Tensor networks as unsupervised generative models

by

Jonas Van Gompel

Promotor	Prof. Dr. Jan Ryckebusch
Co-promotor	Prof. Dr. Jutho Haegeman
Supervisor	Corneel Casert
Supervisor	Tom Vieijra

Thesis submitted in partial fulfillment for the degree of
Master in Physics and Astronomy

GHENT UNIVERSITY
Faculty of Sciences
Department of Physics and Astronomy

4th June 2020

Declaration of Authorship

The author grants permission to make this master's thesis available for consultation and to copy parts of the master's thesis for personal use. Any other use falls under the limitations of copyright, in particular with regard to the obligation to explicitly mention the source when citing results from this master's thesis.

Date: 4th June 2020

Abstract

Machine learning has become an increasingly valuable tool in many areas of modern technology and science, including physics. Fields such as experimental particle physics, condensed matter physics, observational astronomy and cosmology have seen a growing number of machine learning applications. In parallel, concepts and methods from statistical and quantum many-body physics have inspired numerous theoretical and practical machine learning techniques.

This thesis will focus on the modelling of probability distributions in an exponentially large configuration space. This is a core problem in both quantum many-body physics and generative modelling, which is a subject related to applied statistics and machine learning. The goal of unsupervised generative modelling is to obtain a probability distribution that generates samples similar to observed samples in a dataset. Samples can refer to many types of data, such as images of faces or English sentences. Typically only a tiny fraction of the exponentially many possible samples are sensible. For instance, the number of possible images grows exponentially with the amount of pixels in the image. However, only a tiny fraction of these possibilities are images of interest, as most combinations of pixel values look like noise. This bears striking resemblance to the description of quantum many-body states, where the physically relevant states live on a small submanifold in their exponentially large Hilbert space. Tensor networks have proven to be an efficient representation of the physically relevant submanifold. This suggests that tensor networks could provide a powerful and well-studied framework for generative modelling. In particular, we will study the use of matrix product states as generative models due to their theoretical and computational simplicity. Matrix product states have revolutionized our understanding of one-dimensional quantum systems, partly thanks to a powerful optimization algorithm called density matrix renormalization group.

In chapters 1, 2 and 3, the necessary concepts and tools of respectively tensor networks, machine learning and generative modelling will be introduced. Chapter 4 discusses how matrix product states can be trained as generative models using an algorithm based on density matrix renormalization group. The sampling procedure described in this thesis allows efficient generation or reconstruction of samples directly from the modelled probability distribution. We also dedicate a section of chapter 3 to generative adversarial networks, which are considered the current state-of-the-art for image processing. These generative models are trained via a minimax game, as opposed to the more common maximum likelihood objective. This is possibly advantageous as maximizing likelihood does not correspond to minimizing a proper metric of distance between the model and data distribution. Therefore we attempt to extend the alternative training method to matrix product states in chapter 5. Furthermore, modifications of the optimization algorithm, such as adaptive learning rate and stochastic gradient descent, are examined and compared to the performance of tree tensor networks. Finally, we designed and analyzed a one-dimensional, discrete toy dataset based on sine waves, as the one-dimensional geometry of the matrix product state formulation is less suitable for two-dimensional data such as images.

Acknowledgements

First of all, I would like to thank Corneel and Tom for the invaluable guidance and feedback during this work. My sincerest gratitude also goes out to professor Ryckebusch for taking the time to give incredibly helpful and detailed feedback. Furthermore, thank you to professor Haegeman, professor Verstraete and dr. Nys for the suggestions and constructive criticism. Without the knowledge and enthusiasm of all aforementioned people, this thesis would not have been possible.

Fellow students, thank you for the many amicable lunches during our studies and the video chats during the pandemic. Suffering alone would have been awfully boring.

Finally, a warm thank you to the people close to me for the unwavering support and love.

Contents

Chapter 1

Tensor Networks

1.1 Quantum many-body systems

Representing and investigating properties of quantum many-body systems is a central problem in condensed matter physics. Subjects such as superconductivity, quantum computation and molecular modelling are at their core all quantum many-body problems [1]. These subjects are of great interest for both science and industry, which necessitates the accurate description of quantum many-body systems. This has proven to be a challenging task, mainly because of the high-dimensional Hilbert space of the quantum mechanical systems involved. In quantum mechanic, interacting systems can be in a superposition, which means that the Hilbert space of the composite is the tensor product of the Hilbert space of each constituent [2]. For example, the Hilbert space of N particles with p degrees of freedom has a dimension of p^N. To put this into perspective, the dimension of the Hilbert space of 300 spin $\frac{1}{2}$ particles fixed on a lattice is $2^{300} \approx 10^{90}$, which is already larger than the estimated number of atoms in the observable universe, namely 10^{80} [3].

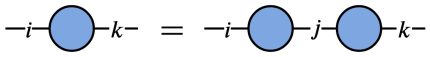
It follows that, generally speaking, exponentially many variables are required to describe an N-particle system. Fortunately, most physical systems live on a small manifold of the Hilbert space due to additional structure in their Hamiltonian, such as the local nature of physical interactions [4]. For instance, evolving a quantum N-body state for a time polynomial in N with a local Hamiltonian can merely reach an exponentially small volume in the Hilbert space [5]. This suggests that most states in the Hilbert space are not physical as they can only be reached after a time $t \sim \mathcal{O}(e^N)$. Consequently, an efficient parametrization of the physically relevant manifold could describe a quantum N-body system using a number of variables that grows polynomially with N as opposed to exponentially. Tensor networks are a successful example of such a parametrization [2]. We will now introduce some concepts of tensor networks and their notation which will be used later. For a more comprehensive introduction to tensor networks, we refer to Ref. [6].

1.2 Tensor diagram notation

For our purposes, an index contraction is the sum over all possible values of a repeated index of a set of tensors. A simple example of a contraction is the matrix product

$$C_{ik} = \sum_j A_{ij} B_{jk} \,. \tag{1.1}$$

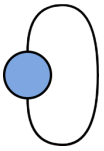
Tensor diagram notation can be used to graphically represent the matrix product as



. (1.2)

In tensor diagram notation, each index of a tensor is represented by a line and an index shared by two tensors denotes a contraction. In Eq. (1.2) the index j is contracted. Tensor diagram notation can be seen as an extension of the Einstein summation convention. The rank of a tensor is determined by its number of indices, or equivalently the number of lines connected to the tensor. Tensors with 0, 1, 2, and 3 connected lines are respectively scalars, vectors, matrices and rank-3 tensors. The names of the tensors and indices are often not denoted in tensor diagram notation. A trace of a matrix can be written as

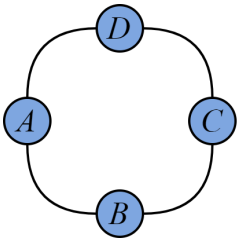
$$\mathrm{Tr}A = \sum_i A_{ii} =$$



. (1.3)

For a product of tensors, the trace becomes

$$\mathrm{Tr}(ABCD) =$$



$$= \mathrm{Tr}(DABC) = \mathrm{Tr}(CDAB) = \mathrm{Tr}(BCDA)\,. \tag{1.4}$$

Note that the invariance of the trace under cyclic permutations is immediately apparent in the tensor diagram notation.

A tensor network is a set of tensors which are contracted according to a diagram to form a single composite tensor. The result is a scalar if there are no unconnected indices in the tensor network. The structure of the contraction of the constituent tensors is readily apparent by using tensor diagram notation. A high-dimensional tensor T can be rewritten exactly as a tensor network, some examples of which are shown in Fig. 1.1. For instance, T could contain the coefficients of the wave function of a quantum many body system with respect to an orthonormal basis

$$|\Psi\rangle = \sum_{j_1 j_2 \cdots j_N} T_{j_1 j_2 \ldots j_N} \, |j_1\rangle \otimes |j_2\rangle \otimes \cdots \otimes |j_N\rangle \,. \tag{1.5}$$

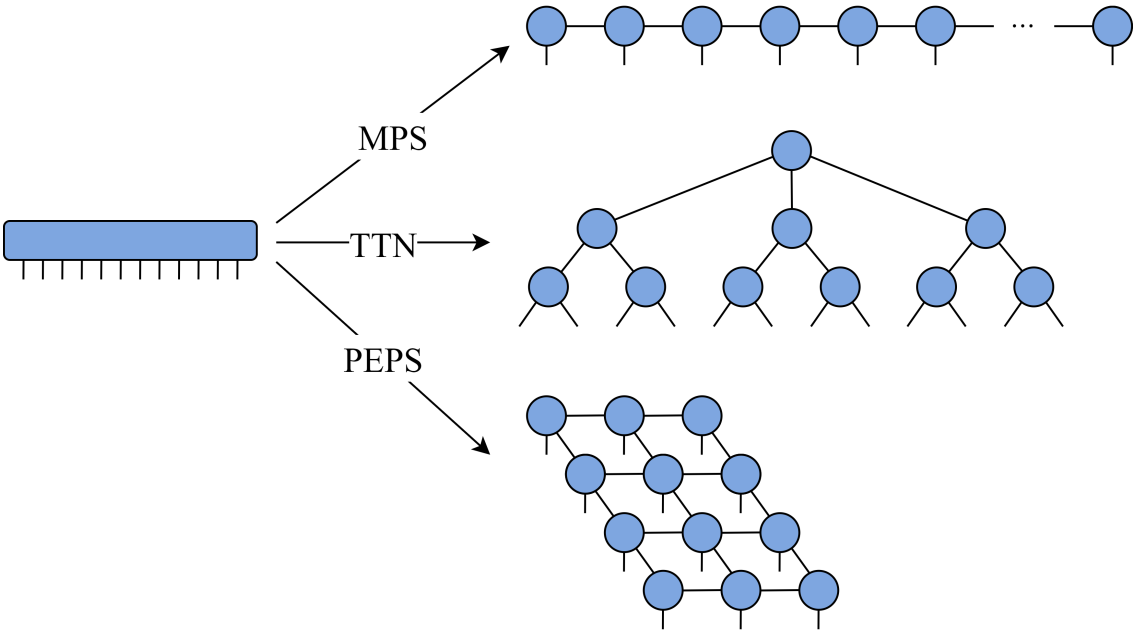


Figure 1.1 A high-dimensional tensor rewritten as some common classes of tensor networks: matrix product state (MPS), tree tensor network (TTN) and projected entangled pair state (PEPS). All tensor networks in this figure have open boundary conditions.

The main advantage is that T, which describes the entire Hilbert space using $\mathcal{O}(e^N)$ parameters, can in some cases be efficiently approximated with tensor networks using only $\mathcal{O}(\text{poly}(N))$ parameters. This procedure will be further explained in the following sections, where we will also introduce matrix product states (MPSs) [7] and tree tensor networks (TTNs) [8]. Note that other important classes will not be discussed, such as the two-dimensional extension of MPSs, the so-called projected entangled pair state (PEPS) [9], and a more general form of TTNs called multiscale entanglement renormalisation ansatz (MERA) [10].

1.3 Matrix product states

To represent T in Eq. (1.5) in terms of a tensor network, one has to 'split' the high-dimensional tensor into constituent tensors of lower rank. Any rank r matrix M can be factorised using the singular value decomposition (SVD)

$$M = USV^\dagger\,, \tag{1.6}$$

with $M \in \mathbb{C}^{m\times n}$, $U \in \mathbb{C}^{m\times r}$ with $U^\dagger U = \mathbb{1}_{r\times r}$, $V \in \mathbb{C}^{n\times r}$ with $V^\dagger V = \mathbb{1}_{r\times r}$, and finally $S = \text{diag}(s_1, \ldots, s_r) \in \mathbb{R}^{r\times r}$, which contains the strictly positive singular values of M [1]. These singular values are the square root of the eigenvalues of $M^\dagger M$ and are thus positive real numbers. By convention, S is sorted in descending order: $s_1 \geqslant \cdots \geqslant s_r$. We can now

[1] This form is called the compact SVD. In the full SVD, the diagonal matrix S in Eq. (1.6) contains both the zero and non-zero singular values of M.

efficiently approximate M with a lower rank matrix M_t using truncated SVD [11], which consists of taking only the t largest singular values into account:

$$M \approx M_t = U_t S_t V_t^\dagger \,. \tag{1.7}$$

where $t < r = \mathrm{rank}(M)$, $M_t \in \mathbb{C}^{m\times n}$, $U_t \in \mathbb{C}^{m\times t}$, $V_t \in \mathbb{C}^{n\times t}$ and $S = \mathrm{diag}(s_1, \ldots, s_t)$.

We can partition the Hilbert space of $|\Psi\rangle$ in Eq. (1.5) as $|j_1 \ldots j_n\rangle \otimes |j_{n+1} \ldots j_N\rangle$, with $1 \leqslant n < N$

$$|\Psi\rangle = \sum_{j_1 j_2 \ldots j_N} T_{j_1 j_2 \ldots j_N} |j_1 \ldots j_n\rangle \otimes |j_{n+1} \ldots j_N\rangle \,. \tag{1.8}$$

The tensor T can be flattened to a matrix by using

$$(j_1 \ldots j_n) = j_1 + j_2 p_1 + \cdots + j_n \prod_{i=1}^{n-1} p_i \,, \tag{1.9}$$

where p_i is the maximum value the index j_i can take. For example, the $2 \times 2 \times 2$ tensor A with elements a_{ijk} can be rewritten as the 4×2 matrix

$$\begin{pmatrix} a_{(11)1} & a_{(11)2} \\ a_{(12)1} & a_{(12)2} \\ a_{(21)1} & a_{(21)2} \\ a_{(22)1} & a_{(22)2} \end{pmatrix} \,. \tag{1.10}$$

The elements of T in matrix form will be denoted as $T_{(j_1 \ldots j_n),(j_{n+1} \ldots j_N)}$. After reshaping T to a matrix, SVD can be applied

$$T_{(j_1 \ldots j_n),(j_{n+1} \ldots j_N)} = \sum_{b_n=1}^{d_n} U_{(j_1 \ldots j_n),b_n} S^n_{b_n,b_n} \overline{V}_{b_n,(j_{n+1} \ldots j_N)} \,, \tag{1.11}$$

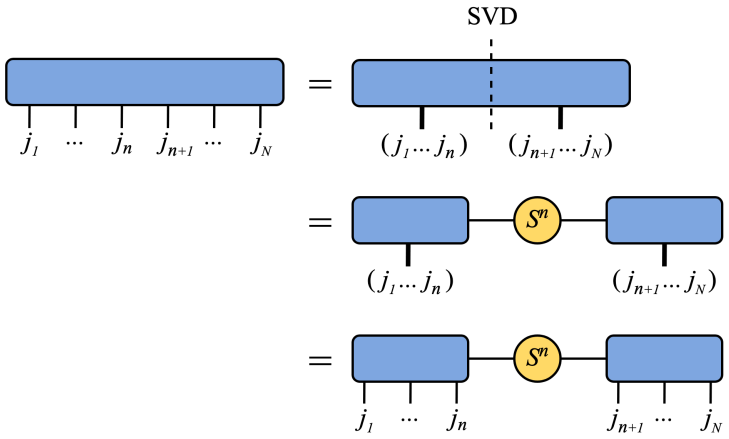
with $S^n = \mathrm{diag}(s_1, \ldots, s_{d_n})$ and

$$d_n = \min\left(\prod_{i=1}^{n} p_i, \prod_{i=n+1}^{N} p_i \right) \,. \tag{1.12}$$

If $p_1 = \cdots = p_N = p$, Eq. (1.12) simplifies to $d_n = p^{\min(n,\ N-n)}$. After transforming the basis $|j_1 \ldots j_n\rangle' = U |j_1 \ldots j_n\rangle$ and $|j_{n+1} \ldots j_N\rangle' = V |j_{n+1} \ldots j_N\rangle$, we obtain the Schmidt decomposition of $|\Psi\rangle$

$$|\Psi\rangle = \sum_{b_n=1}^{d_n} s_{b_n} |j_1 \ldots j_n\rangle' \otimes |j_{n+1} \ldots j_N\rangle' \,. \tag{1.13}$$

In tensor diagram notation, the equality between Eq. (1.5) and (1.13) is also apparent.

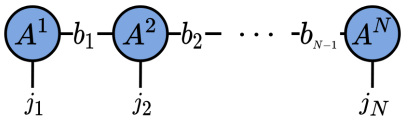


(1.14)

Successively repeating this procedure for $n = 1, 2, \ldots, N-1$ results in an MPS with open boundary conditions

$$|\Psi\rangle = \sum_{j_1=1}^{p_1} \cdots \sum_{j_N=1}^{p_N} \sum_{b_1=1}^{d_1} \cdots \sum_{b_{N-1}=1}^{d_{N-1}} A^{(1)}_{j_1 b_1} A^{(2)}_{b_1 j_2 b_2} \ldots A^{(N)}_{b_{N-1} j_N} |j_1 j_2 \ldots j_N\rangle \,, \tag{1.15}$$

where each $S^{(n)}$ was contracted with an adjacent matrix, see section 1.3.1 for more details. In the notation $A^{(i)}$, i is not a power but an index used to distinguish the tensors $A^{(i)}$ from each other. In tensor diagram notation, the coefficients of the state (1.15) are



. (1.16)

An MPS with periodic boundary conditions contains an additional contraction over the index b_N, which is shared by the tensors $A^{(1)}$ and $A^{(N)}$. Open boundary conditions are obtained when $d_N = 1$, which means $A^{(1)}$ and $A^{(N)}$ become matrices. In what follows, we will only encounter MPSs with open boundary conditions. When representing the state of a 1D system which exhibits translational invariance, it is useful to construct an MPS which also has this property by demanding that all of its tensors $A^{(i)}$ are identical.

Henceforth we restrict ourselves to situations with $p_1 = \cdots = p_N = p$. We will refer to p as the physical dimension and to $j_1, \ldots, j_N$ as the physical indices. For instance, the Ising model has $p = 2$ since each site $|j_i\rangle$ can be either $|\uparrow\rangle$ or $|\downarrow\rangle$. The contraction over $b_1, \ldots, b_N$, which are called virtual or bond indices, can be written as matrix multiplications. This allows us to reformulate Eq. (1.15) more elegantly as

$$|\Psi\rangle = \sum_{j_1 \ldots j_N} A^{(1)}_{j_1} A^{(2)}_{j_2} \ldots A^{(N)}_{j_N} |j_1 j_2 \ldots j_N\rangle \,, \tag{1.17}$$

where we used an abbreviated notation for the summations: $\sum_{j_1 \ldots j_N} = \sum_{j_1=1}^{p} \cdots \sum_{j_N=1}^{p}$. The maximum value of the bond indices are called the bond dimensions, denoted as

$d_1, \ldots, d_N$ in Eq. (1.15). The maximum of these is referred to as the bond dimension D of the MPS. An example of an MPS is depicted in Fig. 1.1.

Representing T in Eq. (1.5) exactly as an MPS is often not desired because, unless a significant amount of singular values are zero, it requires $\mathcal{O}(p^N)$ parameters (see Eq. (1.12)). The exponential scaling of the required number of parameters is precisely what we were trying to resolve. If the bond dimension of the MPS is limited to D, the MPS contains at most NpD^2 parameters. This limit can be imposed by successively applying truncated SVD on T, where at most D singular values are retained at each split. Generally speaking, an MPS can accurately approximate T when the discarded singular values are small compared to the retained singular values for each truncated SVD. By contracting all bond indices of the constructed MPS, we obtain an approximation T_D of the original tensor T. Since T_D is a reconstruction of a tensor with p^N elements using only poly(N) parameters, the coefficients in T_D are not independent. This is not surprising as the construction of tensor networks with poly(N) parameters consists of applying many truncated SVDs, which form a lower rank approximation.

Although an MPS with poly(N) parameters is able to accurately model only an exponentially small volume of the Hilbert space of a quantum N-body system, it can capture many physically relevant states [2]. For instance, gapped low energy states of a system with a local Hamiltonian have many vanishing singular values in their Schmidt decomposition (Eq. (1.13)). To see the reason for this, we first consider a state $|\Psi\rangle$ living in a Hilbert space H and with its density matrix given by $\rho = |\Psi\rangle\langle\Psi|$. A bipartition of the system into two subsystems A and B, with $H = H_A \otimes H_B$, allows us to define the reduced density matrix by taking the partial trace over the basis of B

$$\rho_A = \sum_b \langle b|\Psi\rangle\langle\Psi|b\rangle = \mathrm{Tr}_B[\rho]\,. \tag{1.18}$$

The Von Neumann entropy of ρ_A is given by

$$S = \mathrm{Tr}[\rho_A \log \rho_A]\,. \tag{1.19}$$

As S can be interpreted as a measure of entanglement between the two subsystems [12], it is also referred to as the entanglement entropy. A $|\Psi\rangle$ picked at random from H will likely have an entanglement entropy that scales as the volume of the smallest of the two subsystems A or B [4]. The property of interest for low-energy eigenstates of gapped Hamiltonians with local interactions, is that the entanglement entropy tends to scale as the size of the boundary between A and B [12]. This so-called area-law for the entanglement entropy is somewhat intuitive, as we can expect that local interactions will only entangle degrees of freedom near the boundary of the subsystems. The heavy constraints imposed on the states by the area-law causes many singular values in the Schmidt decomposition to vanish [6]. Since discarding singular values which are zero does not introduce an error, these states can be expressed exactly as an MPS with poly(N) parameters [13].

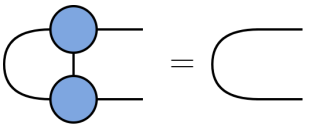
1.3.1 Canonical form

The MPS representation is invariant under the transformation $A^{(i)} \to \left(X^{(i-1)}\right)^{-1} A^{(i)} X^{(i)}$, with $X^{(i)}$ a $d_i \times d_i$ matrix. This can be seen as follows

$$\begin{aligned} &\sum_{j_1 \dots j_N} \mathrm{Tr}\Big[\left(X^{(N)}\right)^{-1} A^{(1)}_{j_1} X^{(1)} \left(X^{(1)}\right)^{-1} A^{(2)}_{j_2} \dots A^{(N)}_{j_N} X^{(N)} \Big] |j_1 j_2 \dots j_N\rangle \\ &= \sum_{j_1 \dots j_N} \mathrm{Tr}\left[A^{(1)}_{j_1} A^{(2)}_{j_2} \dots A^{(N)}_{j_N} \right] |j_1 j_2 \dots j_N\rangle \ , \end{aligned} \tag{1.20}$$

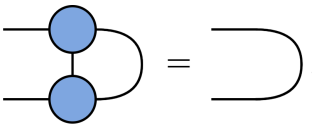
where we used Eq. (1.4). The index $(0) = (N)$ due to periodic boundary conditions. In case of open boundary conditions ($d_N = 1$), the trace operation in Eq. (1.20) is redundant and $X^{(N)}$ becomes a scalar. The freedom in parametrization, also called the gauge freedom, can be used to give the tensors in the MPS convenient properties. For instance, we can impose without loss of generality that

$$\forall i \in \mathbb{N}_0,\ i < N : \sum_{j_i=1}^{p} \left(A^{(i)}_{j_i}\right)^\dagger A^{(i)}_{j_i} = \mathbb{1}_{d_i \times d_i} \tag{1.21}$$

$\Longleftrightarrow$  , (1.22)

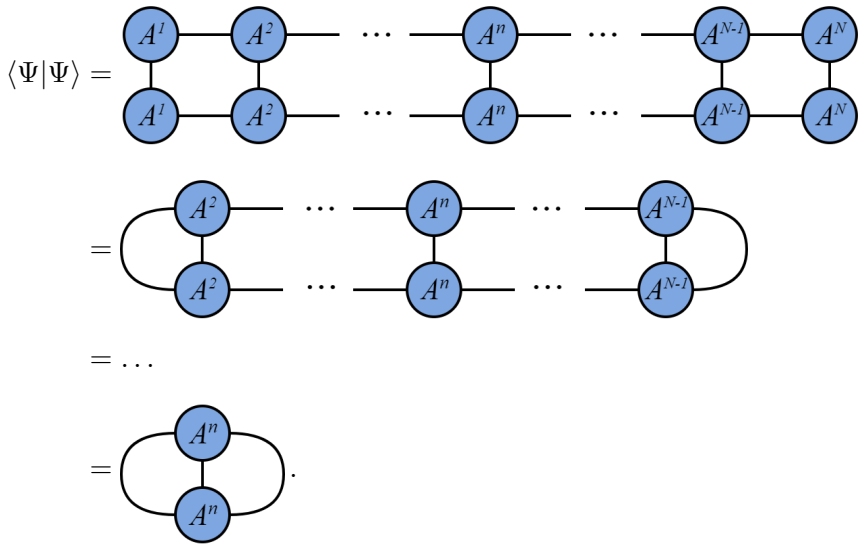
which is the left-canonical form of an MPS. When constructing an MPS with open boundary conditions by successively repeating (1.14) for $n = 1, 2, \dots, N-1$, we can ensure that this MPS is in the left-canonical form by contracting $S^{(n)}$ with the tensor to its right after each SVD. The MPS will then satisfy (1.21) since U in Eq. (1.6) is orthogonal. The right-canonical form is defined analogously

$$\forall i \in \mathbb{N}_0,\ 1 < i \leqslant N : \sum_{j_i=1}^{p} A^{(i)}_{j_i} \left(A^{(i)}_{j_i}\right)^\dagger = \mathbb{1}_{d_i \times d_i} \tag{1.23}$$

$\Longleftrightarrow$  . (1.24)

We can also define a mixed-canonical form with respect to an index n, with $1 < n < N$. The tensors $\{A^{(i)}_{j_i} | i = 1, \dots, n-1\}$ are then left-canonical, while $\{A^{(i)}_{j_i} | i = n+1, \dots, N\}$ are right-canonical. The tensor $A^{(n)}$ is kept neither left nor right-canonical. We will often

use the mixed-canonical state to compute $\langle\Psi|\Psi\rangle$ very efficiently:



(1.25)

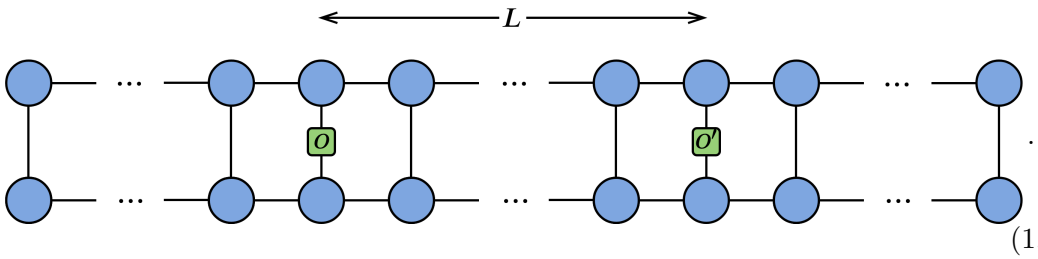
The time required for this computation is now independent of N.

1.3.2 Correlations

To investigate which types of correlation functions are present in the states that an MPS can represent, we consider an MPS $|\Psi\rangle$ of N sites with open boundary conditions. The two-point correlation function of an operator O at site n and an operator O' at site $n+L$ is defined as [6]

$$C(L) = \langle\Psi|O_{j_n} O'_{j_{n+L}}|\Psi\rangle \ , \tag{1.26}$$

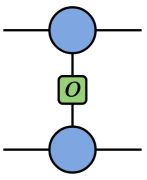
or visually



(1.27)

The tensor

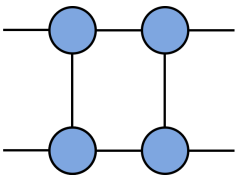
$$E_O^{(m)} = \sum_{j_m=1}^{p} \sum_{j'_m=1}^{p} O_{j_m j'_m} A^{(m)}_{j_m} \otimes \overline{A}^{(m)}_{j'_m} =$$



(1.28)

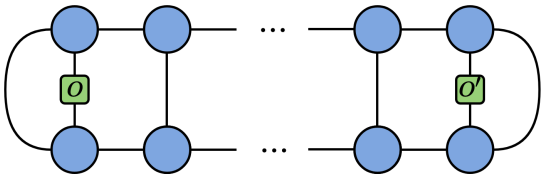
is referred to as the O-transfer matrix. This becomes the transfer matrix $E^{(m)}$ in the case $O = \mathbb{1}$. Graphically, the product of two transfer matrices is given by

$$E^{(m)}E^{(m+1)} = \quad . \tag{1.29}$$



Returning to Eq. (1.27), by bringing the tensors left of site n in left-canonical form and those right of $n + L$ in right canonical-form, the correlator can be simplified to

$$C(L) = \tag{1.30}$$



$$= \mathrm{Tr}\left[E_O^{(n)} \prod_{i=1}^{L-1} E^{(n+i)} E_{O'}^{(n+L)}\right] . \tag{1.31}$$

where the trace operations are performed as depicted in diagram (1.30). For simplicity, assume that the MPS is translationally invariant, implying that $A^{(1)} = \cdots = A^{(N)} = A$. The transfer matrices are then site independent

$$C(L) = \mathrm{Tr}\left[E_O E^{L-1} E_{O'}\right] . \tag{1.32}$$

By using the eigen decomposition $E = \sum_k \lambda_k |k\rangle \langle k|$ and normalising A such that all eigenvalues admit $\lambda_k \leqslant 1$, the correlator becomes

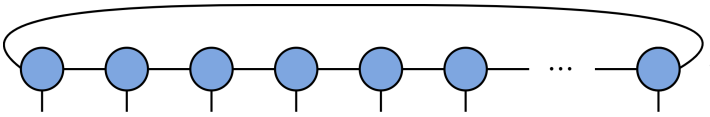
$$C(L) = \mathrm{Tr}\left[\sum_k \lambda_k^{L-1} E_O |k\rangle \langle k| E_{O'}\right] . \tag{1.33}$$

We see that the correlator decays exponentially with L, except for a constant which is determined by the terms with $\lambda_k = 1$. Note that normalizing A is unnecessary if $|\Psi\rangle$ represents a physical state because we then have $\langle\Psi|\Psi\rangle = \mathrm{Tr}\left[E^N\right] = 1$, which implies that all eigenvalues of E already admit $\lambda_k \leqslant 1$.

