Neural quantum states from a computational perspective

by
Sebastian Testanière Overskott

Thesis

for the degree of

Master's of Science

Department of Computer Science
Faculty of Technology, Art and Design
Oslo Metropolitan University

August 2023

Abstract

The main problem of computational quantum physics is the the Curse of Dimensionality, that is the exponential growth of the length of the description, needed to specify a many-body/part quantum state, with the increase of the number of parts the system consists of. Advances in the field on Machine Learning brought new perspectives, and, relatively recently, the idea of Neural Quantum States (NQSs), that are the states which can be encoded with a specific type of artificial neural networks, the so-called Restricted Boltzmann Machines (RBMs), was proposed. It was shown that NQSs are able to capture complex quantum states characterized by a high degree of entanglement, while keeping the length of the state description under control. In this thesis we consider the NQS concept from an 'operational' point of view, by addressing such computational aspects as scaling of time and memory size (the number of parameters needed to specify a NQS) with the size of a quantum model. For benchmarking we use two scalable models, a completely random Hamiltonian and an Ising chain. We demonstrate that the scaling are essentially different for these models which result highlights an interesting link between quantum physics and Machine Learning. We also discuss different strategies which could help to enhance the convergence, while keeping computational cost at a minimum. Especially, we exploit the structure of the Ising Hamiltonian to speed up the training of the network.

This thesis has resulted in a Python framework that implements Neural Quantum State (NQS) as a way to approximate ground state energies for discrete quantum spin systems. The framework can implement NQSs for general Hamiltonians and Ising Hamiltonians, and trains the NQS by using a variational quantum Monte Carlo method to approximate the energy ground state of discrete quantum spin systems. We show that our implementation can achieve promising results with sub-exponential growth of computational cost for the Ising model. We propose a heuristic $N_\lambda = K \cdot N_\sigma^3$ for finding a good number of Markov Chain Monte Carlo (MCMC) steps, which makes the sampling independent of the system size. The experiments done in this thesis shows that the implementation manages to approximate the ground state energy of structured Ising Hamiltonians with sub-exponential number of parameters.

Preface

I consider my self a computer engineer first, but the world of science and physics has always amazed and inspired me. This thesis has been a perfect marriage between programming and physics. At times, it has left me beaten and exhausted, but has also given me a childlike excitement when the pieces finally came together. It's been a wonderful journey, but now I'm glad we're at the end.

First, I want to thank my supervisor Sergiy Denysov for providing excellent notes, interesting discussions, for proposing this project, and having faith in my abilities to complete its challenges. I hope you continue to inspire programmers to take a leap into the quantum world.

My deepest gratitude goes to my co-supervisor Kristian Wold for hours upon hours of bug searching, explaining (and re-explaining) countless topics, providing great feedback, and meaningful conversations in general. You are way to modest and you have been a huge inspiration for me. Without your patience and knowledge, I would not have made it until the end.

I want to thank my fellow student Diedrik Leijenaar Oksnes for 5 years of studies, work and collaboration.

My thanks to professor Sølve Selstø for introducing me to quantum physics and the wonders of mathematics in programming. It has been 5 very interesting years for me leading up to this thesis, much thanks to you.

Huge thanks to my family and friends for amazing support.

A special thanks to my amazing wife Anna Solveig Julia Testanière Overskott. You inspired me to start my studies, and have been supporting me fully, all the way. I could not have managed this without you. Thank you for valuable thoughts on learning, for aiding me in exam training, and proof reading hand-ins for almost 6 years. You are as brilliant as you are beautiful.

Finally, I want to thank my two boys, Léon and Théodore, for showing me that there are, of course, more important things in life than this thesis.

Sebastian Testanière Overskott - Oslo, August 2023

Dedicated to my dad, Andrés Overskott

who saw the beginning, but not the end of this project

List of Abbreviations

AG	Analytical Gradient.
BM	Boltzmann Machine.
FD	Finite Difference.
GD	Gradient Descent.
MBP	Many Body Problem.
MCMC	Markov Chain Monte Carlo.
ML	Machine Learning.
MPS	Matrix Product State.
NN	Neural Network.
NQS	Neural Quantum State.
QC	Quantum computers.
QT	Quantum Technologies.
RBM	Restricted Boltzmann Machine.

Contents

1 | Introduction

There is a substantial progress was made in quantum physics, both theoretical and experimental, during the last two decades. It was boosted by the advances made in Quantum Technologies (QT) which altogether led to what is called now the 'Second Quantum Revolution' [1]. The first revolution, which has happened with the establishment of quantum physics and discoveries of fundamental quantum phenomena (such as tunneling, state superposition, and entanglement) about a century ago, has substantially changed our view on the world we live in. The second revolution marks the beginning of a practical use of these phenomena to our advantage. Quantum computing, quantum communications, quantum metrology, are quantum chemistry are the fields where this revolution is currently ongoing.

During the last decade, Quantum computers (QC) changed their status from the imaginary 'devices' (similar to the Turing Machine at the early stage of computer science) to the real-life computational platforms which can be accessed remotely. QC are expected to open a new way of computing because it was shown that they - as the imaginary devices - are able to solve some problems faster then their classical counterparts. For example, it has been been proven that an ideal QC is able to find a specified element in an unstructured array of N elements in $O(\sqrt{N})$ steps, while the classical computer cannot do it faster then in $O(N)$ steps[2]. The world of cyber security trembled when it was shown that the to crack the widely used RSA encryption algorithm could be cracked an ideal QC in seconds instead of eons [3].

One of the key features which makes quantum computers superior with respect to the classical ones is the exponential growth of the information which can be encoded (which can be encoded into a state of a quantum computer) the number of qubits he computer consists of. This feature creates a vast space to store and process information by implementing such genuine quantum effects as superposition and entanglement. While being a bliss for quantum computers, it is a curse, the Curse of Dimensionality, for the computational quantum physics and quantum chemistry. The is no way one could store the description of an arbitrary state of a system of 50 qubits even on largest existing computational cluster – simply because of the lack of memory.

For quantum physicists, the Curse of Dimensionality is know as the infamous Many Body Problem (MBP). This one of the last problems that still is lacking a standardised high-performance computational framework that most other fields of science do [4]. The history of computational quantum physics can be considered as a story of a continuous struggle against the MBP, by looking for more efficient and sophisticated ways to compactify the descriptions of many-body states and thus grasp more and more complex and interesting quantum states with the same memory size. In this thesis, we will look at one of the latest findings along this line.

1.1 The many body problem

Physics that describe the world as we perceive it, is called non-quantum physics. It is an extremely useful tool that sits at the core of almost every creation or discovery done in the last centuries. It can be used to describe the grand celestial movements in space, and the invisible forces of electromagnetism in our phones. Here's an example: Let us say we want to describe the movement of a ball's movement in a room. We can then utilise its position and velocity at every point in the space. This would require three numbers representing the movement in the three dimensional space. For each additional ball we add, we need three more numbers to describe the movement of all the balls. In short, the number of variables in this "problem" are the number of dimensions times the number of balls. $k \cdot n$, where k is the number of dimensions and n is the number of balls.

When the scale gets on the level of atoms, the laws of non-quantum physics are not capable of accurately describing the physical phenomenons that exist at this scale. Here, the laws of quantum physics reign. If we now look at atom sized particles instead of balls we will need some number of variables to describe them as before. The amazing/terrifying difference is noticeable when more particles are added to the equation. Because of quantum properties (we will look at some later) the complexity grows exponentially as k^n instead of the linear $k \cdot n$ [5]. This also leads to an exponential increase in required resources, like computer memory, and is what makes calculations of quantum systems with only a handful of particles impossible to do accurately. Even numerical approximations struggle when the systems are on a molecular size.

Still, this explosion in complexity is possible to harness and use to our advantage. As the godfather of quantum computing, Richard Feynman said: “Nature isn't classical, dammit, and if you want to make a simulation of nature, you'd better make it quantum mechanical” [6]. He foresaw a possibility to exploit the complex quantum systems to our benefit. QT and especially QC has gotten much attention lately. QC are getting bigger and better and there are predictions they will outperform classical computers in some areas of computing [7]. The computing power of QC is rapidly getting greater, as more and more companies and institutions like IBM [8], Google [7] and Amazon [9] are getting engaged in quantum information technology. This will potentially decrease processing times, and be an energy efficient alternative to the classical super computers. The main reason for this is the exponential increase in processing power we get from an additional quantum bit (qubit), compared to the linear increase in the classical realm. To store all the information in a QC with 50 qubits, we will need to store 2^{50} numbers on a classical platform, which is a petabyte of data! That is, only to store the information, without performing any calculations. This is an excellent example of the complexity of the MBP. A QC with 50-100 perfect qubits (often called logical qubits, meaning the qubits and quantum gates have non-impacting noise and essential infinite coherence times) is needed to make a QC faster than today's computers. In the current scene of QC, we use whats called Noisy Intermediate-Scale Quantum (NISQ) computers. Today's qubits (often called physical qubits) are far from perfect and are susceptible for noise, and have limited coherence times. In this environment we will need a lot more qubits to apply quantum error correction algorithms. The goal for most manufacturers of QC today is to create machines with several thousands (if not millions) of qubits.

1.2 Machine Learning

Machine Learning (ML) has had an enormous growth the last decade. It has proven to be a solution to problems that were unsolvable earlier and given us insight in data in a much more complex manner than traditional statistics. The Two words in the term "machine learning" reveals some of the idea behind its working parts. The machine is a highly dimensional, non-linear function $F(x; p_1 \dots p_{N_p})$ with input x and the parameters $p_1 \dots p_{N_p}$. The learning is the process of tuning the parameters p with a stochastic optimization algorithm, so that it minimizes a loss function for an input, x_n i.e. $\mathcal{L} = (x_i)|F(x_i; p) - y_i|$ [10].

How do we create an algorithm that can distinguish images of cats from images of dogs? Approaching this with a standard programming paradigm seems like an insurmountable task. But we have an abundance of images of what cats and dogs are. The process in general is to produce algorithms driven by data. What we need is a desired outcome, and a machine that has enough flexibility to translate the input data to the desired output. The machine is just a very expressive mathematical function. This means that it has a lot of "knobs to turn" i.e. the parameters p. The process of learning or training is just tuning these knob so that we get as close to the desired output as we can. By utilising already tested and proven methods for optimization (i.e. gradient descent and more sophisticated methods), and the power of modern computers, we can systematically turn the knobs of the machine so that it is sensitive toward the difference between cats and dogs. In the end we understand the method for producing such algorithms, but we have no intuition for how the final, tuned mathematical function actually is able to distinguish cats from dogs.

One such machine that has become quite popular is the Artificial neural networks, more often just called Neural Network (NN). The idea of utilizing NNs, came in the 1950-60. It was presumed the model functioned like neurons in a brain[11] hence its name. They are usually complex connections of nested functions, often with many layers. NNs has shown very good results in a broad array of fields e.g. finance [12], cyber security [13], and recognising microorganisms [14].

