system has not completely converged after five sweeps or due to numerical instabilities of our implementation.

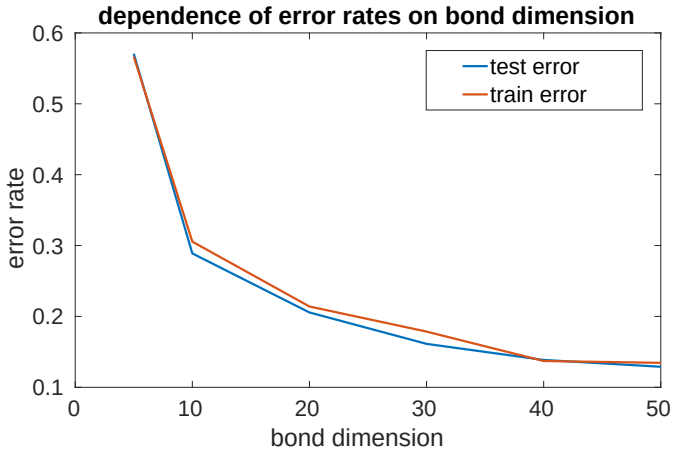
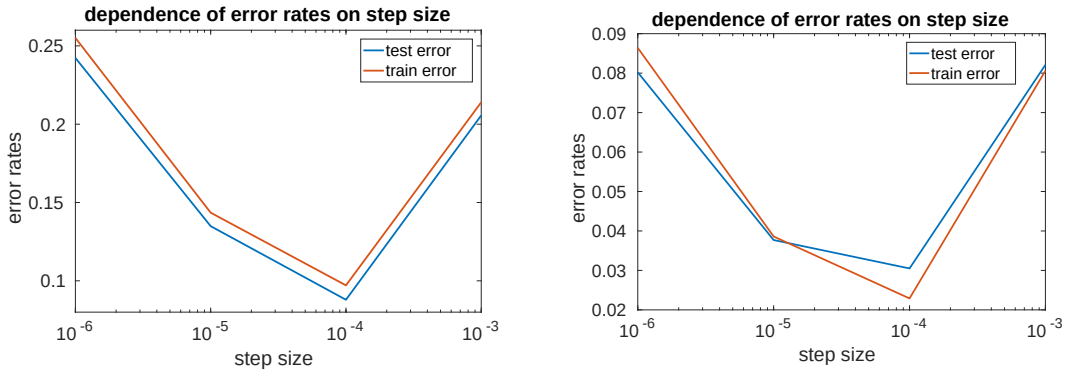


Figure 21: Dependence of the error rates on the bond dimension m after five sweeps. The MPS was trained with 20,000 training images and tested on 10,000 test images with step size $\alpha = 0.001$.

5.3 Step size

Something that is not discussed in detail in Ref. [11] is the strong dependence of the results on the choice of the step size α. This parameter has a strong influence on the error rates achieved and is purely empirical. As can be seen in Fig. 22, the step size changes the error rates greatly and has a clear minimum at $\alpha = 10^{-4}$ for a training set of 20,000 images. However, the optimal value for α changes with the size of the training sample, making it necessary to determine the optimal α every time the sample size is changed (see Fig. 23).



(a) $m = 20$ (b) $m = 80$

Figure 22: Dependence of error rates on parameter α for different bond dimensions m after 5 sweeps. The MPS was trained with 20,000 training images and tested on 10,000 test images.

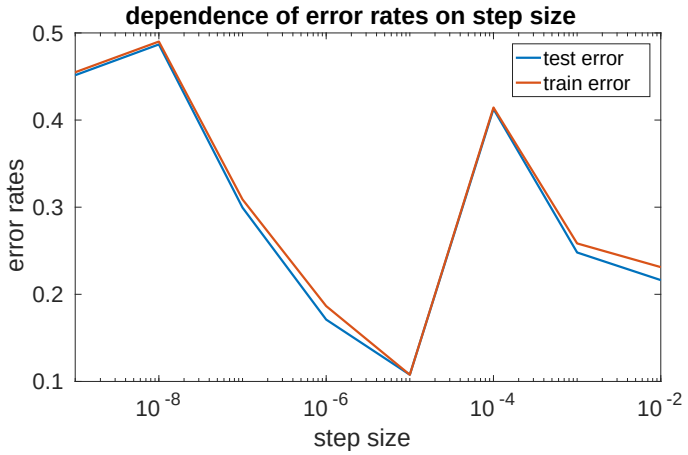


Figure 23: Dependence of error rates on parameter α for an MPS trained with 60,000 training images and tested on 10,000 test images after 5 sweeps with bond dimension $m = 20$.

Note that for some values of α, the error rates increase despite reaching already low values in earlier sweeps. For instance, the data presented in Fig. 23 shows a local maximum for $\alpha = 10^{-4}$ after five sweeps. The test error rate after one sweep is as low as 18.4% before increasing to 38.48% in the second and eventually 41.27% in the fifth and last sweep. The convergence of the algorithm is therefore lost. It is thus very important to stress the high dependence of the algorithm on this empirical parameter.