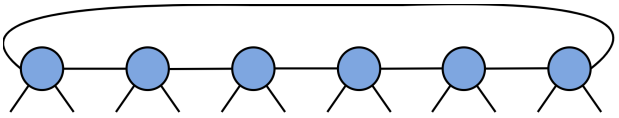
1.4 Tree tensor networks

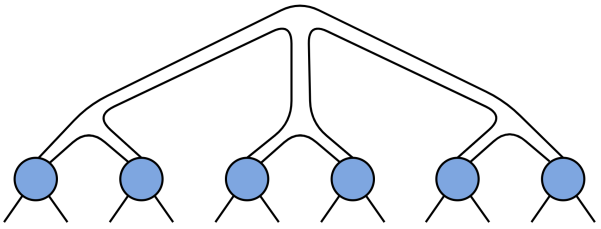
Correlation functions of MPS states decay exponentially, while many states of interest, e.g. ground states of gapless Hamiltonians in 1D, support correlations with power law decay [6]. Since not all quantum states can be represented efficiently by an MPS, alternative forms of tensor networks can also be valuable tools. Here we will consider tensor networks of the shape of a Cayley tree with coordination number c, which determines the order of the tensors. In Fig. 1.1, a tree tensor network (TTN) with coordination number 3 is depicted. Each MPS of bond dimension D can be expressed as a TTN with $c = 3$ and

bond dimension D^2 [2]. To see this, consider an MPS of 12 sites with closed boundary conditions

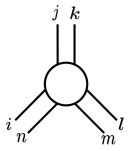
$|\Psi\rangle =$  . (1.34)

By contracting neighbouring tensors, we obtain

$|\Psi\rangle =$  (1.35)

$=$  . (1.36)

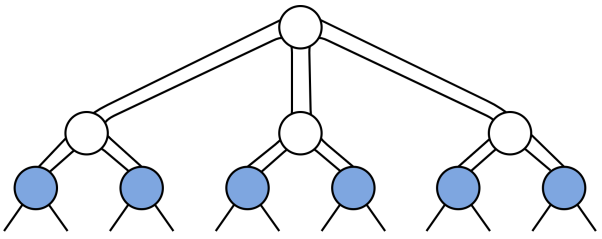
Note that Eq. (1.36) is simply a different way of drawing the tensor network in Eq. (1.35). By defining a tensor with elements given by

 $= \delta_{ij}\delta_{kl}\delta_{mn}$, (1.37)

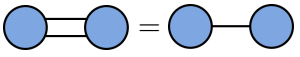
where the Kronecker delta is defined as

$$\delta_{ij} = \begin{cases} 0 & \text{if } i \neq j \\ 1 & \text{if } i = j \,, \end{cases} \tag{1.38}$$

we can rewrite Eq. (1.36) as

$|\Psi\rangle =$  . (1.39)

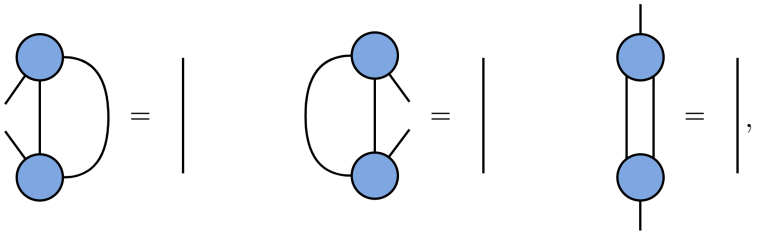
Using Eq. (1.9), it follows that

 . (1.40)

Suppose the indices on the left hand side can take on d and d' different values, then the index on the right hand side can take on dd' values. Using Eq. (1.40), Eq. (1.39)

can be brought into the form of a TTN, as depicted in Fig. 1.1 [2]. Despite the close connection between MPSs and TTNs, we will see that the two tensor networks have different properties.

Similar to the MPS case, the gauge freedom over the bond indices can be used to impose a canonical form of the tensors in a TTN. Depending on which index is retained, we can define a left-, right- and upper-canonical form. These are respectively

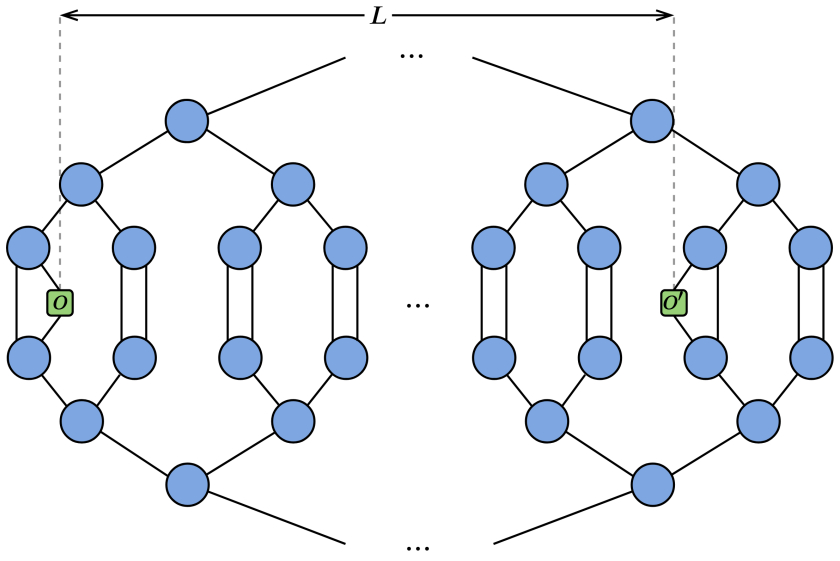


(1.41)

where the single straight lines represent identity matrices.

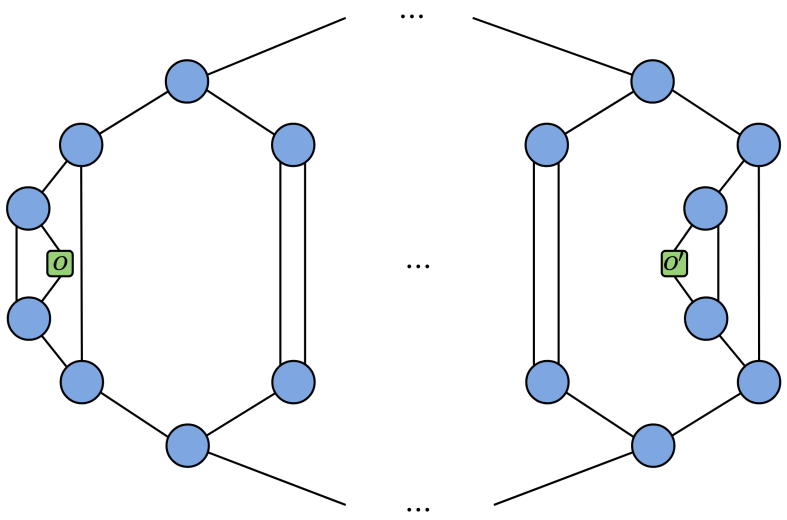
Consider the correlation function $C(L)$ given by (1.26), where $|\Psi\rangle$ is now a TTN of N sites with $c = 3$ and operators O and O' act on sites 2 and $N - 3$ respectively

$C(L) =$

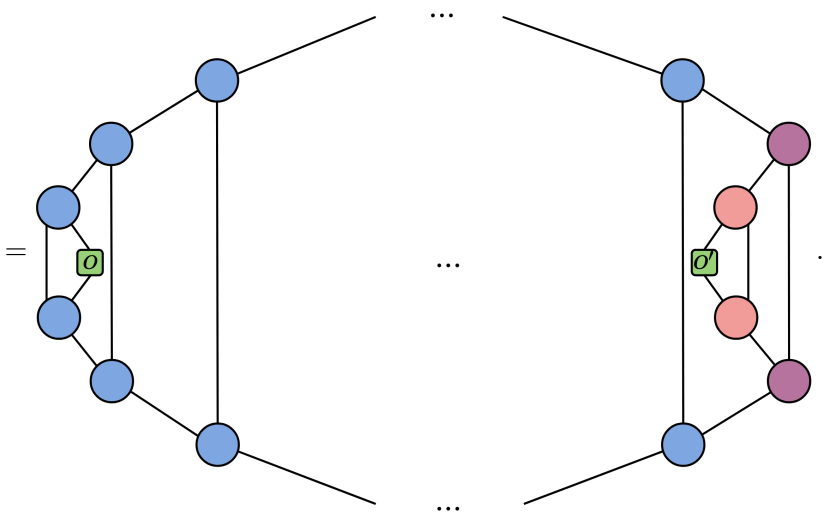


(1.42)

If all tensors which do not lie on the paths from sites 2 and $N-3$ to the root of the tree are in the appropriate canonical form, we can simplify (1.42) as follows

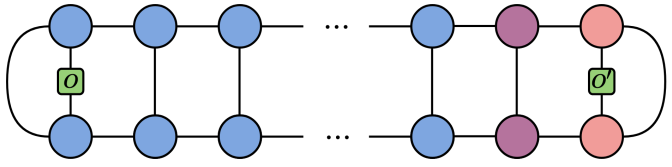
$C(L) =$ 

(1.43)

$=$ 

. (1.44)

The purple and pink colours are used to keep track of the tensors and have no further meaning. We can bring the pink tensors outwards since it doesn't alter which indices are contracted

$C(L) =$ 

(1.45)

Clearly the obtained expression for $C(L)$ has the same form as Eq. (1.30). If we again assume that all tensors in the TTN are identical and perform the traces as shown in diagram 1.45, we obtain

$$C(L) = \mathrm{Tr}\left[E_O E^{\mathcal{O}(\log L)} E_{O'}\right] . \tag{1.46}$$

The exponent is now $\mathcal{O}(\log L)$ instead of $L-1$ because the unique path between sites n and $n+L$ contains $\mathcal{O}(\log L)$ tensors. Eq. (1.46) and the following discussion is valid for any TTN, not just the example used in diagram (1.42). Analogous to Eq. (1.33), we write E as its eigen decomposition $E = \sum_k \lambda_k |k\rangle \langle k|$ and normalise the tensors of the TTN such that all eigenvalues admit $\lambda_k \leqslant 1$.

$$C(L) = \mathrm{Tr}\left[\sum_k \lambda_k^{\mathcal{O}(\log L)} E_O |k\rangle \langle k| E_{O'}\right] \tag{1.47}$$

$$= \mathrm{Tr}\left[\sum_k \mathcal{O}(L^\alpha) E_O |k\rangle \langle k| E_{O'}\right], \tag{1.48}$$

where $\alpha \in \mathbb{R}^-$ is a constant. We see that the correlator in a TTN decays with a power law, meaning a TTN can represent states that an MPS, with its exponentially decaying correlations, cannot.

To see how tensor networks can be used in a machine learning context, in particular for generative modelling, we will first introduce these two subjects in the following chapters.

Chapter 2

Machine learning

2.1 Overview

Machine learning is a field closely related to applied statistics which studies algorithms that perform tasks without requiring explicit instructions. These tasks are often too challenging to be solved by a program designed by humans. Instead, machine learning algorithms rely on patterns and inference from given data to improve their performance in a specific task, which is referred to as learning.

The interest in machine learning has grown significantly in the past decade due to the availability of large amounts of data and the increase in computational capabilities. Specifically the improvements in graphics processing units (GPUs) have made the use of very complex algorithms feasible [14]. The numerical operations required to train most models are essentially matrix multiplications, which GPUs are able to perform efficiently.

Machine learning algorithms can be broadly categorized as supervised or unsupervised depending on how they learn from the data:

- **Supervised learning**: Supervised learning algorithms require a dataset where each sample has a set of features or variables $\boldsymbol{x}$ and an associated label $\boldsymbol{y}$. The aim in supervised learning is to find a function f so that $f(\boldsymbol{x}) = \boldsymbol{y}$, meaning the algorithm is able to predict the correct label of a sample from its features. The given features are for example the pixel values of an image or the configurations of the spins on a lattice. Predicting the label is usually done by estimating the conditional probability $P(\boldsymbol{y}|\boldsymbol{x})$. The goal of any machine learning algorithm is that it must generalize as well as possible [15]. In the case of supervised learning, generalization means that the algorithm must achieve meaningful accuracy when determining the labels of samples that were not used to train the model. This means that it must learn useful patterns from the training data so that the labels of unseen samples can also be inferred. Note that $\boldsymbol{y}$ can be a scalar or a vector. For example when classifying cells as cancerous or benign, the labels $\boldsymbol{y}$ would be scalars. On the other hand, Facebook's

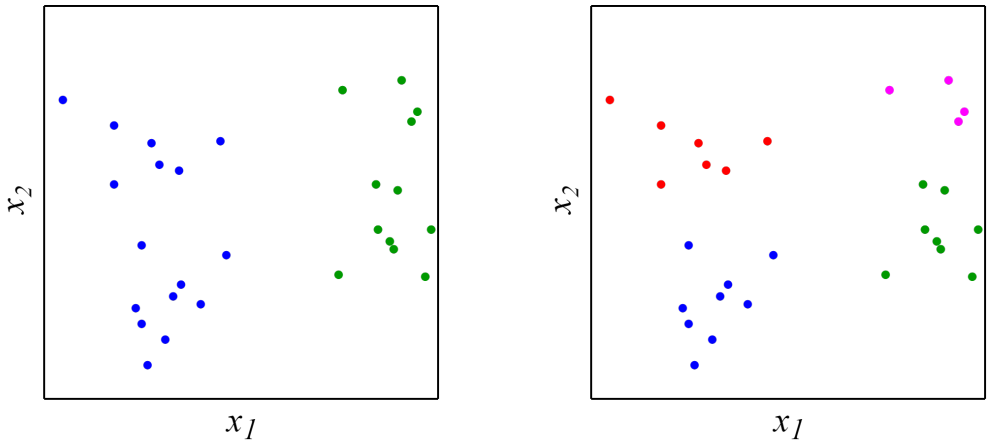


Figure 2.1 Two examples of clustering of points. The preferred type of clustering depends on the problem at hand.

algorithm to automatically tag all people in an image has vectors as labels since each image can contain multiple faces to classify. Both of these examples are called classification tasks because $\boldsymbol{y}$ can only take a limited amount of values. When $\boldsymbol{y}$ is continuous instead of discrete, the task is referred to as regression. As suggested by the name, this field has substantial overlap with regression analysis in statistics. Even for relatively simple regression problems which are exactly solvable, such as linear regression, machine learning approaches can be a more viable solution. This will be discussed in more detail in section 2.2.

- **Unsupervised learning**: In unsupervised learning, the task is to find patterns and structure in unlabeled data. A typical example of unsupervised learning is clustering, where the dataset is divided into groups of similar samples. Another example is dimensionality reduction, which consists of reducing the number of features under consideration while retaining as much information as possible. Unsupervised learning often poses extra challenges compared to the supervised variant because there are no labels available to guide the algorithms [15]. For instance, Fig. 2.1 indicates that the evaluation of a clustering algorithm is not straightforward. It is a priori unclear which of the two clusterings is superior, making the evaluation problem-dependent.

2.2 Linear regression

To introduce the general strategy of machine learning algorithms, it is instructive to first look at a simple example. In linear regression, we are given T samples (data points) where each sample i consists of a vector of features $\boldsymbol{x}^{(i)} \in \mathbb{R}^N$ and a scalar regression label $y^{(i)}$. The goal is to find the function $f(\boldsymbol{x}^{(i)}) = y^{(i)}$, which maps the features $\boldsymbol{x}^{(i)}$ to the desired output $y^{(i)}$ for each sample i. This function is assumed to depend linearly on

the parameters of the model $\boldsymbol{\theta} \in \mathbb{R}^{N+1}$, which results in

$$f(\boldsymbol{x}^{(i)}) = \theta_0 + \theta_1 x_1^{(i)} + \theta_2 x_2^{(i)} + \cdots + \theta_N x_N^{(i)} \,. \tag{2.1}$$

Note that linear regression only restricts f to be linear in $\boldsymbol{\theta}$, not in the features $\boldsymbol{x}$. For instance, a polynomial can also be used as model by setting $\boldsymbol{x} = (x, x^2, \dots)$.

A standard approach to determine the optimal parameters is the method of least squares [16], where we minimize the sum of squared errors

$$\mathcal{L} = \sum_{i=1}^{T} \left(y^{(i)} - f(\boldsymbol{x}^{(i)}) \right)^2 . \tag{2.2}$$

This loss function for the linear regression problem will be further motivated in section 3.2. To make notation easier, we will add a constant feature to each $\boldsymbol{x}^{(i)}$ which will be multiplied with the intercept θ_0, meaning $\boldsymbol{x}^{(i)} = \left(1, x_1^{(i)}, \dots, x_N^{(i)}\right)$. We define a vector $\boldsymbol{y} = \left(y^{(1)}, y^{(2)}, \dots, y^{(T)}\right)$ containing the label of each sample and a $T \times (N+1)$ matrix X containing the features of each sample in its rows. The loss function (2.2) can now be rewritten as

$$\mathcal{L} = \|\boldsymbol{y} - X\boldsymbol{\theta}\|^2 \,. \tag{2.3}$$

The first derivative is given by

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} &= \frac{\partial}{\partial \boldsymbol{\theta}} \left((\boldsymbol{y} - X\boldsymbol{\theta})^\mathsf{T} (\boldsymbol{y} - X\boldsymbol{\theta}) \right) && (2.4) \\ &= \frac{\partial}{\partial \boldsymbol{\theta}} \left(\boldsymbol{y}^\mathsf{T}\boldsymbol{y} - 2\boldsymbol{\theta}^\mathsf{T} X^\mathsf{T} \boldsymbol{y} + \boldsymbol{\theta}^\mathsf{T} X^\mathsf{T} X \boldsymbol{\theta} \right) && (2.5) \\ &= -2X^\mathsf{T}\boldsymbol{y} + 2X^\mathsf{T} X\boldsymbol{\theta} \,. && (2.6) \end{aligned}$$

The second derivative is then

$$\frac{\partial^2 \mathcal{L}}{\partial \boldsymbol{\theta}^2} = 2X^\mathsf{T} X \,, \tag{2.7}$$

which is a positive semi-definite matrix. Therefore, $\mathcal{L}$ is a convex function and we can determine the optimal parameters for which $\mathcal{L}$ is minimal by setting its first derivative to zero.

$$\begin{aligned} & \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} = -2X^\mathsf{T}\boldsymbol{y} + 2X^\mathsf{T} X\boldsymbol{\theta} = 0 && (2.8) \\ \iff \quad & X^\mathsf{T} X\boldsymbol{\theta} = X^\mathsf{T}\boldsymbol{y} && (2.9) \\ \iff \quad & \boldsymbol{\theta} = \left(X^\mathsf{T} X\right)^{-1} X^\mathsf{T}\boldsymbol{y} && (2.10) \end{aligned}$$

This approach requires that $X^\mathsf{T} X$ is invertible, which means the number of samples must at least be larger than the number of features so that X has full rank [15]. The exact solution is also inefficient for regression in high-dimensional spaces since the complexity

of the matrix inversion $\left(X^\intercal X\right)^{-1}$ is $\mathcal{O}(N^3)$. An alternative approach that doesn't suffer from these limitations is gradient descent. In this approximate method, the parameters are updated iteratively by taking steps in the negative direction of the gradient of $\mathcal{L}$.

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \gamma \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}^{(t)}} \tag{2.11}$$

$$= \boldsymbol{\theta}^{(t)} - 2\gamma \sum_{i=1}^{T} \left(f\left(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}^{(t)}\right) - y^{(i)} \right) \boldsymbol{x}^{(i)} \tag{2.12}$$

where the initial values of the parameters $\boldsymbol{\theta}^{(0)}$ can be chosen randomly. The semicolon in $f\left(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}^{(t)}\right)$ is used to separate the inputs $\boldsymbol{x}^{(i)}$ of the function from the parameters $\boldsymbol{\theta}^{(t)}$ which define the function. The factor 2 in Eq. (2.12) can be absorbed in the constant γ, which is the learning rate of the algorithm. The update step (2.11) can be repeated until the gradient becomes zero for all parameters. At that point, the algorithm has converged to a minimum of the loss function. This is the global minimum if $\mathcal{L}$ is convex, which means the solution (2.10) has been found. The number of gradient steps, also called epochs, and the learning rate are hyperparameters in this algorithm: their values control the behaviour of the algorithm and must be set before the learning process begins. Various meta-algorithms exist that monitor the performance of the model for several choices of hyperparameters in an attempt to find the optimal hyperparameter values [17].

2.3 General strategy

As mentioned in section 2.1, the central goal of machine learning is to perform well on previously unseen samples, which is referred to as the ability to generalize. To evaluate this ability, the dataset is split into a train and test set. The learning algorithm must achieve meaningful performance on the test samples by using the training samples to optimize its parameters. This method can only work if each sample in the dataset is independent from each other and if all samples were drawn from the same probability distribution P_d [15]. These i.i.d. (independent and identically distributed) assumptions imply that all samples stem from the same generative process which has no memory of past generated samples.

The set of functions that a machine learning algorithm is able to select as a solution is referred to as its hypothesis space. In linear regression, the hypothesis space is the set of all functions which are linear in the parameters $\boldsymbol{\theta}$. To search the hypothesis space for optimal parameters, a loss function $\mathcal{L}$ is required which quantifies the performance of the model on the given task. For linear regression, the loss function (2.2) is the sum of the squared differences between the predicted and the true label. The loss is minimal (zero) when the predictions match the true labels. Because of the i.i.d. assumptions, the machine learning algorithm can improve its performance on the unseen samples by minimizing its loss on the training set. An additional requirement is that the difference between the training and the test loss must be limited. Since the parameters of the model

are chosen to minimize the training loss, the expected training loss is smaller than or equal to the expected value of the test loss.

Two central challenges in machine learning are underfitting and overfitting. Underfitting occurs when the model cannot adequately capture the underlying structure of the data [15]. The result is that the model is not able to achieve a sufficiently low training loss. Overfitting means that the model has learned properties which are specific to the training samples (e.g. noise) [14]. Such a model will generalize poorly because those specific properties will not be present in the test samples, resulting in a large gap between training and test loss.

Whether a model is more likely to underfit or overfit can be controlled by changing its capacity. The model's capacity is its ability to fit a wide variety of functions, which is predominantly determined by the number of parameters in the model and the used optimization algorithm. Note that the capacity and hypothesis space of a model are not necessarily identical because imperfections in the optimization scheme can make that regions of the hypothesis space cannot be accessed in practice [15].

The top panels in 2.2 illustrate underfitting and overfitting for a linear regression problem. The bottom panel shows a typical relationship between capacity and loss [15]. A model with inadequate capacity, such as a linear fit in the regression example, is unable to describe the data well. This underfitting results in both a high training and test loss. A model with a capacity appropriate for the complexity of the task achieves the best test loss. As the capacity of the model is further increased, the training loss decreases because for instance the noise in the training data is also described. Besides requiring more computational resources, the gap between the training and test loss increases because the the model learned properties specific to the training data. The top right plot shows an example of overfitting, where the fitted polynomial perfectly describes the training data but provides an unlikely description for other x.

Many machine learning algorithms train models with a number of parameters much higher than the number of training samples. For linear regression this would lead to a situation as in the top right pane of Fig. 2.2, where infinitely many functions pass exactly through the training points. As a result, the chosen function is unlikely to resemble the correct function. On the other hand, the famous convolutional neural network *AlexNet* has over 60 million parameters and can be trained on 1.6 million training samples without significantly overfitting [18]. Although it is not fully understood why these models are able to generalize [19], regularization methods play an important role in avoiding overfitting. One of the simplest and most effective regularization methods is early stopping, where the training is ended as soon as the model starts overfitting. To detect overfitting, the performance on a validation set is measured after each epoch. The validation and test data are both not used to train the model, however the hyperparameters of the machine learning algorithm are chosen to achieve the best performance on the validation data, while the test set is meant to be completely independent.

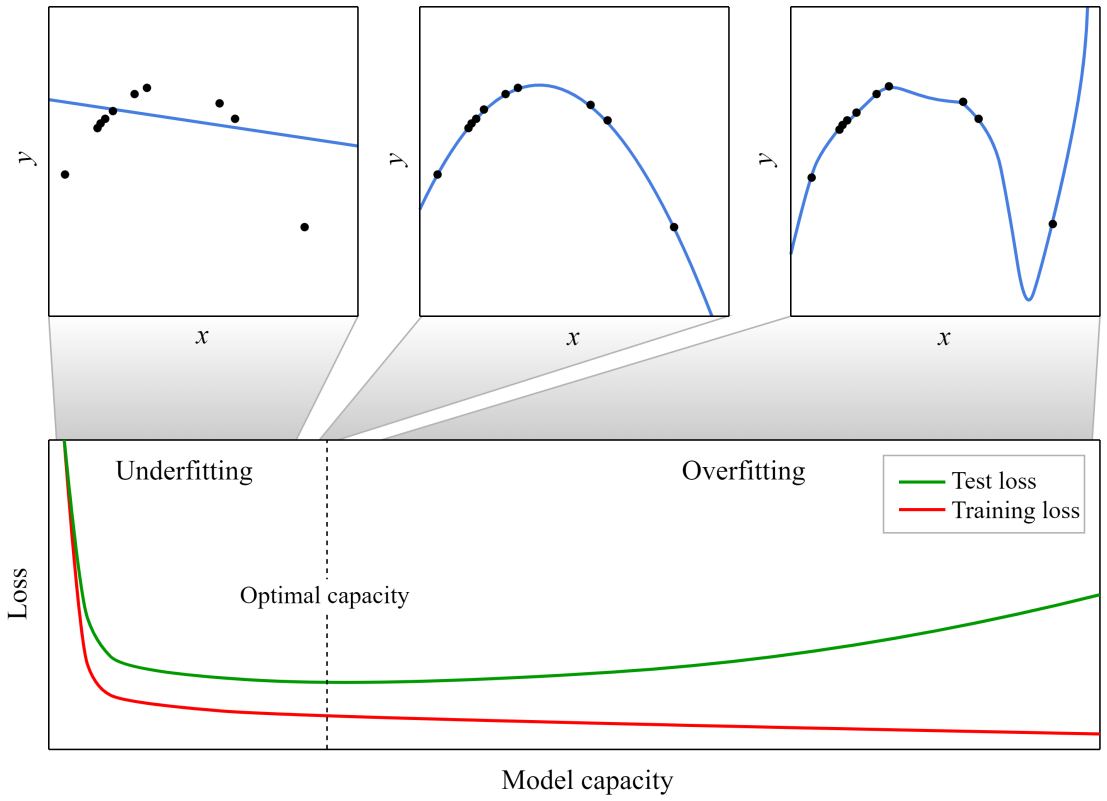


Figure 2.2 An illustration of the typical influence of the model's capacity on the tendency to underfit or overfit. The capacity is optimal when the test loss is minimal, indicated by the dashed line. The top three plots show (from left to right) examples of linear regression with respectively inadequate, appropriate and excessive model capacity. The model capacity increases from left to right and the loss from bottom to top.

We can now summarize the key aspects of machine learning:

- **Problem formulation**: Gathering unbiased[1] data which preferably satisfies the i.i.d. assumptions and choosing a clear task for the algorithm.
- **Representation**: Deciding which model the algorithm will train, for instance a neural network or a tensor network. This also determines the hypothesis space.
- **Optimization**: Choosing an appropriate loss function which quantifies how well the model performs on the given task. Since the loss function depends on the parameters of the model, it is a P-dimensional function with P the number of parameters in the model. The function is generally also not convex. Because evaluating the loss function is usually computationally expensive, first order optimization schemes such as gradient descent are often employed for minimization.

[1] A classic but unverified example of dataset bias is where the U.S. trained a model to classify images of tanks as allied or hostile. However these images were taken during different seasons, so they ended up with a model that classified weather conditions instead of tanks.

- **Generalization**: The main objective of machine learning is performing a given task well on unseen samples. The hyperparameters of the algorithm can be tuned to obtain minimal loss on the validation data. After training the model, its ability to generalize can be quantified by the loss on the test set. If the gap between the loss on training and unseen samples is too large, regularization methods can be used to prevent overfitting.

Chapter 3

Generative modelling

3.1 Overview

This thesis deals with unsupervised generative modelling. Thereby one aims at capturing the underlying probability distribution $P_d(\boldsymbol{x})$ that generated a dataset. The distribution $P_d(\boldsymbol{x})$ is either a probability density function or a probability mass function, depending on whether $\boldsymbol{x}$ is continuous or discrete. A generative model can either explicitly define a distribution $P_m(\boldsymbol{x})$ which it tries to make similar to $P_d(\boldsymbol{x})$ [1], or it can generate new samples from $P_m(\boldsymbol{x})$ without explicitly defining it. In the latter case, the likelihood that $\boldsymbol{x}$ will be generated by the model can not be determined, which makes objective evaluation of the model's performance challenging [15]. To make matters worse, generative models can simultaneously underfit and overfit and yet seem to produce high-quality samples. Imagine a generative model trained to generate images of cars and bikes. If the model produces identical copies of the training images of cars, its generated images would look visually pleasing. However, the model is underfitting since no images of bikes are produced. Simultaneously, it is also overfitting as no images that were not in the training set are produced. For explicit generative models, this situation is easily identified as a bimodal distribution of the likelihood of its training images, where the likelihood of car images is much larger than the likelihood of bike images. Detecting underfitting in implicit models is more challenging, especially for realistic datasets with many different modes instead of the two modes (cars and bikes) in our example. Implicit models are nonetheless useful for tasks like denoising and synthesis, which will be explained below. Some important applications of generative modelling are:

(i) **Anomaly detection**: This task consists of identifying abnormal samples in a dataset. Abnormal samples, also called anomalies or outliers, are defined as instances that differ significantly from the majority of the data. Anomalies are usually detected by modelling the underlying distribution of 'normal' samples followed by marking the samples which have a low likelihood of being generated by said model.

[1] In statistics, this is known as density estimation for continuous $\boldsymbol{x}$ and as probability mass function estimation for discrete $\boldsymbol{x}$.

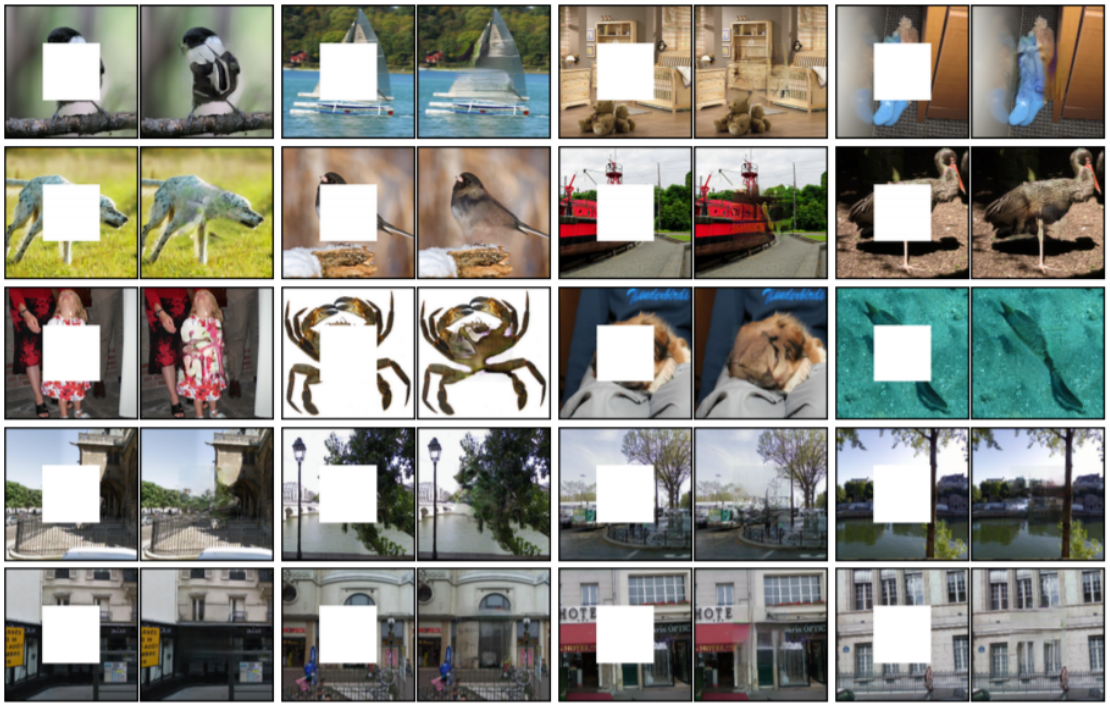


Figure 3.1 Results from Ref. [24] using a type of variational autoencoder (VAE) [25] as generative model. The corrupted samples are images of the ImageNet dataset [26] with their center pixels removed. Right of each corrupted image is the reconstruction by the model. These images were not part of the training set, so the model has not simply memorized the uncorrupted images.

For instance, server intrusions can be detected as deviations from a model that describes regular server activity [20]. A different approach, which doesn't rely on generative modelling, uses the fact that anomalies are more easily isolated than normal samples. This concept is for example used in the isolation forest algorithm [21].