1.3 Methods of approximation

The complexity of quantum many-body systems grows exponentially and is one of the big challenges in modern theoretical physics [15]. Already at ~ 5 particles the distribution will look close to a normal distribution. Although it is a daunting task, the rewards of taming the MBP are great. It is an important problem to solve in almost all areas of QT. Luckily, we do not need to represent (in most cases) the full capacity of a system to gain insight from it. This means we can get a very good estimate without representing the whole system. We will have to beware of what we are loosing in the process. In the last decades, the increase in computational power has made it possible to create some very good methods of approximation. This has lead to many different approaches of solving with each its own benefits and challenges. One of the basic approximation of quantum states is the product state 2.10.1. It is the most crude approximation and lack the power to express quantum phenomena, but can still be relevant for some non-interacting systems. This representation grows linearly with the number of particles in the system ($\sim N_\sigma$), and is thus a quite cheap method computational-wise. A more sophisticated method is the Matrix Product State (MPS) 2.10.2. Sometimes called "tensor states", it was a very innovative approach to model many-body quantum states when it was introduced almost two decades ago [16]. It is based on the idea of low-rank approximations of high-dimensional tensors. The main control parameter is the *bond degree* N. Once N is fixed the MPS grows as ($\sim N \cdot N_\sigma$). The MPS is able to capture more complex quantum states i.e. interactions between the particles, but *short-ranged*, meaning the particles only interact with neighbouring particles. Long-ranged entanglement demands exponentially large N. Last, we have the NQS. Also called Neural Network Quantum States [11], it is a ML approach to solve the MBP. There's been many success in using neural networks and machine learning on many-body system [17] [18][19][20]. The NQS can capture long-range entangled states, possibly at a lower cost than MPS. The main parameter is the size of the *hidden layer* N_h. Once N_h is fixed, the number of parameters (length of vector θ) scales ($\sim N_h \cdot N_\sigma$). The reach each method has into the space is different, with the parameter N fix for the three methods. If figure 1.1, we see that the product state only captures a line in the space. The MPS manages to reach into the space, capturing more of the quantum states. Finally we see the NQS reaching even further into the space, thus being able to reach even longer.

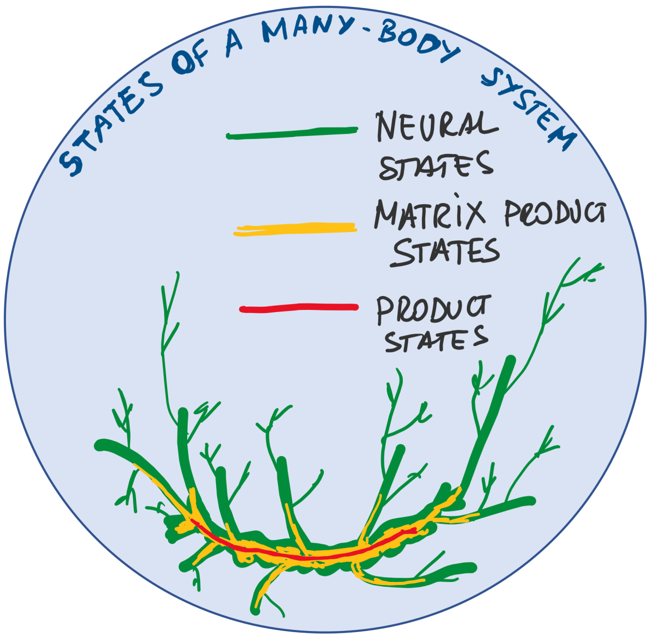


Figure 1.1: A sketch of the MBP space, and how far each of the methods presented in 1.3 can reach with the same, fixed value N.

1.4 An analogy

The idea behind using a neural network to represent or approximate a quantum state is somewhat analogous to how we compress digital images. Digital images are built up of millions (usually) of pixels i.e. small squares on the computer screen that can take on one color. If the image is just noise, that is all the pixels take on a random colors, we would need information of all the pixels to recreate the image. All the pixels are equally important. If we have a picture of something more concrete, let's say a dog, not all the pixels have the same importance. It the dog picture we find *structure*. The fur of the dog will have more or less the same color, so all the pixels representing the fur will have almost the same value. The same applies if it's a tree in the background, or anything else we can recognise from noise. In an image compression procedure, this structure is utilised to remove information from some of the pixels, and instead generate the values again when needed, based on the neighbouring pixels' values. The image is not identical after compression, i.e. the pixels are not the same, but the *information* is approximately preserved. We can still see that it is a dog.

Now here's how we can relate a microscopic quantum system to digital images. Nature, molecules and atoms that surround us, have a lot of structure. An electron is equal to all other electrons in terms of e.g. mass and charge. The same applies to neutrons, protons and all other parts that make up matter. The repetitiveness in the particles properties creates predictable patterns. Our aim is that the NQS will be able to find these patters and compress the state for us. Just as a image compression algorithm does with a digital image. Almost all real-life quantum systems have, as the dog picture, much structure. This makes it possible in most cases to "remove" some of the information from the system, and a bit like how we retain the important information in a compressed digital image, we can retain the important information of a compressed quantum system.

1.5 Study objective

The goal of this thesis is to explore the RBM's ability to approximate an unknown quantum system by imitating its probability distribution. In particular, we want to examine *discrete spin systems* as found in QCs, and the RBM's ability to learn the ground state 2.9 of these systems. The first steps are to explore the theory and state-of-the-art methods for using ML based approaches to solve the MBP i.e. NQSs. We are then going to derive the mathematical expressions we need to implement these theories in code. Especially the analytical expression for the gradients of the RBM will be derived and presented. The second step is to implement a NQS in the programming language Python along with optimization techniques and a stochastic sampling algorithm. We will use a Monte-Carlo algorithm as a cheaper alternative than calculating the whole wave function explicitly. We will use this cheaper approach to estimate the systems *local energies* 5.1 which can be used as an estimator for a quantum systems energy. This is a exponentially cost for generic Hamiltonians, but under strong assumptions of the Hamiltonian we can make it linearly cheap. A Ising Model Hamiltonian will be used for this purpose 2.7.3. Lastly, we will run experiments with our implementation of both models. The impressive accuracy of the NQS on real-life problem has been extensively studied [21] [11] [20] [22]. The ability to encode many of the otherworldly quantum effect in a classical way makes the RBM very interesting tool in quantum sciences. In this thesis we want to study the importance and computational impact of the different "moving parts" of the

NQS form a computational perspective.

To summarize:

- Derive and present expression for the mathematical expressions RBM in the computational basis 2.1.
- Create a Python module that implements RBM neural network that can encode information of given quantum system models.
- Implement a sampling method that potentially is more efficient than brute force calculation of the probability distribution.
- Implement two optimization algorithms: One well known from numerical approaches Finite Difference (FD), often used for machine learning optimization, and an analytical expression of the RBM that potentially is more efficient.
- Explore performance of the RBM a physical system model and on a generic model.
- See how the parameter space of the RBM impacts accuracy and performance.

1.6 Outline

In part I of this thesis we will present the theory behind the MBP. Here we give a quick introduction to relevant theory on quantum mechanics, neural networks, stochastic sampling, and relevant error measures for performance testing. Part II describes the implementation of the theory into code as a Python framework. In part III we present the results of experiments done with the implemented code, and discuss the results in context with the theory. Finally, in part IVwe summarize and conclude the project and have a look at what can be done with future work on this project.

Part I

Theoretical background

2 | Quantum mechanics

To get an understanding of the many body problem we need to start by looking at its smallest components, the quantum particles. A quantum particle is an entity at the atomic scale, e.g. electrons, molecules and atoms, that has *quantum properties*. These quantum properties are phenomena that often baffles due to their un-intuitive behaviour. Being everywhere at once, moving through impassable barriers and behaving like a particle and a wave at the same time are all perfectly normal for a quantum particle. Throughout this thesis there will be several encounters with quantum mechanics. For that purpose I want to introduce some common notation used in the field and also introduce some important concepts that will be referred to throughout the paper. This chapter is based on [5] if noting else is mentioned.

2.1 The wave function

A quantum system is best described by *quantum mechanics*. Here, an isolated quantum system (meaning we ignore disturbance from outside the system) can be described by the wave function ψ. It is a description that can be used to derive properties like position, momentum, and energy, even if, as is often is, it is hidden from us. It also encapsulates the quantum particle's wave-like behavior. ψ can also be written as a complex vector of amplitudes, where each amplitude corresponds to one particular classical state, called the *state vector*. Even if the wave function contains information of all the different possible states it can take, "looking" i.e. measuring the particle will only show us one of the states.

2.2 Particle spin and qubits

There are many entities in the quantum world, but in this project we want to limit us to one type. Some particles show a property called spin. It is related til angular momentum, but the import part is that it can "spin" in either of two directions. We call these directions spin up ($\uparrow$), and spin down ($\downarrow$). This makes spin an binary property with binary outcomes i.e we can define $(\uparrow) \equiv 0$ and $(\downarrow) \equiv 1$. We already know very well from classical computing that binary values can be used to do calculations, and the same applies to particles with spin. they can be used to store a value of spin up or spin down. Or as we more commonly know them, 1 and 0. When we use a particle in this way we call them quantum bits, or *qubits*.

In this project, we are going to be working with representing quantum systems that has discrete spin properties as found in quantum computers. This means a particle and a qubit will be meaning the same thing forward. A qubit represented by the ψ can only take the values 0 or 1 when we investigate it.

2.3 Hilbert spaces and some notation

Since the states of a quantum system exists in a vector space, they are most often represented as vectors. In quantum mechanics these vectors are commonly written using *Dirac notation*. Here, a row vector is denoted

$$\langle A| \equiv [A_0^*, A_1^*, \dots, A_n^*], \tag{2.1}$$

where the * operator denotes complex conjugation. And a column vector is written

$$|B\rangle \equiv \begin{bmatrix} B_0 \\ B_1 \\ \vdots \\ B_n \end{bmatrix} \tag{2.2}$$

The basis of a vector space is the collection of vector $|\psi_n\rangle$so that any vector $|\psi\rangle$ can be rewritten as a linear combination of vectors from the basis.

$$|\psi\rangle = \sum_i c_i \, |wf_i\rangle \tag{2.3}$$

The basis formed by a discrete spin system is dependent on the number of particles. for n qubits we will have 2^n possible states that can be expressed as a collection of all the states where each state is one of the integers $\in (0, 2^n - 1)$ as shown in table 2.1. This basis is the *operational basis*, or sometimes called the standard basis, is the basis we will utilize for this thesis.