5.4 Additional remarks

The optimization used here represents the gradient descent step, where B^l was updated by simply adding $\alpha \Delta B^l$, which is not the optimal update scheme for such a optimization problem. Here it is mainly employed for simplicity and its similarity to standard supervised learning techniques (see Ch. 2.3.1). In fact, the conjugate gradient descent method would improve performance [11] and could be implemented in a straightforward way. However, this is beyond the scope of this thesis.
The one dimensional mapping of the two dimensional images of handwritten digits might not be ideal to capture the correlations between pixels. Even though MPS are optimized for one-dimensional patterns of correlations they still offer powerful performance for a two-dimensional system such as the images we are dealing with [32]. However, while an MPS can still approximate power-law decays over quite long distances [11] the choice of tensor network may also influence the efficiency of the algorithm. Other two-dimensional networks might offer superior modeling capacity, e.g. a MERA (multi-scale entanglement renormalization ansatz) network [26], which can explicitly model power-law decaying correlations or Projected Entangled Pair States (PEPS) [25], which are explicitly designed for two-dimensional systems. It would be interesting to investigate this further and find the best tensor network for a given task.
As mentioned above, it is not necessary to truncate with respect to a fixed bond dimension. One key advantage of the MPS representation is that the bond dimension can be chosen *adaptively* based on some threshold. In this way, a variable number of singular values larger than some threshold are kept depending on how much entanglement (i.e., correlations) is in the system. This feature enables us to compress the MPS form of W^l as much as possible while still ensuring an optimal decision function [11]. However, we found that the typical singular value spectra (see Fig. 24) are not decreasing fast enough to find an suitable threshold that does not exceed our computational limitations. For an optimal representation the SVD spectra of the MPS would have to decrease exponentially. A couple of test were performed with different thresholds but none of them had any success. Either the MPS bond dimensions exploded during the first sweep or so few singular values were kept that error rates became very high.

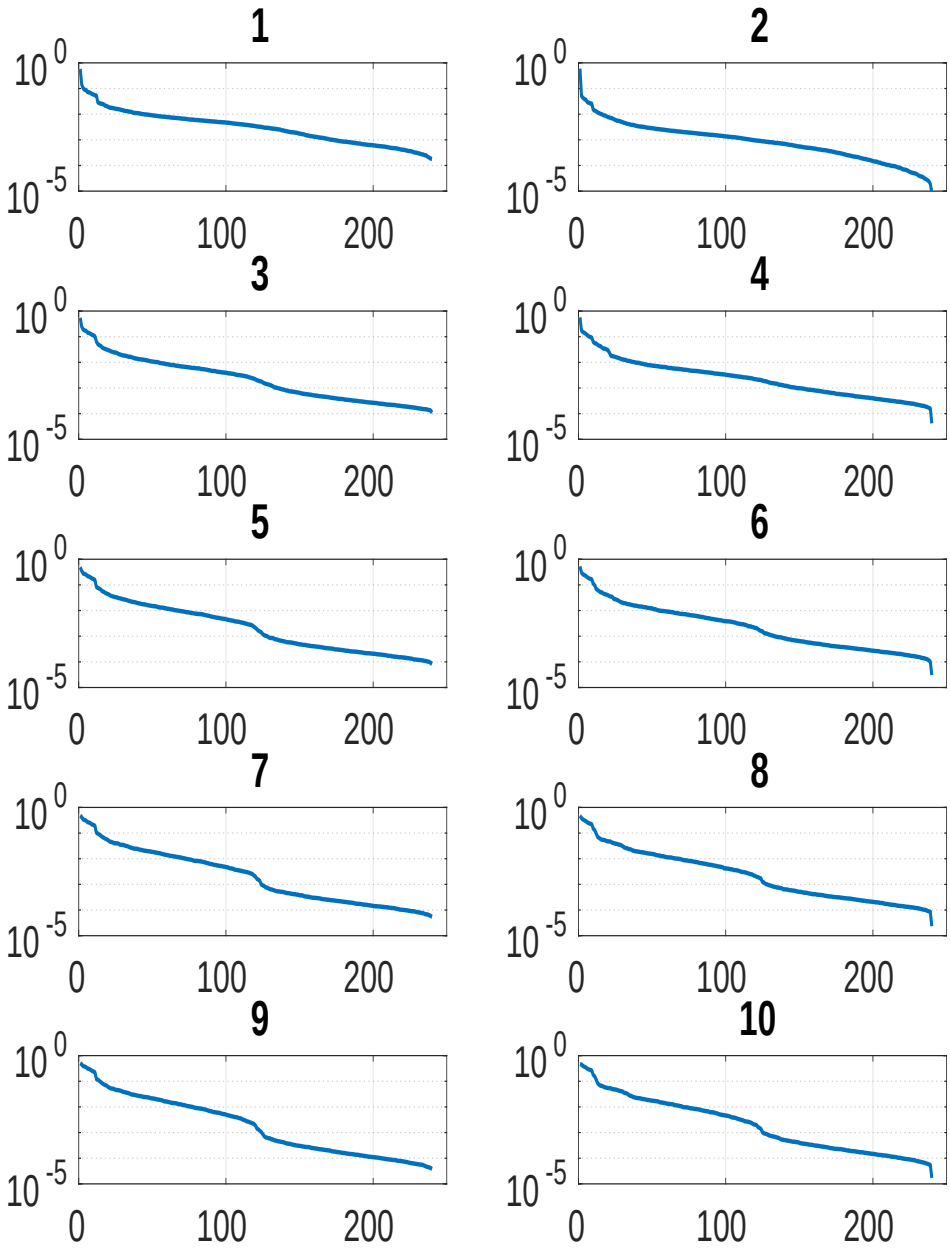


Figure 24: Normalized singular value spectrum before truncation at the 100^{th} site of the MPS recorded over five sweeps for a bond dimension of $m = 120$. Each row represents one sweep, with the left panels being the sweep from the right side of the chain to the left side and the right panels being the sweep back.