(ii) **Missing data imputation**: Here, a machine learning algorithm must provide a prediction for the values of some features which are missing from a sample. If a generative model is able to approximate the unobservable $P_d(\boldsymbol{x})$ with $P_m(\boldsymbol{x})$, we can estimate the conditional distribution of the values of missing features. For instance, if a corrupted sample $\boldsymbol{x}_{\not\alpha}$ is given which contains each feature except x_α, the estimated distribution of the missing value is given by $P_m(x_\alpha|\boldsymbol{x}_{\not\alpha})$. In practice, missing data is a common problem in for instance clinical research, where many medical tests are only done when strictly necessary because the procedure is expensive or invasive. Imputation is often required to analyze data with missing features using statistical methods [22]. When trained on time-series data, e.g. the evolution of stock market prices, a generative model can be used to simulate likely futures [23]. Another example of missing data imputation is image reconstruction, which is illustrated in Fig. 3.1.

(iii) **Representation learning**: Obtaining a useful representation of data is of vital importance for any task. For instance, dividing two numbers is straightforward in the Arabic numeral representation, while the same task becomes a challenge when representing the numbers as Roman numerals. Manual feature-engineering to extract a useful representation from raw data is expensive and becomes infeasible for complex tasks. Say we want to detect whether there is a car present in an image, which is important information e.g. for self-driving cars. It would be helpful to know if there are wheels in the image. The variation in lighting, angle, obscuring objects and so on, make it difficult to describe exactly what a wheel looks like in terms of pixel values. One way to accomplish such complex tasks is to use multiple layers, e.g. of a neural network, to extract increasingly abstract features from the data. This is shown in Fig. 3.2, which uses results of a generative model from Ref. [27]. As represented in the directed graph in Fig. 3.2, each node in the first hidden layer is simply a linear combination of the input features, possibly added with a constant which is referred to as the bias. A non-linear function, the so-called activation function, is then applied to all nodes of hidden layer 1. The result is then used as input for the second layer, after which the same procedure is repeated for all layers. A common activation function for hidden layers is ReLu [28], which sets negative values to zero: $f_{\mathrm{ReLu}}(x) = \max(0, x)$. The remarkable capability of neural networks to extract useful representation explains the success of deep learning [15]. We will also see that certain generative models learn a meaningful low-dimensional representation of the input data by modelling its empirical probability distribution. Figure 3.4, which is a result from Ref. [29], is another example of the connection between representation learning and generative modelling.

(iv) **Synthesis**: A straightforward application of generative modelling is generating new samples which are similar to the those in the training data. This task is referred to as synthesis or sampling. An important example is speech synthesis, where a text must be converted to audio that reads the given text aloud. Because there are many different ways to say a sentence, the output requires a large amount of variation to seem natural and realistic. Manually developing a program that performs varied speech synthesis is close to impossible. However, developing a generative model that learns these variations from audio data is possible.

(v) **Denoising**: Given a corrupted sample $\boldsymbol{x}_c$, a generative model is expected to predict the uncorrupted version $\boldsymbol{x}$ by estimating $P_d(\boldsymbol{x}|\boldsymbol{x}_c)$. The corrupted samples could be $\boldsymbol{x}_c = \boldsymbol{x} + \boldsymbol{\epsilon}$, where $\boldsymbol{\epsilon}$ is noise with unknown distribution.

(vi) **Classification**: In section 2.1 we mentioned that a label $\boldsymbol{y}$ can be predicted from features $\boldsymbol{x}$ by estimating the conditional probability $P_d(\boldsymbol{y}|\boldsymbol{x})$. This is referred to as the discriminative approach. We can also take a generative approach by modelling the joint probability distribution $P_d(\boldsymbol{x}, \boldsymbol{y})$ using an explicit generative model. The prediction is then obtained from Bayes' rule

$$P_m(\boldsymbol{y}|\boldsymbol{x}) = \frac{P_m(\boldsymbol{x}|\boldsymbol{y})P_m(\boldsymbol{y})}{P_m(\boldsymbol{x})} = \frac{P_m(\boldsymbol{x}, \boldsymbol{y})}{\sum_{\boldsymbol{y}} P_m(\boldsymbol{x}, \boldsymbol{y})}\,. \tag{3.1}$$

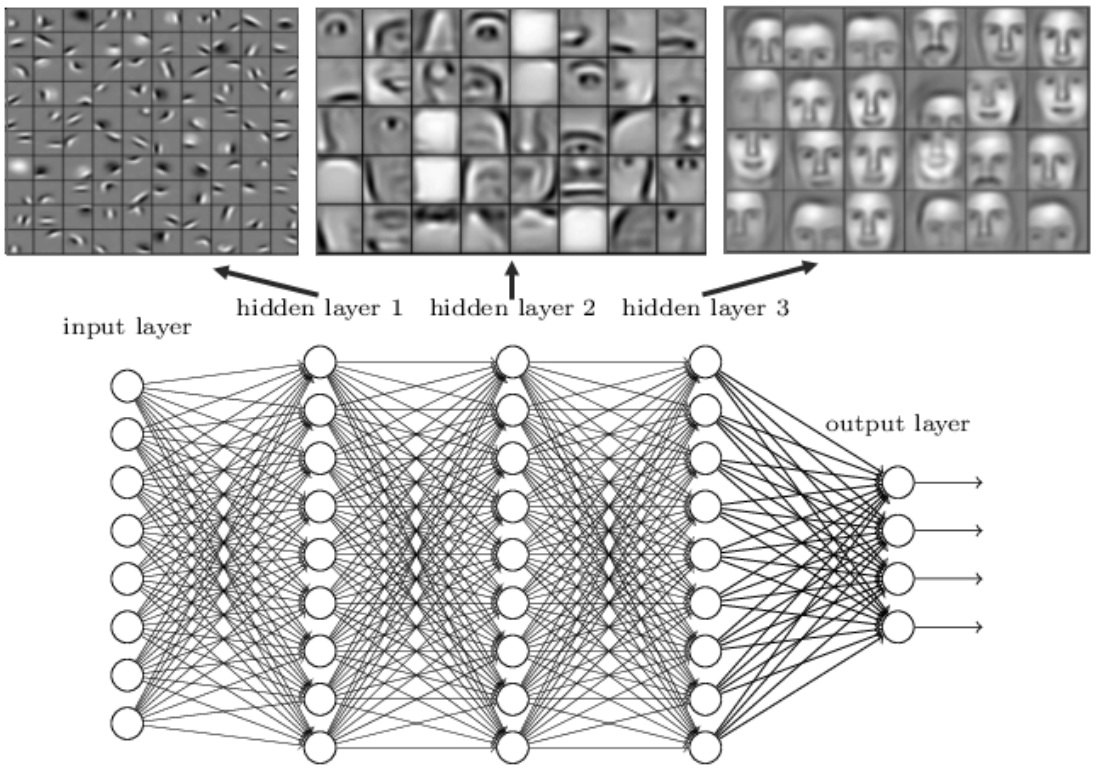


Figure 3.2 A visualization of hierarchical feature learning by a deep neural network, represented by the directed graph. The top images are generated by successive hidden layers of a deep belief network designed by Ref. [27] which was trained to model the probability distribution of greyscale images of faces. The first hidden layer combines nearby pixels to form edges and contrasts. Given the first layer's description of edges, the second layer is able to represent more complex features of the data, such as eyes and mouths, which are subsequently fed into the next layer. Instead of learning the complicated mapping from pixels to faces directly, a deep network can learn a series of simpler mappings. Note that the directed graph shows a generic fully connected neural network, it does not represent the specific model used to generate the top images containing the features.

Classic examples of this approach are naive Bayes classifiers [30] and linear discriminant analysis [31].

3.2 Maximum likelihood estimation

To train a generative model, we would like to find parameters $\boldsymbol{\theta}$ so that the probability of generating the given samples $\boldsymbol{x}^{(i)}$ is large. More formally, we want to maximize the likelihood of the given data, which we can write as a product thanks to the i.i.d. assumptions

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \prod_{i=1}^{T} P_m(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}) \,. \tag{3.2}$$

A product of many probabilities, which can become small for some $\boldsymbol{\theta}$, is prone to arithmetic underflow, where very small numbers are rounded to zero. Since the logarithm is a monotonically increasing function, we can avoid numerical issues by maximizing the log-likelihood instead [15]

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{T} \log P_m(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}) \tag{3.3}$$

$$= \arg\min_{\boldsymbol{\theta}} -\mathbb{E}_{\boldsymbol{x}\sim P_d} \left[\log P_m(\boldsymbol{x}; \boldsymbol{\theta})\right] . \tag{3.4}$$

Equation (3.4) shows that we want to minimize the mean negative log-likelihood (NLL), which is a common loss function in generative modelling. The minus sign was introduced because the loss tends to be a function that we wish to minimize.

The use of maximum likelihood methods is not limited to generative modelling. It is a common principle in other fields of machine learning and in statistics [32]. For instance in supervised leaning, the optimal parameters of a model which infers $\boldsymbol{y}^{(i)}$ from $\boldsymbol{x}^{(i)}$ are the parameters which maximize the conditional log-likelihood

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{T} \log P_m(\boldsymbol{y}^{(i)}|\boldsymbol{x}^{(i)}; \boldsymbol{\theta}) . \tag{3.5}$$

Suppose we have scalar continuous labels $y^{(i)}$ a Gaussian model

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{T} \log \left[\frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{\left(y^{(i)} - f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta})\right)^2}{2\sigma^2}\right)\right] \tag{3.6}$$

$$= \arg\max_{\boldsymbol{\theta}} \left(-\frac{T}{2}\log 2\pi - T\log\sigma - \sum_{i=1}^{T} \frac{\left(y^{(i)} - f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta})\right)^2}{2\sigma^2}\right) \tag{3.7}$$

$$= \arg\min_{\boldsymbol{\theta}} \sum_{i=1}^{T} \left(y^{(i)} - f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta})\right)^2 . \tag{3.8}$$

Therefore maximizing the log-likelihood yields the same optimal parameters as minimizing the sum of squared errors (2.2), which justifies the use of Eq. (2.2) as loss function in least squares linear regression.

3.2.1 Limitations

To capture underlying probability distribution $P_d(\boldsymbol{x})$, introduced in section 3.1, we must be able to measure how different it is from our model $P_m(\boldsymbol{x})$. One such a measure is the Kullback-Leibler (KL) divergence, also referred to as the relative entropy. The KL divergence of P with respect to Q is defined as

$$D_{\mathrm{KL}}(P\|Q) = \mathbb{E}_{\boldsymbol{x}\sim P}\left[\log \frac{P(\boldsymbol{x})}{Q(\boldsymbol{x})}\right] = \mathbb{E}_{\boldsymbol{x}\sim P}\left[\log P(\boldsymbol{x}) - \log Q(\boldsymbol{x})\right] . \tag{3.9}$$

Minimizing the KL divergence of P_d with respect to P_m results in

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} D_{\mathrm{KL}}(P_d(\boldsymbol{x}) \| P_m(\boldsymbol{x}; \boldsymbol{\theta})) \tag{3.10}$$

$$= \arg\min_{\boldsymbol{\theta}} \mathbb{E}_{\boldsymbol{x} \sim P_d} \left[\log P_d(\boldsymbol{x}) - \log P_m(\boldsymbol{x}; \boldsymbol{\theta}) \right] , \tag{3.11}$$

which is identical to Eq. (3.4) since the first term doesn't depend on $\boldsymbol{\theta}$. In other words, maximizing the log-likelihood of the data is equivalent to minimizing the KL divergence $D_{\mathrm{KL}}(P_d \| P_m)$ [33].

It is straightforward to prove that the KL divergence is nonnegative [34]. We will make use of Jensen's inequality [35], which states that for a random variable $X \in \mathbb{R}$ and a convex function φ

$$\varphi\left(\mathbb{E}[X]\right) \leqslant \mathbb{E}\left[\varphi(X)\right] . \tag{3.12}$$

As log is a concave function, we can use Jensen's inequality to write

$$\mathbb{E}_{\boldsymbol{x} \sim P}\left[\log \frac{Q(\boldsymbol{x})}{P(\boldsymbol{x})}\right] \leqslant \log\left(\mathbb{E}_{\boldsymbol{x} \sim P}\left[\frac{Q(\boldsymbol{x})}{P(\boldsymbol{x})}\right]\right) \tag{3.13}$$

$$= \log\left(\sum_{\boldsymbol{x}} P(\boldsymbol{x}) \frac{Q(\boldsymbol{x})}{P(\boldsymbol{x})}\right) \tag{3.14}$$

$$= \log\left(\sum_{\boldsymbol{x}} Q(\boldsymbol{x})\right) \tag{3.15}$$

$$= \log 1 = 0 \tag{3.16}$$

$$\iff D_{\mathrm{KL}}(P \| Q) \geqslant 0 . \tag{3.17}$$

The particular situation $D_{\mathrm{KL}}(P_d \| P_m) = 0$ corresponds with $P_d(\boldsymbol{x}) = P_m(\boldsymbol{x})$ if $\boldsymbol{x}$ is discrete. For continuous variables, D_{KL} is zero if P_d and P_m are equal almost everywhere[2] [15]. Due to these properties, it could be tempting to think of $D_{\mathrm{KL}}(P_d \| P_m)$ as a distance between the two distributions. However, measuring the distance between $P_d(\boldsymbol{x})$ and $P_m(\boldsymbol{x})$ should give the same result as measuring the distance between $P_m(\boldsymbol{x})$ and $P_d(\boldsymbol{x})$, while the KL divergence generally does not exhibit this symmetry: $D_{\mathrm{KL}}(P_d \| P_m) \neq D_{\mathrm{KL}}(P_m \| P_d)$. The KL divergence also does not satisfy the triangle inequality: we generally have that

$$D_{\mathrm{KL}}(P_d \| P_m) \not\leqslant D_{\mathrm{KL}}(P_d \| Q) + D_{\mathrm{KL}}(Q \| P_m) \tag{3.18}$$

[2] In measure theory, a property that holds *almost everywhere* means that a property is valid for all elements in a set except a subset of measure zero. In this context, we mean that $P_d(\boldsymbol{x}) = P_m(\boldsymbol{x})$ holds throughout $\mathbb{R}^n$ except for a set of points in $\mathbb{R}^n$ which occupies no volume [15].

for some distribution Q [14]. For simplicity, assume that $\boldsymbol{x}$ is discrete[3]. Suppose we have for some $\boldsymbol{x}_1$ that $P_d(\boldsymbol{x}_1) \approx 0$ and $P_m(\boldsymbol{x}_1) > 0$, the contribution of $\boldsymbol{x}_1$ to

$$D_{\mathrm{KL}}(P_d \| P_m) = \sum_{\boldsymbol{x}} \left(P_d(\boldsymbol{x}) \log \frac{P_d(\boldsymbol{x})}{P_m(\boldsymbol{x})} \right) \tag{3.19}$$

will be negligible because

$$\lim_{a \to 0^+} (a \log a) = \lim_{a \to 0^+} \frac{\log a}{a^{-1}} = \lim_{a \to 0^+} \frac{a^{-1}}{-a^{-2}} = 0\,. \tag{3.20}$$

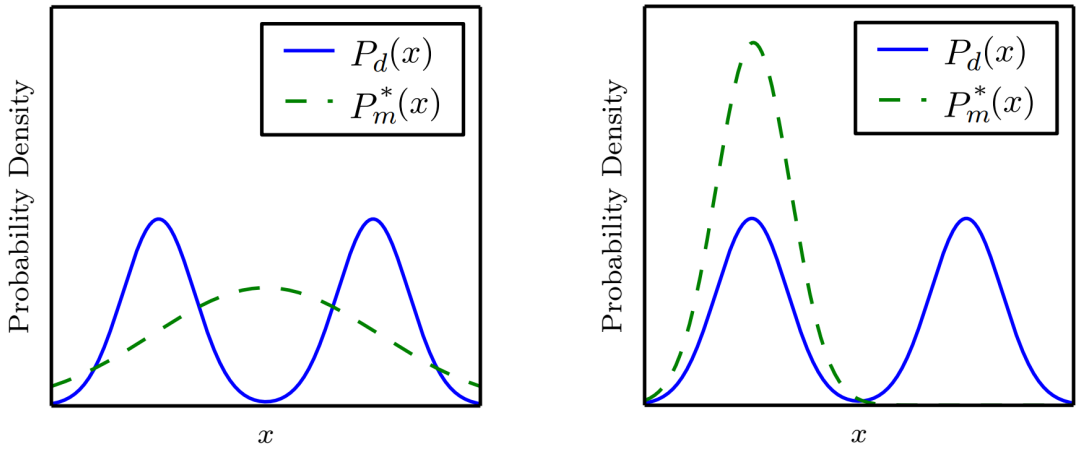
On the other hand, for $D_{\mathrm{KL}}(P_m \| P_d)$ the $\boldsymbol{x}_1$ term will be very large. Similarly, a point $\boldsymbol{x}_2$ with $P_d(\boldsymbol{x}_2) > 0$ and $P_m(\boldsymbol{x}_2) \approx 0$ will greatly enlarge $D_{\mathrm{KL}}(P_d \| P_m)$ while being insignificant for $D_{\mathrm{KL}}(P_m \| P_d)$. In Fig. 3.3 we see the differences that can arise when minimizing either $D_{\mathrm{KL}}(P_d \| P_m)$ or $D_{\mathrm{KL}}(P_m \| P_d)$. Maximum likelihood optimization will focus on placing high probability in regions in $\boldsymbol{x}$-space with many training samples [36], however we also want P_m to have low probability where no training samples are observed. In other words, we want to minimize a true distance measure between distributions instead of $D_{\mathrm{KL}}(P_d \| P_m)$. The issue is that $D_{\mathrm{KL}}(P_m \| P_d)$ is impossible to compute because $P_d(\boldsymbol{x})$ is of course unknown. We will now introduce generative adversarial networks (GANs), which punishes models for placing high probability where no samples are observed without explicitly calculating $D_{\mathrm{KL}}(P_m \| P_d)$. This is achieved by penalizing the model for generating new samples that can be easily discriminated from the training samples.

3.3 Generative adversarial networks

GANs were designed as an alternative to generative models which are trained by maximizing likelihood. The current state-of-the-art algorithms for tasks such as image synthesis are based on GANs [37]. The central idea behind GANs is a scenario in game theory where two players compete with each other [38]. One player, the generator, takes a continuous latent variable $\boldsymbol{z}$ as input and produces new samples $\boldsymbol{x} = g(\boldsymbol{z}; \boldsymbol{\theta}^{(g)})$ that are intended to come from the same distribution as the training samples. The other player, the so-called discriminator, attempts to determine whether a given sample is drawn from the training data or from the generator. The output $d(\boldsymbol{x}; \boldsymbol{\theta}^{(d)})$ of the discriminator is therefore an estimation of the probability that $\boldsymbol{x}$ is a real sample, as opposed to a fake sample produced by the generator. The functions g and d must be differentiable to allow optimization by gradient descent and are typically represented by deep neural networks.

The latent variable $\boldsymbol{z}$ is introduced to encode complicated correlations between the features of a sample [14]. The idea is that $P(\boldsymbol{x}|\boldsymbol{z})$ is easier to estimate than $P(\boldsymbol{x})$ if $\boldsymbol{z}$ corresponds to abstract concepts of the generated sample [14]. For instance when generating images of faces, $\boldsymbol{z}$ could encode the gender, age, hair color, pose, etc. It is more straightforward for

[3] For a probability density function P of a continuous variable $\boldsymbol{x}$, we must integrate over an interval of $\boldsymbol{x}$-values to obtain the probability that $\boldsymbol{x}$ falls within this interval. The probability of a continuous variable taking on any particular value is always zero.



(a) $P_m^*(\boldsymbol{x}) = \arg\min_{\boldsymbol{\theta}} D_{\mathrm{KL}}(P_d(\boldsymbol{x}) \| P_m(\boldsymbol{x}; \boldsymbol{\theta}))$ (b) $P_m^*(\boldsymbol{x}) = \arg\min_{\boldsymbol{\theta}} D_{\mathrm{KL}}(P_m(\boldsymbol{x}) \| P_d(\boldsymbol{x}; \boldsymbol{\theta}))$

Figure 3.3 Figure adapted from Ref. [36] illustrating the asymmetry of the KL divergence. The differences between $D_{\mathrm{KL}}(P_d \| P_m)$ and $D_{\mathrm{KL}}(P_m \| P_d)$ are most obvious when the model has too little capacity to fit P_d, as is the case in this figure. (a) The result of fitting a Gaussian P_m to a mixture of two Gaussians P_d by minimizing $D_{\mathrm{KL}}(P_d \| P_m)$, which is equivalent to the maximum likelihood estimation. To minimize this KL divergence, P_m has high probability where P_d has high probability, which results in blurring the two modes. (b) Here, $D_{\mathrm{KL}}(P_m \| P_d)$ is minimized instead. Now P_m has low probability where P_d has low probability, meaning one of the modes is selected to avoid the region of low probability between the modes. Although P_m chose the left mode in this figure, the KL divergence is equally low when choosing the right mode. If the two modes of P_d are not separated by a sufficiently large region of low probability, its possible that both directions of the KL divergence blur the modes together as in (a).

a model to generate a coherent image when given that it should correspond to a blonde woman of 24 years old, instead of simply asking to generate a face. This is equivalent to saying that $P(\boldsymbol{x}|\boldsymbol{z})$ is easier to model than $P(\boldsymbol{x})$. Since it is intended that $\boldsymbol{z}$ only represents some general properties of a sample and not each detail, the dimensionality of $\boldsymbol{z}$ is much smaller than that of $\boldsymbol{x}$. $P(\boldsymbol{x})$ can be recovered by marginalizing over $\boldsymbol{z}$

$$P(\boldsymbol{x}) = \int P(\boldsymbol{x}|\boldsymbol{z}) \, \mathrm{d}\boldsymbol{z} \,. \tag{3.21}$$

In physics, Eq. (3.21) corresponds to integrating out degrees of freedom. The concept of integrating out a subset of variables leading to sophisticated correlations between the remaining variables is also common in physics. For instance, phenomena such as superconductivity can arise from integrating out phonon excitations of electrons in a lattice [39]. Latent variable generative models attempt to reverse this logic to encode highly non-linear interactions between the features of $\boldsymbol{x}$ by integrating out the newly introduced variables $\boldsymbol{z}$ [14]. A similar trick is for example used in the Hubbard-Stratonovich transformation in condensed matter physics, where a particle theory is converted to its respective field theory by introducing an auxiliary scalar field [40, 41].

Since a GAN consists of a generator (g) and its adversary, the discriminator (d), there is no single loss function being minimized. Instead, the optimization procedure can be formulated as a zero-sum game with the objective determined by a function $v(\boldsymbol{\theta}^{(d)}, \boldsymbol{\theta}^{(g)})$ [38]. During training, both players attempt to maximize their own payoff, given respectively by $v(\boldsymbol{\theta}^{(d)}, \boldsymbol{\theta}^{(g)})$ and $-v(\boldsymbol{\theta}^{(d)}, \boldsymbol{\theta}^{(g)})$ for the discriminator and the generator. At convergence, the generated samples are indistinguishable from those in the dataset and the discriminator always outputs $\frac{1}{2}$ [36]. The parameters of the generator are then given by

$$\boldsymbol{\theta}^{*(g)} = \arg\min_{\boldsymbol{\theta}^{(g)}} \arg\max_{\boldsymbol{\theta}^{(d)}} v(\boldsymbol{\theta}^{(d)}, \boldsymbol{\theta}^{(g)}) \,. \tag{3.22}$$

At this point, the discriminator can be discarded as its sole purpose is to train the generator. We will see later that convergence is rarely reached in practice [36]. Although many variants of GANs exist with various choices for v [42], the original proposal is [38]

$$\begin{aligned} v(\boldsymbol{\theta}^{(d)}, \boldsymbol{\theta}^{(g)}) &= \mathbb{E}_{\boldsymbol{x}\sim P_d} \left[\log d(\boldsymbol{x})\right] + \mathbb{E}_{\boldsymbol{z}\sim P_{\boldsymbol{z}}} \left[\log(1 - d(g(\boldsymbol{z})))\right] && (3.23) \\ &= \mathbb{E}_{\boldsymbol{x}\sim P_d} \left[\log d(\boldsymbol{x})\right] + \mathbb{E}_{\boldsymbol{x}\sim P_m} \left[\log(1 - d(\boldsymbol{x}))\right] \,. && (3.24) \end{aligned}$$

Note that P_d is the distribution of the data, while $d(\boldsymbol{x})$ represents the discriminator. We did not denote the dependence of g and d on their respective parameters to avoid clutter. The distribution $P_{\boldsymbol{z}}$ is a prior for the latent variables. Often, $P_{\boldsymbol{z}}$ adopts the form of a Gaussian [36]. The model distribution P_m is implicitly represented by the generator. For a fixed generator, the discriminator is a supervised classifier with labels $y = 1$ for all samples from the dataset and labels $y = 0$ for generated samples. Training the discriminator simply means applying Eq. (3.5) on both real and generated samples, so maximizing the log-likelihood of outputting $d(\boldsymbol{x}) = 1$ if $\boldsymbol{x} \sim P_d$ and outputting $d(\boldsymbol{x}) = 0$ if $\boldsymbol{x} \sim P_m$. The generator can only influence the second term of v, where it minimizes the log-probability of the discriminator being correct [38].

Again assuming a fixed generator, we can determine the optimal strategy for the discriminator by taking the first and second functional derivatives of Eq. (3.24) for a given sample $\boldsymbol{x}$.

$$\begin{aligned} \frac{\delta v}{\delta d(\boldsymbol{x})} &= \frac{P_d(\boldsymbol{x})}{d(\boldsymbol{x})} - \frac{P_m(\boldsymbol{x})}{1 - d(\boldsymbol{x})} && (3.25) \\ \frac{\delta^2 v}{\delta d(\boldsymbol{x})^2} &= -\frac{P_d(\boldsymbol{x})}{d(\boldsymbol{x})^2} - \frac{P_m(\boldsymbol{x})}{(1 - d(\boldsymbol{x}))^2} && (3.26) \end{aligned}$$

Since $P_d(\boldsymbol{x}) \geqslant 0$ and $P_m(\boldsymbol{x}) \geqslant 0$, we have that $\frac{\delta^2 v}{\delta d(\boldsymbol{x})^2} \leqslant 0$ and therefore v is concave when g is fixed. Setting the first derivative to zero gives the discriminator strategy which obtains maximal payoff for a fixed generator

$$\begin{aligned} & 0 = (1 - d^*(\boldsymbol{x}))\, P_d(\boldsymbol{x}) - d^*(\boldsymbol{x}) P_m(\boldsymbol{x}) && (3.27) \\ \iff \quad & d^*(\boldsymbol{x}) = \frac{P_d(\boldsymbol{x})}{P_d(\boldsymbol{x}) + P_m(\boldsymbol{x})} \,. && (3.28) \end{aligned}$$

This derivation is equivalent to proof 1 of Ref. [38]. By plugging Eq. (3.28) in Eq. (3.24) we obtain

$$v = \mathbb{E}_{\boldsymbol{x}\sim P_d}\left[\log\frac{P_d(\boldsymbol{x})}{P_d(\boldsymbol{x})+P_m(\boldsymbol{x})}\right] + \mathbb{E}_{\boldsymbol{x}\sim P_m}\left[\log\frac{P_m(\boldsymbol{x})}{P_d(\boldsymbol{x})+P_m(\boldsymbol{x})}\right] \tag{3.29}$$

$$= \mathbb{E}_{\boldsymbol{x}\sim P_d}\left[\log\frac{P_d(\boldsymbol{x})}{\frac{1}{2}\left(P_d(\boldsymbol{x})+P_m(\boldsymbol{x})\right)}\right] + \mathbb{E}_{\boldsymbol{x}\sim P_m}\left[\log\frac{P_m(\boldsymbol{x})}{\frac{1}{2}\left(P_d(\boldsymbol{x})+P_m(\boldsymbol{x})\right)}\right] - 2\log 2. \tag{3.30}$$

In this expression we can recognise the definition of KL divergence, given by Eq. (3.9), twice

$$v = D_{\mathrm{KL}}\left(P_d\middle\|\frac{P_d+P_g}{2}\right) + D_{\mathrm{KL}}\left(P_m\middle\|\frac{P_d+P_g}{2}\right) - 2\log 2 \tag{3.31}$$

$$= 2D_{\mathrm{JS}}(P_d\|P_m) - 2\log 2\,, \tag{3.32}$$

where D_{JS} is the Jensen–Shannon (JS) divergence, defined as

$$D_{\mathrm{JS}}(P\|Q) = \frac{1}{2}D_{\mathrm{KL}}\left(P\middle\|\frac{P+Q}{2}\right) + \frac{1}{2}D_{\mathrm{KL}}\left(Q\middle\|\frac{P+Q}{2}\right). \tag{3.33}$$

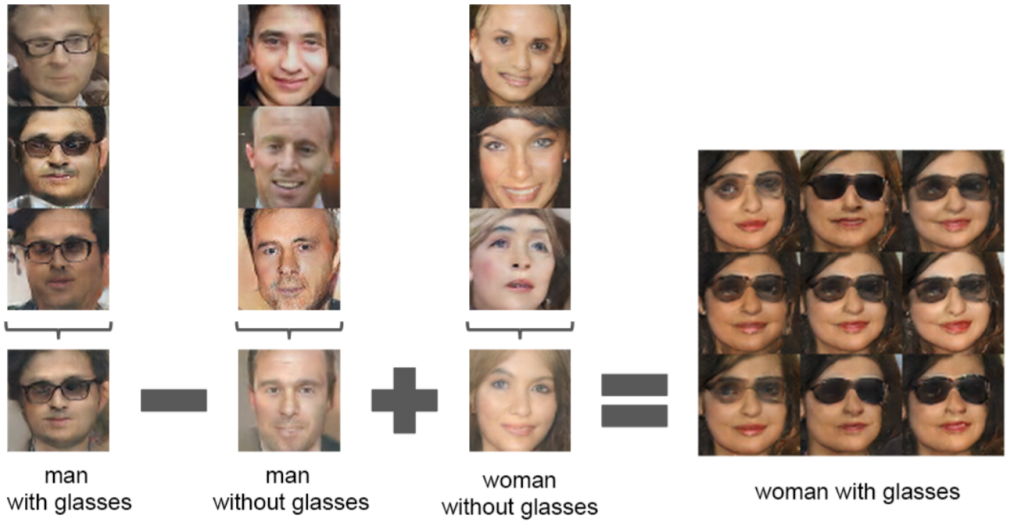
The JS divergence can be seen as a symmetrized version of the KL divergence, with $D_{\mathrm{JS}}(P\|Q) = D_{\mathrm{JS}}(Q\|P)$. Thanks to the properties of the KL divergence, we have that $D_{\mathrm{JS}}(P\|Q) \geqslant 0$ and $D_{\mathrm{JS}}(P\|Q) = 0 \iff P = Q$. Furthermore, The JS distance given by $\sqrt{D_{\mathrm{JS}}}$ also satisfies the triangle inequality and is a proper metric for the distance between two distributions [43]. We also see that the generator is optimal if $v = -\log 4$ and most importantly $P_m = P_d$. Under the made assumptions, which we will soon come back to, a GAN minimizes the JS divergence [38], in contrast to maximum likelihood which minimizes the KL divergence. Therefore we can hope that a GAN doesn't suffer from the same limitations as maximum likelihood, such as being prone to improperly describing data distributions with multiple modes [36]. The empirical observation that GANs tend to outperform maximum likelihood based methods in e.g. realistic image generation could then be explained by having access to estimations of both directions of the KL divergence, allowing GANs to avoid high P_m in regions of low P_d [14].