The operational basis is orthogonal. This meaning it has a Krönecker product $\langle A_n|B_m\rangle = \delta_{nm}$ i.e. equals 1 if states are the same and 0 otherwise. To reiterate, the basis or state space is describing all the possible states for the quantum

Operational basis
$0 = \|0...00\rangle \equiv \|0\rangle$
$1 = \|0...01\rangle \equiv \|1\rangle$
$2 = \|0...10\rangle \equiv \|2\rangle$
$\vdots$
$n-1 = \|1...11\rangle \equiv \|2^{n-1}\rangle$

Table 2.1: A table showing how we represent the basis-states for n qubits

system. The vector space of quantum mechanics is a *Hilbert space*. A Hilbert space is a vector space with a well defined inner product and an outer product. The compact nature of Dirac notation also makes it convenient to write these. As a consequence of the orthogonal basis, the inner product is written in the following way:

$$\langle A|B\rangle \equiv \sum_{j,i} \langle i| A_i^* B_j |j\rangle = \sum_{i,j} A_i^* B_j \langle i|j\rangle = \sum_i A_i^* B_i \tag{2.4}$$

The outer product is a operator and is written:

$$|B\rangle \langle A| \equiv \sum_{i,j} |i\rangle B_i \langle j| A_j^* = \sum_{i,j} B_i A_j^* |i\rangle \langle j| \tag{2.5}$$

2.4 Superposition

Superposition is perhaps the most famous of the quantum properties. A few sections ago, we explained how the system could contain information of several states, but only one of the state would be revealed to us if we measured the system. Superposition is the single particle ability contain information about several (or all) classical states at once. Superposition is mathematically written as a linear combination of the states:

$$|\psi\rangle = \alpha |0\rangle + \beta |1\rangle \tag{2.6}$$

We call α and β are the probability *amplitudes* of each state α and β. They differ from regular probabilities in that they can be complex, not just real, positive values. This property makes it possible to add quantum states with destructive interference. This has no analogy in standard probability. One very fascinating definition is that the probability for each of the outcomes to occur is $P(0) = |\alpha|^2$ and likewise with $P(1) = |\beta|^2$. The absolute value squared of the amplitudes can be interpreted as a standard probability. We therefore require then that the sum of amplitudes squared is one, meaning it is normalized and preserves that a probability never is more than 1 (it has to be in one state or the other).

$$|\alpha|^2 + |\beta|^2 = 1 \tag{2.7}$$

More generally we write the superposition as a sum of many states and amplitudes.

$$|\psi\rangle = \sum_i \Psi_i |\psi_i\rangle \tag{2.8}$$

$$\sum_i |\Psi_i|^2 = 1 \tag{2.9}$$

Where Ψ_i is the complex amplitude associated with state ψ_i i.e. α and β from the two-state system (2.6), and ψ_i is the respective state i.e. $|0\rangle$ and $|1\rangle$ from the same example. When we want to get information out of the QC we need to *measure* the qubits. This interference with the quantum system forces it to "choose" one of the outcomes 1 or 0, even if it stored information about both states at the time of measurement. This means we can never know the underlying probability after just one measurement. We need to prepare the qubit several times in the exactly the same way and measure again. This is repeated some thousand times. Collecting the results of each measure, we get an probability distribution that represent the state i.e. $| |\psi\rangle |^2$.

2.5 Observables and expectation values

Observables are operators that act on the wave function in question. Often denoted $\bar{O}$. It serves as a method for extracting a deterministic value from a quantum state [23]. The observable can be any information we want to gather knowledge about, this being position, momentum or, as we are going to utilize in plenty during this project: energy.

The expectation value, a word and propertyfrom statistics, of an observable is denoted $\langle\bar{O}\rangle$. For an unnormalized wave function, it can be found by calculating:

$$\langle\bar{O}\rangle = \frac{\langle\psi|\bar{O}|\psi\rangle}{\langle\psi|\psi\rangle} \tag{2.10}$$

We are going to encounter this expression several times throughout this thesis.

2.6 Entanglement

One of the most important, if not **the** most important quantum property in quantum computing, is entanglement. In short it means that we can prepare two or more qubits in such a way that if you measure one of the entangled qubits, the other qubits will be forced to "choose" a state at the exact same time. Even if you didn't measure the second qubit. This apply even if the qubits are separated by great distances or isolated from each other in other ways. An example of an entangled state.

$$|\psi\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}} \tag{2.11}$$

The equation (2.11) is one of the *Bell states* that are famous in quantum mechanics. It shows the absurdity of entanglement in a very compact and easy to read way. Here we can see that there are two states, $|00\rangle$ and $|11\rangle$, that both have the amplitude $\frac{1}{\sqrt{2}}$ meaning it would be 50/50 chance to get one of the two outcomes. If we measure the first qubit to be zero state, the other one is going to be zero state as well. And if the first is one state, the second is one state as well. The strange implications this shows is that is it not possible to have an outcome of $|01\rangle$ or $|10\rangle$ even if the individual qubits can be either $|0\rangle$ or $|1\rangle$.

2.7 Hamiltonian

The total system energy, both kinetic and potential energy, and any external forces influencing the system, are represented by the system Hamiltonian. it is most often represented mathematically as a matrix $\hat{H}$ that fulfils certain properties. It is a *hermitian* operator meaning $\hat{H}_{ij} = \hat{H}^*_{ji}$, with all of its eigenvalues representing all possible energies the Hamiltonian system can have and the eigenvectors are the respective eigen-states. It is more often than not that we do not know the complete description of the system energy, and thus the Hamiltonian. For many physical system the Hamiltonian can be very intricate.

2.7.1 Matrix Hamiltonian

Given the notation introduced in section 2.3, expressing the Hamiltonian as a matrix fits the mathematics very well. We can then easily use the operator on the vectors as we will see in further sections. One drawback with the matrix Hamiltonian representation is the its size scales exponentially with the system it the describes. For the operational basis this is 2^n where n is the number of qubits.

If the system described by the Hamiltonian is completely generic i.e. describing noise, we would need this increase in size to grasp it completely. More often than not, physical systems have a lot of structure, and as we shall see contain a lot of zero elements in its matrix representation.

2.7.2 Generic Hamiltonian

A generic or random Hamiltonian is a Hamiltonian where all elements are drawn from a random distribution. In this thesis they are from a normal distribution with expectation value 0 and variance 1 $N \sim (0,1)$. Solving a generic Hamiltonian can be used as a very strict benchmark, since it has no structure and requires a more flexible approach than physical systems do. In other words: If your approach is able solve the generic Hamiltonian of a given size, it can solve a real life problem of the same size as well.

2.7.3 The Ising model

The Ising model is a model for spin systems with interacting and external forces. It has been used to describe several phase shifting physical systems e.g. polarization in ferromagnets [24]. It consists of a lattice of spins (not confined to quantum spins) which can only take on directions $\uparrow$, $\downarrow$ often represented as $\{-1, 1\}$ or as we will do in this thesis, $\{|0\rangle, \langle 1|\}$. The state of the system is the combined spin configuration e.g. a six spin configuration $(\uparrow\downarrow\downarrow\uparrow\downarrow\uparrow)$. The spins interact with their nearest neighbour and can also be influenced by an external field. the general Ising Hamiltonian can be written.

$$\hat{H}_{\sigma_i} = \sum_{i>j} -\gamma_{i,j}\sigma_i\sigma_j - \sum_i \mu\sigma_i \tag{2.12}$$

Where $J_{i,j}$ is the interaction between neighboring spins, σ are the spins, γ is the external force applied on each spin.

In this thesis we have looked at a more concrete Ising system as a model for our qubits. It has no external field.

$$\hat{H} = \sum_{i=1}^{n-1} \gamma_{i,i+1} \cdot \mathbb{1}^{(1)} \otimes \mathbb{1}^{(2)} \otimes \cdots \otimes X^{(i)} \otimes X^{(i+1)} \otimes \cdots \mathbb{1}^{(n)}, \tag{2.13}$$

where $\mathbb{1} = \left[\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right]$ is the identity matrix, $X = \left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right]$ is the Pauli-X operator, and $\gamma \in \mathbb{R}$ is a scaling factor. It can also be written as

$$\hat{H} = \sum_{i=1}^{n-1} -\gamma_{i,i+1} \mathbb{X}_i \mathbb{X}_{i+1}, \tag{2.14}$$

where $\mathbb{X}$ is the X operator acting an the whole space. This representation of the Hamiltonian are we calling *Tensor product Hamiltonian.*

The Ising Hamiltonian matrix of dimensions $d \times d$ will have $\sim d$ number of non-zero elements meaning of all the elements in the matrix it is only a square root of total elements that are giving us any information. Matrix representation of this Ising Hamiltonian is an very inefficient way to both store, access and calculate/diagonalize. Recall that our Ising model requires γ-values numbering $n-1$ where n is the system size. For a two spin system requiring only one γ-value has a Hamiltonian matrix that look like this:

$$\begin{bmatrix} 0 & 0 & 0 & \gamma \\ 0 & 0 & \gamma & 0 \\ 0 & \gamma & 0 & 0 \\ \gamma & 0 & 0 & 0 \end{bmatrix}$$

We need to store 16 values where four of them are relevant. For a slightly bigger system where $n = 3$, we would need a matrix of this caliber:

$$\begin{bmatrix} 0 & 0 & 0 & \gamma_a & 0 & 0 & \gamma_b & 0 \\ 0 & 0 & \gamma_a & 0 & 0 & 0 & 0 & \gamma_b \\ 0 & \gamma_a & 0 & 0 & \gamma_b & 0 & 0 & 0 \\ \gamma_a & 0 & 0 & 0 & 0 & \gamma_b & 0 & 0 \\ 0 & 0 & \gamma_b & 0 & 0 & 0 & 0 & \gamma_a \\ 0 & 0 & 0 & \gamma_b & 0 & 0 & \gamma_a & 0 \\ \gamma_b & 0 & 0 & 0 & 0 & \gamma_a & 0 & 0 \\ 0 & \gamma_b & 0 & 0 & \gamma_a & 0 & 0 & 0 \end{bmatrix}$$

Here, of the 64 elements, only 16 are relevant. The structure in these systems are also visible in the matrix representation.

2.8 The Schrödinger equation

Like Newton's equations govern how objects move in time at our scale, Schrödinger's equation does the same for atomic and subatomic particles. It has several forms, but in this paper we will use the *time independent* Schrödinger equation presented as a linear algebra equation in (2.15) below.

$$E\,|\psi\rangle = \hat{H}\,|\psi\rangle \tag{2.15}$$

Where E is the energy, ψ is the wave function, and $\hat{H}$ is the system Hamiltonian. The time independent equation describes a system that does not change in time, only in space. Solving the Schrödinger equation will give us the eigenvalues and eigenvectors of the Hamiltonian. The eigenvalues are the systems possible energies, with eigenvectors describing its related wave function i.e.

2.9 Ground state of a quantum system

One of the many properties we find in quantum system that is not present in classical physics, is the quantized energy levels of a quantum system. It is this behavior that origins the "quantum" in quantum mechanics. The first level is often called the *ground state* and other states *excited states*. Given an Hamiltonian $\hat{H}$, we can find all the energy states and the corresponding wave functions or system states by solving the Schrödinger equation. This is analytically done by *diagonalization*. We can also find the expectation value for the Hamiltonian by solving 2.16. The expectation value of the Hamiltonian is the system energy. If we have an unnormalized wave function (or we are not sure it is normalized) which is most often the case when choosing an ansatz we will need to normalize it the following way.

$$E = \frac{\langle\psi|H|\psi\rangle}{\langle\psi|\psi\rangle} \tag{2.16}$$

The most interesting of these energy levels is the *ground state.*

$$E_{gs} = \underset{|\psi\rangle}{min}\left(\frac{\langle\psi|H|\psi\rangle}{\langle\psi|\psi\rangle}\right) \tag{2.17}$$

where E_{gs} is the ground state energy. The ground state tells us the energy level of the system when it is in it lowest energy configuration. Around room temperature, the state of a molecule is very, very close to its ideal ground state. This is very use full in fields as chemistry and material design. The ground state is the lowest energy the system can have. All other energy levels are called excited and there can be many of them. the ground state is also the state we most often will find systems in. Nature is 'lazy' and will try to contain as little energy as possible.