6 Conclusion

In this thesis, we studied recent developments in the field of computational condensed matter physics, where it becomes increasingly popular to combine machine learning with well-established tensor network techniques. To this end, we first discussed some background knowledge about machine learning and neural networks and elaborated on the basis of a specific class of tensor networks, so-called matrix product states (MPS). Then we introduced a quantum-inspired tensor network approach to multi-class supervised learning tasks. While the one-dimensional MPS ansatz worked well even for the two-dimensional MNIST handwritten digit data, much work remains to determine the optimal tensor network for a given domain. Other tensor networks may be more suitable and offer superior performance, such as PEPS, which are explicitly designed for two-dimensional systems. Also finding the optimal set of parameters, especially the best step size α requires more work and tests. It would be helpful to understand the effects of the applied normalization and the regularization through the tensor network parametrization.
Still, the representation of the weights of a neural network as a tensor network has many interesting implications. Most notably, it enables non-linear kernel learning with a cost that scales only linear with training set size for optimization, and is independent of the training set size for evaluation, while still using a very expressive feature map (the dimension of feature space scales exponetially with the size of the input space). It could also prove extremely useful for issues of interpretability, since tensor networks are composed of linear operations only.
There is also much room to improve our optimization algorithm through standard approaches known from machine learning, such as batch-learning or adaptive learning rates. Another simple ways of improving the algorithm would be implementing a conjugate gradient descent update instead of our gradient descent inspired update of the bond tensor B^l. It would be very interesting to investigate the possibility to apply unsupervised learning techniques to initialize the tensor network.
We are convinced there is great potential in investigating the power of tensor networks for machine learning tasks and vice-versa exploring the use of neural networks in physical applications in the future.