However, the assumptions we made are not valid in practice, making it difficult to theoretically explain the success of GANs [36]. Note that Eq. (3.32) depends on the assumption that the discriminator is trained until optimality after each update of the generator, which is often too computationally expensive in practice. It can also lead to vanishing gradients for the generator, where the gradient of $\mathbb{E}_{\boldsymbol{z}\sim P_{\boldsymbol{z}}}\left[\log(1-d(g(\boldsymbol{z})))\right]$ becomes too small to improve the generator with gradient descent when the discriminator confidently labels generated samples as fake. This situation is especially common at the start of training, when the generated samples are clearly different from the training samples [36]. In practice, a heuristically motivated objective for g is often used to avoid vanishing gradients in early learning [38]. Instead of minimizing $\mathbb{E}_{\boldsymbol{z}\sim P_{\boldsymbol{z}}}\left[\log(1-d(g(\boldsymbol{z})))\right]$, the generator is trained by maximizing

$$v_g = \mathbb{E}_{\boldsymbol{z}\sim P_{\boldsymbol{z}}}\left[\log d(g(\boldsymbol{z}))\right], \tag{3.34}$$

while the discriminator still maximizes Eq. (3.24). Intuitively, this means that instead of minimizing the log-probability of the discriminator being correct, the generator maximizes the log-probability of the discriminator being mistaken [36]. The alternative objective for the generator means that the game is no longer zero-sum. Furthermore, the theoretical results that GAN training corresponds to minimizing $D_{\mathrm{JS}}(P_d \| P_m)$ is no longer valid. Even if the zero-sum game (3.22) is used, the expression for the optimal discriminator (Eq. (3.28)) depends on the convexity of v for a fixed generator. Although this is valid if $d(\boldsymbol{x}; \boldsymbol{\theta}^{(d)})$ and $g(\boldsymbol{x}; \boldsymbol{\theta}^{(g)})$ are directly optimized in function space, we are instead updating the parameters $\boldsymbol{\theta}^{(d)}$ and $\boldsymbol{\theta}^{(g)}$ of the neural networks which represent these functions. Since v is generally not convex in parameter space, results (3.28) and (3.32) are not valid in practice [38].

Besides the lack of theoretical guarantees, GANs also suffer from mode collapse, where some modes of P_d are not present in P_m. An example of mode collapse would be generating only images of fives when trained on MNIST, a dataset of images with handwritten digits ranging from zero to nine (see section 5.3.1). Since the discriminator is a supervised classifier, it is also susceptible to issues commonly encountered in supervised learning, namely underfitting and overfitting. All these factors make GANs notoriously difficult to train. However the architectures and algorithms that do work tend to be very successful, such as the deep convolutional GAN (DCGAN) and its successors [44]. Fig. 3.4 is a remarkable result of DCGANs, obtained in Ref. [29], indicating that $\boldsymbol{z}$ can encode meaningful concepts of the dataset. Unlike the discriminator, the generator is fairly resistant against overfitting. Instead of learning by increasing the likelihood of the training samples, the generator adapts its parameters to increase the probability, assigned by the discriminator, that the generated samples are from the dataset. Since the generator does not directly observe the training data, it is unlikely to simply output a copy of a training sample [36]. Much work has gone into improving the theoretical understanding of GANs and into stabilizing their training process [42][45], however these remain open issues [15].



(a)

(b)

Figure 3.4 (a) Latent vector arithmetic of a deep convolutional GAN trained on images of faces. The latent vectors of 3 generated images are averaged for each column, producing the lower row. These are then added or subtracted to produce the center image on the right, while the surrounding 8 images are produced by adding uniform noise. This result shows that the latent space of the GAN has captured remarkably abstract concepts of the data, namely gender and glasses [29]. (b) Doing the same arithmetic in input space instead of latent space, meaning we average and add or subtract pixel values, produces a much worse result. These figures are results of Ref. [29], where the connection between generative models and representation learning was investigated.

Chapter 4

Matrix product states as generative models

4.1 Tensor networks for machine learning

Generative modelling requires estimating a probability distribution which lives in an exponentially large space. For instance, the number of possible combinations of words grows exponentially with the length of a sentence. However only a minute fraction of these possibilities form a sensible sentence. Similarly, almost all combinations of pixel values look like noise and not as actual images. Thus the probability distributions of interest only occupy a tiny volume of the immense space [15]. This task shows significant similarities with representing quantum many-body states, which live on a small submanifold in their exponentially large Hilbert space (see section 1.1). The success tensor networks have shown in parameterizing the physically relevant submanifold, using a number of parameters that grows polynomially in system size, suggests that tensor networks could be a powerful architecture for generative models.

Recently, there have been many interesting applications of tensor networks in machine learning. For example, matrix product states and tree tensor networks have been used for classification tasks [46, 47], natural language processing [48] and as generative models [49, 50]. An MPS has also been used as a memory efficient way to represent the parameters of large neural networks [51]. Moreover, the perspective of quantum information, and in particular entanglement, has proven useful to gain a better theoretical understanding of both the models [52] and data [53] used in machine learning.

Tensor networks are also a promising framework for machine learning with quantum computers [16]. Near-term quantum computers will have a limited number of qubits and a high error rate [54]. Although machine learning techniques are generally resilient to noise caused by these errors [55], the input data in machine learning tends to require a significant amount of memory. Such an input is difficult to process using a limited number of qubits. Tensor networks can be realized as quantum circuits where the number

of qubits required scales at most logarithmically with the size of the input data [55]. Therefore tensor networks could be one of the first practical quantum generative models, making use of a limited amount of noisy qubits [56].

In what follows, we will discuss and further build on the work of Ref. [49], where an MPS is used for generative modelling. The code written for this thesis is available on Github [57]. The CPU version of the code used in Ref. [49], which can also be found on Github [58], was used as a framework.

4.2 Representation

In section 3.1 it was outlined that generative modelling requires approximating a probability distribution $P_d(\boldsymbol{x})$ by a model distribution $P_m(\boldsymbol{x})$ given T samples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(T)}\}$ which are drawn from $P_d(\boldsymbol{x})$. Each sample is $N \times p$ array, where N is the system size and p the physical dimension. For instance, a binary image of 196 pixels is encoded as 196 vectors: $\binom{1}{0}$ for black pixels and $\binom{0}{1}$ for white pixels. The entire image is then one-hot encoded as a 196×2 matrix. In one-hot encoding, the i'th feature value of p possible values is represented as a binary vector of length p, where the i'th element is 1 and all other vector elements are 0. The training set of T samples is now represented as a $T \times N \times p$ array.

A Born machine, inspired by Born's rule [59], represents the model distribution as

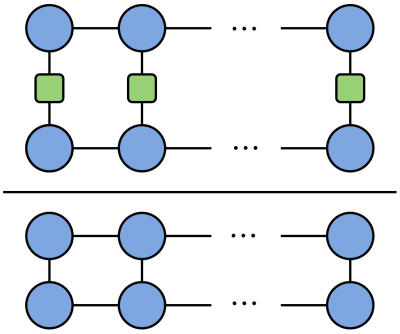
$$P_m(\boldsymbol{x}) = \frac{|\langle \boldsymbol{x}|\Psi\rangle|^2}{Z} . \tag{4.1}$$

In our case, $|\Psi\rangle$ is an MPS with real elements and open boundary conditions (see Eq. (1.17)) and Z is the normalization factor, analogous to a partition function in statistical physics [49]. Calculating Z generally requires to sum over all p^N possible configurations of $\boldsymbol{x}$, which is intractable. Most machine learning algorithms which use the maximum likelihood estimation approximate the normalization factor, for instance by using annealed importance sampling (AIS) [60]. Implicit generative models can even entirely avoid the need to compute Z, which is the approach GANs adopt (see section 3.3) [38]. One of the benefits of using an MPS model is that Z can be determined in a straightforward manner

$$Z = \sum_{\boldsymbol{x}} |\langle \boldsymbol{x}|\Psi\rangle|^2 = \sum_{\boldsymbol{x}} \langle \Psi|\boldsymbol{x}\rangle \langle \boldsymbol{x}|\Psi\rangle = \langle \Psi|\Psi\rangle . \tag{4.2}$$

By making use of the mixed canonical form, $\langle \Psi|\Psi\rangle$ can be computed efficiently with Eq. (1.25).

In tensor diagram notation, Eq. (4.1) reads

$P_m(\boldsymbol{x}) =$  , (4.3)

where the green squares represent the features $x_1, \ldots, x_N$ in one-hot encoding. In (4.1), the green squares are diagonal $p \times p$ matrices containing a one-hot encoded feature on its diagonal.

Instead of representing the model probability distribution by the square of an MPS, one could also use an MPS with non-negative tensor elements: $P_m(\boldsymbol{x}) = \langle \boldsymbol{x} | \Psi_+ \rangle / Z$. The tensors of $|\Psi_+\rangle$ could be constructed using nonnegative matrix factorization [61] instead of SVD to ensure all tensor elements are nonnegative. It has been suggested however that this has little impact on the capabilities of the generative model [62].

4.3 Optimization

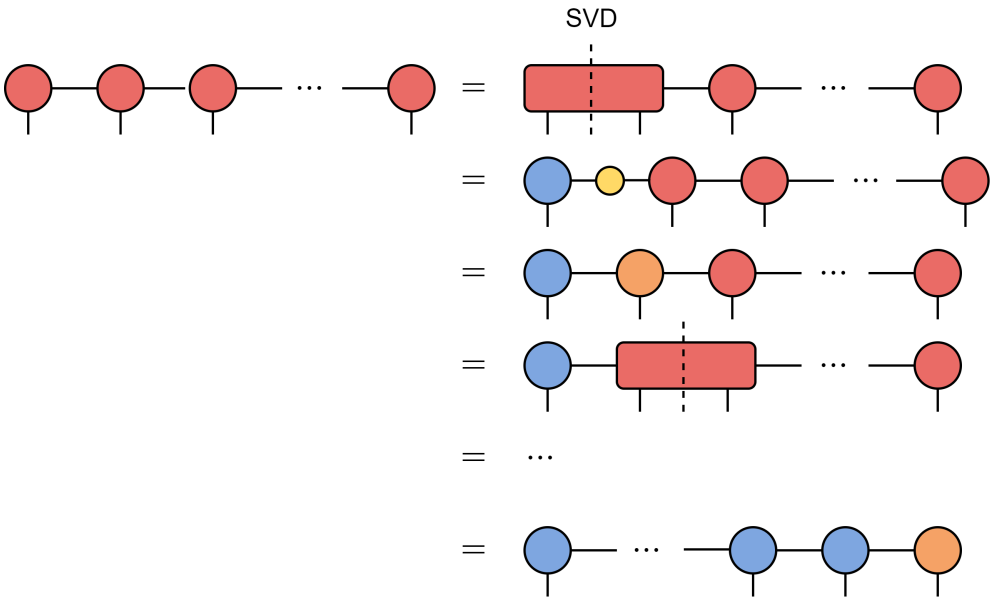
Approximating $P_d(\boldsymbol{x})$ can be achieved by maximising the likelihood of the training samples in the model probability distribution (see section 3.2). The loss function is then the mean negative log-likelihood (NLL)

$$\mathcal{L} = -\frac{1}{T} \sum_{i=1}^{T} \log P_m(\boldsymbol{x}^{(i)}) \,, \tag{4.4}$$

where T is the number of training samples. The loss $\mathcal{L}$ is a function of all parameters that define P_m. Here, the parameters are all tensor elements in the MPS (see Eq. (4.1)). To minimize $\mathcal{L}$ efficiently, we will use an adaptation of the celebrated density matrix renormalization group algorithm (DMRG) [63] with two-site update. The MPS is initialized with a small bond dimension and random tensor elements. How exactly the tensor elements are initialized is important and will be discussed in section 4.6. The MPS is then brought in the left canonical form (1.21) as follows

1. Form a merged tensor by contracting over the leftmost bond index b_1.
2. Reshape the merged tensor as a matrix (see Eq. (1.10)) and apply SVD. The leftmost tensor is now canonicalized as the U and V in Eq. (1.6) are orthogonal matrices.
3. Contract the matrix S containing the singular values into $A^{(2)}$, which is subsequently divided by its Frobenius norm.

This procedure is repeated until the MPS is in the left canonical form:

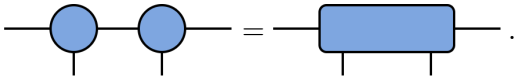


(4.5)

Where we used the following colour code: red is a tensor which is not canonicalized, blue is a canonicalized tensor, yellow is a matrix containing singular values and orange is a (not canonicalized) tensor contracted with singular values. Note that by normalizing the orange tensor, the left-canonical form now has $Z = 1$.

A DMRG two-site update starts by merging an adjacent pair of tensors

$$A^{(n)}_{j_n} A^{(n+1)}_{j_{n+1}} = A^{(n,n+1)}_{j_n j_{n+1}} \tag{4.6}$$



The parameters of the merged tensor are updated using gradient descent. To make notation easier, we will denote derivation to $A^{(n,n+1)}_{j_n j_{n+1}}$ as a prime

$$A^{(n,n+1)}_{j_n j_{n+1}} = A^{(n,n+1)}_{j_n j_{n+1}} - \gamma \mathcal{L}' \,, \tag{4.7}$$

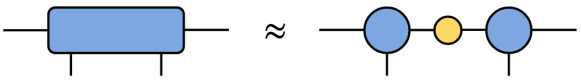
with the derivative given by

$$\mathcal{L}' = \left(-\frac{1}{T} \sum_{i=1}^{T} \log \frac{\left| \langle \boldsymbol{x}^{(i)} | \Psi \rangle \right|^2}{Z} \right)' \tag{4.8}$$

$$= \frac{Z'}{Z} - \frac{2}{T} \sum_{i=1}^{T} \frac{\langle \boldsymbol{x}^{(i)} | \Psi \rangle'}{\langle \boldsymbol{x}^{(i)} | \Psi \rangle} \,. \tag{4.9}$$

After the merged tensor has been updated using Eq. (4.7), it is subsequently decomposed using truncated SVD, where a singular value s_i is discarded if $s_i / s_1 < \epsilon_{\text{cut}}$. Here, s_1 is the largest singular value since the singular values are sorted in descending order and ϵ_{cut} is a hyperparameter which determines the cutoff. Note that this definition of truncated SVD

is slightly different from the usual one (Eq. (1.7)), where the t largest singular values are retained. The rows and columns which would be multiplied with the discarded singular values are also deleted, so the number of retained singular values corresponds to the bond dimension between the two tensors.

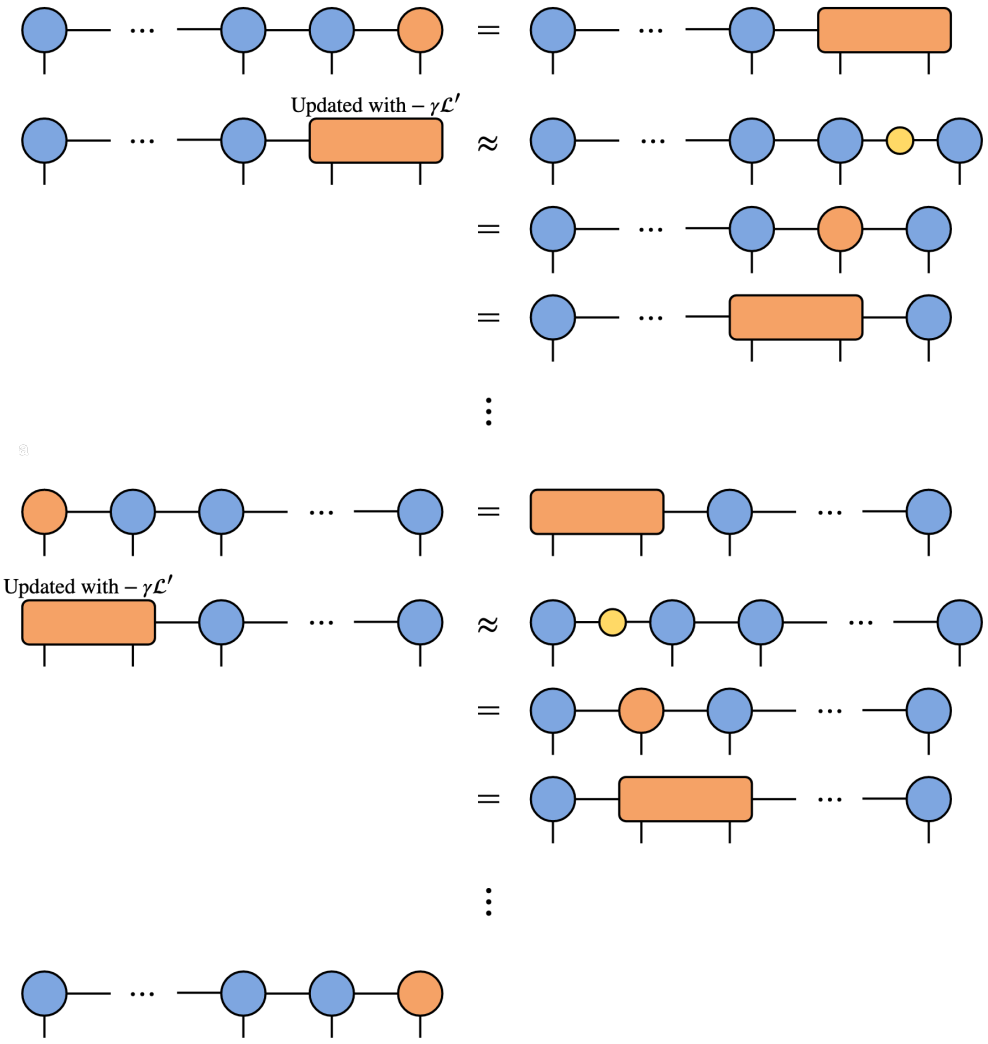


(4.10)

During training, the number of nonvanishing singular values, and therefore the bond dimensions, will generally increase as the MPS tries to capture more and more correlations in the data [49]. At most $D_{\max}$ singular values will be retained, with $D_{\max}$ a prescribed hyperparameter, meaning that s_i will be discarded when $i > D_{\max}$ even if $s_i/s_1 \geqslant \epsilon_{\text{cut}}$. This limit is set to avoid an excessively large bond dimension, which can cause overfitting and is computationally demanding.

The procedure of merging, updating and decomposing is successively applied for $n = N-1, N-2, \ldots, 1$. The singular values are each time contracted into the tensor to its left, which is then divided by its Frobenius norm to ensure that $Z = 1$. We then do the same for $n = 2, 3, \ldots, N-1$, sweeping back to its left-canonical from. Note that the singular values are now contracted into the right tensor after each truncated SVD.

In tensor diagram notation, the procedure is

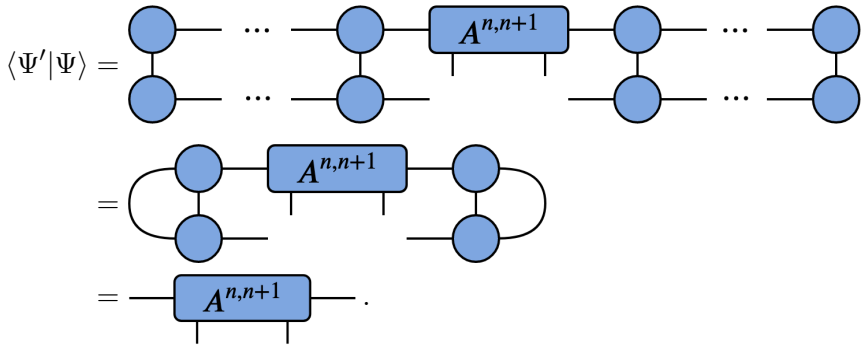


(4.11)

where orange is again used to indicate that the tensor is not canonical. After the back and forth sweep, also called an epoch, the MPS is again in its left-canonical form. All parameters in the MPS were updated using gradient descent and the number of parameters was possibly adapted as the bond dimension increased or decreased. Note that during an epoch (4.11), the MPS is always in left, mixed or right canonical form.

It is vital that $\mathcal{L}'$ can be computed efficiently because it will dictate how computationally expensive the training algorithm is. From Eq. (4.2) we obtain $Z' = 2\langle\Psi'|\Psi\rangle$. If the MPS is in the appropriate canonical form (right for $n = 1$, mixed for $1 < n < N - 1$ and left

for $n = N - 1$), the expression for Z' is greatly simplified

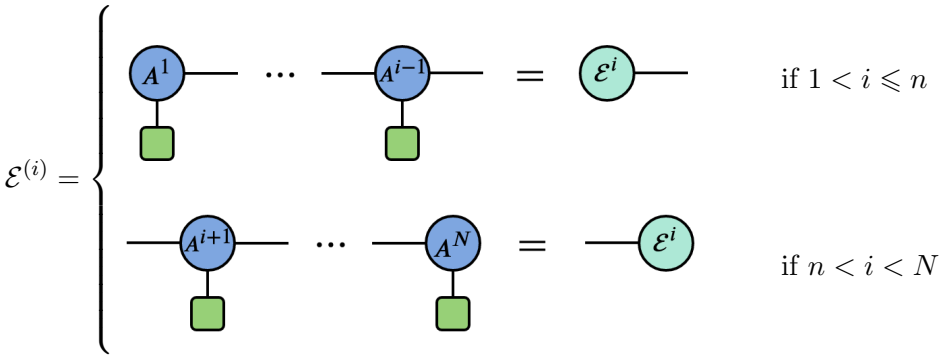


(4.12)

With $Z = 1$ and Eq. (4.12), the derivative (4.9) becomes

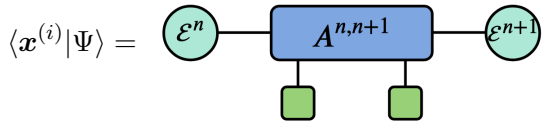
$$\mathcal{L}' = 2A^{(n,n+1)}_{j_n j_{n+1}} - \frac{2}{T}\sum_{i=1}^{T} \frac{\langle \boldsymbol{x}^{(i)}|\Psi\rangle'}{\langle \boldsymbol{x}^{(i)}|\Psi\rangle}\,. \tag{4.13}$$

Naively calculating the second term in Eq. (4.13) from scratch would require to contract the MPS with T samples in each gradient step. This can be avoided by keeping an environment $\mathcal{E}$ in memory which is updated after each gradient step. Entry i of the list $\mathcal{E}$ when updating $A^{(n,n+1)}_{j_n j_{n+1}}$ is defined as

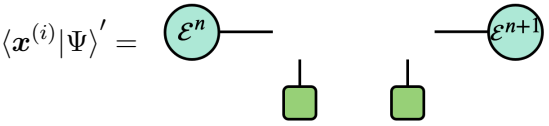


(4.14)

Let $\boldsymbol{t}$ be a vector of length T with all components 1, the first and last entry of the environment are then respectively $\mathcal{E}^{(1)} = \boldsymbol{t}$ and $\mathcal{E}^{(N)} = \boldsymbol{t}^{\mathsf{T}}$. Note that we have T environments since each sample consists of different features, represented by the green squares in Eq. (4.14). We can now easily compute the second term in Eq. (4.13) using

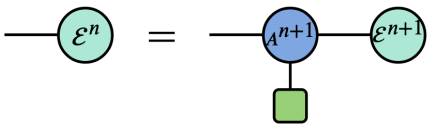


(4.15)



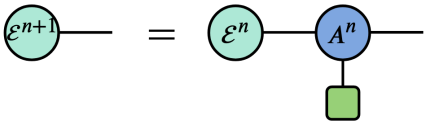
(4.16)

After gradient descent has been performed on the merged matrix $A^{(n,n+1)}_{j_n j_{n+1}}$, the environments must be updated so Eqs. (4.15) and (4.16) can be used in the next gradient descent step. When the merged tensor has been decomposed into $A^{(n)}_{j_n}$ and $A^{(n+1)}_{j_{n+1}}$, if sweeping leftward the entry n of the environment is changed to



. (4.17)

When sweeping rightward, we instead update $\mathcal{E}^{(n+1)}$ as



. (4.18)

Determining the gradient $\mathcal{L}'$ now requires a minimal amount of computations, allowing the MPS to be trained time-efficiently.

The two-site update enables the MPS to dynamically adapt the bond dimensions to the complexity of the data during training. This allows the model to focus computational resources on the most important features of the data, as will be discussed in section 5.3.3. Since the number of parameters in the MPS is not fixed, it is a so-called non-parametric model. The successful k-nearest neighbours algorithms [64] and certain support vector machines [65] are other examples of non-parametric models in machine learning. This is in contrast to parametric models such as neural networks, where the number of parameters must be fixed before the training process.

4.4 Sampling

Generating new samples from the model probability distribution is not necessarily straightforward for generative models. For instance, Boltzmann Machines rely on Markov Chain Monte Carlo (MCMC) for sampling [66]. Although many variants of MCMC exist, the algorithm generally starts from initial $\boldsymbol{x}^{(t)}$ and randomly changes the values of features to obtain $\boldsymbol{x}^{(t+1)}$. The ratio $P_m(\boldsymbol{x}^{(t+1)})/P_m(\boldsymbol{x}^{(t)})$ is then used to decide whether or not the changes are accepted. Repeating this procedure results in a random walk in feature space. Some MCMC methods guarantee convergence to a sample from $P_m(\boldsymbol{x})$, however this can require many computationally expensive steps and the generated samples are not necessarily independent [36]. This so-called *slow mixing problem* is especially relevant for the high-dimensional features spaces commonly found in machine learning tasks [67].

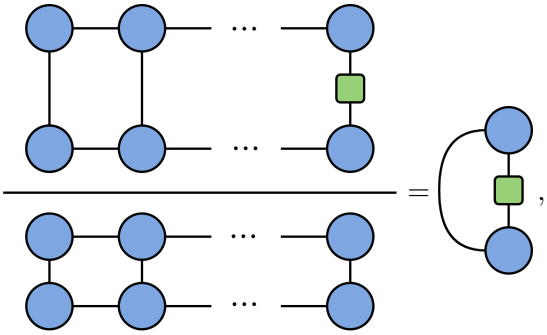
Fortunately, an MPS explicitly defines the model probability distribution and has a tractable partition function, allowing us to directly sample from $P_m(\boldsymbol{x})$. The generation process uses an autoregressive approach, where the chain rule of probability is used to decompose the joint probability distribution of the features into a product of one-dimensional

conditional probability distributions

$$P_m(\boldsymbol{x}) = P_m(x_N, \ldots, x_1) = \prod_{i=1}^{N} P_m(x_i | x_{i-1}, \ldots, x_1)\,. \tag{4.19}$$

In the left canonical form, it is natural to start from the Nth feature because the marginal probability $P_m(x_N)$ simplifies to

$$P_m(x_N) = \sum_{x_1=1}^{p} \ldots \sum_{x_{N-1}=1}^{p} P_m(\boldsymbol{x}) =$$

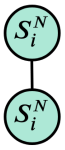


$$=$$

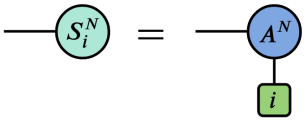
, (4.20)

where we used Eq. (4.3) and $Z = 1$. The green square is a diagonal matrix with the one-hot encoded x_N as its diagonal. For simplicity, we will assume that the p possible values of each feature are $1, 2, \ldots, p$. In practice, we compute

$$P_m(x_N = i) =$$



with



(4.21)

for $i = 1, \ldots, p-1$. With the right diagram in Eq. (4.21) we mean that $S^{(N)}$ is the ith column of the $d_{N-1} \times p$ matrix $A^{(N)}$. The probability of $x_N = p$ is determined using

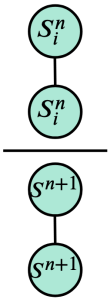
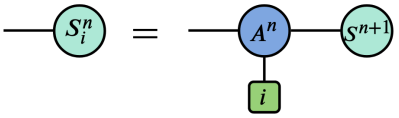
$$P_m(x_N = p) = 1 - \sum_{i=1}^{p-1} P_m(x_N = i)\,. \tag{4.22}$$

A value $c \in \{1, \ldots, p\}$ for x_N is then drawn from these probabilities and only the state $S_c^{(N)} = S^{(N)}$ is kept in memory. Therefore the computation of $S_p^{(n)}$ is only required if $x_n = p$ is chosen. This is not very useful for $n = N$ as we simply need to select column p from $A^{(N)}$, however for the following features the computation of $S_p^{(n)}$ is non-trivial.

Now that x_N is determined, we can obtain the probability mass function of feature $N-1$ using the definition of conditional probability

$$P_m(x_n | x_{n+1}, \ldots, x_N) = \frac{P_m(x_n, x_{n+1}, \ldots, x_N)}{P_m(x_{n+1}, \ldots, x_N)}\,. \tag{4.23}$$

In tensor diagram notation, this becomes

$P_m(x_n = i|x_{n+1}, \ldots, x_N) =$  with 

(4.24)

and with $n = N - 1$. If we again set $S^{(N-1)}_{c'} = S^{(N-1)}$, with $c' \in \{1, \ldots, p\}$ the drawn value for feature x_{N-1}, an entire sample can be generated by repeating Eq. (4.24) for $n = N - 2, \ldots, 1$.

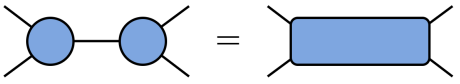
Besides generating samples from scratch, an MPS can also be used for missing data imputation. For instance if the boundaries of a spin system are given, the entire spin configuration according to $P_m(\boldsymbol{x})$ can be inferred from it. Unfortunately, canonicalization can not be used to simplify the expression of marginal probabilities (see Eq. (4.20)) if a segment of missing features is bordered by given features [49].

4.5 Time complexity

To get an idea of the scalability of the generative model discussed in section 4.3, it is informing to look at the asymptotic time complexity of the training and sampling algorithms. Here we will consider an MPS with bond dimension D and a dataset of T training samples, each with N sites with a physical dimension of p. Thanks to the environment defined in Eq. (4.14), the computational requirements of each update of the parameters in a merged tensor is independent of N. Since N merged tensors are updated per sweep, the complexity of the training algorithm scales as $\mathcal{O}(N)$. The training also scales linearly in the number of training samples. The number of operations required to multiply a $l \times n$ matrix with a $n \times m$ matrix is of the order lnm. This can be seen by considering that the resulting matrix contains lm elements and that each of those elements is the result of an inner product of two length n vectors. If we assume all bond dimensions of the MPS are D, we need to contract over a common bond index of two adjacent $D \times p \times D$ tensors to form a merged tensor. By reshaping the tensors to $(pD) \times D$ and $D \times (pD)$ matrices using Eq. (1.9), we can see that this computation is equivalent to a matrix multiplication which scales as $\mathcal{O}(p^2 D^3)$. The time complexity of performing compact SVD on a $m \times n$ matrix is $\mathcal{O}(mn^2)$ (if $m \geqslant n$, otherwise the complexity is $\mathcal{O}(m^2 n)$) [11]. Therefore decomposing a merged matrix with dimension $D \times p \times p \times D$, which is first reshaped to a $(pD) \times (pD)$ matrix, scales as $\mathcal{O}(p^3 D^3)$. This dwarfs the complexity of other operations performed during training, e.g. contracting over the physical index of a

tensor scales as $\mathcal{O}(pD^2)$. Generating q samples entails qN operations of the form (4.24), meaning the sampling scales as $\mathcal{O}(qNpD^2)$.