2.10 Methods of approximation

This is a short section describing in some more detail the alternative quantum system description presented in the introduction 1.3. It is meant to illustrate some of the points made in more technical terms, without dive deep into definition.

2.10.1 Product states

The most crude approximation of a quantum system.

$$\psi_{\sigma_1,\sigma_2,\dots,\sigma_n} = \psi_{\sigma_1} \cdot \psi_{\sigma_2} \cdot \dots \cdot \psi_{\sigma_n}, \tag{2.18}$$

where If we use this way of representing let's say a two spin system: two qubits. They can be measured to be in one of four states: $|00\rangle, |01\rangle, |10\rangle$ or$|11\rangle$. We can

$$\begin{aligned}\psi &= \frac{a|0\rangle + b|1\rangle}{\sqrt{2}} \otimes \frac{c|0\rangle + d|1\rangle}{\sqrt{2}} \\ &= \frac{ac|00\rangle + ad|01\rangle + bc|10\rangle + bd|11\rangle}{2}\end{aligned} \tag{2.19}$$

If we now look at a Bell state which is entangled, presented in section 2.6. The wave function of this state is $\psi = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$. We see that the ad amplitude from (2.19) has to be 0. This means either $a = 0$ or $d = 0$. If $a = 0$, then ac would be 0 not $\frac{1}{\sqrt{2}}$. In other words: there is no possible way to express the entangled Bell state by using a product state notation. This is in fact a way to think of entanglement. It is the information that cannot be expressed with product states.

2.10.2 Matrix product state

Also known as tensor networks. Describes the quantum system by using a network of interconnected tensors. [25]

$$\psi = \sum_{i_1,i_2,\dots,i_N} C_{i_1,i_2,\dots,i_n} |i_1\rangle \otimes |i_2\rangle \otimes \dots \otimes |i_N\rangle \tag{2.20}$$

where C are p^N complex numbers, p is the number of degrees of freedom for each particle N e.g. $p = 2$ for qubits. And $|i\rangle$ are complex, square matrices of order B. The bond degree $B = 1$ is a product state, and B grows exponentially with system size for a generic system. B need to be small for MPS to be computational effective.

3 | Variational method

Finding the exact ground state of a quantum system by using the Schrödinger equation is only possible in a few special cases which are all systems with some constraints. Either in size or with complexity of interaction. This makes the analytical approach impossible in near all real life cases, and we need to find approximations that handles the quantum system complexity well [26]. The variational method is one such method of approximation. Aiming to find the best possible approximation of the ground state of a quantum system given by a Hamiltonian and a *trial wave function* often called *ansatz*. The ansatz (German word meaning approach) is a word used for an educated guess of what a solution might be. In the case of the variational method, this is the trial wave function selected to approximate the ground state. The educated guess depends on what type of system that is in question, and can be any mathematical approach. before the use of computers and their power, these guesses need to be more educated, since the number of free variables would need to be constrained. Today we can have the option for a much more flexible guess with more options for tuning, like the RBM. The important part is that the ansatz has to be an easier beast to tame, an thus can be calculated more efficiently than the original system. This will in our case be to use a lot fewer parameters than the 2^{σ} states of the system.

3.1 Variational principle

Given a wave function ψ and an Hamiltonian $\hat{H}$ with the ground state energy E_{gs} and its eigenfunction $|\psi_{gs}\rangle$ i.e. $\hat{H}\,|\psi_{gs}\rangle = E_{gs}\,|\psi_{gs}\rangle$. Any new trial wave function φ with a lowest energy E_0 that is *normalized*, has the same *boundary conditions* , and same *variables* as ψ, the *variational theorem* applies.

$$E_0 \geq E_{gs} \tag{3.1}$$

For any φ we choose, it's lowest energy E_0 is *always equal or higher* than the ψ ground state energy. Knowing this we can make changes to our ansatz and calculate the ground state again and comparing it to our previous ground state.

If our new ground state is lower, then we know our new ansatz is closer to the true system ground state than our old. One drawback with this method is that we can never know how close we are to the actual E_{gs} without calculating the actual ground state it self.

3.2 Proof

Proof. Let $\hat{H}$ be the known but possibly complicated Hamiltonian of the system, and φ be any ansatz. We can then suppose that φ_n and E_n are the true eigenstates and eigenvalues of $\hat{H}$ so $\hat{H}\left|\varphi_n\right\rangle = E_n\left|\varphi_n\right\rangle$. We then order the energy states in the following way $E_0 < E_1 < \cdots < E_n$ so that E_0 is the ground state energy, E_1 is the energy of first excited state, and so on. With a normalized ansatz $|\varphi\rangle$, we expand the trial wave function in terms of the eigenstates of the Hamiltonian we are looking at. We know that we can always do this, because the eigenvstates of a Hermitian matrix, i.e. the Hamiltonian, is always a basis. Any state in that space can be written as a linear combination of states and amplitudes as shown in (2.8).

$$\begin{aligned}
\langle\varphi|\hat{H}|\varphi\rangle &= \left\langle \sum_n \Psi_n\varphi_n \middle| \hat{H} \middle| \sum_m \Psi_m^*\varphi_m \right\rangle \\
&= \sum_{n,m} \Psi_n\Psi_m^* \left\langle\varphi_n|\hat{H}|\varphi_m\right\rangle
\end{aligned}$$

Now we can use the fact the $\hat{H}$ is hermitian and equation (2.15) and use $\langle\varphi| E = \langle\varphi|\hat{H}$ and we get.

$$\begin{aligned}
&= \sum_{n,m} \Psi_n\Psi_m^* \left\langle\varphi_n|E_n|\varphi_m\right\rangle \\
&= \sum_{n,m} \Psi_n\Psi_m^* E_n \left\langle\varphi_n|\varphi_m\right\rangle \\
&= \sum_n |\Psi_n|^2 E_n,
\end{aligned}$$

where $\langle\varphi_n|\varphi_m\rangle$ is the Krönecker delta δ_{nm} and picks put $n = m$ of the sum. We can now rearrange by extracting terms from the summation.

$$\langle\varphi|\hat{H}|\varphi\rangle = |\Psi_0|^2 E_0 + \sum_{n>0} |\Psi_n|^2 E_n \tag{3.2}$$

$$|\Psi_0|^2 = 1 - \sum_{n>0} |\Psi_n|^2 \tag{3.3}$$

now we combine

$$\langle\varphi|\hat{H}|\varphi\rangle = \left(1 - \sum_{n>0} |\Psi_n|^2\right) E_0 + \sum_{n>0} |\Psi_n|^2 E_n \tag{3.4}$$

$$= E_0 - \sum_{n>0} |\Psi_n|^2 E_0 + \sum_{n>0} |\Psi_n|^2 E_n \tag{3.5}$$

$$= E_0 + \sum_{n>0} |\Psi_n|^2 (E_n - E_0) \tag{3.6}$$

The second term can only be positive given that $E_n \geq E_0$. This shows that

$$\langle\varphi|\hat{H}|\varphi\rangle \geq E_0, \tag{3.7}$$

which was the result we wanted to prove. $\square$

So no matter what ansatz we try, we know we can never have a lower energy than the actual wave function.

4 | Restricted Bolztmann Machine

In this chapter we want to present the idea behind the Restricted Boltzmann Machine (RBM) and how it is mathematically expressed. We will also discuss how we can utilise the RBM to create an approximation of a quantum system, and how we use it as an ansatz with the variational method.

4.1 The Restricted Boltzmann Machine

The Boltzmann Machine (BM) is a neural network with stochastic nodes [27]. The nodes can take on one of two values, and there is a *energy* associated with each *configuration* of neurons i.e. a state. The BM is updated stochastically by using discrete time steps t. A lower energy configuration is emerging more often than a high energy configuration. The collection of observed configurations when $t \to \inf$ will converge. The occurrence of each configuration in the collection is then the probability for the configuration to be found in the BM. While the fully connected structure makes in very capable of learning highly complex structures, it also makes the training of the network practically very hard [21].

The challenge of training the BM inspired the use of a RBM. It differs from the Non-restricted Boltzmann machine by eliminating the inter-layer node connections i.e. no visible node is connected to another visible node and the same with hidden layer. This make the training process easier [28] as we will see later, while still keeping many of the BMs properties. It is a *unsupervised* model used to discover hidden structures in data. The network is usually made out of two parts: Visible layer and hidden layer. RBM belong to a class known as energy-based models. The energy-based models differ from other NN by estimating probability densities instead of values of the inputs. i.e. they find many points instead of one value. This is also called a generative model and is mathematically a function that takes a configuration as an input, and returns a probability

$$f(|i\rangle) = P, \tag{4.1}$$

where $|i\rangle$ is a binary configuration of the visible layer.

RBMs can approximate any distribution in an N-dimensional space [19]. The process of an RBM called "reconstruction" where it learns to recreate data by passing information back and fourth between the visible and the hidden layer. This generative property of the RBM is at the core of this method of approximating quantum ground states. We Want the RBM to act as an machine outputs the probability when we insert a state (configuration of the visible layer) equal to the same probability of measuring the same state in the quantum system.

Term	**Description**
Nodes	The NN building blocks that hold information.
Connections	Describe how the nodes can interact with each other
Visible layer	The set of nodes we encode and read information from
Hidden layer	The set on nodes that allows the network its flexibility
Bias	A value directly influences a specific node
Weigths	A value describing the strength of a connection
Energy	A term used in RBM theory related to the probability for a specific configuration of the visible layer to appear
Parameters	All the values that we change during training (biases and weigths)

Table 4.1: A glossary of commonly used terms in RBM theory

4.2 Structure of the RBM

There is only two layer in the RBM called visible layer and hidden layer as seen in figure 4.1. In our case the visible layer representing the spin states or qubits of a quantum system. The hidden layer provides the network with the flexibility to express the properties of entanglement or correlation between the visible nodes [17]. The possible values for both the visible and hidden layer is binary $\{0, 1\}$. We do not have a output node with the RBM which are commonly found in NNs. Instead we want to retrieve the RBM energy by looking at how the visible layer changes "over time".
The nodes and connections in the network are influenced by a set of parameters. these parameters are the collection of the node biases and connection weights. It is important to point out that the RBM is an *mathematical function.* It is not an actual network of nodes firing with activation functions. It is this mathematical description we are going to provide us with the probabilities and energies. Not actually sampling a network firing nodes.