A Code

This is the main code used in this thesis. It starts off by initializing the different input values, converting them for the compiler and setting up different parameters like file names etc.
Then it initializes the MPS and the feature maps for both training and test data before implementing the right and left sweeps for optimization. Finally it contracts the MPS to classify the test and training data and check the error rates.

```
function [mps,spec,dev,trainerror,testerror] = mps_mnist(Ntrain,DB,Nsweep,alpha,
    Ntest)
%<Description>
%
%<Input>
% Ntrain: [scalar] Number of training examples (max 60.000)
% DB: [scalar] bond dimension m (max m= 120)
% Nsweep: [scalar] number of (rl—lr) sweeps
% alpha: [scalar] empirical step size parameter the gradient update
% Ntest: [scalar] number of test examples for verification (max 10.000)
%
%<Output>
% mps: [cell]  minimized matrix product state
% spec: [cell] svd spectra for the 2nd and 100th site of the mps
% dev: [vector] (1xNsweep) reduction of global cost in respective sweep
% trainerror: [vector] classification error for training data
% testerror: [vector] classification error for test data
%
% last edited D.Maier Jul6,2017


%% converting input for compiler
if isdeployed %take care of command line arguments
    if ischar(Ntrain), Ntrain = str2num(Ntrain); end
    if ischar(DB), DB = str2num(DB); end
    if ischar(Nsweep), Nsweep = str2num(Nsweep); end
    if ischar(alpha), alpha = str2num(alpha); end
    if ischar(Ntest), Ntest = str2num(Ntest); end
end

%% reading MNIST dataset
disptime('reading MNIST dataset');
train = csvread('mnist_train.csv', 0, 0);   % read train.csv
test = csvread('mnist_test.csv', 0, 0);     % read test.csv

% downscaling to 14x14
disptime('downscaling dataset');
tr = train;
te = test;
train = zeros(size(tr,1),197);
test = zeros(size(te,1),197);
train(:,1) = tr(:,1);
test(:,1) = te(:,1);
%resizeing training pictures
for i = 1:size(tr,1)
    ibig = reshape(tr(i, 2:end), [28,28]);
```

```
46      ismall = imresize(ibig,1/2);
47      ismall = reshape(ismall,[1,196]);
48      train(i,2:end) = ismall;
49  end
50  %resizeing test oictures
51  for i = 1:size(te,1)
52      ibig = reshape(te(i, 2:end), [28,28]);
53      ismall = imresize(ibig,1/2);
54      ismall = reshape(ismall,[1,196]);
55      test(i,2:end) = ismall;
56  end
57
58
59  %% Setting inputs
60  train = train(1:Ntrain,:); %cutting train and test examples to size
61  test = test(1:Ntest,:);
62  imglen = 14;                %defining image length do the downscaled 14x14 pixels
63  spec = cell(2,Nsweep*2);    %initializing cell for the two spectra
64  trainerror = ones(1,Nsweep); %initializing vectors for errors of with training
        and test data
65  testerror = ones(1,Nsweep);
66
67  D = DB;         %bond dimension m , paper: max m= 120, truncate to fixed
        dimension
68  d = 2;           % dimension of feature map
69
70  % feature map
71  ftype = 'Normal';
72
73  % setting labels
74  labels = [0,1,2,3,4,5,6,7,8,9];
75  l = size(labels,2);
76  % ouputting parameters for easier identification in the log file
77  parameters={'Ntrain',Ntrain;'DB',DB;'Nsweep',Nsweep;'alpha',alpha;'Ntest',Ntest;
        'imglen',imglen};
78  parameters
79
80  %% Setting folder name etc.
81  folder=sprintf('results/mps_mnist/alphafinal/Ntrain%iDB%iNsweep%ialpha%.gNtest%i
        ',Ntrain,DB,Nsweep,alpha,Ntest);
82  filename=strcat(folder,'/results.mat');
83  if ~exist(filename,'file')
84      mkdir(folder);
85      savedexist=0;
86  else
87      savedexist=1;
88  end
89
90
91
92  %% Beginning of the actual algorithm
93
94  %% setting up MPS
95  disptime('Initializing with random MPS');
```

```
96  mps = randommps_lN(imglen^2,D,d,l); %initializing MPS with index l on last site
        N
97  mps = prepareM(mps,DB,'twosite_L'); %preparing MPS to be left-normalized with l
        still on last site
98  for i = 1:imglen^2-1
99      [mps{i},mps{i+1}]=prepare_twosite_L(mps,i,DB,'lr');
100 end
101
102 %% calculating feature map
103 disptime('Calculating feature map for training'); %mapping training inputs
104 fm = cell(Ntrain,imglen^2);
105 for i = 1:Ntrain
106     for j= 1:imglen^2
107         fm{i,j} = featuremap2(ftype,d,train(i,j+1));
108     end
109 end
110 disptime('Calculating feature map for testing'); %mapping test inputs
111 ft = cell(Ntest,imglen^2);
112 for i = 1:Ntest
113     for j= 1:imglen^2
114         ft{i,j} = featuremap2(ftype,d,test(i,j+1)); %j+1 beause first entry is
                label
115     end
116 end
117
118
119 %% initializing data storage  from left >> right
120 disptime('Initializing data storage -->>');
121 data = cell(Ntrain,imglen^2);
122 for i = 1:Ntrain
123     data{i,1} = contract(fm{i,1},2,2,mps{1,1},3,3); %contracting first entry
            [1,1,D]
124     data{i,1} = permute(data{i,1},[1,3,2]);     % [1,D]
125     data{i,1} = data{i,1}/max(abs(data{i,1}(:))); %normalize through max
126 end
127 %initializing the rest of the chain (except last 2, they are not needed )
128 for j = 2:(imglen^2 - 2)
129     for i = 1:Ntrain
130         data{i,j} = updateData(mps,fm,data,i,j-1,'lr');
131     end
132 end
133
134 %% Sweeps
135 %initializing cost starage with some arbitrary value
136 cstore = cell(size(mps));
137 for i = 1:size(cstore,2)
138     cstore{1,i} = 100000;
139 end
140
141 dev = zeros(1,Nsweep); % initializing vector tracking developement of cost over
        sweeps
142
143 for itS = (1:Nsweep)
144     costold = cstore; %backing up old cost
```

```
145     % right sweep (right->left)
146     disptime(['rl <<--- sweep #',sprintf('%i/%i',[itS Nsweep])]);
147     for j = imglen^2:-1:2
148     %updating mps locally
149     [mps{1,j},mps{j-1},cstore,spec] = updateBSVD(mps,j,DB, 'rl',fm,cstore,train,
            data,spec,itS,alpha);    % initializing last entry for sweep back
150     if j == imglen^2
151         for i = 1:Ntrain
152             data{i,imglen^2} = contract(fm{i,imglen^2},2,2,mps{1,imglen^2},3,3);
                    %contracting first entry [1,1,D]
153             data{i,imglen^2} = permute(data{i,imglen^2},[1,3,2]);     % [1,1,D]
154             data{i,imglen^2} = squeeze(data{i,imglen^2});                %[1,D]
155             data{i,imglen^2} = data{i,imglen^2}/max(abs(data{i,imglen^2}(:))); %
                    normalize
156         end
157     %updating data
158     else
159         for i = 1:Ntrain
160             data{i,j} = updateData(mps,fm,data,i,j+1,'rl');
161
162         end
163     end
164     end
165
166     % left sweep (left->right)
167     disptime(['lr --->> sweep #',sprintf('%i/%i',[itS Nsweep])]);
168
169     for j = 1:imglen^2-1
170
171         [mps{1,j},mps{j+1},cstore,spec] = updateBSVD(mps,j,DB, 'lr',fm,cstore,
                train,data,spec,itS,alpha);
172         %initializing first entry
173         if j == 1
174             for i = 1:Ntrain
175                 data{i,1} = contract(fm{i,1},2,2,mps{1,1},3,3); %contracting
                        first entry [1,1,D]
176                 data{i,1} = permute(data{i,1},[1,3,2]);     % [1,D]
177                 data{i,1} = data{i,1}/max(abs(data{i,1}(:))); %normalize through
                        max
178             end
179             %updating data
180         else
181             for i = 1:Ntrain
182                 data{i,j} = updateData(mps,fm,data,i,j-1,'lr');
183
184             end
185         end
186     end
187     %evaluating cost difference
188     cd = 0;
189     for i = 1:size(cstore,2)
190         cd = cd + (costold{1,i}-cstore{1,i});
191     end
192     dev(1,itS) = cd;
```

```
193     disptime(['In sweep #', sprintf('%i/%i',[itS Nsweep]),' the total cost has
            been reduced by:',sprintf('%i',cd)]);
194
195 %% contracting MPS with test data for check of error rates
196
197 disptime('contracting MPS with test data-->>');
198 cont = cell(Ntest,imglen^2);
199 for i = 1:Ntest
200     cont{i,1} = contract(ft{i,1},2,2,mps{1,1},3,3); %contracting first entry
            [1,1,D]
201     cont{i,1} = squeeze(cont{i,1});     % [D,1]
202     cont{i,1} = permute(cont{i,1},[2,1]); %[1,D]
203     cont{i,1} = cont{i,1}/max(abs(cont{i,1}(:)));
204 end
205 for i = 1:Ntest
206     for j = 2:imglen^2-1
207         cont{i,j}= updateData(mps,ft,cont,i,j-1,'lr');
208
209     end
210 end
211 for i = 1:Ntest
212     last = contract(ft{i,imglen^2},2,2,mps{1,imglen^2},4,3); %contracting last
            entry [1,D,1,l]
213     cont{i,imglen^2} = contract(cont{i,imglen^2-1},2,2,last,4,2); % [1,1,l]
214     cont{i,1} = cont{i,imglen^2};
215     cont{i,1} = squeeze(cont{i,1});
216     cont{i,1} = cont{i,1}/max(abs(cont{i,1}(:)));
217 end
218
219
220 %% comparing with test data
221 disptime('comparing results to correct answers');
222 correct = test(:,1)+1;
223 results = zeros(Ntest,1);
224 for i = 1:Ntest
225     results(i,1) = find(cont{i,1} == max(cont{i,1}));
226 end
227 answers = correct ==results;
228 testerror(1,itS) = 1-sum(answers)/Ntest;
229 disptime(['Test examples correctly classified:  ',sprintf('%i/%i',[sum(answers)
        Ntest]), ' trained with: ',sprintf('%i',size(train,1)),' training examples.'
        ]);
230
231 %% comparing with training results
232 correct2 = train(:,1)+1;
233 B = contract(mps{imglen^2-1},3,2,mps{imglen^2},4,1); % size(B) = [D1,d,1,d,l]
234 B = permute(B,[1,3,2,4,5]);% [D1,1,d,d,l]
235
236 f = cell(Ntrain,1);
237 for i = 1:Ntrain
238     f{i} = B;
239     f{i} = squeeze(f{i});                       %squeeze to [D,d,d,l]
240     f{i} = contract(f{i},4,1,data{i,imglen^2-2},2,2);     %contract with left
            wing
```

```
241     f{i} = squeeze(f{i});                          %squeeze to [d,d,l]
242     f{i} = contract(f{i},3,1,fm{i,imglen^2},2,2);     %contract next bottom leg
243     f{i} = squeeze(f{i});                          %squeeze to [d,l]
244     f{i} = contract(f{i},2,1,fm{i,imglen^2-1},2,2);     %contract last bottom
            leg ->[l,1]
245 end
246
247 results2 = zeros(Ntrain,1);
248 for i = 1:Ntrain
249     results2(i,1) = find(f{i,1} == max(f{i,1}));
250 end
251 answers2 = correct2 ==results2;
252 trainerror(1,itS) = 1-sum(answers2)/Ntrain;
253 disptime(['Training examples correctly classified:  ',sprintf('%i/%i',[sum(
        answers2) Ntrain])]);
254
255 %saving results [mps,spec;dev,trainerror,testerror]
256 save(filename,'mps','spec','dev','trainerror','testerror','parameters');
257 end
```