As we have seen in section 1.4, an MPS can be seen as a TTN with coordination number $c = 2$. Using subsequent SVD's, a TTN can also be brought in canonical form with a method analogous to Eq. (4.5). An optimization algorithm similar to the procedure described in section 4.3 can train a TTN, as was already implemented by Ref. [50] for a TTN with $c = 3$. Since c determines the order of the tensors in the network, it will impact the complexity of the optimization algorithm significantly. In the case of $c = 3$, the most expensive operations are of the form



(4.25)

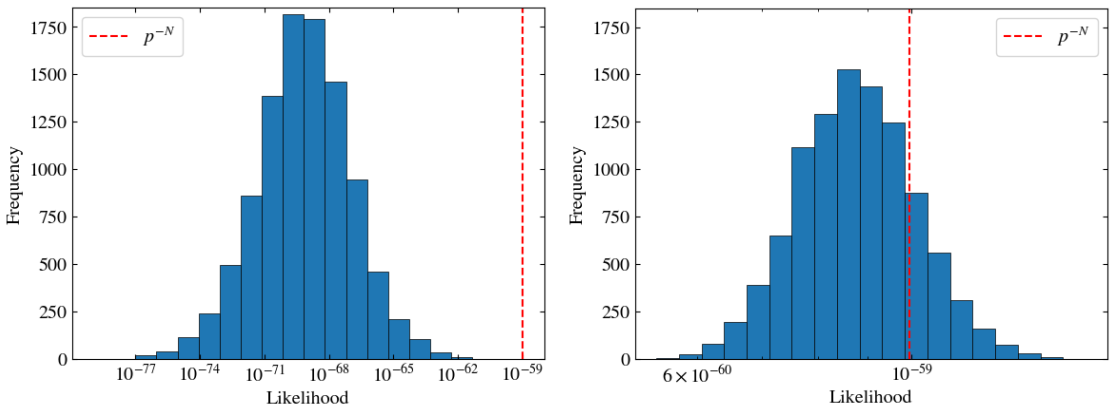
where we either contract the common index to form a merged tensor or apply SVD to decompose it after an update. If all tensors have bond dimension D, the operations scale respectively as $\mathcal{O}(D^5)$ and $\mathcal{O}(D^6)$. If one of the tensors is a leaf of the tree, the scaling becomes respectively $\mathcal{O}(p^2D^3)$ and, assuming that $D \geqslant p$, $\mathcal{O}(p^4D^2)$. Note that at most one of the two tensors has dimensions $D \times p \times p$ since the leaves are not directly connected (see Fig. 1.1). Despite the generally higher computational demands of a TTN, it is also a promising alternative for MPS in machine learning applications, mainly due to its power law decay of correlations (see section 1.4) [50].

4.6 Additions to the code

To develop the algorithms described above in Python, we first used the TensorNetwork library [68] which is being developed by Google. Although it is a promising framework to encourage the use of tensor networks for physics and machine learning applications, it is currently too unstable and inflexible for our purposes. Instead, we used and extended the code of Ref. [49]. Most notably, we generalized the code to learn from and generate data with arbitrary p, whereas it was previously limited to $p = 2$ (binary data). We also modified the initialization of the tensor elements, which are the parameters of the MPS model. These were previously drawn uniformly from the interval $[0, 1]$

$$\begin{gathered}\forall n, b_{n-1}, j_n, b_n \in \mathbb{N}_0;\ 1 \leqslant n \leqslant N;\ 1 \leqslant b_{n-1} \leqslant d_{n-1};\ 1 \leqslant j_n \leqslant p;\ 1 \leqslant b_n \leqslant d_n : \\ A^{(n)}_{b_{n-1} j_n b_n} = \epsilon \qquad \text{with} \qquad \epsilon \sim \mathcal{U}(0, 1)\,.\end{gathered} \tag{4.26}$$

However, in Fig. 4.1 (a) we see that an MPS with parameters given by Eq. (4.26) shows a wide range of likelihoods of generating different samples of the downscaled MNIST dataset, which will be introduced in section 5.3.1. Besides the large difference between the probability of generating the most and least likely sample, the model also performs significantly worse than what we would expect from randomly generating samples. There are p^N possibilities, so generating samples at random should assign a likelihood of p^{-N}



(a) (b)

Figure 4.1 The distribution of the likelihood of 10000 downscaled MNIST images in an initialized MPS . (a) MPS initialized using Eq. (4.26); (b) MPS initialized via Eq. (4.27).

to each possibility. To resolve these issues, we instead used the initialization

$$\begin{aligned} &\forall n, b_{n-1}, j_n, b_n \in \mathbb{N}_0;\ 1 \leqslant n \leqslant N;\ 1 \leqslant b_{n-1} \leqslant d_{n-1};\ 1 \leqslant j_n \leqslant p;\ 1 \leqslant b_n \leqslant d_n : \\ &\qquad A^{(n)}_{b_{n-1} j_n b_n} = \delta_{b_{n-1} b_n} + \epsilon \qquad \text{with} \qquad \epsilon \sim \mathcal{N}(0, 10^{-3})\,, \end{aligned} \tag{4.27}$$

where $\mathcal{N}(\mu, \sigma)$ is a Gaussian with mean μ and standard deviation σ. The standard deviation of the Gaussian noise is a hyperparameter, however the algorithm is fairly insensitive to which σ is chosen. Any reasonable value $(10^{-1} - 10^{-6})$ for σ gives very similar results. If we initialize the MPS with the same dimension D for all bonds and we look at a tensor $A^{(n)}$ as an array of p matrices, we see that Eq. (4.27) means we initialize each of these matrices as the identity matrix + Gaussian noise. Using Eq. (4.27), all samples now have a similar likelihood close to p^{-N} (see Fig. 4.1). Gradient descent finds a local minimum of the loss function, so the initialization is important as it dictates the starting point of the optimization in parameter space [14]. Indeed, in section 5.2 we will see that an MPS initialized with Eq. (4.27) obtains a lower NLL at convergence compared to an MPS initialized with Eq. (4.26).

Finally we improved the computational performance of the algorithm, for instance by parallelizing the sampling procedure. The procedure to generate a single sample is inherently sequential, since we need $x_{n+1}, \ldots, x_N$ to sample x_n via Eq. (4.24). However when generating multiple samples, we now sample each x_n simultaneously instead of generating the images one by one. The code was also modified to work both on CPU and GPU, although the required memory is often too large for a single GPU.

Chapter 5

Results

5.1 Adverserial learning with MPS

The initial goal of this thesis is to design an algorithm that uses adversarial learning with tensor networks as generator and discriminator. In the literature, adversarial learning can refer to a regularization method for classifiers where adversarial samples are added to its training set [69]. Adversarial samples are training samples which are perturbed in such a way that they are misclassified. By adding adversarial samples to the training set, the classifier is more resistant against small input perturbations (e.g. noise) [69]. In this thesis, the term adversarial learning refers to the procedure of training a generative model using a discriminator, as described in section 3.3.

Considering the limitations of maximizing likelihood (see section 3.2.1) and both the resistance of GANs against overfitting and its ability to produce realistic samples (see section 3.3), adversarial learning could prove a useful alternative to maximum likelihood in order to train a tensor network as generative model. Meanwhile, having an MPS as generator allows us to explicitly represent the model distribution, which could resolve some theoretical and practical issues relating GANs. The generator of a GAN does not explicitly represent the a model distribution P_m and is in fact a mapping from latent variables to features of a sample. Therefore an MPS as generator has a more straightforward interpretation and allows us to determine the likelihood of generating a given sample. The likelihood of test samples is an important performance measure for generative models. This is the major reason why implicit models are difficult to objectively evaluate and compare to other models. Furthermore, having access to the likelihood could help detect mode collapse and permit a new way to investigate why it is such a common problem in adversarial learning [36].

Another advantage of explicit models is that we can complete samples of which some features are already known (see section 4.4), allowing the model to perform for example image reconstruction or time series prediction. In GANs, there is no straightforward way to obtain values for the latent variables $\boldsymbol{z}$ corresponding to some given features. Since the generator in GANs requires $\boldsymbol{z}$ as input, it is limited to synthesis tasks. Some

extensions of GANs do allow an approximate mapping from feature space to latent space by additionally training an encoder [70]. Recently, flow-based generative models have been proposed which use an invertible neural network as generator [71]. These models allow us to determine the exact likelihood of given samples, similar to MPS, and have an exact feature space to latent space mapping, namely g^{-1}. Although flow-based generative models are usually trained using maximum likelihood estimation, they have also been extended to adversarial training [72].

The unsupervised learning problem of estimating $P_d(\boldsymbol{x})$ can be reduced to that of learning a probabilistic binary classifier that distinguishes between samples from $P_d(\boldsymbol{x})$ and the model distribution $P_m(\boldsymbol{x})$ [15]. This supervised learning task can be solved using the maximum likelihood estimation, which consists of maximizing

$$v = \mathbb{E}_{\boldsymbol{x}\sim P_d}\left[\log P_c(y=1|\boldsymbol{x})\right] + \mathbb{E}_{\boldsymbol{x}\sim P_m}\left[\log P_c(y=0|\boldsymbol{x})\right] , \tag{5.1}$$

where $P_c(y=1|\boldsymbol{x})$ the probability assigned by the classifier that $\boldsymbol{x}$ comes from $P_d(\boldsymbol{x})$ rather than $P_m(\boldsymbol{x})$. The idea behind GANs is that a good generative model should generate samples that no classifier can distinguish from samples of the dataset [38]. In a GAN, $P_c(y=1|\boldsymbol{x})$ is modelled by the discriminator $d(\boldsymbol{x})$. Recall that the discriminator and generator take turns to respectively maximize and minimize the objective function, which is given by

$$v_{\text{GAN}} = \mathbb{E}_{\boldsymbol{x}\sim P_d}\left[\log d(\boldsymbol{x})\right] + \mathbb{E}_{\boldsymbol{z}\sim P_{\boldsymbol{z}}}\left[\log(1-d(g(\boldsymbol{z})))\right] . \tag{5.2}$$

Alternatively, the generator can be trained by maximizing $\mathbb{E}_{\boldsymbol{z}\sim P_{\boldsymbol{z}}}\left[\log d(g(\boldsymbol{z}))\right]$ to avoid vanishing gradients during early learning (see section 3.3).

In an attempt to train an MPS with an adversarial loss instead of maxmimum likelihood estimation, we proposed the objective

$$v_1 = \mathbb{E}_{\boldsymbol{x}\sim P_d}\left[\log \frac{\langle \boldsymbol{x}|d\rangle^2}{\overline{Z}}\right] - \log\langle g|d\rangle^2 , \tag{5.3}$$

with $|d\rangle$ and $|g\rangle$ the discriminator and generator represented as MPSs, normalized so that $\langle d|d\rangle = \langle g|g\rangle = 1$. The partition function is given by $\overline{Z} = \sum_{\boldsymbol{x}}\langle \boldsymbol{x}|d\rangle^2$. Like in GANs, the optimal parameters of $|g\rangle$ are found with Eq. (3.22). Since $|g\rangle$ explicitly represents P_m via Eq. (4.1) instead of the implicit generator in GANs, we no longer have latent variables $\boldsymbol{z}$. Since Eq. (5.3) does not contain an expectation value with respect to $\boldsymbol{x}\sim P_m$, generating samples during training is not required. The downside is that we need to compute $\langle g|d\rangle$. In practice, this contraction of two different MPSs can actually be more computationally expensive than generating samples. Both $|d\rangle$ and $|g\rangle$ can take turns either maximizing or minimizing v_1 using gradient descent with an algorithm based on DMRG, analogous to the optimization described in section 4.3. An extra environment $\widetilde{\mathcal{E}}$ was introduced to efficiently compute the gradient with respect to $\log\langle g|d\rangle^2$ for each merged tensor. The environment of the contraction between the two MPS is defined analogously to Eq. (4.14), except that each entry of $\widetilde{\mathcal{E}}$ is now a matrix instead of a vector.

Figure 5.1 The 30 binary images of the bars and stripes dataset.

Figure 5.2 Example of images generated by a model which suffers from mode collapse. The generator's output is clearly less varied than would be expected from the original bars and stripes dataset in Fig. 5.1. The model was trained using Eq. (5.3) and hyperparameters $D_{\max} = 16$, $\gamma = 0.1$ and $\epsilon_{\text{cut}} = 10^{-7}$.

Although this method converges, the mean negative log-likelihood (NLL) of the train and test set obtained by the generator are consistently higher than those of an MPS trained with maximum likelihood estimation. Additionally, the learning rate must be fine-tuned to avoid mode collapse. Even when trained on the simple bars and stripes dataset, it is common that the likelihood of generating some training samples reduces to zero during training. The used bars and stripes dataset consists of 30 binary images which each have 16 pixels, as shown in Fig. 5.1. The toy dataset requires almost no computational resources, making it suitable for testing new code and for hyperparameter tuning. Despite the simplicity of bars and stripes, a generative model trained on it must learn to generate 30 images out of the 2^{16} possibilities. This makes it a useful proof of concept for a model. An example of the output of a mode collapsed generator is shown in Fig. 5.2.

Looking more closely at the expression for v_1 proposed in Eq. (5.3) reveals some issues that explain the poor results. Since $-\log(x)$ is a convex function for $x \in]0, \infty[$ and we assume $\langle g|d\rangle \neq 0$, we can use the Jensen's inequality (Eq. (3.12)) to rewrite the second term of Eq. (5.3) as

$$\log \langle g|d\rangle^2 = \log \mathbb{E}_{\boldsymbol{x}\sim P_m} \left[\langle \boldsymbol{x}|d\rangle^2\right] \tag{5.4}$$

$$\geqslant \mathbb{E}_{\boldsymbol{x}\sim P_m} \left[\log \langle \boldsymbol{x}|d\rangle^2\right] . \tag{5.5}$$

The generator is trained to minimize v_1, corresponding to maximizing $\log \langle g|d\rangle^2$. It is now apparent that we are maximizing an upper bound of the function we should be maximizing, namely the expectation value of the logarithm as in Eq. (5.1). As maximizing an upper bound of a function does not necessarily correspond to maximizing the function itself, the proposed expression for v_1 is flawed. This is also indicated by looking at the optimal generator for Eq. (5.3). For a fixed discriminator $|d\rangle$, the generators objective $\log \langle g|d\rangle^2$ is maximal if $|g\rangle = |d\rangle$. Therefore instead of optimizing the generator with gradient ascent,

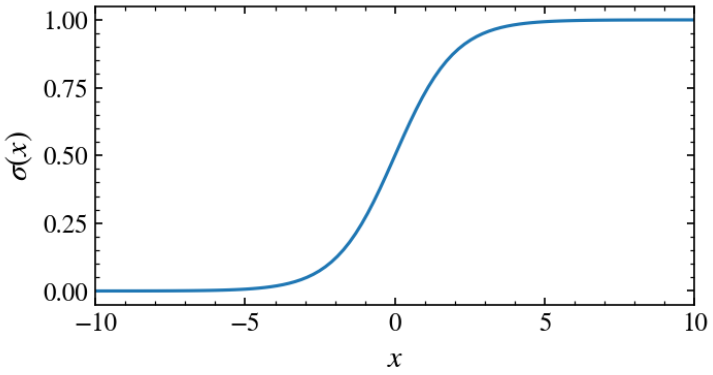


Figure 5.3 The logistic sigmoid function.

we can simply copy the discriminator MPS as generator. As $\langle d|d\rangle = 1$, the second term in Eq. (5.3) is then zero and we are left with the familiar maximum log likelihood objective Eq. (4.4). Note that copying the discriminator as generator is not possible in GANs because d and g are completely different functions: d maps features $\boldsymbol{x}$ to a scalar, while the g maps $\boldsymbol{z}$ to $\boldsymbol{x}$.

In an attempt to resolve the flaws of Eq. (5.3), we propose the following expression for the objective

$$v_2 = \mathbb{E}_{\boldsymbol{x}\sim P_d}\left[\log \frac{\langle \boldsymbol{x}|d\rangle^2}{\overline{Z}}\right] + \mathbb{E}_{\boldsymbol{x}\sim P_m}\left[\log\left(1 - \frac{\langle \boldsymbol{x}|d\rangle^2}{\overline{Z}}\right)\right]. \tag{5.6}$$

After each epoch (4.11) of training the discriminator with gradient ascent, the generator which defines P_m via Eq. (4.1) is set to $|g\rangle = |d\rangle$. Although the expectation values are now positioned as in Eq. (5.1), the interpretation is still not the same. As $|d\rangle$ is normalised, $\langle \boldsymbol{x}|d\rangle^2 / \overline{Z}$ cannot represent $P_c(y = 1|\boldsymbol{x})$ from Eq. (5.1), which is the probability that a sample $\boldsymbol{x}$ is drawn from P_d. For instance if P_m does not resemble P_d, we want $P_c(y = 1|\boldsymbol{x})^{(i)} \approx 1$ for all training samples $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(T)}$. This is not possible for $\langle \boldsymbol{x}|d\rangle^2 / \overline{Z}$ due to the constraint $\langle d|d\rangle = 1$. When letting go of this constraint, the tensor elements and therefore the norm of the discriminator tend to diverge.

A common method to represent a probability using a model is to apply the logistic sigmoid function, defined as

$$\sigma(x) = \frac{1}{1 + e^{-x}}, \tag{5.7}$$

as the final step of a model. The sigmoid $\sigma(x)$ guarantees that the output of the model will be in the interval $[0, 1]$, as can be seen in Fig. 5.3. The discriminator neural network in a GAN uses $\sigma(x)$ as its output function. We will use the notation

$$P_{\overline{m}}(\boldsymbol{x}) = \frac{\langle \boldsymbol{x}|d\rangle^2}{\overline{Z}} \tag{5.8}$$

$$P_m(\boldsymbol{x}) = \frac{\langle \boldsymbol{x}|g\rangle^2}{Z}\,, \tag{5.9}$$

where similar symbols are used because after each epoch we set $|g\rangle = |d\rangle$, or equivalently $P_m(\boldsymbol{x}) = P_{\overline{m}}(\boldsymbol{x})$. Due to the properties of the logistic sigmoid, the following function is suitable to represent the probability that $\boldsymbol{x}$ is drawn from the dataset

$$P_c(y=1|\boldsymbol{x}) = \sigma\left(\log \frac{P_{\overline{m}}(\boldsymbol{x})}{P_m(\boldsymbol{x})}\right) \tag{5.10}$$

$$= \frac{1}{1+\exp\left(-\log \frac{P_{\overline{m}}(\boldsymbol{x})}{P_m(\boldsymbol{x})}\right)} \tag{5.11}$$

$$= \frac{1}{1+\frac{P_m(\boldsymbol{x})}{P_{\overline{m}}(\boldsymbol{x})}} \tag{5.12}$$

$$= \frac{P_{\overline{m}}(\boldsymbol{x})}{P_{\overline{m}}(\boldsymbol{x})+P_m(\boldsymbol{x})}\,. \tag{5.13}$$

Using Eq. (5.1), our objective function is now given by

$$v_3 = \mathbb{E}_{\boldsymbol{x}\sim P_d}\left[\log P_c(y=1|\boldsymbol{x})\right] + \mathbb{E}_{\boldsymbol{x}\sim P_m}\left[\log(1-P_c(y=1|\boldsymbol{x}))\right] \tag{5.14}$$

$$= \mathbb{E}_{\boldsymbol{x}\sim P_d}\left[\log \frac{P_{\overline{m}}(\boldsymbol{x})}{P_{\overline{m}}(\boldsymbol{x})+P_m(\boldsymbol{x})}\right] + \mathbb{E}_{\boldsymbol{x}\sim P_m}\left[\log \frac{P_m(\boldsymbol{x})}{P_{\overline{m}}(\boldsymbol{x})+P_m(\boldsymbol{x})}\right]. \tag{5.15}$$

This objective function v_3 is a variant of the objective of noise contrastive estimation (NCE) [73], where a generative model is trained by learning it to distinguish training samples from samples drawn from a fixed noise distribution [15]. The performance of NCE strongly depends on the choice of noise distribution [74]. A poor choice of noise distribution means its samples are likely to be too obviously distinct from the data, in which case our model is not forced to adequately capture P_d [15]. We circumvent this difficulty in Eq. (5.15) by using samples generated by the model P_m itself as noise samples. Unfortunately, this training procedure is not useful since its expected gradient is identical to the expected gradient of maximum likelihood [74]. To see this, we simplify the gradient of Eq. (5.15), given by

$$\begin{aligned}\frac{\partial v_3}{\partial \boldsymbol{\theta}^{(d)}} = \frac{\partial}{\partial \boldsymbol{\theta}^{(d)}}\Big(&\mathbb{E}_{\boldsymbol{x}\sim P_d}\left[\log P_{\overline{m}}(\boldsymbol{x}) - \log(P_{\overline{m}}(\boldsymbol{x})+P_m(\boldsymbol{x}))\right] \\ &+ \mathbb{E}_{\boldsymbol{x}\sim P_m}\left[\log P_m(\boldsymbol{x}) - \log(P_{\overline{m}}(\boldsymbol{x})+P_m(\boldsymbol{x}))\right]\Big)\,,\end{aligned} \tag{5.16}$$

with $\boldsymbol{\theta}^{(d)}$ the parameters of $|d\rangle$. It is important to note that, even though we set $P_m = P_{\overline{m}}$ at the end of each epoch, P_m is kept fixed during an epoch, meaning that

$$\frac{\partial P_m}{\partial \boldsymbol{\theta}^{(d)}} = 0\,. \tag{5.17}$$

Therefore Eq. (5.16) can be rewritten as

$$\begin{aligned}\frac{\partial v_3}{\partial \boldsymbol{\theta}^{(d)}} = \mathbb{E}_{\boldsymbol{x}\sim P_d}\left[\frac{\partial}{\partial \boldsymbol{\theta}^{(d)}}\Big(\log P_{\overline{m}}(\boldsymbol{x}) - \log(P_{\overline{m}}(\boldsymbol{x}) + P_m(\boldsymbol{x}))\Big)\right] \\ - \mathbb{E}_{\boldsymbol{x}\sim P_m}\left[\frac{\partial}{\partial \boldsymbol{\theta}^{(d)}}\log(P_{\overline{m}}(\boldsymbol{x}) + P_m(\boldsymbol{x}))\right].\end{aligned} \tag{5.18}$$

Assuming $P_{\overline{m}}(\boldsymbol{x})$ is nonzero, we can rewrite the second term as

$$\mathbb{E}_{\boldsymbol{x}\sim P_m}\left[\frac{\partial}{\partial \boldsymbol{\theta}^{(d)}}\log(P_{\overline{m}}(\boldsymbol{x}) + P_m(\boldsymbol{x}))\right] = \mathbb{E}_{\boldsymbol{x}\sim P_m}\left[\frac{\frac{\partial}{\partial \boldsymbol{\theta}^{(d)}}P_{\overline{m}}(\boldsymbol{x})}{P_{\overline{m}}(\boldsymbol{x}) + P_m(\boldsymbol{x})}\right] \tag{5.19}$$

$$= \mathbb{E}_{\boldsymbol{x}\sim P_m}\left[\frac{\frac{\partial}{\partial \boldsymbol{\theta}^{(d)}}\exp\left(\log P_{\overline{m}}(\boldsymbol{x})\right)}{P_{\overline{m}}(\boldsymbol{x}) + P_m(\boldsymbol{x})}\right] \tag{5.20}$$

$$= \mathbb{E}_{\boldsymbol{x}\sim P_m}\left[\frac{P_{\overline{m}}(\boldsymbol{x})\frac{\partial}{\partial \boldsymbol{\theta}^{(d)}}\log P_{\overline{m}}(\boldsymbol{x})}{P_{\overline{m}}(\boldsymbol{x}) + P_m(\boldsymbol{x})}\right] \tag{5.21}$$

$$= \frac{1}{2}\mathbb{E}_{\boldsymbol{x}\sim P_m}\left[\frac{\partial}{\partial \boldsymbol{\theta}^{(d)}}\log P_{\overline{m}}(\boldsymbol{x})\right]. \tag{5.22}$$

In the last step we used the fact that, outside of the differentiation, $P_m(\boldsymbol{x}) = P_{\overline{m}}(\boldsymbol{x})$ if we set $|g\rangle = |d\rangle$ after each gradient step. This can further be used to show that Eq. (5.22) is zero

$$\frac{1}{2}\mathbb{E}_{\boldsymbol{x}\sim P_m}\left[\frac{\partial}{\partial \boldsymbol{\theta}^{(d)}}\log P_{\overline{m}}(\boldsymbol{x})\right] = \sum_{\boldsymbol{x}} P_m(\boldsymbol{x})\frac{\frac{\partial}{\partial \boldsymbol{\theta}^{(d)}}P_{\overline{m}}(\boldsymbol{x})}{P_{\overline{m}}(\boldsymbol{x})} \tag{5.23}$$

$$= \frac{\partial}{\partial \boldsymbol{\theta}^{(d)}}\sum_{\boldsymbol{x}} P_{\overline{m}}(\boldsymbol{x}) \tag{5.24}$$

$$= \frac{\partial}{\partial \boldsymbol{\theta}^{(d)}}1 = 0\,. \tag{5.25}$$

From Eq. (5.22), we can also see that

$$\mathbb{E}_{\boldsymbol{x}\sim P_d}\left[\frac{\partial}{\partial \boldsymbol{\theta}^{(d)}}\log(P_{\overline{m}}(\boldsymbol{x}) + P_m(\boldsymbol{x}))\right] = \frac{1}{2}\mathbb{E}_{\boldsymbol{x}\sim P_d}\left[\frac{\partial}{\partial \boldsymbol{\theta}^{(d)}}\log P_{\overline{m}}(\boldsymbol{x})\right]. \tag{5.26}$$

Using Eq. (5.25) and (5.26), the gradient (5.18) reduces to

$$\frac{\partial v_3}{\partial \boldsymbol{\theta}^{(d)}} = \frac{1}{2}\mathbb{E}_{\boldsymbol{x}\sim P_d}\left[\frac{\partial}{\partial \boldsymbol{\theta}^{(d)}}\log P_{\overline{m}}(\boldsymbol{x})\right]. \tag{5.27}$$

If we absorb the factor $\frac{1}{2}$ in the learning rate, the obtained gradient is equivalent to that of maximum likelihood (see Eq. (3.4)). We previously set $|g\rangle = |d\rangle$ after every epoch instead of every gradient step, which is every merged tensor update for the algorithm described in section 4.3. Therefore, even outside of the differentiation, P_m is not equal to $P_{\overline{m}}$ and thus result (5.27) is not valid. This explains why our trained models performed similar but consistently worse, e.g. in terms of NLL, than those trained with maximum likelihood.

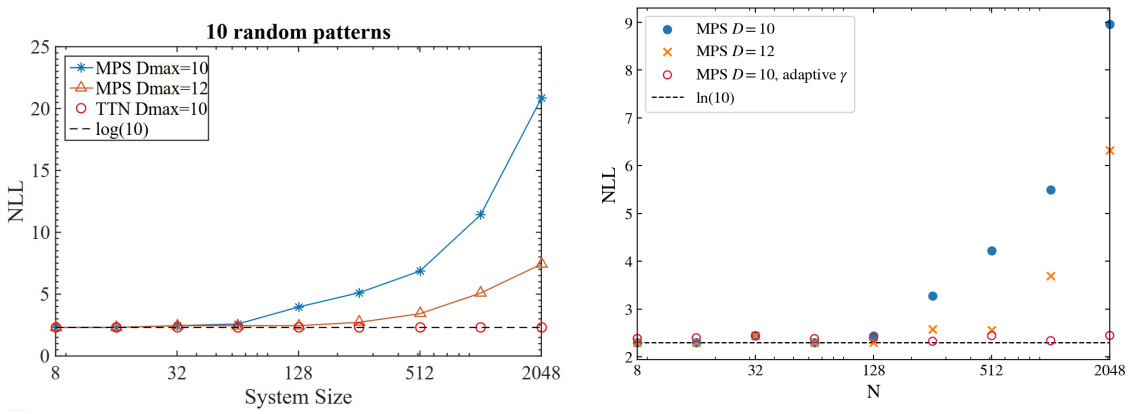
Our effort to use adversarial learning for MPS resulted in the objective function (5.15), which was shown to be equivalent to maximum likelihood estimation. Unfortunately, our alternative formulation is less intuitive and provides no benefit over standard maximum likelihood. Moreover, one of the main benefits of both adverserial learning and NCE is that it avoids the need for computing the partition function of our generative model. Note that e.g. in our formulation and analysis of GANs (section 3.3), we were never required to compute Z. This is a major advantage over maximizing likelihood for models that generally have intractable partition functions, such as neural networks [15]. In contrast, the partition function of an MPS model is easy to compute, especially when taking advantage of the canonical form. Therefore all further results are obtained with the familiar maximum likelihood objective (4.4).

5.2 Memorizing random bits

Memorizing T specific arrangements of N random bits is a useful task to test the capacity (see section 2.3) of a generative model. Memorizing means that the likelihood of generating each of the T samples becomes $\frac{1}{T}$, assuming that all training samples are unique. This is equivalent to saying that the NLL has reached its theoretical lower bound $\log T$. When the lower bound is reached, the distribution of the model is exactly the same as the empirical distribution of the training data. The model will then generate identical copies of the training samples, with an equal probability assigned to each training sample. Note that this is not useful for any practical tasks, as the model is completely overfitting when it has memorized all training samples. Nonetheless, it is interesting to evaluate whether the optimization scheme is able to converge to the lower bound of the loss function, namely $\log T$.

It has been claimed in section III.A of Ref. [50] that an MPS with $D_{\max} = 10$ cannot be trained to memorize 10 samples of N random bits when the system size N is too large. Fig. 5.4 (a), which is copied from Ref. [50], shows this experimentally. This was explained by the fact that the correlation length in the MPS decays exponentially fast, which causes "the information contained in the middle bond to be more saturated when the system size becomes very large, making the maximum likelihood training less efficient" [50]. It is further argued that a TTN "is almost unaffected by system size because it can better capture the long range dependences". Fig. 5.4 (a) indeed shows that the TTN is able to minimize the NLL close to its lower bound. These results are unexpected since an MPS in superposition of T random samples can be determined using Schmidt decompositions, as described in section 1.3. This is equivalent to saying that the constructed MPS with $D = T$ has memorized T samples. Therefore a solution where the NLL reaches its lower bound is clearly in the hypothesis space of an MPS with $D = T$. Recall that this doesn't necessarily mean such a solution is found by our learning algorithm, since gradient descent only finds a local minimum of the loss function [14].

To further investigate the surprising results of Ref. [50], we attempted to reproduce their figure 5.4 (a). Our results, shown in Fig. 5.4 (b), have a significantly smaller NLL for an



(a) (b)

Figure 5.4 Experimental results of the mean NLL when trained on 10 different patterns of N random bits. The theoretical lower bound of the NLL for this task is log(10), shown as a dashed black line in both plots. (a) Plot taken from Ref. [50] where the performance of MPSs and TTNs is compared. (b) The blue dots and orange crosses show our results of the same MPSs used in plot (a). The red circles use the adaptive learning rate of Eq. (5.30) with $\beta = \frac{1}{4}$. Each MPS in figure (b) was trained for 200 epochs with $\gamma = 0.1$. The hyperparameters and the code used in plot (a) were not made public in Ref. [50].

MPS with both $D_{\max} = 10$ and $D_{\max} = 12$. When we initialize the tensor elements using Eq. (4.26) instead of Eq. (4.27), the obtained NLL are approximately 10% higher for each N. As this cannot explain the observed differences in NLL between Fig. 5.4 (a) and (b), it is likely that either the learning rate γ used to produce Fig. 5.4 (a) was not 0.1 and performed significantly worse, or that the MPSs were not trained to convergence. In our experiments, the models were trained for 200 epochs. The obtained NLLs are very close to the NLLs after just 100 epochs, so the training seems to have converged for each point in Fig. 5.4 (b). Another possibility is that our sequences of randomly generated bits are incidentally easier to memorize than those used in Ref. [50], although one would expect this becomes unlikely for large N.