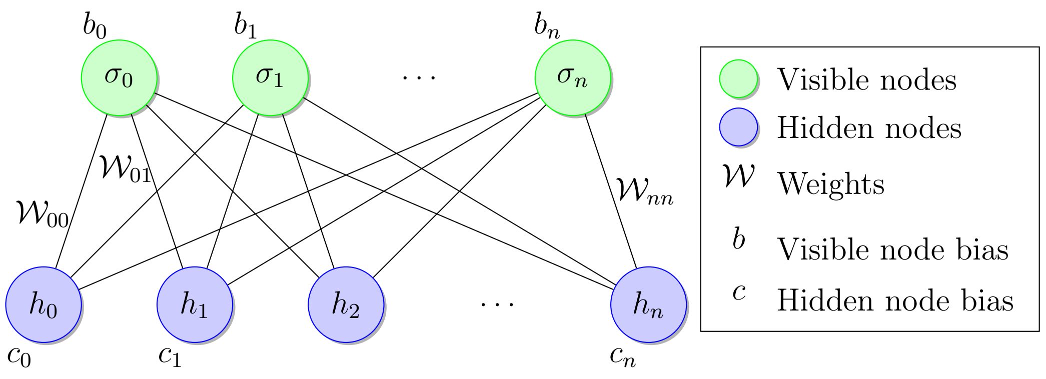


Figure 4.1: Visualization of the RBM network layout showing how the connection between the visible and hidden layer is imagined. Figure by the author.

4.3 The mathematical RBM

Here we will present the mathematical foundation that is building the RBM. Below is the equation for the probability P of finding the RBM in a state χ. A state is a configuration of the visible and hidden layer

$$P(\sigma, h) = \frac{1}{Z} e^{-E(\sigma,h)}, \tag{4.2}$$

where E is the energy for the state, and $\frac{1}{Z}$ is a normalization constant described in (4.3). This probability is a Boltzmann probability distribution, and where the network has its name from. The term energy is commonly used in literature both for RBMs and for Boltzmann distributions. It is worth noting that this energy is the RBM energy, and not the energy of the quantum system we want to learn.

$$Z = \sqrt{\sum_{\forall} P} \tag{4.3}$$

The probability E is calculated in follow way:

$$E(\sigma, h) = \sum_{i,j=1}^{N_h,N_\sigma} h_i \mathcal{W}_{ij} \sigma_j + \sum_{i=1}^{N_h} c_i h_i + \sum_{j=1}^{N\sigma} b_j \sigma_j, \tag{4.4}$$

where σ is the binary vector representing the nodes in the visible layer and h the binary vector in the hidden layer. The $\chi = \{\sigma, h\}$ is the set of all binary vectors. $\mathcal{W}$ is the weights between the visible and hidden layer. b, c are the biases for respectively the visible and hidden layer. N_σ, N_h is the number of nodes in each layer. An alternate way of writing this is by using matrix representations in stead on sums:

$$E(\sigma, h) = h^T \mathcal{W} \sigma + c^T h + b^T \sigma \tag{4.5}$$

This representation is more closely related to both notation from quantum mechanics (vectors/(matrices) and how we will implement the framework in code.

4.4 Marginalization of hidden layers

One of the benefits the RBM gives us over a regular BM are the possibility to trace away the hidden layer from our calculation. This is solely due to the lack of connection internally in the hidden layer. If we look at the formula for finding the probability of a given state in the RBM (4.2). If we want to find probability to find a specific visible layer state σ, we only need to check all possible values of the hidden layer h to get the probability.

$$P(\sigma) = \sum_{\forall h} P(\sigma, h) \tag{4.6}$$

Where $\forall h$ are all the possible values of vector h i.e. $\{0, 1\}$. This is the marginalization, that is, one of the degrees of freedom in the density is being summed over i.e. h. We can expand the expression for energy into separate terms for the biases and weights.

$$\begin{aligned} &= \frac{1}{Z} \sum_{\forall h} e^{-(h^T \mathcal{W} \sigma + c^T h + b^T \sigma)} \\ &= \frac{1}{Z} \underbrace{\sum_{\forall h} e^{-(h^T \mathcal{W} \sigma + c^T h)}}_{\text{Part A}} e^{b^T \sigma} \end{aligned} \tag{4.7}$$

A closer look at part A.

$$\frac{1}{Z} \sum_{\forall h} e^{-(h^T \mathcal{W} \sigma + c^T h)} = \frac{1}{Z} \sum_{\forall h} e^{-h^T \mathcal{W} \sigma} e^{-c^T h} \tag{4.8}$$

We know that h can only take the values $\{0, 1\}$, so we can quickly check all possible outcomes of (4.8) by inserting.

$$h_i = 0 \quad \Rightarrow \quad \begin{aligned} e^{-c_i h_i} &= 1 \\ e^{-h_i \mathcal{W}_{ij} \sigma_j} &= 1 \end{aligned} \tag{4.9}$$

$$h_i = 1 \quad \Rightarrow \quad \begin{aligned} e^{-c_i h_i} &= e^{-ci} \\ e^{-h_i \mathcal{W}_{ij} \sigma_j} &= e^{-\mathcal{W}_{ij} \sigma_j} \end{aligned} \tag{4.10}$$

This means we can rewrite (4.2) to an expression which is not dependent on h at all, only the weights between the visible and hidden layer and the biases.

$$P(\sigma) = \frac{1}{Z} \prod_{i=1}^{N_h} \left(1 + e^{-\mathcal{W}^{[i]}\sigma - c_i}\right) \cdot e^{b^T\sigma} \tag{4.11}$$

$$\text{Where } \mathcal{W}^{[i]} \text{ is the i-th row of } \mathcal{W} = \begin{bmatrix} \mathcal{W}_{11} & \mathcal{W}_{12} & \cdots & \mathcal{W}_{1N_\sigma} \\ \mathcal{W}_{21} & \mathcal{W}_{22} & \cdots & \mathcal{W}_{2N_\sigma} \\ \vdots & \vdots & \ddots & \vdots \\ \mathcal{W}_{i1} & \mathcal{W}_{i2} & \cdots & \mathcal{W}_{iN_\sigma} \\ \mathcal{W}_{N_h1} & \mathcal{W}_{N_h2} & \cdots & \mathcal{W}_{N_hN_\sigma} \end{bmatrix}$$

Now we have an expression to find the probability for a given state to appear from the RBM. We still don't know what the normalization factor $\frac{1}{Z}$ is, and calculating it will require us to know the complete distribution of the RBM. This is a problem for large systems, since the number of states that need to to be checked grows exponentially.

4.5 Complex Parameters

We can treat $\sqrt{\mathcal{P}(\sigma)}$ as an amplitude $\Psi(\sigma)$ of the state $|\psi_\theta\rangle = \sum \Psi(\sigma)|\sigma\rangle$ of N_σ qubits represented by the RBM. However, amplitudes we get are real and positive since P is real and positive. Strictly positive amplitudes are not a problem in the case that the Hamiltonian has a positive-valued ground state. However, this is not the case for the systems we are interested in, and thus we need to generalize this to complex amplitudes.

There are two ways to introduce complex amplitudes. First method is that we can use a second RBM an approach called *phase modulus RBM* [20]. This extra network will give us phases Φ, so that $\psi^\theta = \sqrt{P^\theta_{RBM1}}e^{i\Phi^\theta}$, where $\Phi^\theta = -\log\frac{P^\theta_{RBM2}}{2}$. Where P_{RBM1} and P_{RMB2} are the probabilities retrieved from the two RBMs. Both the RBMs are real valued, but in combination they can express the complex space of the quantum system.The second approach is to assume that all parameters of the RBM (biases and weights) are complex numbers. This approach is called *complex RBM.* Based on the study done by Viteritti et. al. we will go for the latter approach. It requires a fewer amount of parameters and also have better accuracy for ground state energies [20].

We now have complex amplitudes form the RBM. We still want avoid calculating the scaling factor $\frac{1}{Z}$ form equation (4.11), since it will require an exponential growth in computation time. Thus we omit the scaling factor and introduce an unnormalised but still useful as we will see, amplitude ϕ.

$$\phi(\sigma) = \prod_{i=1}^{N_h} \left(1 + e^{-\mathcal{W}^{[i]}\sigma - c_i}\right) \cdot e^{b^T\sigma} \tag{4.12}$$

5 | Local Energy

5.1 Local observables and local energies

As we've seen, the accurate energy of the RBM requires us to know the complete probability of the system to calculate the scaling factor Z 4.3. To achieve this, we would need all information about said distribution, and thus we would not need to create an ansatz in the first place. We can circumvent this paradox by estimating the energy for a local state of the RBM. Recall that a RBM encodes a quantum state with a set of parameters $|\psi_\theta\rangle$. The state can be expressed as in (2.8) of states $|\sigma\rangle$ and amplitudes Ψ^θ. Recall, $\Psi^\theta_i = \sqrt{P^\theta_i}$, and P^θ_i is the probability to find neurons of the visible layer in the state $|i\rangle$. The states can be represented as integers $i = \sigma_1 \cdot 2^0 + \sigma_1 \cdot 2^1 + \cdots + \sigma_{N_\sigma} \cdot 2^{n_\sigma - 1}$ where $\sigma_1, \sigma_2, \ldots, \sigma_N$ ($\sigma_s \in \{0, 1\}$) is a binary vector specifying the state of visible neurons like the operational basis discussed in 2.3. So we have a RBM with a certain parameter vector θ, and we want to estimate the energy E_θ of our current NQS, $|\psi_\theta\rangle$.

$$E_\theta = \frac{\langle\psi_\theta|\hat{H}|\psi_\theta\rangle}{\langle\psi_\theta|\psi_\theta\rangle} \tag{5.1}$$

$$\langle\psi_\theta| = \sum_i \Psi^{\theta*}_i \langle i| \quad \text{and} \quad |\psi_\theta\rangle = \sum_i \Psi^\theta_i |i\rangle \tag{5.2}$$

Where Ψ^θ_i is the amplitude of expansion of the states in the operational basis. This gives

$$E_\theta = \frac{\sum_i \Psi^\theta_i \langle i| \hat{H} \sum_{i'} \Psi^{\theta*}_{i'} |i'\rangle}{\sum_i \Psi^\theta_i \langle i| \sum_{i'} \Psi^{\theta*}_{i'} |i'\rangle} \tag{5.3}$$

$$= \frac{\sum_i \sum_{i'} \Psi^\theta_i \cdot \Psi^{\theta*}_{i'} \langle i| \hat{H} |i'\rangle}{\sum_i \sum_{i'} \Psi^\theta_i \cdot \Psi^{\theta*}_{i'} \langle i|i'\rangle} \tag{5.4}$$

Recall, $\langle i|i'\rangle = \delta_{ii'}$ is the Kronecker product i.e. is 1 if $i = i'$ and 0 otherwise

$$= \frac{\sum_i \sum_{i'} \Psi_i^\theta \cdot \Psi_{i'}^{\theta *} \cdot \frac{\Psi_i^\theta}{\Psi_i^\theta} \langle i| H |i'\rangle}{\sum_i |\Psi_i^\theta|^2} \tag{5.5}$$

$$= \frac{|\Psi_i^\theta|^2}{\sum_i |\Psi_i^\theta|^2} \sum_i \cdot \frac{\Psi_{i'}^\theta}{\Psi_i^\theta} \langle i|\hat{H}|i'\rangle \tag{5.6}$$