A.1 Update function

```
1  function [A,R,cstore,spec] = updateBSVD(mps,id,DB, direction,fm,cstore,train,
       data,spec,itS,alpha)
2  %<Description>
3  %
4  % function [A,R,cstore,spec] = updateBSVD(mps,id,DB, direction,fm,cstore,train,
       data,spec)
5  %
6  %<Input>
7  % mps: [array] the entire mps
8  % id: [scalar] index of tensor to be updated
9  % DB : [scalar] bond dimension for truncation
10 % direction : [char array] Must be either 'lr' or 'rl'. Determines the
11 %       direction for the canonical form.
12 %fm: [array] mapped input data
13 %cstore: [array] cost storage
14 %train: training data (this is given to cost.m as the correct labels)
15 %data: [array] storage of left and right 'blocks' (this is also handed to cost.m
       )
16 %spec: [array] cell with svd spetra
17 %itS: [scalar] iteration of sweeps (to put spectrum into correct place in spec)
18 %alpha: [scalar] empirical step size for update
19 %
20 % < Output >
21 % A : [tensor] The canonical form of input tensor M.
22 % R : [tensor] Tensor to be transformed next. (indec l is moved)
23 %cstore: [array] cost storage with minimized cost
24 %spec: [array] cell with svd spetra
25 %Written by D.Maier (Jun08,2017) edited Jun26,2017
26
27 switch direction
28     case 'lr'
```

```
29          M = mps{id};
30          N = mps{id+1};          % M    |l           N
31          [D1,~,d,l]= size(M);    % D1 – o – D2   D3 –o – D4
32          [~,D4,~]= size(N);      %      |d           |d
33
34
35          %form two-site tensor B
36          B = contract(M,4,2,N,3,1); % size(B) = [D1,d,l,D4,d]
37          B = permute(B,[1,4,2,5,3]);% [D1,D4,d,d,l]
38
39          %% update here
40          costold = cstore{1,id};
41          [cstore{1,id}, delB] = cost(B,data,fm,id,'lr',train(:,1));
42          if cstore{1,id} < costold %only update if cost is lowered
43              B = B + alpha*delB;
44          else
45              cstore{1,id} = costold;
46          end
47
48          %% svd and truncation
49          B = permute(B,[1,3,2,4,5]);% size(B) = [D1,d,D4,d,l]
50          B = reshape(B, [D1*d,D4*d*l]);
51          if isempty(DB)      %no DB input, no truncation
52              [A,S,R]=svd(B,'econ');
53              ds = size(S,1);
54
55              R = R';
56              S = S/trace(S*S);
57              if id == 2  % putting svd spectrum into cell for later plotting
58                  spec{1,2*itS-1} = diag(S);
59              elseif id == 100
60                  spec{2,2*itS-1} = diag(S);
61              end
62              %reshape to final form
63                  A = reshape(A,[D1,d,ds]);      %D1,d = o – ds
64                  A = permute(A,[1,3,2]);        %   -> (|d)
65                  R = S*R;            %[ds,D4*d*l]
66                  R = reshape(R,[ds,D4,d,l]);
67
68          else    %truncation
69              [A,S,R]=svd(B,'econ');
70              R = R';
71              ds = size(S,1);
72              S = S/trace(S*S);
73
74              if id == 2  % putting svd spectrum into cell for later plotting
75                  spec{1,2*itS-1} = diag(S);
76              elseif id == 100
77                  spec{2,2*itS-1} = diag(S);
78              end
79
80              if ds <=DB %no truncation
81                  %reshape to final form
82                  A = reshape(A,[D1,d,ds]);      %D1,d = o – ds
```

```
83                  A = permute(A,[1,3,2]);         %   -> (|d)
84                  R = S*R;            %[ds,D4*d*l]
85                  R = reshape(R,[ds,D4,d,l]);
86
87              else %truncation
88                  A = A(:,1:DB);      %[D1*d,DB]
89                  S = S(1:DB,1:DB);
90                  R = R(1:DB,:);
91                  %reshape to final form
92                  A = reshape(A,[D1,d,DB]);       %D1,d = o - DB
93                  A = permute(A,[1,3,2]);         %   -> (|d)
94                  R = S*R;            %[DB,D4*d*l]
95                  R = reshape(R,[DB,D4,d,l]);
96              end
97          end
98
99      case 'rl'
100         M = mps{id};
101         N = mps{id-1};          %      N            |l   M
102         [~,D2,d,l]= size(M);    % D3 - o - D4   D1 -o - D2
103         [D3,~,~]= size(N);      %      |d           |d
104         B = contract(N,3,2,M,4,1); % size(B) = [D3,d,D2,d,l]
105         B = permute(B, [1,3,2,4,5]); %[D3,D2,d,d,l]
106
107         %% update here
108         costold = cstore{1,id};
109         [cstore{1,id}, delB] = cost(B,data,fm,id,'rl',train(:,1));
110         if cstore{1,id} < costold %only update if cost is lowered locally
111             B = B + alpha*delB;
112         else
113             cstore{1,id} = costold;
114         end
115
116         %% svd and truncation
117         B = permute(B, [1,3,5,2,4]); %[D3,d,l,D2,d]
118         B = reshape(B, [D3*d*l,D2*d]);
119         if isempty(DB)      %no DB input, no truncation
120             [R,S,A]=svd(B,'econ');
121             A = A';
122             S = S/trace(S*S);
123             if id == 2  % putting svd spectrum into cell for later plotting
124                 spec{1,2*itS} = diag(S);
125             elseif id == 100
126                 spec{2,2*itS} = diag(S);
127             end
128             DB = size(S,1);
129             %reshape to final form
130             A = reshape(A,[DB,D2,d]);
131             R = R*S;
132             R = reshape(R,[D3,d,l,DB]);
133             R = permute(R,[1,4,2,3]);
134
135         else    %truncation
136             [R,S,A]=svd(B,'econ');
```

```
            A = A';

            ds = size(S,1);
            S = S/trace(S*S);

            if id == 2  % putting svd spectrum into cell for later plotting
                spec{1,2*itS} = diag(S);
            elseif id == 100
                spec{2,2*itS} = diag(S);
            end

            if ds <=DB %no truncation
                %reshape to final form
                A = reshape(A,[ds,D2,d]);          % ds -o= D2*d
                R = R*S;
                R = reshape(R,[D3,d,l,ds]);
                R = permute(R,[1,4,2,3]);

            else %truncation
                R = R(:,1:DB);      %[D1*d,DB]
                S = S(1:DB,1:DB);
                A = A(1:DB,:);
                %reshape to final form
                A = reshape(A,[DB,D2,d]);
                R = R*S;
                R = reshape(R,[D3,d,l,DB]);
                R = permute(R,[1,4,2,3]);
            end
        end

    otherwise
        error('ERR: ''direction'' should be either ''lr'' or ''rl''.');
end
```