Although the NLL reached by the MPSs in our experiments is consistently lower than those of Fig. 5.4 (a), it indeed doesn't reach its theoretical limit log(10) when the system size becomes too large. Since an MPS with $D = 10$ which is able reach this limit can be analytically determined, it is clear that the problem does not stem from the expressive capabilities of an MPS, but in the optimization scheme which doesn't converge to the lower bound of the loss function. It was already noted in Ref. [49] that the gradients of the loss function (4.4) can become very small even though the optimization is not the vicinity of a local minimum of the loss function. In that situation, a plateau or saddle point in the loss function may have been encountered [14]. Since the gradients in these regions become very small, it can take many epochs to traverse them. Besides being computationally inefficient, the vanishing gradients cause the model to stop improving as the parameters are barely changing, which could be mistaken for convergence. In machine learning literature, this phenomenon is often referred to as the vanishing gradient problem. Here we resolve the issue by increasing the learning rate so that the norm of the gradient

step is a function of the number of elements η in the merged tensor $A^{(n,n+1)}_{j_n j_{n+1}}$ which is being updated. In other words, Eq. (4.7) is adapted to

$$A^{(n,n+1)}_{j_n j_{n+1}} = A^{(n,n+1)}_{j_n j_{n+1}} - \gamma \eta^{\beta} \frac{\mathcal{L}'}{\|\mathcal{L}'\|} \tag{5.28}$$

$$= A^{(n,n+1)}_{j_n j_{n+1}} - \gamma_a \mathcal{L}' \,, \tag{5.29}$$

where we have defined the adaptive learning rate as

$$\gamma_a = \gamma \frac{\eta^{\beta}}{\|\mathcal{L}'\|} \,. \tag{5.30}$$

The downside of artificially increasing the norm of each gradient descent step by using the adaptive learning rate, is that the training can oscillate around a minimum of $\mathcal{L}$ instead of converging to it. Fig. 5.4 (b) shows that an MPS trained with the adaptive learning rate is able to almost perfectly memorize the 10 random samples. Also note the small deviations from log(10) which seem to be unrelated to the system size. These are probably caused by the mentioned oscillations around a minimum. The hyperparameters used for this result are $\beta = \frac{1}{4}$ and $\gamma = 0.1$. The value of β was chosen as a compromise: a larger β causes the oscillations around a minimum to increase, whereas a smaller β fails to traverse a saddle point or plateau in the loss function in a reasonable amount of epochs. The adaptive learning rate method could be further improved by letting γ_a decay over time. This would allow the training to converge to a minimum instead of oscillating around it because the gradient steps are too large. Unfortunately the TTN results in Fig. 5.4 (a) could not be reproduced because the code of Ref. [50] was not made public.

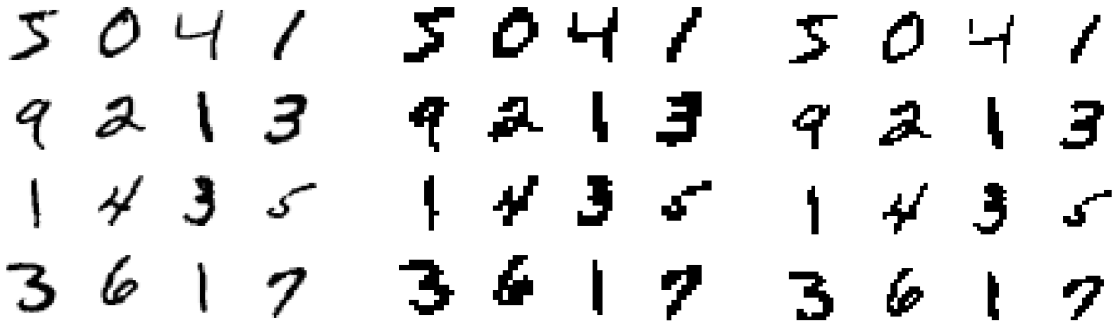
We can conclude that an MPS with $D_{\text{max}} = 10$ is able to memorize 10 samples of random bits, but that an adaptive learning rate is required to avoid getting stuck in a saddle point or plateau of the loss function. Furthermore our results show that a TTN is not necessarily more suitable for this task than an MPS for this task, as was claimed in Ref. [50]. Note that a TTN is still expected to outperform an MPS as a generative model for data with long range correlations because its correlation function shows an algebraic decay with separation distance, as opposed to exponential for an MPS (see sections 1.3.2 and 1.4).

5.3 Downscaled MNIST

5.3.1 Dataset

The MNIST dataset [75] is a standard benchmark for image recognition and generation in machine learning [14]. It consists of 60000 greyscale 28×28 images of handwritten digits. To limit the required computational resources, the images were downscaled to 14×14 pixels. Since $\boldsymbol{x}$ must be one-dimensional for an MPS, the images are flattened row by row into a vector of length 196. The ordering of the pixels is depicted in Fig. 5.5. On average, the zig-zag ordering keeps spatially neighboring pixels as close to each other as possible

Figure 5.5 Figure from Ref. [76] showing the ordering of the 14×14 images as one-dimensional vectors.



(a) (b) (c)

Figure 5.6 Results of downscaling the first 16 MNIST images: (a) the original 28×28 images; (b) images obtained by downscaling to 14×14 using max pooling followed by binarizing; (c) results of cropping to 20×20, then downscaling to 14×14 using a bicubic filter and finally binarizing.

along the one-dimensional path [76]. The greyscale images must then be binarized because the MPS can only generate discrete data. This is done by simply setting all pixel values larger than the threshold 0.5 to 1 and the rest to 0. The downscaling can be performed easily by 2×2 max pooling or average pooling, where respectively the maximum or the average of all 2×2 squares of pixels is taken as one pixel in the downscaled image. If we represent a hypothetical 4×4 image as a matrix, where the matrix elements represent the pixel values ranging from 0 (white) to 1 (black), max pooling results in

$$\begin{pmatrix} 0 & 0.3 & 0.8 & 0 \\ 0.1 & 0.4 & 1 & 0 \\ 0 & 0 & 0.9 & 0 \\ 0 & 0 & 0.7 & 0 \end{pmatrix} \xrightarrow{\text{Max pooling}} \begin{pmatrix} 0.4 & 1 \\ 0 & 0.9 \end{pmatrix} . \tag{5.31}$$

Max pooling severely reduces the quality of the images, as can be seen in Fig. 5.6 (b). Average pooling produces downscaled images of similar quality. Since we want all digits to be recognizable, even after binarization, we opted for a different method for downscaling. First, the white borders of the original images were cropped before downscaling. To make sure no parts of the handwriting was cropped, the digits were centered by translating the center of mass of the pixel values to the center of the images. This allows us to crop the images from 28×28 to 20×20 without losing meaningful information. The cropped images were then downscaled to 14×14 using a bicubic filter, which is commonly used

for image up or downscaling [77]. After binarization, the result of the downscaling shown in Fig. 5.6 (c) resembles the original images more clearly compared to the result obtained by pooling.

5.3.2 Stochastic gradient descent

Even for a dataset of limited size, such as downscaled MNIST, calculating $\mathcal{L}'$ for each gradient descent step is computationally expensive. The limiting factor is the second term in Eq. (4.13). In the limit $T \to \infty$, Eq. (4.13) can be rewritten as

$$\mathcal{L}' = 2A^{(n,n+1)}_{j_n j_{n+1}} - 2\mathbb{E}_{\boldsymbol{x}\sim p_d} \frac{\langle \boldsymbol{x}|\Psi\rangle'}{\langle \boldsymbol{x}|\Psi\rangle} \,. \tag{5.32}$$

Estimating the expectation value by averaging over all T training samples becomes prohibitively expensive for large datasets. Stochastic gradient descent resolves this issue by approximating the expectation value using a so-called minibatch, which is a subset of samples drawn uniformly from the training data [78]. The number of samples in each minibatch is treated as a hyperparameter.

Stochastic gradient descent could be implemented in our algorithm by repeatedly drawing a minibatch and updating all tensors in the MPS, using the gradients from that minibatch, in a left and right sweep (see Eq. (4.11)). However this method proved to be unstable, meaning that the NLL of the training set increased and that tensor elements tend to diverge. The reason for this is that the same minibatch is used for $2N$ gradient steps, as opposed to drawing a new batch after every step. A possible solution is to perform several steps of gradient descent (Eq. (4.7)) before decomposing a merged tensor, where each time $\mathcal{L}'$ is approximated using a different batch.

Note that repeating Eq. (4.7) can be useful even without stochastic gradient descent. At first sight it might seem redundant to determine $\mathcal{L}'$ using the entire training set multiple times, however this is not equivalent to taking one step with larger γ. Although updating the elements of a merged tensor doesn't change the environment $\mathcal{E}$ and therefore $\langle \boldsymbol{x}|\Psi\rangle'$ (see Eq. (4.16)), it does alter the result of $\langle \boldsymbol{x}|\Psi\rangle$ due to Eq. (4.15).

In practice, we perform 5 subsequent gradient descent steps on each merged tensor using partitions of $T/5$ samples of the training set. The minibatches are used in random order. The relatively large batch size is required to ensure convergence of the optimization. Fig. 5.7 (a) and 5.7 (b) show the results of respectively regular and stochastic gradient using $T = 10^4$ training samples. The orange curves show the mean NLL of 10^4 test images which are not used for training. Stochastic gradient descent reaches a lower NLL for both the training and test images without requiring more computational resources than the standard variant. Both models were trained using the same hyperparameters. Most notably, the bond dimensions are too similar to explain the difference in performance: the average bond dimension of the model in Fig. 5.7 (a) at the final epoch was 278, while that of Fig. 5.7 (b) was 279. The slower rate of descent of the training NLL during the first epochs is caused by the small initial bond dimensions of the MPSs. Most bonds reach

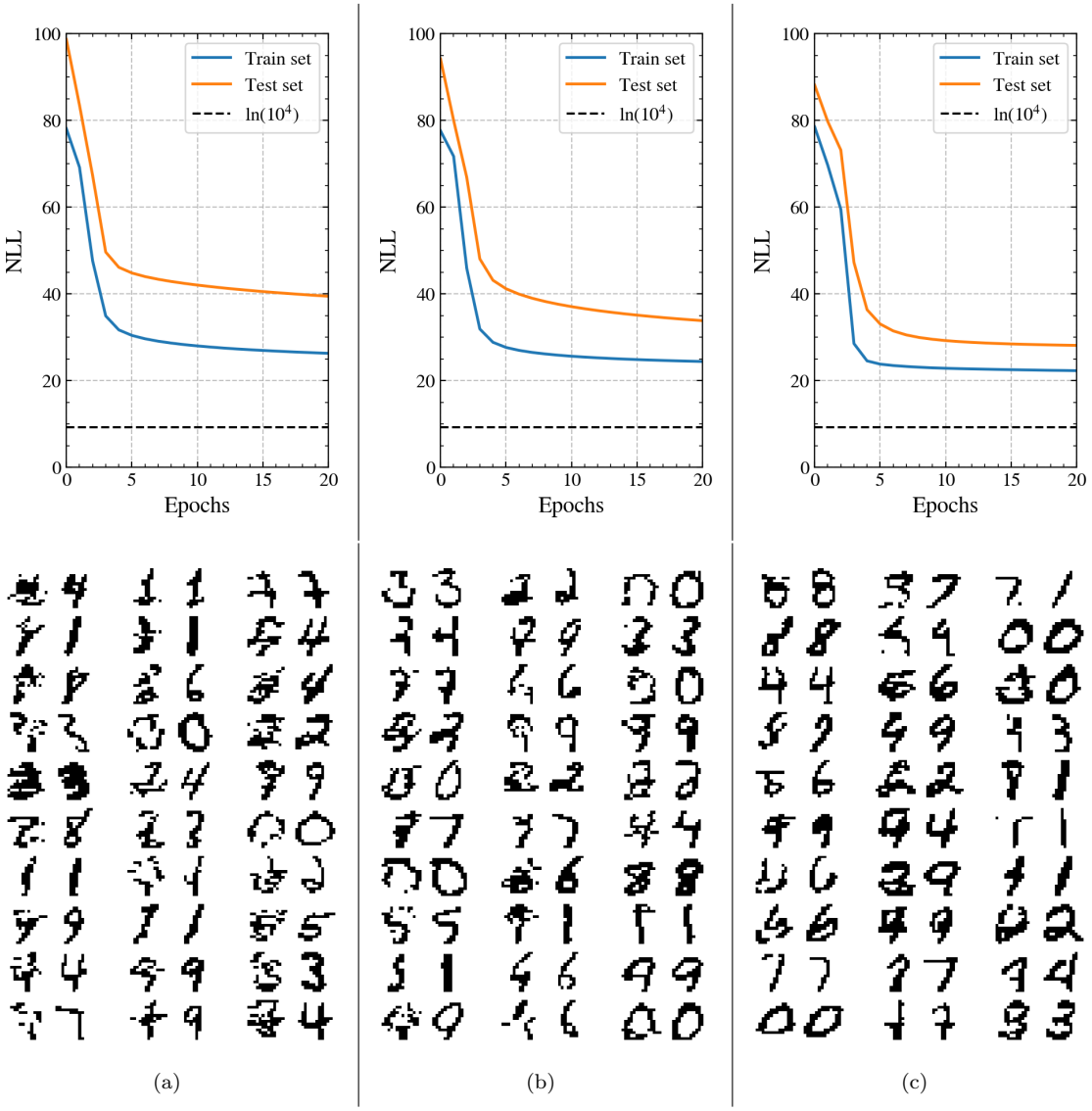


Figure 5.7 The top plots show the mean NLL of 10^4 training images and 10^4 test images when training an MPS using various optimization algorithms: (a) gradient descent; (b) stochastic gradient descent; (c) stochastic gradient descent and adaptive learning rate. The dashed black line depicts the smallest possible value of the NLL. The bottom figures represent 3 columns of images generated by the model above at epoch 20. Each generated image is bordered by its nearest neighbour found in the training data, with the similarity measured using euclidean distance in pixel space. The hyperparameters used in all experiments are: $D_{\text{max}} = 300$, $\gamma = 10^{-3}$ and $\epsilon_{\text{cut}} = 10^{-7}$. Stochastic gradient descent trains each merged tensor 5 times using batches of 2000 images. The adaptive learning rate in (c) uses $\beta = \frac{1}{4}$.

their maximum allowed dimension $D_{\text{max}} = 300$ after 3 epochs. In contrast, parametric models typically show the steepest decline in $\mathcal{L}$ at the start of training because their number of parameters remains constant during training.

Figure 5.7 also shows 3 columns of generated samples, each with its most similar training sample to its right. Here, the similarity is quantified as the Euclidean distance in pixel

space. In other words, the training sample most similar to a given generated sample $\widetilde{\boldsymbol{x}}$ is the $\boldsymbol{x}^{(i)}$ for which $\|\widetilde{\boldsymbol{x}} - \boldsymbol{x}^{(i)}\|$ is minimal. Showing the nearest neighbour of each generated image is useful to examine the quality of the generated images. It is also a visual method to detect overfitting: if the synthesised images consistently look very similar to its nearest neighbour, it is an indication that the model has memorized images of the training set. Although some of the generated images do resemble handwritten numbers, the overall quality of both models' output is poor. In some generated samples, black horizontal lines are noticeable which deteriorate the images' quality. These flaws could be an artefact of generating a 14×14 image sequentially as a vector of length 14^2 (see Fig. 5.5).

Motivated by the results of section 5.2, we used both stochastic gradient descent and the adaptive learning rate (5.30) with $\beta = 0.25$, resulting in Fig. 5.7 (c). The NLL curves now flatten after fewer epochs, meaning the optimization converges faster. This suggests that the adaptive learning rate again helps in traversing saddle points of the loss function. Since a lower NLL is reached, the generated samples are significantly better than those in Fig. 5.7 (a). Nevertheless, most generated images still look worse than those from the dataset.

When 10 times fewer images are used to train the MPS, standard gradient descent achieves similar train NLL and sample quality as before. This is shown in Fig. 5.8 (a). The same test set is used as in Fig. 5.7, meaning the test NLL curves of the figures can be readily compared. The NLL of the unseen test images now slowly increases during later epochs, in contrast to the decrease seen in Fig. 5.7 (a). As mentioned in section 2.3, this behaviour is typical in machine learning. When too few samples are used to train a model, it struggles to detect properties which are relevant to all samples of the dataset, not just to the training samples. In this context, overfitting means that the probability of generating images from the test set increases during training. The failure to generalize is even more apparent in Fig. 5.7 (c), where stochastic gradient descent and adaptive learning rate is used. Although the images generated by this model look fairly good, they often show clear resemblance to a training image. This is to be expected from the large gap between training and test NLL. Although generally we want the loss to be small, here a minimal training loss of $\mathcal{L} = \log T$ would lead to an infinitely large test NLL. This would be an extreme case of overfitting, where the model will only generate the memorized training samples as all other possible images, including those from the test set, are assigned zero probability.

As previously mentioned, we can perform several steps of gradient descent using the entire training set to determine $\mathcal{L}'$, instead of only using a minibatch as in stochastic gradient descent. Fig. 5.8 (b) is the result of performing 5 such updates on each merged tensor. Compared to stochastic gradient descent with a batch size of 200, using the entire training set does yield better results. Note that this comparison is hardly fair because the epochs of the standard gradient descent algorithm took almost three times as long as the stochastic variant, which is caused by evaluating the gradient of the entire dataset for each of the 5 steps.

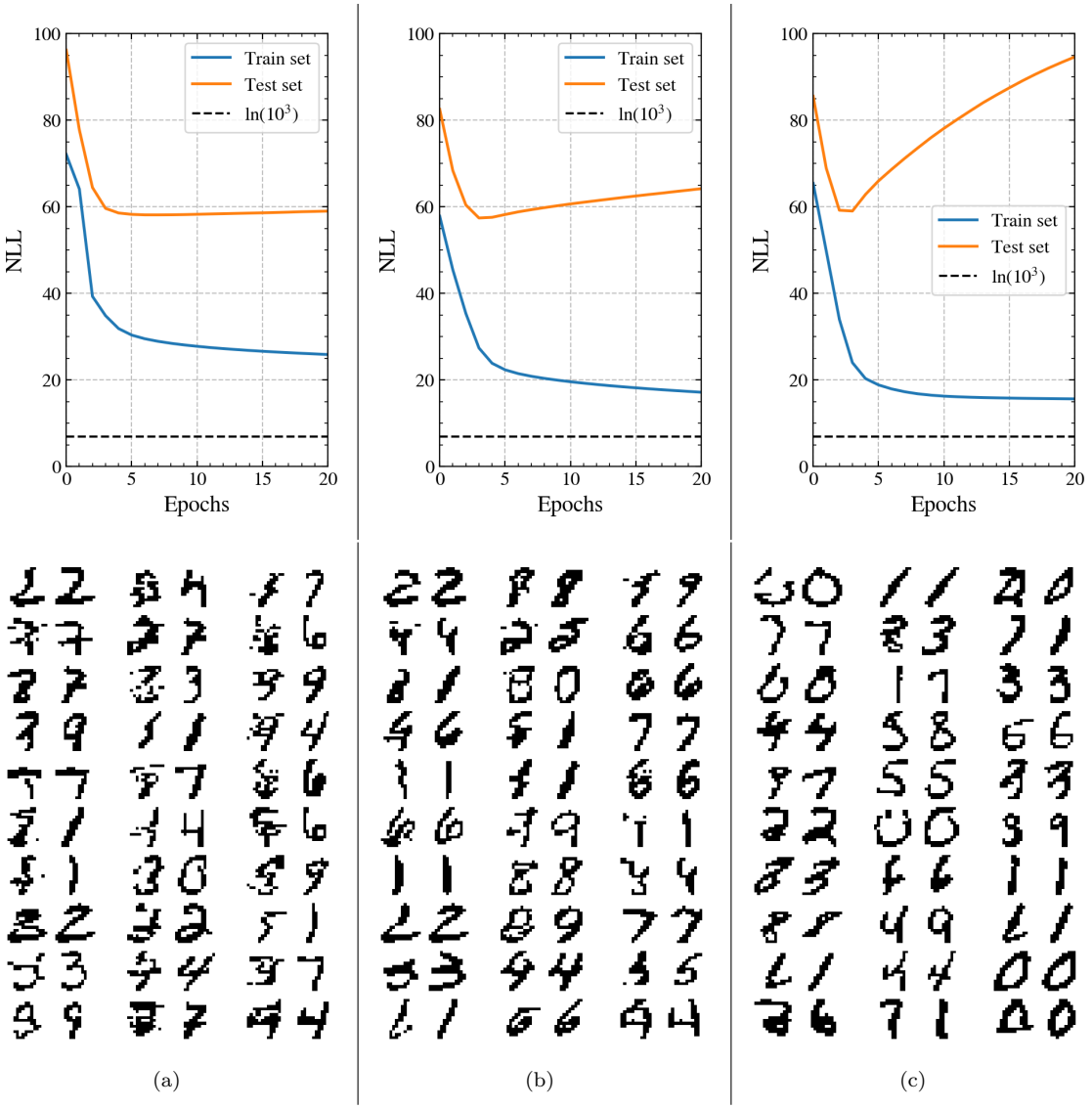


Figure 5.8 The mean NLL of 10^3 training images and 10^4 test images when training an MPS using various optimization algorithms: (a) gradient descent; (b) gradient descent where each merged tensor is updated 5 times; (c) stochastic gradient descent and adaptive learning rate. The dashed black line depicts the smallest possible value of the mean NLL of the training data. The bottom figures represent 3 columns of images generated by the model above at epoch 20. Each generated image is bordered by its nearest neighbour found in the training data, with the similarity measured using euclidean distance in pixel space. The used hyperparameters are described in Fig. 5.7, except that the batch size of stochastic gradient descent is now 200.

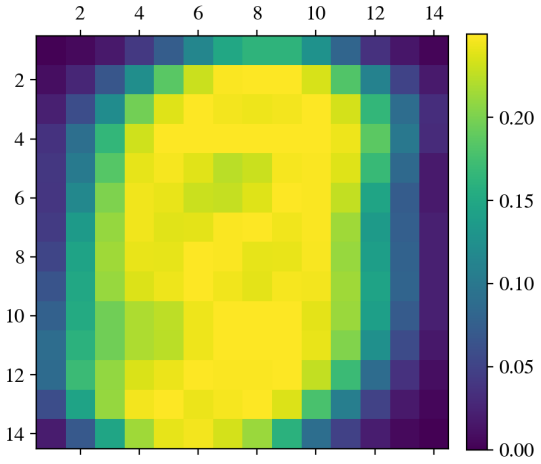


Figure 5.9 The variance of each pixel of the first 10^4 binarized, downscaled MNIST images.

As stochastic gradient descent combined with adaptive learning rate showed the best results for this dataset, we will use these techniques to train all following MPSs. The hyperparameters described in Fig. 5.7 will be used in the results that follow, unless mentioned otherwise.

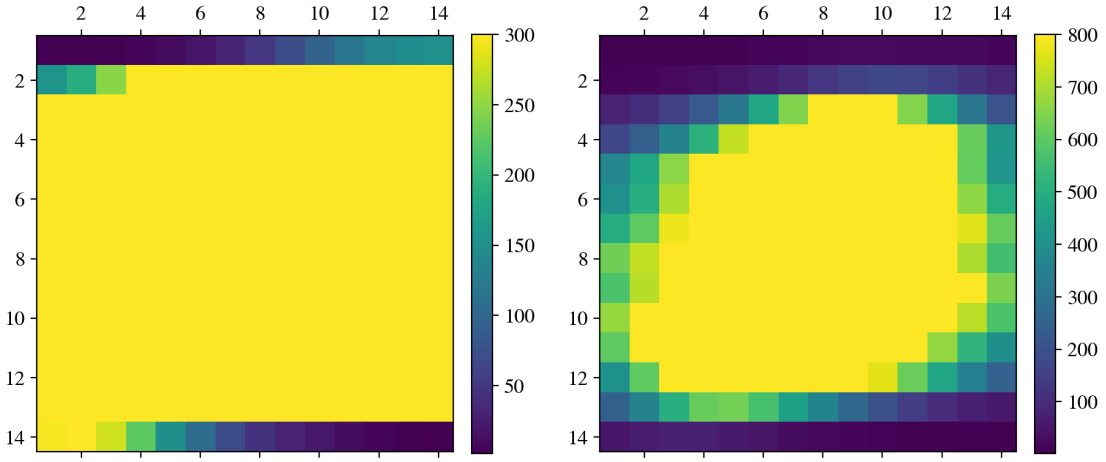
5.3.3 Adaptive bond dimensions

A major advantage of the two-site update, described in section 4.3, is that the bond dimensions can adapt to the complexity of the task [46]. In downscaled MNIST, we expect the most important pixels to be at the center of the image. Indeed, the variance of pixel x_i, given by $\mathbb{E}\big[(x_i - \mathbb{E}[x_i])^2\big]$, is significantly smaller at the borders because these are almost always white. The variance of the pixels over 10000 images is shown in Fig. 5.9. The MPS of Fig. 5.7 (c), which was trained with the relatively small cutoff 10^{-7} and $D_{\text{max}} = 300$, has reached its maximum allowed dimension for almost all bonds after 20 epochs, as shown in Fig. 5.10 (a). The bond dimensions near the borders of the MPS are smaller because of Eq. (1.12). This is even more clear when using $\epsilon_{\text{cut}} = 0$: the bond dimensions starting from the left are then 2, 4, 8, 16, 32, 64, 128, 256, 300, 300, ... after 10 epochs. In other words, the bond dimensions for $\epsilon_{\text{cut}} = 0$ are given by

$$d_n = \min\left(p^{\min(n,n-N)}, D_{\text{max}}\right) , \tag{5.33}$$

with $p = 2$ and $D_{\text{max}} = 300$. This is not surprising because, when partitioning a 1D system $|j_1 \dots j_N\rangle$ into $|j_1 \dots j_n\rangle$ and $|j_{n+1} \dots j_N\rangle$, the dimension of the Hilbert space of the smallest partition is given by $p^{\min(n,n-N)}$.

By using a larger cutoff, only the relatively large singular values will be retained after updating a merged tensor. This ensures that not all bond dimensions will reach their maximum, enabling us to use a larger D_{max} without requiring an excessive amount of computational resources. Fig. 5.10 (b) shows the bond dimensions after training an MPS



(a) (b)

Figure 5.10 The bond dimensions of MPSs trained with the same 10^4 images used in Fig. 5.9. Element i of the raster, counted as in Fig. 5.5, represents the bond dimension d_i. For instance, the top left corner shows the dimension of the bond connecting sites (pixels) 1 and 2. As we use MPSs with open boundary conditions, the bottom right corner representing d_N will always be 1. (a) MPS trained with $D_{\max} = 300$ and $\epsilon_{\text{cut}} = 10^{-7}$ trained for 20 epochs, which results are shown in Fig. 5.7 (c). (b) MPS trained for 10 epochs with $D_{\max} = 800$ and $\epsilon_{\text{cut}} = 1.4 \cdot 10^{-3}$. See Fig. 5.11 for its results.

with $\epsilon_{\text{cut}} = 1.4 \cdot 10^{-3}$ and $D_{\max} = 300$. We see that the largest bond dimensions are indeed centered around the area with high pixel variance (see Fig. 5.9). The model generalizes remarkably well as its test NLL is significantly closer to the train NLL, which can be seen by comparing Fig. 5.11 with Fig. 5.7 (c). A possible explanation is that the larger cutoff acts as a form of regularization by making it less likely that the model learns details of the training images instead of properties relevant for all images. It is also possible that the model simply underfits less because of the larger $D_{\max}$, however the train NLL did not improve significantly.

5.3.4 Pixel grouping

Although MPSs optimized with DMRG have also been used to study certain two-dimensional quantum lattice systems [79], the one-dimensional arrangement of its tensors make MPSs less suitable for 2D systems. Pixels in natural images are most strongly correlated with its neighbours, which is also typical for spin systems [5]. Since it is impossible to rearrange a 2D system to 1D while retaining all neighbours, the 1D arrangement of a 2D system will exhibit longer ranged correlations. For instance, after flattening a 14×14 image via Fig. 5.5, there will be 13 sites between initially vertically neighbouring pixels. Combined with the fact that correlations in an MPS exhibit an exponential decay (see section 1.3.2), we can expect an MPS to struggle with 2D data.

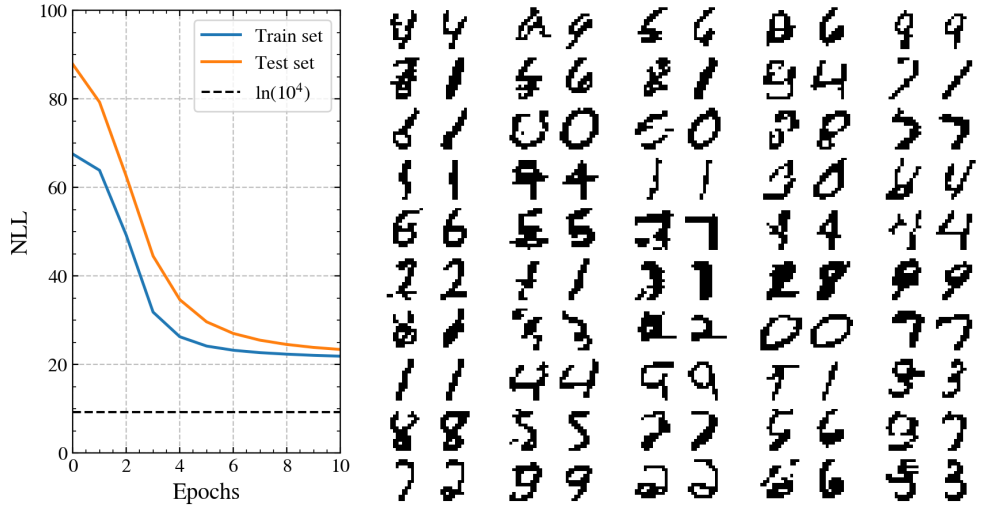


Figure 5.11 Results of the MPS of Fig. 5.10 (b). Each generated image is bordered by its nearest neighbour found in the training data, with the similarity measured using euclidean distance in pixel space.

By generalizing the code to work with an arbitrary physical dimension, we are able to train an MPS to predict areas of 2×2 pixels at once. This was attempted to reduce the handicap induced by modelling 2D data using a 1D architecture. The reasoning is that using groups of pixels would give more spatial information to the MPS, making it easier to also model the vertical correlations in the images. Because a group of 4 binary pixels can have 16 different configurations, it is encoded as a one-hot vector of size 16. On the other hand, the number of sites required will be 4 times smaller due to the grouping. Note that we only altered the way the images are fed into the MPS, not the images themselves. For instance, the total number of possible configurations images remains identical: $p^N = 16^{196/4} = 2^{196}$. After generating new samples from the MPS, the one-hot vectors are mapped back to pixels to obtain the generated images.

Fig. 5.12 (a) shows the result of training an MPS on 1000 training samples with 2×2 pixel grouping for 10 epochs. The number of training images and epochs is relatively small because increasing p comes at a significant computational cost (see section 4.5). Compared to Fig. 5.8 (c), the train and test NLL start lower because the mean bond dimension after the first epoch is fairly large: 183.9 as opposed to 7.9 without grouping. All hyperparameters, and in particular ϵ_{cut}, used in Fig. 5.8 (c) are identical to those in Fig. 5.12. Therefore the faster growth of bond dimensions is caused by the grouping itself, more specifically by the larger p combined with Eq. (1.12). At the final epoch, the mean bond dimension is approximately the same when trained with or without pixel grouping, meaning the 2×2 grouping achieves slightly lower NLL with the same bond dimension. The downside is that training takes almost 4 times longer. Furthermore, some of the generated images still don't resemble digits.