Recall the amplitude absolute valued squared is interpreted as the probability for that state to be found $|\Psi_i^\theta|^2 = P_i^\theta$

$$\sum_i P_i^\theta \cdot \left(\sum_{i'} \langle i| H |i'\rangle \cdot \frac{\Psi_{i'}^\theta}{\Psi_i^\theta} \right) \tag{5.7}$$

We then have an expression for the energy based on the probability for a state and its *local energy*,

$$E_\theta = \sum_i P_i^\theta \cdot \varepsilon_i^\theta \tag{5.8}$$

where ε_i^θ is the local energy of the state $|i\rangle$,

$$\varepsilon_i^\theta = \sum_{i'} \langle i|H|i'\rangle \frac{\Psi_{i'}^\theta}{\Psi_i^\theta} \tag{5.9}$$

The factors $\frac{1}{Z}$ terms found in cancel out from the fraction because they are a common factor, we can use a quantity we can obtain from (4.12).

$$\varepsilon_i^\theta = \sum_{i'} \langle i|H|i'\rangle \frac{\phi_{i'}^\theta}{\phi_i^\theta} \tag{5.10}$$

We can't calculate P_i^θ itself without summing over all states which is exponential. Therefore, the summation (5.8) is a problem. This is solved with a MCMC 6.1 procedure. Let's introduce $\hat{\varepsilon}_i$

$$\hat{\varepsilon}_i = \varepsilon_j, \quad \text{where } |j\rangle \sim P_{NQS}(j) \tag{5.11}$$

Meaning $\hat{\varepsilon}_i$ is the local energy of a state $|j\rangle$ which is generated by the probability distribution of the NQS. Finally, we estimate the energy from the sample states by averaging.

$$E_\theta \simeq \frac{1}{n_\lambda} \sum_{i=1} \hat{\varepsilon}_i^\theta, \tag{5.12}$$

where n_λ is the number of samples retrieved form the MCMC procedure. If the state for the lowest energy for the NQS, $|E_0\rangle$ is *localized* over a few neighbouring states from the set $\{|i\rangle\}$, then the sampling is very efficient, because the energy can be efficiently estimated bu just these few relevant states. This again meaning we need a lot fewer samples than the size of the Hilbert space. if not, then it isn't practical since we have to consider a exponential growing part of the Hamiltonian.

5.2 Local energies with the Ising model

Let's introduce the local energies of the sparse Hamiltonian in section 2.7.3. The power of MCMC sampling really shines when the Hamiltonian describing the system is focused on a few states. Instead of considering the whole Hilbert space of the system, we can use a more efficient way to calculate the energies needed. Recall the Ising Hamiltonian introduced in in equation (2.14).

$$\varepsilon(\sigma) = \sum_{\sigma} \langle \sigma | \hat{H} | \sigma' \rangle \frac{\Psi(\sigma')}{\Psi(\sigma)} = \sum_{i=1}^{n-1} \gamma_i \frac{\Psi(\mu_i)}{\Psi(\sigma)} \tag{5.13}$$

where μ_i is the same state as σ except the values on index i and $i+1$ are flipped i.e. $i = 2, \sigma = [01101] \Rightarrow \mu_2 = [01011]$. $\mu_i = |\sigma_1, \sigma_2, \ldots, \bar{\sigma}_i, \bar{\sigma}_{i+1}, \ldots, \sigma_n\rangle$ This means we can "skip" most of the elements that would be present in the matrix Hamiltonian, and save precious computational time and power.

6 | Quantum Monte Carlo

In real life many-body problems, most of the distribution contributes little or nothing to the total probability. The amount of "emptiness" grows exponentially with the increase of dimensions. Let's say you have a one dimensional distribution where only 10% is contribution to the total in a significant way. In two dimensions the same distribution will probably only $\sim 1\%$ contribute. For three dimensions $\sim$0.1% and so on. This is a property we can exploit by sampling the system. By having denser sampling in the areas that contribute more to the distribution, and less inn areas that contribute less, we can get better accuracy with much less samples. Chapter is based on [29] and [30]

6.1 Quantum Monte Carlo

Quantum Monte Carlo is the name of the group of approximation methods used to solve complex quantum systems by using *Monte Carlo* methods. As Monte Carlo once was the world capital of hazard games and gambling, the name is being used due to the stochastic nature of the methods. The target is to produce samples efficiently from an unnormalized probability distribution. We want to create a chain of samples of state from the target distribution i.e. $|\psi|^2$, so that the probability of one state to appear in the chain is the same as the probability for that state to be measured in the quantum system. There are several approaches to how to achieve this, and in this chapter we will present one that is used togehter with the variational method 3.1 in our implementation and training of the NQS.

6.2 Markov Chain Monte Carlo

The MCMC is the name of a group sampling algorithms that samples from a probability distribution. They provide a straightforward way to simulate values from any known distribution and use the same samples for other tasks. More specifically it samples over *Markov Random fields.* In short, a Markov random field is a space where the probability to move from a point to the next is only dependent on the point you are on. If we store every point we visit we will create a Markov chain. If we were to collect infinitely many points, the distribution of the points in the chain will converge to the probability distribution that were sampled. The RBM is a probabilistic undirected graphical model and thus a Markov random field [28].

6.3 Hamming step and random walk

A *Hamming distance* is in general a metric for comparing two equal-length data strings, by quantifying the element-wise deviation between them [31]. In the case of binary strings or arrays, as we are using in this project, the Hamming distance $\mathcal{H}$ between two binary string of same length, A and B, can be calculated the following way.

$$\mathcal{H} = \sum_i |A_i - B_i| \tag{6.1}$$

Where i is the index in the string or array. We have to perform a *random walk* over the set of operational states, $\{|i\rangle\}$. The walk in random walk is done by taking *steps* in the distribution landscape. A step is a move from one state to the next. In the case of the RBM. To move in our landscape, we are going to take *Hamming steps*. This is done by moving the state by a Hamming distance of 1, i.e. "flipping" the value of a randomly chosen node in the visible layer e.g. $|01101\rangle \rightarrow |01001\rangle$. When a new state is reached, we need to check if the new sample is going to be collected in our Distribution chain or not.

6.4 Metropolis-Hastings algorithm

Metropolis-Hasting is a subgroup of MCMC algorithms. It describes how we select our next entry in our state-collection while doing a random walk to ensure that the resulting samples aproximates the target ditribution. Here we will present the algorithm as it is used in this project, i.e. not the most general one.

$$P(i \rightarrow i') = min\left(1, \frac{P(i')}{P(i)}\right) \tag{6.2}$$

where P is the probability to accept the new state $|i'\rangle$ coming from state $|i\rangle$, $P(i')$ is the probability that the RBM is state $|i'\rangle$, and $P(i)$ is the RBM probability to be in state $|i\rangle$ given the same set of parameters for the RBM.

The Metropolis-Hasting Algorithm can be then summarized in the following steps:

1. Generate a random state σ^t where $t = 0$ and add it to the chain $D(t) = \sigma^t$.

2. Take a hamming step in the distribution landscape and find a new state σ^{t+1}

3. Compute the quantity $R = \frac{P(\sigma^{t+1})}{P(D(t))}$.

4. Draw a uniformly distributed random number $\eta \in [0, 1)$.

5. If $R > \eta$, accept the new states, i.e. $D(t + 1) = \sigma^{t+1}$. Otherwise, the following state in the chain stays the current one: $D(t + 1) = \sigma^t$.

6. Increase t by one and repeat form step 2. When the number of desired samples are collected, the algorithm is done.

If we let $n_\lambda \to \infty$, the collection of samples will approximate the target distribution.

6.5 Standard deviation of the MCMC and choosing number of samples

The collected samples from the MCMC algorithm can be used as an estimator $\hat{x}$.

$$\hat{x} = \frac{1}{n_\lambda} \sum_i^{n_\lambda} x_i, \tag{6.3}$$

where n_σ is the number of samples in the collection, and λ are the collection of samples. The variance of the estimator $V(\hat{x}9$ will then be

$$V(\hat{x}) = \frac{1}{n_\lambda} \sum_i^{n_\lambda} V(x_i), \tag{6.4}$$

and the standard deviation $std(\hat{x})$

$$std(\hat{x}) = \frac{1}{\sqrt{n_\lambda}} \sum_i^{n_\lambda} \sqrt{V(x_i)} \tag{6.5}$$

The standard deviation or error of the estimation is only dependent on the number of samples, and an 4 time increase in n_λ will give a halving of the MCMC error.

6.6 Warm up

One nice thing about the MCMC algorithm is that it converges form any staring point in the distribution if the number of samples is big enough. Downside is that it does require a number of samples that are not a representation of the distribution before it actually begins converging. A normal practice is to throw away some of the first samples of the sample walk i.e. not collect them for the final distribution. This is called the thermalization, burn in steps, or later preferred, warm up period [32] of the algorithm. There is no obvious way in knowing how many steps the warm up should last, and is a subject of trial and error. After some experimenting we chose set the number of warm up steps to be $10/N_sampels$, a tenth of the number we want to samples. Since the number of samples needed scales with system size, the warm up steps scales with this as well.

7 | Training the RBM

The RBM, as most neural networks require training. Training is the process of adjusting the models parameters so that the model performs as desired. The generative properties of the RBM in combination with the variational method makes defining the training in one of the three traditional umbrella terms for machine learning: Supervised, unsupervised and reinforcement

This is done by choosing an objective function. Also often called loss function it's functions as a measure of how well the model performs. A lower output of the loss function means the model is performing better. Then the parameters are adjusted in an iterative manner, so that the objective score is as low as possible. In this project we have utilized *gradient descent* to perform the training.

7.1 Gradient descent

Gradient Descent (GD) also knows as steepest descent is an optimization technique for finding minimum (or optimum) of a function $f(x)$ that is differentiable in an neighborhood around x, where x can be a many variable vector. For each time step n, we can intuitively see that a movement $f(x_n) \to f(x_{n+1})$ in the direction where the gradient is lowest will lead to $f(x_{n+1}) \leq f(x_n)$. Note that this is only true when each step is short enough so that we overshoot

$$x_{n+1} = x_n - \eta \cdot \nabla_x f(x_n) \tag{7.1}$$

$\nabla_x f(x_n)$ are the gradients of f w.r.t. x. The parameter η is often called *learning rate*. The learning rate adjust how long (or short) each step will be. It is most often used in the $0.001 - 0.1$ range, although others can be used. A too low value will lead to slow convergence toward an minimum. While a too high value will lead the algorithm to "jump" over minimum, and also not converge correctly. Changing the sign of η will lead to an steepest **ascent** method, that find maxima. GD is relatively easy to understand and implement. Still. it is a powerful tool when the grandient landscape is "easily traversable" meaning the gradient is not close to 0 in most on the space. Downsides with the gradient decent method is that it is somewhat naive, can take long to converge, and is easily stuck in local minimum. To improve performance improved variants of GD algorithm like Adam 7.2 can be used.

7.2 Adam optimization

Adam is a stochastic gradient descent algorithm to add momentum to the often called "vanilla" gradient descent (7.1) proposed by Kinga & Ba [33]. It uses the moving averages of the gradients to dynamically adjust the individual gradient step size. This leads to faster convergence and also helps with avoiding local minima.