A.2 Cost function

```
function [cost,grad] = cost(B,data,fm,id,direction,label)
%<Description>
% cost function
%<Inputs>
% B: [tensor] rank 5 tensor being optimized (D,D,d,d,l)
% labels: [cell array]: Ntrain x 1 vector with correct labels
% data: [cell array] with left right blocks (Ntrain x #sites)
% fm: [cell array]
% id: [scalar] index of site being optimized
% direction: [string] direction we are going

%<Outputs>
% cost: [scalar]
% grad: [tensor] to update B
%written by D.Maier(Jun11,2017)

[Ntrain,N] = size(data);
f = cell(Ntrain,1);
```

```
del = cell(Ntrain,1);
cost = 0;
grad = zeros(size(B));
[D1,D2,d,~,~] = size(B);

switch direction
    case'lr'
        for i = 1:Ntrain
            if id == N-1
                f{i} = B;
            else
                f{i} = contract(B,5,2,data{i,id+2},2,1);        %contract with
                    right wing
            end
            f{i} = squeeze(f{i});                       %squeeze to [D,d,d,l]

            if id ~= 1
                f{i} = contract(f{i},4,1,data{i,id-1},2,2);     %contract with
                    left wing
                f{i} = squeeze(f{i});                       %squeeze to [d,d,l]
            end
                f{i} = contract(f{i},3,1,fm{i,id+1},2,2);     %contract next
                    bottom leg
                f{i} = squeeze(f{i});                       %squeeze to [d,l]
                f{i} = contract(f{i},2,1,fm{i,id},2,2);     %contract last
                    bottom leg ->[l,1]

                del{i,1} = zeros(10,1);

                del{i,1}(label(i,1)+1,1) = 1;               %Kronecker l,L_n  +1
                     to map 0 to 1...
                del{i} = (f{i} - del{i});                  %difference [10,1]
                cost = cost + 0.5*del{i}'*del{i};           %calculate total
                    cost

                %% calculate gradient
                % del[10,1], data{left}[1xD1], data{right}[D2x1] ,fm[1xd]

                if id == 1 %leftmost entry , left bit is 1
                    a = mkron(-del{i},data{i,id+2},fm{i,id+1},fm{i,id}); %[10*D2
                        ,d*d]
                    a = reshape(a,[1,10,D2,d,d]);
                    a = permute(a,[1,3,4,5,2]); %a [1,D,d,d,l]
                elseif id == N-1 %rightmost entry , right bit is 1
                    a = mkron(-del{i},data{i,id-1},fm{i,id+1},fm{i,id}); %[10,D1
                        *d*d]
                    a = reshape(a,[10,D1,1,d,d]);
                    a = permute(a,[2,3,4,5,1]);
                else %all others
                    a = mkron(-del{i},data{i,id+2},data{i,id-1},fm{i,id+1},fm{i,
                        id}); %[10*D2,D1*d*d]
                    a = reshape(a,[10,D2,D1,d,d]);
                    a = permute(a,[3,2,4,5,1]);
                end
```

```
64
65                  grad = grad + a;
66          end
67
68      case 'rl'
69          for i = 1:Ntrain
70              if id == N
71                  f{i} = B;
72              else
73                  f{i} = contract(B,5,2,data{i,id+1},2,1);        %contract with
                        right wing
74              end
75              f{i} = squeeze(f{i});                       %squeeze to [D,d,d,l]
76              if id ~= 2
77                  f{i} = contract(f{i},4,1,data{i,id-2},2,2);     %contract with
                        left wing
78                  f{i} = squeeze(f{i});                       %squeeze to [d,d,l]
79              end
80              f{i} = contract(f{i},3,1,fm{i,id},2,2);     %contract next bottom
                    leg
81              f{i} = squeeze(f{i});                       %squeeze to [d,l]
82              f{i} = contract(f{i},2,1,fm{i,id-1},2,2);     %contract last bottom
                    leg ->[l,1]
83
84              del{i,1} = zeros(1,10);
85              del{i,1}(1,label(i,1)+1) = 1;               %Kronecker l,L_n  +1 to
                    map 0 to 1...
86              del{i} = (f{i} - del{i}');                  %difference [10,1]
87              cost = cost + 0.5*del{i}'*del{i};           %calculate total cost
88
89              %% calculating gradient
90
91              if id == 2 %leftmost entry , left bit is 1
92                  a = mkron(-del{i},data{i,id+1},fm{i,id},fm{i,id-1}); %[10*D2,d*d
                        ]
93                  a = reshape(a,[1,10,D2,d,d]);
94                  a = permute(a,[1,3,4,5,2]); %a [1,D2,d,d,l]
95              elseif id == N %rightmost entry , right bit is 1
96                  a = mkron(-del{i},data{i,id-2},fm{i,id},fm{i,id-1}); %[10,D1*d*d
                        ]
97                  a = reshape(a,[10,D1,1,d,d]);
98                  a = permute(a,[2,3,4,5,1]);
99              else %all others
100                 a = mkron(-del{i},data{i,id+1},data{i,id-2},fm{i,id},fm{i,id-1})
                        ; %[10*D2,D1*d*d]
101                 a = reshape(a,[10,D2,D1,d,d]);
102                 a = permute(a,[3,2,4,5,1]);
103             end
104
105             grad = grad + a;
106         end
107     otherwise
108         error('ERR: ''direction'' should be either ''lr'' or ''rl''.');
109 end
```