To reduce the computational cost while retaining the vertical grouping, we attempted a 2×1 grouping in Fig. 5.12 (b). The physical dimension is now 4, making it 5 times faster

than 2×2 grouping and even 40% faster than the MPS trained in Fig. 5.8 (c). Keep in mind that we only trained for 10 epochs in Fig. 5.12, so processing the images without grouping is still faster. The results of 2×1 grouping appears to be significantly worse than 2×2 or even no grouping. Furthermore, the NLL of the training set increased at epoch 3, indicating instability in the training process. Unlike the test NLL, we are directly minimizing the mean NLL of the training images with gradient descent, so a rising train NLL indicates unstable optimization. This can usually be resolved by using a smaller learning rate.

Finally, we also tried a 4×1 grouping. To make the number of rows divisible by 4, we padded the training images with two rows of zeros at the bottom. The obtained NLL is slightly better than in Fig. 5.8 (c) and slightly worse than those in Fig. 5.12 (a). Because of the extra computational cost associated with using images of 16×14, the 2×2 should be preferred. From these experiments, it seems that pixel grouping only slightly improves the results. Due to the significant computational costs caused by the higher p, we can conclude that allowing higher bond dimension without pixel grouping is a more effective method than using lower bond dimensions with grouping. Note that these results are obtained with downscaled MNIST and that grouping might prove more useful for other datasets.

5.4 Discrete sine waves

5.4.1 Dataset

Due to the one-dimensional geometry of a MPS, it is not very well suited to model images or other types of data with more than one spatial dimension. Therefore it would be beneficial to model discrete one-dimensional data, preferably also with a small physical dimension p to limit computational requirements. The physical dimension of a dataset corresponds to the number of possible values for a feature. Both the bars and stripes and downscaled MNIST datasets consist of binary images, meaning each pixel can take on two possible values and therefore $p = 2$. Popular and useful one-dimensional datasets in machine learning context are time series, such as stock market prices or audio samples. However, these datasets are not suitable for standard MPS models since they are continuous. Generative models for text are also a central focus in recent research [80], however encoding all punctuation marks, letters and their capitalized variants requires a physical dimension close to 100. Many state-of-the-art machine learning algorithms for text also use a character level embedding [81], which further increases the physical dimension. The combination of large bond dimensions to capture sufficient correlations and large physical dimensions would be too computationally expensive for the purpose of this thesis (see section 4.5).

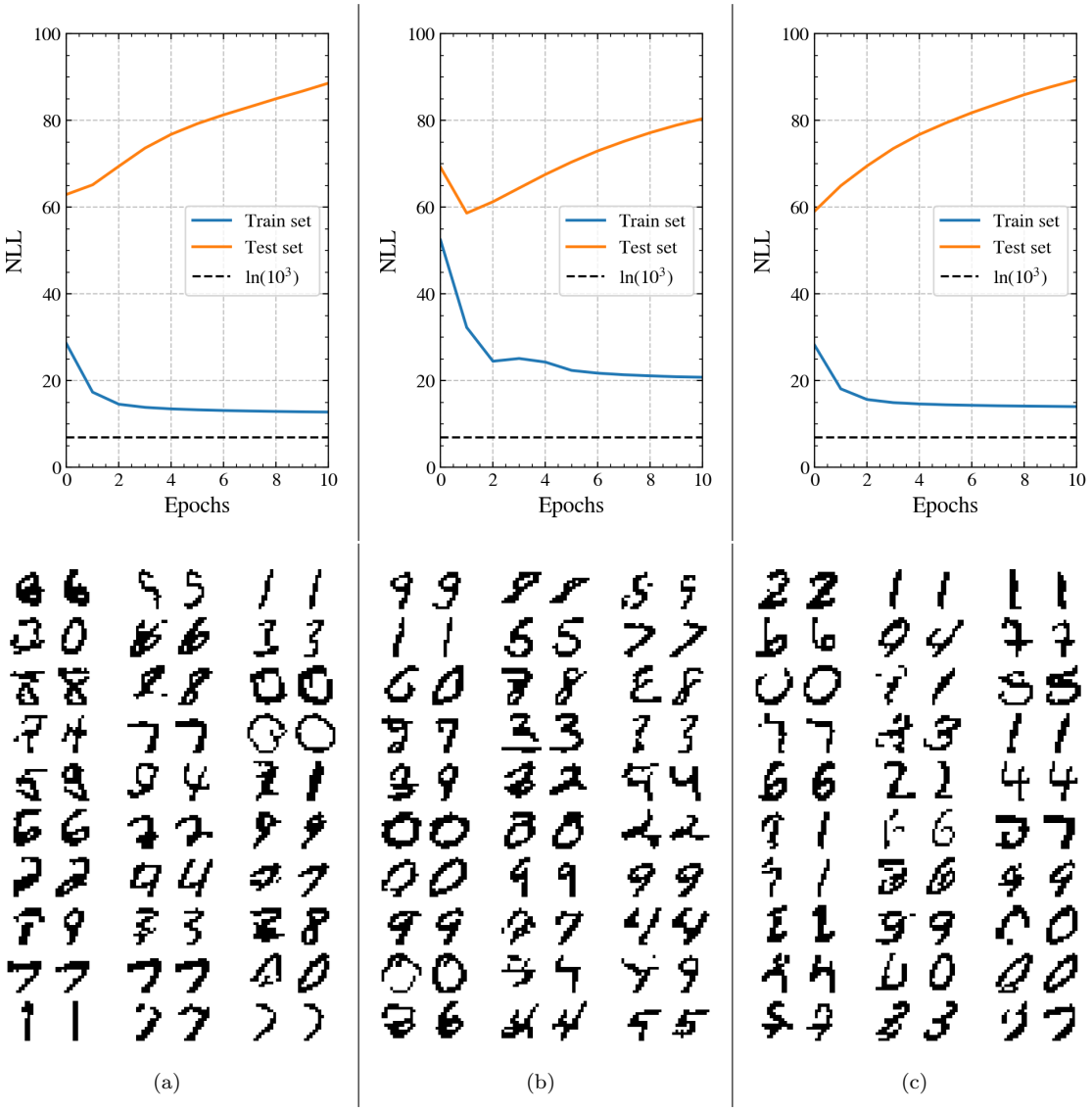
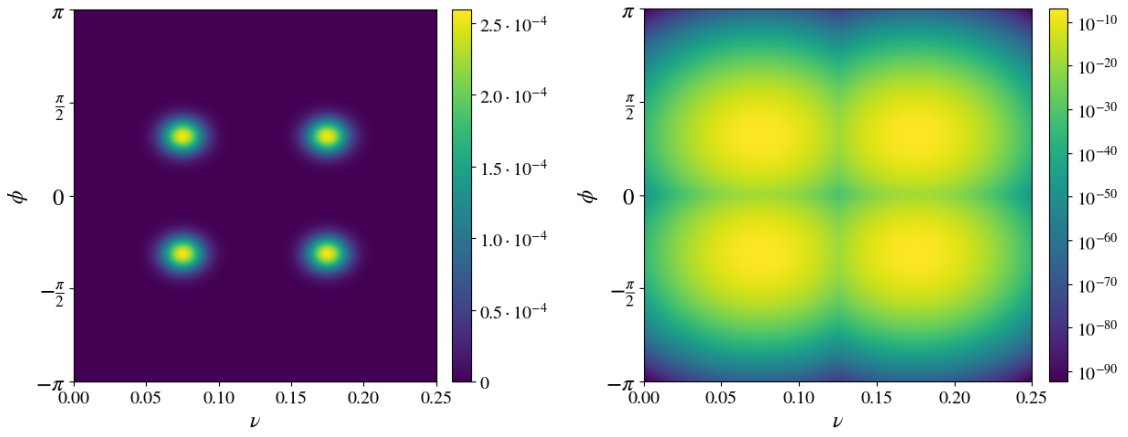


Figure 5.12 The mean NLL of 10^3 training images and 10^4 test images when pixels are grouped together: (a) 2×2 grouping, (b) 2×1 grouping and (c) 4×1 grouping. The bottom figures show 3 columns of images generated by the model above at epoch 10. Each generated image is bordered by its nearest neighbour found in the training data, with the similarity measured using euclidean distance in pixel space. The hyperparameters and optimization scheme of Fig. 5.8 (c) are used.



(a) (b)

Figure 5.13 The joint probability distribution of the frequencies and phases of the discrete sine waves on (a) linear and (b) logarithmic scale.

Since no suitable dataset was found, we created a toy dataset which consists of discretized sine waves of the form

$$\boldsymbol{x} = \sin(\frac{\pi}{2}\nu\boldsymbol{t} + \phi) \qquad \text{with } \boldsymbol{t} = \begin{pmatrix} 0 \\ 1 \\ 2 \\ \vdots \\ N \end{pmatrix}, \tag{5.34}$$

which means that an MPS trained on this dataset will consist of N matrices. For each sample $\boldsymbol{x}$, a frequency ν and a phase ϕ are drawn from a double Gaussian probability density function

$$Q(\nu) = \frac{\mathcal{N}(\frac{3}{40}, \frac{1}{10}) + \mathcal{N}(\frac{7}{40}, \frac{1}{10})}{2} \tag{5.35}$$

$$R(\phi) = \frac{\mathcal{N}(-1, \frac{1}{5}) + \mathcal{N}(1, \frac{1}{5})}{2}\,. \tag{5.36}$$

The joint probability distribution is plotted in Fig. 5.13. This figure is obtained by evaluating the joint probability density function for both 300 frequencies in the interval $[0, \frac{1}{4}]$ and 300 phases in $[-\frac{\pi}{2}, \frac{\pi}{2}]$. The joint probability is then obtained by multiplying with step sizes $\delta\nu = \frac{1/4}{299}$ and $\delta\phi = \frac{\pi}{299}$ [1]

$$P(\nu, \phi) = Q(\nu)R(\phi)\delta\nu\delta\phi\,. \tag{5.37}$$

Repeating this procedure with a 700×700 grid of ν and ϕ instead of 300×300 produces an identical figure, implying that the used $\delta\nu$ and $\delta\phi$ are sufficiently small.

[1] The denominator is 299 because we want both edges of the interval to be included.

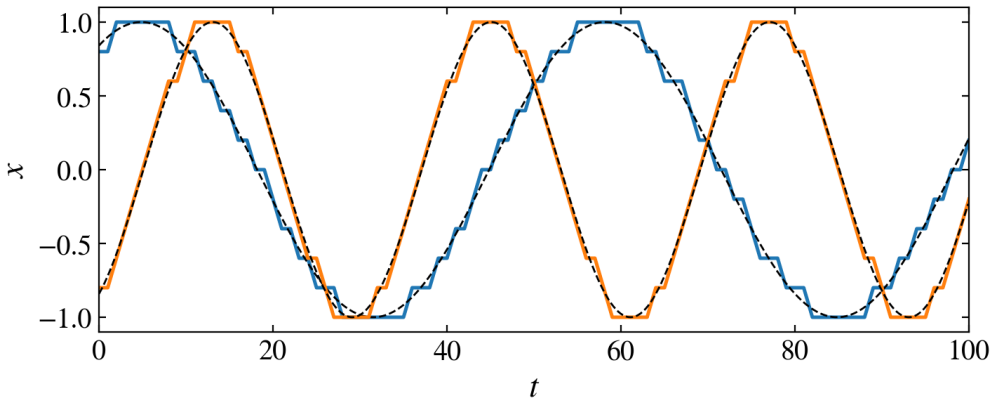


Figure 5.14 Examples of discretized sine waves with $p = 11$ and their continuous form in black dashed lines. (blue) Sine wave with $\nu = \frac{3}{40}$ and $\phi = 1$; (orange) sine wave with $\nu = \frac{7}{40}$ and $\phi = 1$.

As an example, a wave with $\nu = \frac{3}{40}$ and $\phi = 1$ and its discretized form of $p = 11$ is shown in blue in Fig. 5.14. The discretized wave in one-hot encoding takes the form of the $p \times N$ matrix

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & \cdots \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & \cdots \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & \cdots \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \cdots \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & \cdots \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & \cdots \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots \end{pmatrix} . \tag{5.38}$$

If the frequency and phase of two samples is close to each other, its possible that their discretized forms are identical. This is usually only a problem when the used physical dimension is low and can be avoided by increasing p.

5.4.2 Results

Estimating the probability density function of the discrete sine waves dataset, shown in Fig. 5.13, is straightforward in (ν, ϕ) space, which we will refer to as the phase space of the data. Given sufficient (ν, ϕ) points drawn from the underlying distribution (5.37), an approximation of Fig. 5.13 could be obtained by simply creating a two-dimensional histogram of the data points. The fact that each sample is characterized by the two parameters ν and ϕ is convenient for visualization. For more realistic datasets, e.g. MNIST, we have no access to such information, making density estimation significantly more challenging.

Similarly, in this section we will use an MPS to estimate the underlying probability density using only the raw data, namely T discrete sine waves of the form (5.38).

As mentioned, the log-likelihood that our model assigns to each point in the phase space of our toy dataset can be readily visualized and compared with the true distribution in Fig. 5.13 (b). Each (ν, ϕ) point in the 300×300 grid, which was used to create Fig. 5.13 (see section 5.4.1), is converted to its corresponding discretized sine wave in one-hot encoding, which we will denote as $\boldsymbol{x}(\nu, \phi)$. The likelihood of each wave is then determined by Eq. (4.1), which becomes

$$P_m(\nu, \phi) = \frac{|\langle \boldsymbol{x}(\nu, \phi) | \Psi \rangle|^2}{Z} \,. \tag{5.39}$$

Before taking the logarithm of the resulting likelihoods, possible zeros are converted to 10^{-700} to avoid infinities. Finally, bicubic interpolation is used to ensure the plots do not look pixelated.

The results of training an MPS using various T can be seen in Fig. 5.15. The top left pane show that our model does not overfit when trained with 10^4 samples, as the test loss decreases monotonically. The top right pane shows the probability density assigned to each $\boldsymbol{x}(\nu, \phi)$ of the grid by the MPS, which was trained for 10 epochs. Although the model generally appears to have captured the underlying distribution in Fig. 5.13 (b), there are some remarkable differences to note.

Firstly, the striped patterns that can be seen in some regions of the phase space are artefacts of the discretization of the waves. More specifically, not all (ν, ϕ) points in the 300×300 grid result in unique $\boldsymbol{x}(\nu, \phi)$. For instance, there are 300 'waves' with $\nu = 0$, of which at most 11 can be unique because only p different constant functions can be represented in the discrete form. In total, about 3% of the discretized waves in the grid are not unique, which explains the artefacts in the phase space plots.

Secondly, note the areas of relatively high likelihood around $\nu = 0$ and $\phi = \pm\frac{\pi}{2}$, which are not present in the true distribution. These areas correspond to the constant functions $x = 1$ and $x = -1$. Their unexpectedly high likelihood can be understood by considering Fig. 5.16, which shows the frequency of the 11 possible x-values averaged over 10^4 waves. Clearly, the values 1 and -1 are the most common in the dataset because the inflection points of the waves, which all have amplitude 1, are at $x = 1$ and $x = -1$. Therefore the relatively high likelihood assigned by our model to functions like $x = 1$ and $x = -1$ appears to be a sign of underfitting, where the model learns the most likely value of x instead of wave patterns. The fact that our model assigns a higher likelihood to $\nu = 0, \phi = -\frac{\pi}{2}$ compared to $\nu = 0, \phi = -\frac{\pi}{2}$ can be explained by the asymmetry between -1 and 1 seen in Fig. 5.16. The asymmetry is an artefact of the finite length of our waves, as for $N = 1001$ both values of x become equally common. Training MPSs with such large N would require too much computational resources for our purposes.

Finally, we note the significant difference in scale compared to Fig. 5.13 (b). This becomes even more apparent when considering the sum of the probability density of each $\boldsymbol{x}(\nu, \phi)$

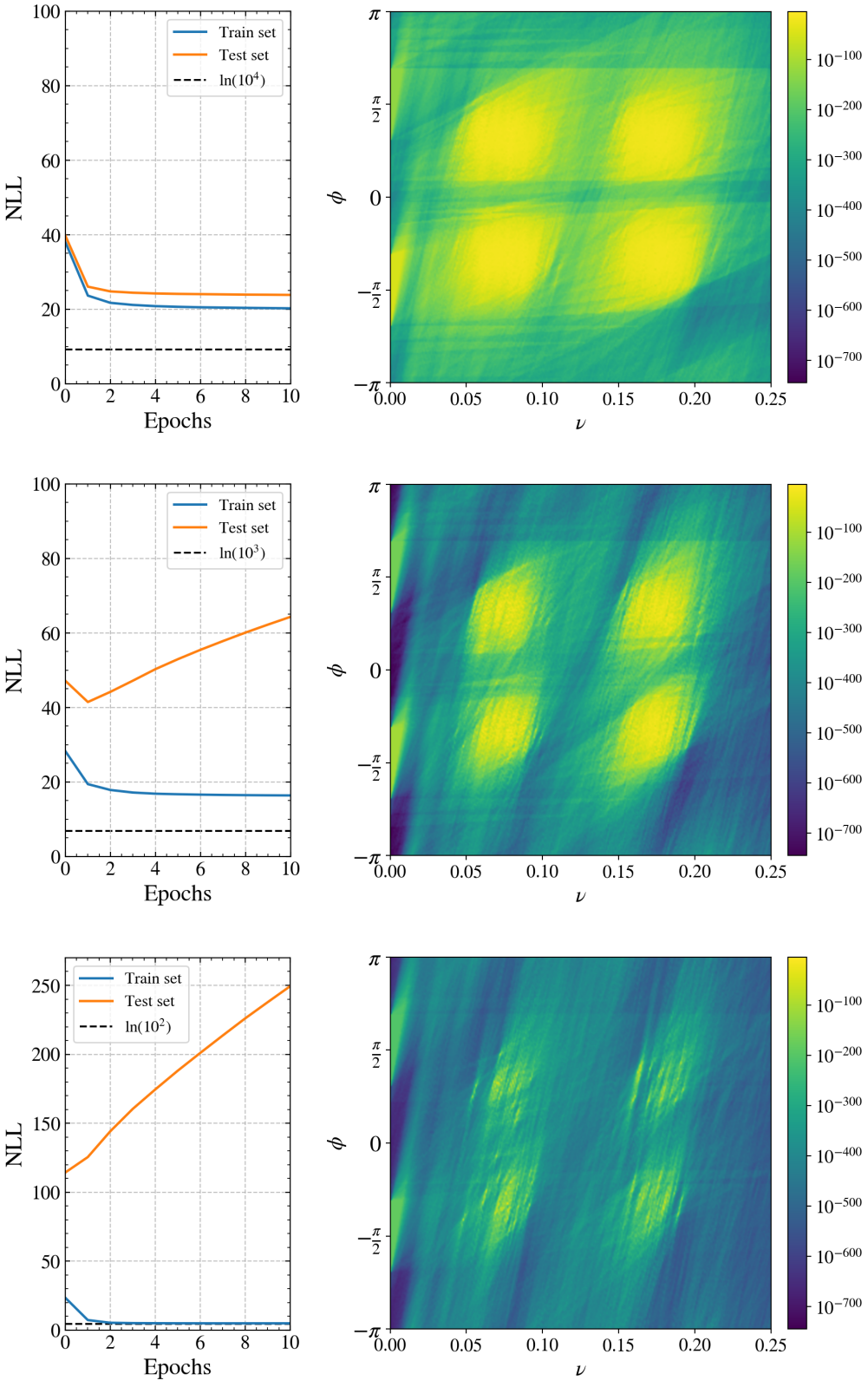


Figure 5.15 NLL and log of the probability density assigned to the phase space of the discrete sine waves dataset by MPSs trained for 10 epochs, using 10^4 test samples and (top) $T = 10^4$; (middle) $T = 10^3$; (bottom) $T = 10^2$. All right plots use the same colour scale to make comparison easier. For each MPS, $D_{\max} = 150$ and $\epsilon_{\text{cut}} = 10^{-7}$ was used.

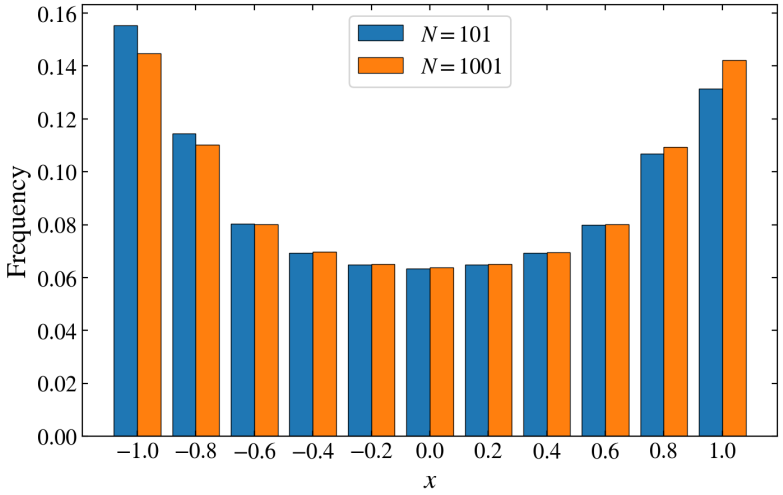


Figure 5.16 Frequency of each possible value of x averaged over 10^4 discrete sine waves. Note that the asymmetry between the frequencies of $x = -1$ and $x = 1$ for $N = 101$ vanishes if the waves are sufficiently long, e.g. $N = 1001$.

in the grid

$$\sum_{i=0}^{299}\sum_{j=0}^{299} \frac{|\langle \boldsymbol{x}(i\delta\nu, j\delta\phi)|\Psi\rangle|^2}{Z}\delta\nu\delta\phi = 3.03 \cdot 10^{-4}\,. \tag{5.40}$$

We can do the same for the probability density of the data, defined by Eq. (5.37)

$$\sum_{i=0}^{299}\sum_{j=0}^{299} P(i\delta\nu, j\delta\phi) = 0.9867\,. \tag{5.41}$$

This suggests that the probability of drawing a sample $\boldsymbol{x}(\nu, \phi)$ from the true data distribution with $\nu \in [0, \frac{1}{4}]$ and $\phi \in [-\frac{\pi}{2}, \frac{\pi}{2}]$ is almost 99%, while the probability of our model generating such a sample is merely 0.03%. A possible explanation is that our model performs poorly and is likely to generate noise, however the generated samples shown in Fig. 5.17 indicate that the model does generate wave-like samples. A more likely explanation is that the resolution of the used (ν, ϕ) grid is too small, meaning a smaller $\delta\nu$ and $\delta\phi$ would result in a larger sum in Eq. (5.40). Another important factor is that the slightest deviation in the waves of the grid, e.g. a different x for a single site t_i, would not contribute to the sum even though the wave doesn't differ noticeably. Since the scale of the phase space visualizations strongly depends on the resolution of the grid which is plotted and does not necessarily correspond to the quality of the generated samples, it should not be interpreted as an evaluation metric of the model.

When fewer training samples are used, the overfitting problem becomes apparent in both the NLL and the phase space plots. For $T = 10^3$, shown in Fig. 5.15 (middle), the test loss quickly starts to increase and the four modes of the data distribution look somewhat distorted. In particular, there are regions of relatively low likelihood between training

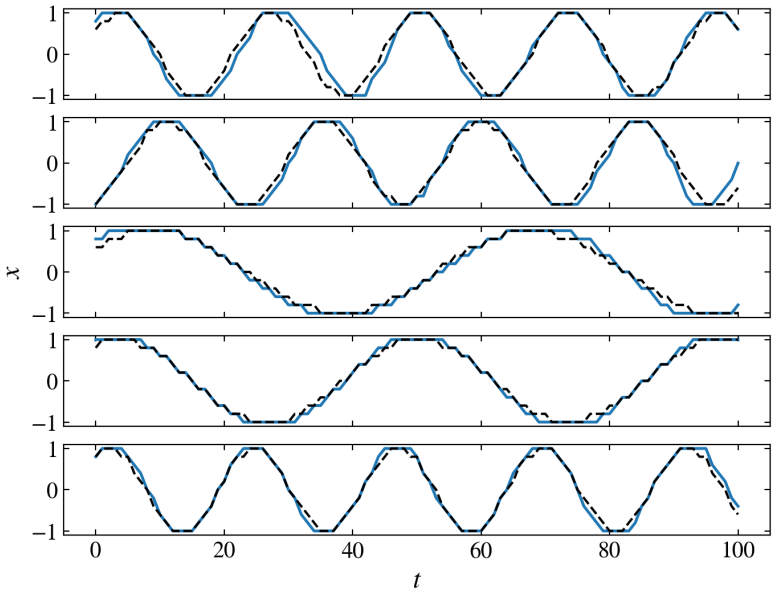
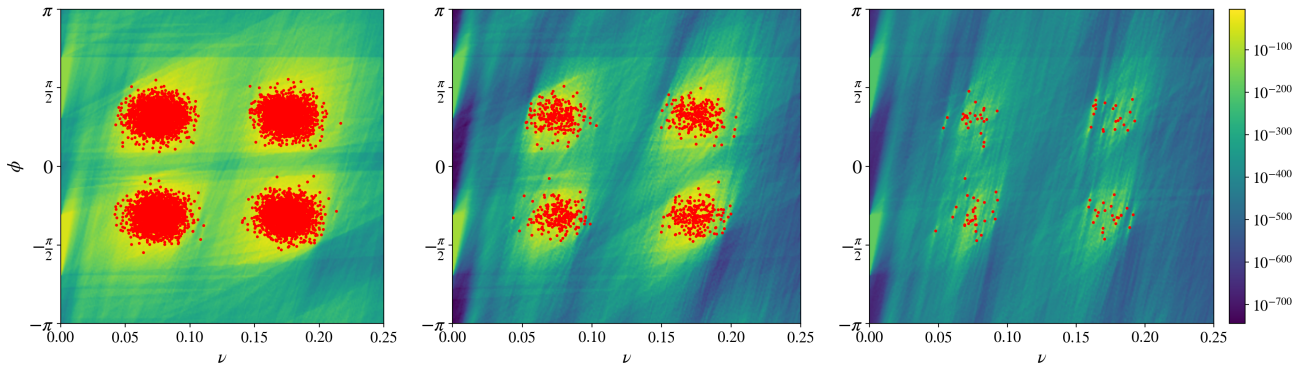


Figure 5.17 Samples generated by the MPS of Fig. 5.15 (top) are plotted in blue. The dashed black waves represent the nearest training samples corresponding to each generated sample, using Euclidean distance as metric.

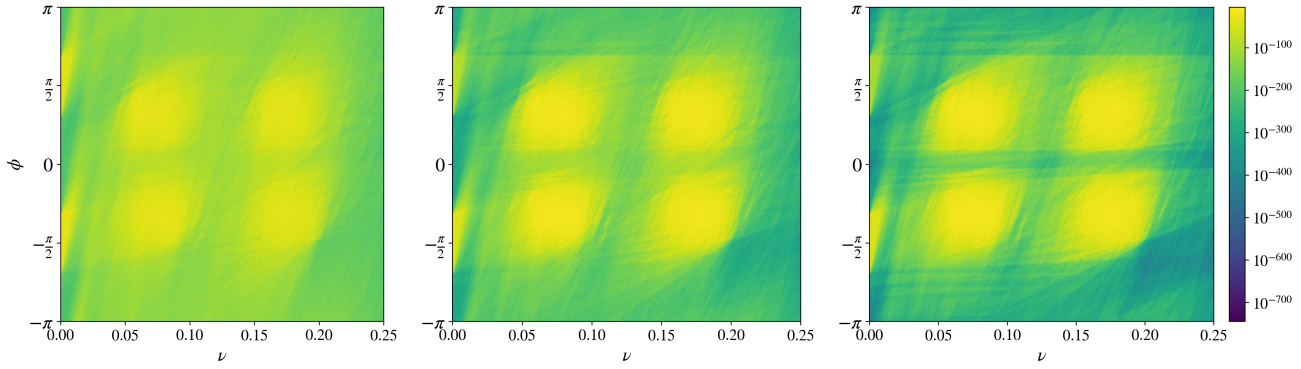
samples originating from the same mode. This can be seen more clearly in Fig. 5.18 (b), where the (ν, ϕ) points of the training samples are indicated as red dots. Also note that the areas of high likelihood near $\nu = 0$, which we assumed to be a sign of overfitting, is now less prominent. This is even more apparent in Fig. 5.15 (bottom), which shows the results of an MPS trained with $T = 100$. The NLL of the training set is now close to its minimal value $\log(100)$, meaning the model overfits significantly. We can even distinguish individual training samples in the phase space plot (see Fig. 5.18 (c)), meaning the model was clearly unable to capture the entire data distribution. As expected, the waves generated by the MPSs trained with 1000 and 100 training samples become increasingly more similar to their nearest neighbours in the training set.

The evolution of the probability density the MPS of Fig. 5.15 (top) assigns to the phase space can be seen in Fig. 5.19. During training, the modes of the underlying data distribution become more prominent as the likelihood of the $\boldsymbol{x}(\nu, \phi)$ which are not near the modes of the data decreases. Although not clearly visible in the figure, the probability of the areas around $(\nu = 0, \phi = \pm\frac{\pi}{2})$ does decrease as the training progresses, indicating that the model underfits less at later epochs, as expected. It is interesting to note that even at epoch 0, the four modes of the data distribution are already visible. This means that the model does not first capture one mode and then continues to improve by capturing more and more modes of the data.



(a) (b) (c)

Figure 5.18 The right plots of Fig. 5.15 with (ν, ϕ) points corresponding to the training samples used for each MPS indicated as red dots. The number of training samples is (a) 10^4; (b) 10^3; (c) 10^2.



(a) (b) (c)

Figure 5.19 Probability density the MPS trained using $T = 10^4$ assigns to each $\boldsymbol{x}(\nu, \phi)$ in the 300×300 grid, evaluated after (a) epoch 0; (b) epoch 2, (c) epoch 8.

Chapter 6

Conclusions and outlook

As suggested by the success of MPSs in simulating one-dimensional quantum systems, MPSs are also able to model the underlying probability distribution of various datasets. An MPS explicitly defines the model distribution, in contrast to defining a mapping of latent variables to features, which is the approach adopted by GANs and variational autoencoders (VAEs) [25]. An MPS model allows us to determine the likelihood of generating a given sample in a straightforward manner, whereas GANs and VAEs must resort to approximations of the likelihood. Consequently, the performance of an MPS model can be objectively evaluated. More generally, generative models based on tensor networks require significantly less hyperparameter tuning, simply because there are less hyperparameters compared to neural network models. Neural networks can have a varying number of layers, number of nodes per layer, number of filters in case of convolutional layers and so on. Moreover, the performance of neural networks often strongly depends on these architectural choices [14]. On the other hand, tensor network models only require optimization of the values of the hyperparameters γ and ϵ_{cut}. A major disadvantage of tensor networks as generative models is that, so far, the obtained generated samples are significantly worse compared to GANs and VAEs [50]. Note that tensor networks were only recently proposed as generative models, so further work might improve their results.

One approach for improving generative models based on tensor networks is further development of the training algorithm. Our results indicate that the DMRG algorithm with two-site updates is an excellent framework for optimizing MPSs, in part because it allows the model to focus on the most important features of the data via adaptive bond dimensions. However, we have also shown that an alternative parameter initialization method, stochastic gradient descent and adaptive learning rate can significantly improve the obtained NLL and convergence speed. These results indicate that MPSs benefit from optimization techniques commonly used in machine learning. Therefore the use of optimization techniques such as decaying learning rate or Adam [82] would be interesting to examine in the future.