Algorithm 1 Adaptive Movement Estimation algorithm (Adam), from [33]

```
Require: $\gamma$ : Learning rate
Require: $\beta_1, \beta_2 \in [0, 1)$ : Exponential decay rates for the moment estimates
Require: $f(\theta)$ : Stochastic objective function with parameters $\theta$
Require: $\theta_0$ : Parameter vector to be optimized
  $m_0 \leftarrow 0$ moment 1
  $v_0 \leftarrow 0$ moment 2
  $t \leftarrow 0$ time step
  while $\theta_t$ not convergent do
      $t \leftarrow (t + 1)$ Increase time step
      $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ Find the gradients
      $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ Updating moment 1 using $g_t$ and $\beta_1$
      $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ Updating moment 2 using $g_t$-squared and $\beta_2$
      $\hat{m}_t \leftarrow m_t/(1 - \beta_1^t)$ Bias corrected estimate 1
      $\hat{v}_t \leftarrow v_t/(1 - \beta_2^t)$ Bias corrected estimate 2
      $\theta_t \leftarrow \theta_{t-1} - \gamma \cdot \hat{m}_t/(\sqrt{\hat{v}_t} + \epsilon)$ Update the parameters
  end while
  return $\theta_t$
```

From literature we will use the standard parameters $\beta_1 = 0.9, \beta_2 = 0.999$ in this project.

7.3 Finite difference scheme

Finite difference is a numerical differentiation method for estimation gradients [34]. It comes in many flavors in accuracy and complexity and is a tried and true method for approximating gradients. It is based on the definition of the derivative,

$$f'(x) = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h} \tag{7.2}$$

where $f(x)$ is the function to be evaluated and x is the point where $f(x)$ is evaluated. When the difference h approaches zero, the result is the derivative of

f at point (x). Since most of real life problems cant be evaluated analytically this way, we can use FD to estimate the derivative. This approach is instead of having h reaching 0, we let h be a small enough number for the approximation to be sufficiently accurate.

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \tag{7.3}$$

There are improvements to this one-point method. The FD scheme we are using in this project is a two-point method (7.4). Meaning we average the gradient over two points instead of one. The error in this method is scaling with $\frac{1}{h^2}$ instead of $\frac{1}{h}$ as in (7.3) meaning we get a much faster convergence for the same value choice of h.

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \tag{7.4}$$

7.4 The Analytic Gradient of the RBM

We use the RBM to encodes a state $\varphi_\theta(\sigma)$ that we hope will match a unknown wave function $\psi(\sigma)$. This trial wave function φ (the RBM) is parameterized with parameter set $\theta = \{\nu_1, \nu_2, ..., \nu_m\}$, i.e. weights and biases. The states $\sigma = \sigma_1, \sigma_2, .., \sigma_n$. is all possible configurations of the visible layer of the RBM. The parameters are in general initially a set of random numbers and the likeness between the ansatz and the target wave function is most definitely poor. We have seen in section 3.1, the variational method tells us that $E(\theta) \geq E_{gs}$. Thus our goal is to find the set of parameters θ that minimizes $E(\theta)$ as much as achievable. To tune these parameters and train the RBMwe can use gradient decent over the parameter space:

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla_\theta E(\theta_t) \tag{7.5}$$

Where $\nabla_\theta E(\theta) = \{\nabla_{\theta_1} E(\theta), \nabla_{\theta_2} E(\theta), ..., \nabla_{\theta_m} E(\theta)\}$ is the vector of gradients for all the parameters. The gradients can be found by using numerical methods as discussed in section 7.3, but the mathematical nature of the RBM makes it natural to find and use the analytical gradient as well. Before we show the derivation of the expression for the RBM gradient, we will quickly discuss handling the complex valued θ in the gradients.

7.4.1 Complex parameters and real valued gradients

Since quantum systems are described with complex amplitudes, we will need to also be parameterizing the RBM with complex parameters $\theta = \{\mathcal{W}, b, c\} \in \mathbb{C}$.

If we were to operate with complex parameters θ we would need to take into account that there is no "straightsforward" derivative with respect to complex arguments.one whould have to use Writinger derivatives

$$\frac{\partial f}{\partial \theta} = \frac{1}{2}\left(\frac{\partial f}{\partial \theta^R} - i\frac{\partial f}{\partial \theta^I}\right) \tag{7.6}$$

$$\frac{\partial f}{\partial \theta^*} = \frac{1}{2}\left(\frac{\partial f}{\partial \theta^R} + i\frac{\partial f}{\partial \theta^I}\right) \tag{7.7}$$

Where $\theta = \theta^R + i\theta^I$. In the end we would have to handle everything with real parameters in implementation. We can instead splitt the parameter vector θ into its real and imaginary parts $\theta = \nu_1, \nu_2, ...,$ where $\nu_1 = Re(\theta_1), \nu_2 = Im(\theta_1)$. This will double the amount of parameters, but keep everything real for derivations. A drawback is that we need to pay extra attention during implementation. We need to recombine and separate the lists of real and imaginary parameters as complex number in other to do calculation correctly.

7.4.2 Analytical expression for the gradients of the RBM

Finding an analytical gradient of the RBM will hopefully let us evaluate the gradient faster and more precisely for each step in the gradient descent method than FD. We have an ansatz which we can write as an linear combination of all the states and their respective amplitude/probability.

$$|\psi_\theta\rangle = \sum \psi(\sigma)\,|\sigma\rangle \tag{7.8}$$

where the probability ψ is calculated with (4.11). Looking at the gradient decent of the parameters 7.5 we can rewrite the energy $E(\theta)$ as a expectation value (**??**), and we can find the gradients.

$$\nabla_\theta E(\theta) = \frac{\partial}{\partial \theta}\left(\frac{\langle\psi_\theta|\hat{H}|\psi_\theta\rangle}{\langle\psi_\theta|\psi_\theta\rangle}\right) \tag{7.9}$$

$$= \frac{\overbrace{\frac{\partial}{\partial \theta}\left(\langle\psi_\theta|\hat{H}|\psi_\theta\rangle\right)}^{\text{Part A}}\langle\psi_\theta|\psi_\theta\rangle}{\langle\psi_\theta|\psi_\theta\rangle^2} - \frac{\langle\psi_\theta|\hat{H}|\psi_\theta\rangle\overbrace{\frac{\partial}{\partial \theta}\left(\langle\psi_\theta|\psi_\theta\rangle\right)}^{\text{Part B}}}{\langle\psi_\theta|\psi_\theta\rangle^2} \tag{7.10}$$

We now look at part A.

$$\overbrace{\frac{\partial}{\partial\theta}(\langle\psi_\theta|\hat{H}|\psi_\theta\rangle)}^{\text{Part A}} = \langle\frac{\partial\psi_\theta}{\partial\theta}|\hat{H}|\psi_\theta\rangle + \langle\psi_\theta|\hat{H}|\frac{\partial\psi_\theta}{\partial\theta}\rangle \tag{7.11}$$

$$= \sum_\sigma\left[\langle\sigma|\frac{\partial\psi^*_\theta(\sigma)}{\partial\nu_j}\hat{H}|\psi_\theta\rangle + \langle\psi_\theta|\frac{\partial\psi_\theta(\sigma)}{\partial\theta}\hat{H}|\sigma\rangle\right] \tag{7.12}$$

We use the linear expansion from (7.8). Next, we look at part B

$$\overbrace{\frac{\partial}{\partial\nu_j}\langle\psi_\theta|\psi_\theta\rangle}^{\text{Part B}} = \sum_\sigma\left[\frac{\partial\psi^*_\theta(\sigma)}{\partial\nu_j}\langle\sigma|\psi_\theta\rangle + \frac{\partial\psi_\theta(\sigma)}{\partial\nu_j}\langle\psi_\theta|\sigma\rangle\right] \tag{7.13}$$

$$= \sum_\sigma\frac{\partial}{\partial\nu_j}|\psi_\theta(\sigma)|^2 \tag{7.14}$$

$$= \frac{\partial}{\partial\nu_j}\left(\sum_\sigma|\psi_\theta(\sigma)|^2\right) \tag{7.15}$$

$$\frac{\overbrace{\sum_\sigma\left[\langle\sigma|\frac{\partial\psi^*_\theta(\sigma)}{\partial\nu_j}\hat{H}|\psi_\theta\rangle + \langle\psi_\theta|\frac{\partial\psi_\theta(\sigma)}{\partial\nu_j}\hat{H}|\sigma\rangle\right]}^{\text{Part A}}}{\langle\psi_\theta|\psi_\theta\rangle} - \tag{7.16}$$

$$\frac{\overbrace{\sum\sigma\,\langle\psi_\theta|\hat{H}|\psi_\theta\rangle\,\frac{\partial}{\partial\nu_j}\left(\sum_\sigma|\psi_\theta(\sigma)|^2\right)}^{\text{Part B}}}{|\,\langle\psi_\theta|\psi_\theta\rangle\,|^2} \tag{7.17}$$

We continue with part A. To easier rewrite the expression as expectation values further on, let's introduce:

$$\frac{\partial}{\partial\nu_j}\psi_\theta(\sigma) = \omega^j_\theta(\sigma)\psi_\theta(\sigma) \tag{7.18}$$

$$\text{where,}\quad \omega^j_\theta(\sigma) = \frac{1}{\psi_\theta(\sigma)}\frac{\partial}{\partial\nu_j}\psi_\theta(\sigma) \tag{7.19}$$

$$\frac{\overbrace{\sum_{\sigma}\left[\langle\sigma|\omega_{\theta}^{j*}(\sigma)\psi_{\theta}^{*}(\sigma)\hat{H}|\psi_{\theta}\rangle+\langle\psi_{\theta}|\omega_{\theta}^{j}(\sigma)\psi_{\theta}(\sigma)\hat{H}|\sigma\rangle\right]}^{\text{Part A}}}{\langle\psi_{\theta}|\psi_{\theta}\rangle} \tag{7.20}$$

We can rewrite ω as a matrix

$$\Omega_{\theta}^{j}=\begin{bmatrix}\omega_{\theta}^{j}(00...0) & 0 & 0 & 0\\ 0 & \omega_{\theta}^{j}(00...1) & 0 & 0\\ 0 & 0 & \ddots & 0\\ 0 & 0 & 0 & \omega_{\theta}^{j}(11...1)\end{bmatrix} \tag{7.21}$$

Part A can be simplified to a very compact expression the following way.

$$\frac{\overbrace{\sum_{\sigma}\left[\langle\sigma|\Omega_{\theta}^{j*}(\sigma)\psi_{\theta}^{*}(\sigma)\hat{H}|\psi_{\theta}\rangle+\langle\psi_{\theta}|\Omega_{\theta}^{j}(\sigma)\psi_{\theta}(\sigma)\hat{H}|\sigma\rangle\right]}^{\text{Part A}}}{\langle\psi_{\theta}|\psi_{\theta}\rangle} \tag{7.22}$$

$$=\frac{\langle\psi_{\theta}|\Omega_{\theta}^{j*}\hat{H}+\hat{H}\Omega_{\theta}^{j}|\psi_{\theta}\rangle}{\langle\psi_{\theta}|\psi_{\theta}\rangle} \tag{7.23}$$