A.3 Data block update function

```
function UD = updateData(mps,fm,data,i,j,direction)
%<Description>
% function that updates data blocks
% [left-o-]-o-o-[-o-right]
% -> [left]-o-o-[-o-o-right] ('rl') or [left-o-o-]-o-o-[-right] ('lr')
%
%<Input>
% mps: [cell array]
% fm : [vector] to contract physical index of mps
% data [cell array]
% i,j: [scalar] index of the data cell to be updated
% direction:  [string] direction the block is growing
%
%<Output>
% UD : [cell] updated data cell
%
%Written by D.Maier(Jun13,2017)

DC = data{i,j}; % [1,D] or [D,1]
% (DC : [cell] data cell to be updated)


switch direction
    case 'lr'
        next = contract(fm{i,j+1},2,2,mps{1,j+1},3,3);  %[1,D,D]
        next = permute(next,[2,3,1]);                   %[D,D]
        UD = contract(DC,2,2,next,2,1);                 %[1,D]
        UD = UD/max(abs(UD(:)));                        %normalization
    case 'rl'
        next = contract(fm{i,j-1},2,2,mps{1,j-1},3,3);  %[1,D,D]
        next = permute(next,[2,3,1]);                   %[D,D]
        UD = contract(next,2,2,DC,2,1);                 %[1,D]
        UD = UD/max(abs(UD(:)));                        %normalization
    otherwise
        error('ERR: ''direction'' should be either ''lr'' or ''rl''.');
end
```

A.4 Additional functions

Additional functions like *contract.m*, to contract two tensors, or $featuremap2.m$, to compute the feature map (37), were used but are not appended here as they do not contain any important information for the algorithm and were just used for convenience.

References

[1] M. Rupp, A. Tkatchenko, K.-R. Müller, and O. A. von Lilienfeld, "Fast and Accurate Modeling of Molecular Atomization Energies with Machine Learning," *Phys. Rev. Lett.*, vol. 108, p. 058301, Jan. 2012.