Another promising research direction is the implementation of other types of tensor networks as generative models. It is possible that the examined techniques and obtained

hyperparameters for MPSs are also beneficial for generative models based on different tensor networks. As the two-dimensional generalization of MPS, PEPS are a more natural choice for image modelling. In fact, during the writing of this thesis Ref. [83] was published, where a PEPS model was trained on 5 downscaled MNIST images. Due to the connections of the tensors in PEPS (see Fig. 1.1), large intermediate tensors are formed during the contraction with data. Although this becomes infeasible for large bond dimension and/or system size, approximate contraction schemes can be used. For instance, each intermediate tensor can be decomposed as an MPS, as was opted for in Ref. [83]. Due to the fact that each tensor in a PEPS is connected to more than one other tensor, we can no longer efficiently compute marginal probabilities as in the MPS case (see Eq. (4.20)). Therefore exact sampling, as described in section 4.4, becomes intractable for PEPS models and we must resort to MCMC. TTN models do not suffer from these limitations and can use algorithms similar to those described in chapter 4. We have shown that correlations functions in TTN states decay as a power law as opposed to exponentially in MPSs. In contrast to what was claimed in Ref. [50], this does not seem to influence the models' capability to memorize random samples. It would be interesting to examine the effects of the differing correlator decay for more useful tasks, for instance by comparing the performance of TTN and MPS on the discrete sine wave dataset we have designed. Furthermore, our dataset could be used to compare tensor networks to other generative models.

Inspired by the success of GANs, our efforts to formulate an adversarial learning objective for MPS models unfortunately resulted in an alternative formulation of the maximum likelihood estimation. A better theoretical understanding of GANs and adversarial learning could clarify whether it is possible to extend adversarial learning to tensor network models and how useful this would be. Despite the fact that maximizing likelihood corresponds to minimizing the KL divergence, which is not a metric for the distance between the model and data distribution, it remains the most common optimization method for generative models.

Recently, a continuous matrix product state (cMPS) [84] has been implemented as generative model [85]. While MPSs are mostly used to study quantum states on a one-dimensional lattice, cMPSs are defined without any reference to an underlying lattice parameter [84]. cMPSs correspond to the continuum limit of MPSs and are mostly used to study quantum field theories with one spatial dimension [84]. This means that generative models based on tensor network are not necessarily restricted to discrete data. The cMPS model is also trained by log-likelihood maximization. However, the optimization is no longer performed by the DMRG-based algorithm described in chapter 4. Instead, all tensor elements in the cMPS are updated at once using stochastic gradient descent. The downside of this approach is that the bond dimensions are no longer adaptive. Moreover, the algorithm requires an extra set of parameters which describes the time evolution of the cMPS state [85]. In other words, the cMPS model is required to learn a Hamiltonian which describes the time evolution of the data. The cMPS model shows promising results on toy datasets [85]. Testing its performance on more realistic data, e.g. audio samples,

would allow comparison to popular generative models for audio, such as WaveNet [86] and WaveGAN [87].

Finally, the close connection between tensor networks and quantum states could prove essential for the migration of machine learning to quantum computers. Algorithms to prepare certain tensor network states on quantum computers have already been developed [55, 88]. This opens up the possibility of resolving the computational limitations of PEPS, allowing us to efficiently exploit the 2D ansatz provided by PEPS.

Appendices

A Nederlandse samenvatting

Machinaal leren is een belangrijk onderdeel geworden van vele domeinen van moderne technologie en wetenschap, waaronder fysica. In onderzoeksgebieden zoals experimentele deeltjesfysica, vastestoffysica, observationele astronomie en kosmologie wordt machinaal leren steeds meer toegepast. Tezelfdertijd dienen ook concepten en methoden uit statistische fysica en kwantum veeldeeltjesfysica als inspiratie voor vele theoretische en praktische technieken in machinaal leren.

Deze thesis focust op het modelleren van waarschijnlijkheidsdistributies in een exponentieel grote configuratieruimte. Dit is een centraal probleem in zowel kwantum veeldeeltjesfysica als generatief modelleren, wat een deelgebied is van toegepaste statistiek en machinaal leren. Het doel van niet-gesuperviseerd generatief modelleren is om een waarschijnlijkheidsdistributie te verkrijgen die samples genereert die gelijkaardig zijn aan de samples in een bepaalde dataset. Samples kan verwijzen naar vele vormen van data, zoals afbeeldingen of Nederlandse zinnen. Gewoonlijk is slechts een kleine fractie van de exponentieel vele mogelijke samples relevant. Bijvoorbeeld het aantal mogelijke afbeeldingen groeit exponentieel met het het aantal pixels in de afbeelding, maar slechts een kleine fractie van deze mogelijkheden zijn interessant want het overgrote deel van combinaties van pixelwaarden ziet er uit als ruis. Dit toont opvallend veel gelijkenissen met het beschrijven van kwantum veeldeeltjestoestanden, waarbij de fysisch relevante toestanden op een kleine submanifold in de exponentieel grote Hilbert ruimte leven. Het is aangetoond dat tensornetwerken de fysisch relevante submanifold efficiënt beschrijven. Dit impliceert dat tensornetwerken een krachtig en grondig onderzocht instrument kunnen bieden voor generatief modelleren. We zullen het gebruik van matrix product toestanden als generatieve modellen bestuderen vanwege hun theoretische en computationele simpliciteit. Matrix product toestanden hebben ons inzicht in eendimensionele kwantumsystemen significant vergroot, mede dankzij een krachtig optimalisatie algoritme genaamd density matrix renormalization group.

In hoofdstukken 1, 2 en 3 introduceren we de nodige concepten van respectievelijk tensornetwerken, machinaal leren en generatief modelleren. In hoofdstuk 4 bespreken we hoe matrix product toestanden kunnen getraind worden als generatieve modellen met behulp van een algoritme gebaseerd op density matrix renormalization group. De beschreven

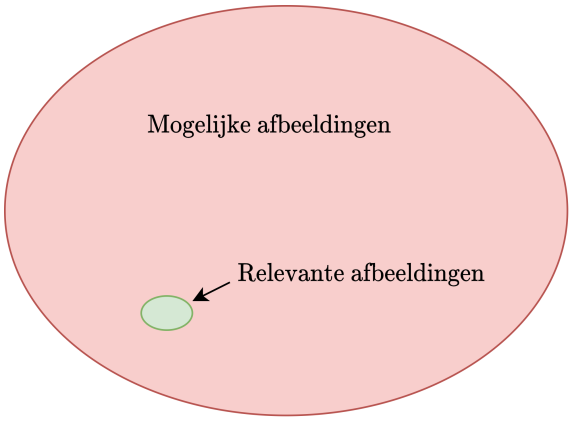
generatie procedure staat ons toe om uit de gemodelleerde waarschijnlijkheidsdistributie efficiënt samples te genereren of reconstrueren. We besteden ook een sectie van hoofdstuk 3 aan zogenaamde generative adverserial networks, die momenteel beschouwd worden als de state-of-the-art generatieve modellen voor afbeeldingen. Deze modellen worden getraind aan de hand van een minimax spel, in tegenstelling tot de gewoonlijke maximum likelihood schatting. Dit kan voordelig zijn omdat we via de maximum likelihood methode niet een afstandsmaat tussen de model en data distributie minimaliseren. Daarom proberen we in hoofdstuk 5 de alternatieve trainingsmethode ook toe te passen op matrix product toestanden. Verder onderzoeken we ook aanpassingen aan het optimalisatie algoritme, zoals een adaptief leertempo en stochastische gradiënt afdaling. Het adaptief leertempo wordt ook vergeleken met de performantie van een ander type tensornetwerk, genaamd boomtensornetwerk. Ten slotte hebben we een eendimensionale dataset ontwikkeld en geanalyseerd die gebaseerd is op discrete sinusgolven. De reden hiervoor is dat de eendimensionale geometrie van matrix product toestanden minder geschikt is voor tweedimensionale data zoals afbeeldingen.

B Populariserende samenvatting

The following science popularization text is part of a collaboration between the thesis students working in the Quantum Group of Ghent University. The goal is to create a 'Quantum Group for Dummies', where the thesis students write Dutch texts that highlight some aspects of the Quantum Group's research and its applications. As the name implies, Quantum Group for Dummies is intended for people who are not familiar with advanced physics. These texts will be combined on the website of the Vereniging voor Natuurkunde [89].

Stel dat je 100 tekeningen van gezichten van mensen moet maken voor een project op de tekenschool, maar je hebt er nog maar 99 gemaakt. Wegens het Coronavirus mag je geen mensen meer bezoeken om te tekenen, dus je gaat zelf een gezicht moeten verzinnen om te tekenen. Je wilt niet dat jouw leerkracht merkt dat de laatste tekening een verzonnen gezicht is, dus de tekening moet zo goed mogelijk bij de andere 99 passen zonder een exacte kopie te zijn, want dat zou de leerkracht opmerken. Stel dat van de 99 gezichten niemand rood haar heeft, dan zorg je er natuurlijk voor dat jouw verzonnen tekening ook geen rood haar heeft zodat de tekening zo weinig mogelijk opvalt tussen de 99 anderen. Ook de kleinere details zijn belangrijk: als de combinatie blond haar en bruine ogen nooit voorkomt in de echte tekeningen, dan zou de verzonnen tekening misschien te opvallend zijn als het wel deze combinatie bevat. Nadat de verzonnen tekening afgewerkt is, belt jouw vriend Harry in volle paniek: hij heeft nog geen enkele tekening gemaakt! Omdat niemand tijd en zin heeft om 100 verzonnen gezichten te tekenen, stel je jezelf de vraag: "Zou een computer dit automatisch kunnen doen voor ons?". Het antwoord is ja! Namelijk met behulp van machinaal leren.

Machinaal leren is een vorm van toegepaste statistiek waarbij algoritmen worden ontwikkeld die taken kunnen uitvoeren zonder expliciete instructies te vereisen. We kunnen in

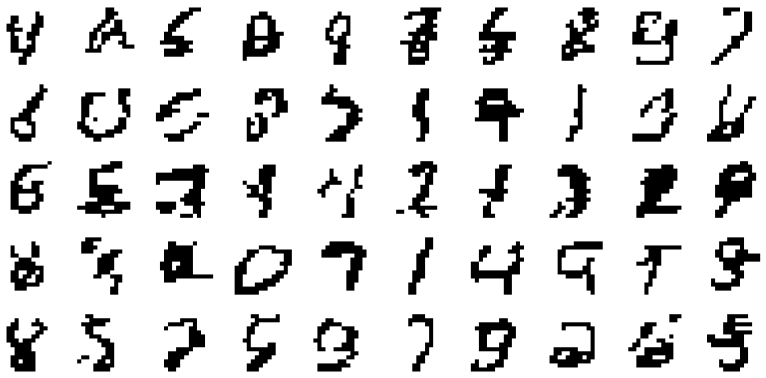


Figuur 1 De relevante afbeeldingen, zoals afbeeldingen van gezichten, zijn maar een kleine fractie van het totaal aantal mogelijke afbeeldingen. De meeste van deze mogelijkheden zien er echter uit als ruis in plaats van afbeeldingen die je bijvoorbeeld met een camera zou maken.

principe een programma schrijven die stukjes van onze 99 echte tekeningen combineert om een nieuwe tekening te maken, maar het resultaat zou geen samenhangend geheel zijn en zou daarom heel makkelijk te onderscheiden zijn van de echte tekeningen. Expliciet een programma schrijven die wel een samenhangende tekeningen creëert is onbegonnen werk. Algoritmen in machinaal leren zoeken naar patronen in data om beter te worden in een bepaalde taak. Dit proces noemen we leren. Machinaal leren is een breed vakgebied en heeft vele mogelijke toepassing. In ons geval zijn onze echte afbeeldingen de data en is het genereren van een afbeelding die op de 99 echte tekeningen lijkt de taak.

Ons doel is nu om een model te verkrijgen van de gezamenlijke kansverdeling van de pixels in de echte afbeeldingen. Zodra we dit model hebben kunnen we bijvoorbeeld pixelsgewijs nieuwe afbeeldingen genereren en is Harry gered. Het generatief model verkrijgen is echter een uitdagend probleem. Het aantal mogelijke afbeeldingen die kunnen gemaakt worden stijgt namelijk exponentieel met het aantal pixels in de afbeelding. In een zwart-wit foto kan elke pixel 256 verschillende waarden aannemen. Dit betekent dat er van een zwart-wit foto van 36 pixels al $256^{36} \approx 10^{86}$ verschillende mogelijkheden bestaan. Dit is meer dan het geschatte aantal deeltjes in het universum, namelijk 10^{80}. Slechts een klein aantal van deze mogelijkheden zijn echter relevant: als je volledig willekeurige pixelwaarden kiest is de kans heel groot dat je afbeelding eruit ziet als ruis in plaats van een gezicht. Dit wordt symbolisch voorgesteld in figuur 1. Een model zoeken dat de relevante mogelijkheden efficiënt beschrijft is een uitdaging. Gelukkig kunnen we ons laten inspireren door fysica.

Kwantum veeldeeltjesfysica beschrijft het gezamenlijk gedrag van vele microscopische deeltjes die elkaar kunnen beïnvloeden via interacties. Een voorbeeld van zulke interacties is atomen die elkaar aantrekken via de magnetische kracht, waardoor bijvoorbeeld een magneet tegen je koelkast blijft hangen. De eigenschappen van kwantum veeldeeltjessystemen beschrijven is over het algemeen heel moeilijk. Gelukkig kunnen we gebruik maken van het feit dat een deeltje enkel zijn dichtste buren zal beïnvloeden. Vergelijk dit met Harry die in Oostende in de zee springt: dit veroorzaakt grote golven in het water



Figuur 2 Resultaat van een generatief model geïnspireerd door kwantum veeldeetljesfysica. Het model werd getraind om afbeeldingen van handgeschreven getallen te genereren.

rondom Harry, maar het water aan de kust van Blankenberge zal er niets van merken. Deeltjes die ver van elkaar verwijderd zijn zullen elkaar dus amper beïnvloeden, of met andere woorden de interacties in veeldeeltjessystemen zijn lokaal. Hierdoor is slechts een kleine fractie van alle mogelijke toestanden fysisch realiseerbaar. In kwantum veeldeeltjesfysica zijn modellen ontwikkeld die op een efficiënte manier de kansverdeling van deze relevante toestanden benaderen. Dit toont opvallend veel gelijkenissen met onze taak, waarbij we enkel de relevante afbeeldingen willen modelleren en naburige pixels elkaar veel meer beïnvloeden dan pixels aan de andere kant van de afbeelding. Bijvoorbeeld als een pixel in het haar bruin is, dan zijn de pixels daarrond waarschijnlijk ook bruin. Daarentegen zegt die bruine pixel bovenaan de afbeelding niets over de pixels onderaan, want haarkleur zegt niets over welke kleur de trui van de persoon heeft.

Wegens de gelijkenissen tussen kwantum veeldeeltjesfysica en generatief modelleren is het logisch om een succesvol model voor veeldeeltjessystemen toe te passen als generatief model. In figuur 2 staan enkele afbeelding van handgeschreven getallen gecreëerd door zo'n generatief model. Aangezien sommige gegenereerde afbeeldingen niet op getallen lijken, is het duidelijk dat er nog veel onderzoek nodig is om deze modellen te verbeteren. Door het nauwe verband met kwantum veeldeeltjesfysica zijn deze modellen een veelbelovende kandidaat om op kwantumcomputers te werken. Kwantumcomputers gebruiken concepten uit de kwantumfysica om bepaalde berekeningen veel sneller te doen dan normale computers. Op dit moment zijn kwantumcomputers nog te beperkt om nuttig te zijn, maar er wordt volop onderzoek gedaan om deze computers verder te ontwikkelen. Machinaal leren vergt veel berekeningen van computers, dus kwantumcomputers zullen waarschijnlijk heel nuttig worden voor machinaal leren.

Bepaalde generatieve modellen hebben de voorbije jaren een heel sterke vooruitgang gekend. De resultaten van een bepaald type generatief model, genaamd GAN, wordt geïllustreerd in figuur 3. De gegenereerde gezichten van modellen uit 2017 en 2018 zijn zo realistisch dat het bijna moeilijk te geloven is dat deze mensen niet echt bestaan! Deze generatieve modellen hebben al vele toepassingen en zijn in staat om onder andere automatisch verzonnen tekeningen te genereren.

Figuur 3 Afbeelding uit Ref. [90] die de snelle vooruitgang van generatief modelleren met behulp van GAN's aantoont. Deze modellen werden getraind om foto's van gezichten te genereren.

Bibliography

[1] Henrik Bruus and Karsten Flensberg. *Many-body quantum theory in condensed matter physics: an introduction*. Oxford University Press, 2004.

[2] F. Verstraete, V. Murg, and J.I. Cirac. Matrix product states, projected entangled pair states, and variational renormalization group methods for quantum spin systems. *Advances in Physics*, 57(2):143–224, 2008.

[3] Edmund Whittaker. Eddington's theory of the constants of nature. *The Mathematical Gazette*, 29(286):137–144, 1945.

[4] Román Orús. A practical introduction to tensor networks: Matrix product states and projected entangled pair states. *Annals of Physics*, 349:117–158, 2014.

[5] David Poulin, Angie Qarry, Rolando Somma, and Frank Verstraete. Quantum simulation of time-dependent Hamiltonians and the convenient illusion of Hilbert space. *Physical Review Letters*, 106:170501, 2011.

[6] Jacob C Bridgeman and Christopher T Chubb. Hand-waving and interpretive dance: an introductory course on tensor networks. *Journal of Physics A: Mathematical and Theoretical*, 50(22):223001, 2017.

[7] D. Perez-Garcia, F. Verstraete, M. M. Wolf, and J. I. Cirac. Matrix product state representations. *Quantum Inf. Comput.*, 7, 2006.

[8] Y.-Y. Shi, L.-M. Duan, and G. Vidal. Classical simulation of quantum many-body systems with a tree tensor network. *Physical Review A*, 74:022320, 2006.

[9] F. Verstraete and J. I. Cirac. Renormalization algorithms for quantum-many body systems in two and higher dimensions. *arXiv e-prints*, pages cond–mat/0407066, 2004.

[10] G. Vidal. Entanglement renormalization. *Physical Review Letters*, 99:220405, 2007.

[11] L. Mirsky. Symmetric gauge functions and unitarily invariant norms. *The Quarterly Journal of Mathematics*, 11(1):50–59, 1960.

[12] J. Eisert, M. Cramer, and M. B. Plenio. Colloquium: Area laws for the entanglement entropy. *Reviews of Modern Physics*, 82(1):277–306, 2010.

[13] F. Verstraete and J. I. Cirac. Matrix product states represent ground states faithfully. *Physical Review B*, 73(9), 2006.

[14] Pankaj Mehta, Marin Bukov, Ching-Hao Wang, Alexandre G.R. Day, Clint Richardson, Charles K. Fisher, and David J. Schwab. A high-bias, low-variance introduction to machine learning for physicists. *Physics Reports*, 810:1–124, 2019.

[15] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[16] Giuseppe Carleo, Ignacio Cirac, Kyle Cranmer, Laurent Daudet, Maria Schuld, Naftali Tishby, Leslie Vogt-Maranto, and Lenka Zdeborová. Machine learning and the physical sciences. *Reviews of Modern Physics*, 91(4):045002, 2019.

[17] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research*, 18(1):6765–6816, 2017.

[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[19] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. *arXiv preprint arXiv:1611.03530*, 2016.

[20] Paul Dokas, Levent Ertoz, Vipin Kumar, Aleksandar Lazarevic, Jaideep Srivastava, and Pang-Ning Tan. Data mining for network intrusion detection. In *Proc. NSF Workshop on Next Generation Data Mining*, pages 21–30, 2002.

[21] F. T. Liu, K. M. Ting, and Z. Zhou. Isolation forest. In *2008 Eighth IEEE International Conference on Data Mining*, pages 413–422, 2008.

[22] Jonathan A C Sterne, Ian R White, John B Carlin, Michael Spratt, Patrick Royston, Michael G Kenward, Angela M Wood, and James R Carpenter. Multiple imputation for missing data in epidemiological and clinical research: potential and pitfalls. *The British Medical Journal*, 338, 2009.

[23] Martin Längkvist, Lars Karlsson, and Amy Loutfi. A review of unsupervised feature learning and deep learning for time-series modeling. *Pattern Recognition Letters*, 42:11–24, 2014.

[24] Deepak Pathak, Philipp Krahenbuhl, Jeff Donahue, Trevor Darrell, and Alexei A. Efros. Context Encoders: Feature Learning by Inpainting. *arXiv e-prints*, page arXiv:1604.07379, 2016.

[25] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[26] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

[27] Honglak Lee, Roger Grosse, Rajesh Ranganath, and Andrew Y Ng. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In *Proceedings of the 26th annual international conference on machine learning*, pages 609–616, 2009.

[28] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[29] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[30] Irina Rish et al. An empirical study of the naive bayes classifier. In *IJCAI 2001 workshop on empirical methods in artificial intelligence*, volume 3, pages 41–46, 2001.

[31] Ronald A Fisher. The use of multiple measurements in taxonomic problems. *Annals of eugenics*, 7(2):179–188, 1936.

[32] In Jae Myung. Tutorial on maximum likelihood estimation. *Journal of mathematical Psychology*, 47(1):90–100, 2003.

[33] Vijay K Rohatgi. *Statistical inference*. Courier Corporation, 2013.

[34] Thomas M Cover and Joy A Thomas. *Elements of information theory*. John Wiley & Sons, 2012.

[35] J. L. W. V. Jensen. Sur les fonctions convexes et les inégalités entre les valeurs moyennes. *Acta Math.*, 30:175–193, 1906.

[36] Ian Goodfellow. Nips 2016 tutorial: Generative adversarial networks. *arXiv preprint arXiv:1701.00160*, 2016.

[37] Andrew Brock, Jeff Donahue, and Karen Simonyan. Large scale gan training for high fidelity natural image synthesis. *arXiv preprint arXiv:1809.11096*, 2018.

[38] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[39] John Bardeen, Leon N Cooper, and J Robert Schrieffer. Microscopic theory of superconductivity. *Physical Review*, 106(1):162, 1957.

[40] John Hubbard. Calculation of partition functions. *Physical Review Letters*, 3(2):77, 1959.

[41] RL Stratonovich. On a method of calculating quantum distribution functions. In *Soviet Physics Doklady*, volume 2, page 416, 1957.

[42] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein gan. *arXiv preprint arXiv:1701.07875*, 2017.

[43] Dominik Maria Endres and Johannes E Schindelin. A new metric for probability distributions. *IEEE Transactions on Information theory*, 49(7):1858–1860, 2003.

[44] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[45] Martin Arjovsky and Léon Bottou. Towards Principled Methods for Training Generative Adversarial Networks. *arXiv e-prints*, page arXiv:1701.04862, 2017.

[46] E Miles Stoudenmire and David J Schwab. Supervised learning with quantum-inspired tensor networks. *arXiv preprint arXiv:1605.05775*, 2016.

[47] Ding Liu, Shi-Ju Ran, Peter Wittek, Cheng Peng, Raul Blázquez García, Gang Su, and Maciej Lewenstein. Machine learning by two-dimensional hierarchical tensor networks: A quantum information theoretic perspective on deep architectures. *New Journal of Physics*, 21(7):073059, 2018.

[48] Angel J Gallego and Roman Orus. Language design as information renormalization. *arXiv preprint arXiv:1708.01525*, 2017.

[49] Zhao-Yu Han, Jun Wang, Heng Fan, Lei Wang, and Pan Zhang. Unsupervised generative modeling using matrix product states. *Physical Review X*, 8:031012, 2018.

[50] Song Cheng, Lei Wang, Tao Xiang, and Pan Zhang. Tree tensor networks for generative modeling. *Physical Review B*, 99:155131, 2019.

[51] Alexander Novikov, Dmitrii Podoprikhin, Anton Osokin, and Dmitry P Vetrov. Tensorizing neural networks. In *Advances in neural information processing systems*, pages 442–450, 2015.

[52] Yoav Levine, David Yakira, Nadav Cohen, and Amnon Shashua. Deep learning and quantum entanglement: Fundamental connections with implications to network design. *arXiv preprint arXiv:1704.01552*, 2017.

[53] Yuhan Liu, Xiao Zhang, Maciej Lewenstein, and Shi-Ju Ran. Entanglement-guided architectures of machine learning by quantum tensor network. *arXiv preprint arXiv:1803.09111*, 2018.

[54] John Preskill. Quantum computing in the nisq era and beyond. *Quantum*, 2:79, 2018.

[55] William Huggins, Piyush Patil, Bradley Mitchell, K Birgitta Whaley, and E Miles Stoudenmire. Towards quantum machine learning with tensor networks. *Quantum Science and Technology*, 4(2):024001, 2019.

[56] Alejandro Perdomo-Ortiz, Marcello Benedetti, John Realpe-Gómez, and Rupak Biswas. Opportunities and challenges for quantum-assisted machine learning in near-term quantum computers. *Quantum Science and Technology*, 3(3):030502, 2018.

[57] Jonas Van Gompel. Thesis. Available: https://github.ugent.be/jpvgompe/thesis, 2020.

[58] Zhao-Yu Han, Jun Wang, Heng Fan, Lei Wang, and Pan Zhang. Unsupervised generative modeling using matrix product states. Available: https://github.com/congzlwag/UnsupGenModbyMPS, 2018.

[59] Max Born. Zur Quantenmechanik der Stoßvorgänge. *Zeitschrift fur Physik*, 37(12):863–867, 1926.

[60] Radford M Neal. Annealed importance sampling. *Statistics and computing*, 11(2):125–139, 2001.

[61] Andrzej Cichocki and Anh-Huy Phan. Fast Local Algorithms for Large Scale Nonnegative Matrix and Tensor Factorizations. *IEICE Transactions on Fundamentals of Electronics Communications and Computer Sciences*, 92(3):708–721, 2009.

[62] Ivan Glasser, Ryan Sweke, Nicola Pancotti, Jens Eisert, and J Ignacio Cirac. Expressive power of tensor-network factorizations for probabilistic modeling, with applications from hidden markov models to quantum machine learning. *arXiv preprint arXiv:1907.03741*, 2019.

[63] Steven R. White. Density matrix formulation for quantum renormalization groups. *Physical Review Letters*, 69:2863–2866, 1992.

[64] J. M. Keller, M. R. Gray, and J. A. Givens. A fuzzy k-nearest neighbor algorithm. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-15(4):580–585, 1985.

[65] Johan Suykens and Joos Vandewalle. Least squares support vector machine classifiers. *Neural Processing Letters*, 9:293–300, 1999.

[66] Asja Fischer and Christian Igel. An introduction to restricted boltzmann machines. In *Iberoamerican congress on pattern recognition*, pages 14–36. Springer, 2012.

[67] David A Levin and Yuval Peres. *Markov chains and mixing times*, volume 107. American Mathematical Soc., 2017.

[68] Chase Roberts, Ashley Milsted, Martin Ganahl, Adam Zalcman, Bruce Fontaine, Yijian Zou, Jack Hidary, Guifre Vidal, and Stefan Leichenauer. Tensornetwork: A library for physics and machine learning. *arXiv preprint arXiv:1905.01330*, 2019.

[69] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[70] Guim Perarnau, Joost Van De Weijer, Bogdan Raducanu, and Jose M Álvarez. Invertible conditional gans for image editing. *arXiv preprint arXiv:1611.06355*, 2016.

[71] Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. Density estimation using real nvp. *arXiv preprint arXiv:1605.08803*, 2016.

[72] Aditya Grover, Manik Dhar, and Stefano Ermon. Flow-gan: Combining maximum likelihood and adversarial learning in generative models. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[73] Michael Gutmann and Aapo Hyvärinen. Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 297–304, 2010.

[74] Ian J Goodfellow. On distinguishability criteria for estimating generative models. *arXiv preprint arXiv:1412.6515*, 2014.

[75] Yann LeCun and Corinna Cortes. The mnist database of handwritten digits. Available: http://yann.lecun.com/exdb/mnist/, 1998.

[76] Edwin Stoudenmire and David J Schwab. Supervised learning with tensor networks. In *Advances in Neural Information Processing Systems*, pages 4799–4807, 2016.

[77] Don P. Mitchell and Arun N. Netravali. Reconstruction filters in computer-graphics. *siggraph Comput. Graph.*, 22(4):221–228, 1988.

[78] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[79] Edwin M Stoudenmire and Steven R White. Studying two-dimensional systems with the density matrix renormalization group. *Annu. Rev. Condens. Matter Phys.*, 3(1):111–128, 2012.

[80] Tsung-Hsien Wen, Milica Gasic, Nikola Mrksic, Pei-Hao Su, David Vandyke, and Steve Young. Semantically conditioned lstm-based natural language generation for spoken dialogue systems. *arXiv preprint arXiv:1508.01745*, 2015.

[81] Yoon Kim, Yacine Jernite, David Sontag, and Alexander M Rush. Character-aware neural language models. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[82] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[83] Feng Pan, Pengfei Zhou, Sujie Li, and Pan Zhang. Contracting arbitrary tensor networks: general approximate algorithm and applications in graphical models and quantum circuit simulations. *arXiv preprint arXiv:1912.03014*, 2019.

[84] Frank Verstraete and J Ignacio Cirac. Continuous matrix product states for quantum fields. *Physical Review Letters*, 104(19):190405, 2010.

[85] Beñat Mencia Uranga and Austen Lamacraft. Schrödingernn: Generative modeling of raw audio as a continuously observed quantum state. *arXiv*, pages arXiv–1911, 2019.

[86] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.

[87] Chris Donahue, Julian McAuley, and Miller Puckette. Adversarial audio synthesis. *arXiv preprint arXiv:1802.04208*, 2018.

[88] Martin Schwarz, Kristan Temme, and Frank Verstraete. Preparing projected entangled pair states on a quantum computer. *Physical Review Letters*, 108(11):110502, 2012.

[89] Bastiaan Aelbrecht. Vereniging voor natuurkunde. Available: https://vvn.ugent.be/, 2020.

[90] David Foster. *Generative deep learning: teaching machines to paint, write, compose, and play.* O'Reilly Media, 2019.

List of Symbols

$\mathbb{1}$	Identity matrix or operator
β	Hyperparameter determining the power in the adaptive learning rate
$\boldsymbol{\theta}$	Parameters of a model
$\boldsymbol{x}$	Vector of features of a sample
$\boldsymbol{y}$	Vector of labels of a sample
$\boldsymbol{z}$	Vector of latent variables
ϵ	Noise drawn from some distribution
ϵ_{cut}	Hyperparameter that determines the cutoff for retaining singular values
η	Number of elements in a merged tensor
γ	learning rate
$\|\Psi\rangle$	State of a system represented by a MPS
$\mathcal{E}$	Vector containing the environment during MPS optimization
$\mathcal{L}$	Loss function
μ	Average of a Gauss
ν	Frequency of a wave
ϕ	Phase of a wave
σ	Standard deviation of a Gauss
$A^{(n)}$	Tensor at site n of a MPS
c	Coordination number of a Cayley tree
$C(L)$	Two point correlation function of operators separated by L sites
D	Bond dimension of a MPS
$d(\boldsymbol{x})$	Discriminator function in a generative adversarial network

E_O	The O-transfer matrix
$g(\boldsymbol{z})$	Generator function in a generative adversarial network
H	Hilbert space
L	Number of sites between two operators
N	System size and number of sites in a MPS
O	An arbitrary operator
p	Physical dimension of a sample, meaning the number of possible values a feature can take
P_d	Probability distribution function of the data
P_m	Probability distribution function defined by a model
S	Von Neumann entropy
s_i	Singular values of a singular value decomposition
T	An arbitrary tensor with a high rank
T	Number of training samples
v	Value function, also referred to as objective function, of a minimax game
Z	Partition function

List of Figures