[2] J. C. Snyder, M. Rupp, K. Hansen, K.-R. Müller, and K. Burke, "Finding Density Functionals with Machine Learning," *Phys. Rev. Lett.*, vol. 108, p. 253002, June 2012.

[3] G. Pilania, K. R. Whittle, C. Jiang, R. W. Grimes, C. R. Stanek, K. E. Sickafus, and B. P. Uberuaga, "Using machine learning to identify factors that govern amorphization of irradiated pyrochlores," July 2016.

[4] L.-F. Arsenault, A. Lopez-Bezanilla, O. A. von Lilienfeld, and A. J. Millis, "Machine learning for many-body physics: The case of the Anderson impurity model," *Phys. Rev. B 90, 155136 (2014)*, Nov. 2014.

[5] M. H. Amin, E. Andriyash, J. Rolfe, B. Kulchytskyy, and R. Melko, "Quantum Boltzmann Machine," Jan. 2016.

[6] J. Carrasquilla and R. G. Melko, "Machine learning phases of matter," May 2016.

[7] P. Mehta and D. J. Schwab, "An exact mapping between the Variational Renormalization Group and Deep Learning," Oct. 2014.

[8] Y. Levine, D. Yakira, N. Cohen, and A. Shashua, "Deep Learning and Quantum Entanglement: Fundamental Connections with Implications to Network Design.," *CoRR*, vol. abs/1704.01552, 2017.

[9] J. Chen, S. Cheng, H. Xie, L. Wang, and T. Xiang, "On the Equivalence of Restricted Boltzmann Machines and Tensor Network States," Jan. 2017.

[10] G. Carleo and M. Troyer, "Solving the Quantum Many-Body Problem with Artificial Neural Networks," *Science 355, 602 (2017)*, June 2016.

[11] E. M. Stoudenmire and D. J. Schwab, "Supervised Learning with Quantum-Inspired Tensor Networks," *Advances in Neural Information Processing Systems 29, 4799 (2016)*, May 2017.

[12] S. S. Haykin, *Neural networks and learning machines*. Upper Saddle River, NJ: Pearson Education, third ed., 2009.

[13] F. Rosenblatt, *Principles of Neurodynamics*. Spartan, New York, 1962.

[14] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *Bulletin of Mathematical Biophysics*, vol. 5, pp. 115–133, 1943.

[15] M. Nielsen, "Neural Networks and Deep Learning." `http://neuralnetworksanddeeplearning.com`, 2017. Accessed: 2017-06-30.

[16] D. H. Ackley, G. E. Hinton, and T. J. Sejnowski, "A Learning Algorithm for Boltzmann Machines," *Cognitive Science*, vol. 9, pp. 147–169, 1985.

[17] N. L. Roux and Y. Bengio, "Representational Power of Restricted Boltzmann Machines and Deep Belief Networks.," *Neural Computation*, vol. 20, no. 6, pp. 1631–1649, 2008.

[18] A. Cichocki, "Tensor Networks for Big Data Analytics and Large-Scale Optimization Problems," Aug. 2014.

[19] A. Novikov, M. Trofimov, and I. Oseledets, "Exponential Machines," Nov. 2016.

[20] R. T. J. F. Trevor Hastie, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* Springer Series in Statistics, Springer, 2 ed., 2013.

[21] G. Hinton, "A practical guide to training restricted Boltzmann machines," *Momentum*, vol. 9, no. 1, p. 926, 2010.

[22] R. Orus, "A Practical Introduction to Tensor Networks: Matrix Product States and Projected Entangled Pair States," *Annals of Physics 349 (2014) 117-158*, June 2014.

[23] G. Evenbly and G. Vidal, "Tensor network states and geometry," *J Stat Phys (2011) 145:891-918*, June 2011.

[24] J. C. Bridgeman and C. T. Chubb, "Hand-waving and Interpretive Dance: An Introductory Course on Tensor Networks," *J. Phys. A: Math. Theor. 50 223001 (2017)*, May 2017.

[25] F. Verstraete and J. I. Cirac, "Renormalization algorithms for Quantum-Many Body Systems in two and higher dimensions," July 2004.

[26] G. Vidal, "Class of Quantum Many-Body States That Can Be Efficiently Simulated," *Phys. Rev. Lett.*, vol. 101, p. 110501, Sep 2008.

[27] U. Schollwoeck, "The density-matrix renormalization group in the age of matrix product states," *Annals of Physics 326, 96 (2011)*, Jan. 2011.

[28] S. R. White, "Density matrix formulation for quantum renormalization groups," *Phys. Rev. Lett.*, vol. 69, pp. 2863–2866, Nov 1992.

[29] S. R. White, "Density-matrix algorithms for quantum renormalization groups," *Phys. Rev. B*, vol. 48, pp. 10345–10356, Oct 1993.

[30] F. Verstraete, D. Porras, and J. I. Cirac, "DMRG and periodic boundary conditions: a quantum information perspective," *Phys. Rev. Lett. 93, 227205 (2004)*, Apr. 2004.

[31] P. J. Grother, "NIST Special Database 19 Handprinted Forms and Characters Database," 1995.

[32] E. M. Stoudenmire and S. R. White, "Studying Two Dimensional Systems With the Density Matrix Renormalization Group," *Annual Review of Condensed Matter Physics, 3: 111-128 (2012)*, Aug. 2011.

Erklärung

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfasst zu haben und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel benutzt zu haben.

München, 26.07.2017

David Maier