Note that the matrix $\sum_{\theta}^{j}=|\Omega_{\theta}^{j*}\hat{H}+\hat{H}\Omega_{\theta}^{j}|$ is hermitian. Therefore part A is real. It is easy to check.

$$\langle\psi_{\theta}|\Omega_{\theta}^{j*}\hat{H}+\hat{H}\Omega_{\theta}^{j}|\psi_{\theta}\rangle \tag{7.24}$$

$$\text{With a sligth abuse of notation :} \tag{7.25}$$

$$=2Re\,\langle\psi_{\theta}|\hat{H}\Omega_{\theta}^{j*}|\psi_{\theta}\rangle \tag{7.26}$$

$$=2Re\,\langle\hat{H}\Omega_{\theta}^{j*}\rangle \tag{7.27}$$

Now, again we look at part B.

$$\frac{\overbrace{\sum_{\sigma}\langle\psi_{\theta}|\hat{H}|\psi_{\theta}\rangle\frac{\partial}{\partial\nu_{j}}\left(\sum_{\sigma}|\psi_{\theta}(\sigma)|^{2}\right)}^{\text{Part B}}}{|\,\langle\psi_{\theta}|\psi_{\theta}\rangle\,|^{2}} \tag{7.28}$$

$$\underbrace{\frac{\langle\psi_{\theta}|\hat{H}|\psi_{\theta}\rangle}{\langle\psi_{\theta}|\psi_{\theta}\rangle}}_{=\langle E_{\theta}\rangle}\underbrace{\frac{\sum_{\sigma}\left[\frac{\partial\psi_{\theta}^{*}(\sigma)}{\partial\nu_{j}}\psi_{\theta}(\sigma)+\frac{\partial\psi_{\theta}^{(}\sigma)}{\partial\nu_{j}}\psi_{\theta}^{*}(\sigma)\right]}{\langle\psi_{\theta}|\psi_{\theta}\rangle}}_{\text{We derive further}} \tag{7.29}$$

$$\frac{\sum_{\sigma}\left[\frac{1}{\psi_{\theta}^{*}\sigma}\frac{\partial\psi_{\theta}^{*}(\sigma)}{\partial\nu_j}\psi_{\theta}(\sigma)\psi_{\theta}^{*}(\sigma)+\frac{1}{\psi_{\theta}\sigma}\frac{\partial\psi_{\theta}(\sigma)}{\partial\nu_j}\psi_{\theta}^{*}(\sigma)\psi_{\theta}(\sigma)\right]}{\langle\psi_{\theta}|\psi_{\theta}\rangle} \tag{7.30}$$

$$= \frac{\left[\omega_{\theta}^{j*}|\psi_{\theta}(\sigma)|^2+\omega_{\theta}^{j}|\psi_{\theta}(\sigma)\right]}{\langle\psi_{\sigma}|\psi_{\sigma}\rangle} \tag{7.31}$$

$$= \frac{|\psi_{\theta}(\sigma)|^2\left[\omega_{\theta}^{j*}(\sigma)+\omega_{\theta}^{j}(\sigma)\right]}{\langle\psi_{\sigma}|\psi_{\sigma}\rangle} \tag{7.32}$$

$$= \frac{\langle\psi_{\theta}|\Omega_{\theta}^{j*}+\Omega_{\theta}^{j}|\psi_{\theta}\rangle}{\langle\psi_{\sigma}|\psi_{\sigma}\rangle} \tag{7.33}$$

$$= \frac{\langle\psi_{\theta}|2Re(\Omega_{\theta}^{j})|\psi_{\theta}\rangle}{\langle\psi_{\sigma})|\psi_{\sigma}\rangle} \tag{7.34}$$

$$= 2Re(\langle\Omega_{\theta}^{j}\rangle) \tag{7.35}$$

We can show that the gradient of a given parameter j can be calculated the follow way:

$$g_j = \frac{\partial E_{\theta}}{\partial\theta_j} = 2Re\left[\langle\hat{H}\Omega_{\theta}^{j*}\rangle - \langle\hat{H}\rangle\langle\Omega_{\theta}^{j*}\rangle\right] \tag{7.36}$$

We need to estimate the energy E_{θ} as discussed earlier by using the local energies based on the MCMC sampling. We can find the expectation values for $\langle\Omega\rangle$ and $\langle\hat{H}\rangle = E$ by using the local operators. 5.1

$$g_j \approx \frac{1}{\lambda}\sum_i \hat{\varepsilon}(\sigma_i)\Omega_{\theta}^{j*}(\sigma_i) - \frac{1}{\lambda}\left(\sum_i \hat{\varepsilon}(\sigma_i)\right)\left(\sum_i \Omega_{\theta}^{j}(\sigma_i)\right)$$

$$g_j \approx 2Re\left[\frac{1}{\lambda}\sum_{j=1}^{\lambda}\hat{\varepsilon}(\sigma_i)\left(\Omega_{\theta}^{j*}-\overline{\Omega}_{\theta}^{j*}\right)\right], \tag{7.37}$$

where λ are the number of samples , and $\overline{\Omega}_{\theta}^{j}$ is the average...

$$\overline{\Omega}_{\theta}^{j} = \sum_{j=1}^{\lambda}\Omega_{\theta}^{j}(\sigma_i) \tag{7.38}$$

Note that everything is complex, including local energies $e_L(\sigma)$ in the general case. Once all derivative g_j are estimated, we can perform one step of *gradient descent.*

Other NQS studies using a RBM operates with basis {-1, 1} [11][20], while we utilise the operational basis. This makes the expressions for the parameter gradients different from the one presented in the previous mentioned papers. Let's introduce logarithmic derivatives:

$$\Omega_\theta^j(\sigma) = \frac{1}{\psi_\theta(\sigma)} \cdot \frac{\partial \psi_{\theta_j}(\sigma)}{\partial \theta_j} \tag{7.39}$$

Recall that $\theta = \mathcal{W}, b, c$ i.e. all weights and biases in the RBM. We then use our ansatz (4.11) as our parameterised trial wave function $\varphi_\theta(\sigma)$. We can then look at the derivative for each of visible bias, hidden bias, and weights. Let's start with the visible biases. The logaritmic derivative derivative then gives us

$$g_{b_j} = \frac{1}{\psi_\theta(\sigma)} \frac{\partial \psi_\theta(\sigma)}{\partial b_j} = -\sigma \tag{7.40}$$

Next we consider hidden biases c

$$g_{c_j} = -\frac{e^{-\mathcal{W}^{[i]}\sigma - c_j}}{1 + e^{-\mathcal{W}^{[i]}\sigma - c_j}} \tag{7.41}$$

And last the weights

$$g_{\mathcal{W}_{ij}} = -\sigma_j \frac{e^{-\mathcal{W}^{[i]}\sigma - c_j}}{1 + e^{-\mathcal{W}^{[i]}\sigma - c_j}} \tag{7.42}$$

With these three expression we can evaluate the gradients for all parameters

7.5 Tools for analyzing

To analyse the effectiveness of our implementation we are going to need some measures for comarison. This section will introduce the error measures used in the experiments with the module.

7.5.1 Relative error

The relative error is a way to measure the error of solutions for different sized problems in a comparable way. It is used for investigating the precision of a NQS of different sizes with success in previous work [11]. Let the true value of a quantity be x and the measured or inferred value x_0. Then the relative error δx is defined by

$$\delta x = \frac{\Delta x}{x} = \left| \frac{x_0 - x}{x} \right| \tag{7.43}$$

where Δx is the absolute error. The relative error is expressed as a percentage. We find the relative error ϵ_{rel} between two energies the following way.

$$\epsilon_{rel} = \left| \frac{E_{NQS}(\theta) - E_{exact}}{E_{exact}} \right| \tag{7.44}$$

Where $E_{NQS}(\theta)$ is the NQS energy at the current configuration of parameters, E_{exact} is the actual ground state energy calculated with diagonalization, a numerically precise method, but slow for large quantum systems.

7.5.2 State error

As a measure of the similarities between two states we can use *fidelity*. For pure states as we are working with, fidelity can be calculated by taking the inner product of two different states $F = \langle\psi|\varphi\rangle$. In our case this will be the RBM wave function and the ground state of the target Hamiltonian. We can find the state error ϵ_{fid} as shown

$$\epsilon_{fid} = 1 - \langle\psi|\varphi_\theta\rangle\,, \tag{7.45}$$

where ψ is the Hamiltonian ground state eigenstate, and φ_θ is the RBM. If they are completly overlapping i.e. the same state, then $\langle\psi|\varphi_\theta\rangle = 1$ so $\epsilon_{fid} = 0$. If the are orthogonal i.e. no overlap, then $\langle\psi|\varphi_\theta\rangle = 0$ and $\epsilon_{fid} = 1$. We can then say something about how close the RBM is to the actual ground state in percentage by the fidelity, expressed in a similar way than other errors we are using in this project.

Since the Ising model we use has *degeneracy*, i.e. it has two states with the same energy, we need to modify (7.45) to consider both ground states.

$$\epsilon_{fid} = 1 - \sqrt{|\langle\psi|\varphi_\theta^a\rangle|^2 + |\langle\psi|\varphi_\theta^b\rangle|^2}, \tag{7.46}$$

where φ_θ^a and φ_θ^b are the two states with the same energy. This makes the error of the Ising system behave as with the non-degenerate states we have with the random Hamiltonian.

7.5.3 Probability error

Another property we are going to investigate is the MCMC algorithms accuracy of representing a state. Here we don't have complex quantum states as in the fidelity error, but probability distribution. These can be compared by using the sum of error between each state probability. Since the probabilities we are using always are normalized to 1, we can use this quite simple error measure ϵ_{prob}

$$\epsilon_{prob} = \frac{\sum |P(\varphi) - P(\psi)|}{2}, \tag{7.47}$$

where, $P(\varphi)$ is the probability distribution for the RBM, and $P(\psi)$ is the probability distribution of the target Hamiltonian. A ϵ_{prob} of 1 will mean the distributions have no overlap, while a ϵ_{prob} of 0 will mean they are identical.

Part II

Implementation

8 From theory to code

In this chapter we will present the how the theory is implemented in code. For clarity we have put the name of attributes, methods/functions, parameters/arguments and classes will be written in `verbose text`. For the sake of clarity and avoid misunderstanding we will refer to function parameters as *arguments*. This is to let the word parameters only be used for the Restricted Boltzmann Machine (RBM) parameters. Code snippets will be presented as listings. A major part of this project is to investigate the computational aspects of using an RBM to solve for the ground state of a system described by a Hamiltonian. Since there are few readily available frameworks to realize this apart from traditional approaches, much of the project has been spent on implementing, testing and running experiments on the framework we developed during this thesis. The artifact from this process is a Python/Numpy module for creating RBMs, sampling these with an Markov Chain Monte Carlo (MCMC) scheme, training the RBM with different methods, and estimating ground states and ground state energies of some different styles of Hamiltonians. The framework is organized as a small module consisting of two file `nqs.py` and `utils.py`. It is written object oriented to be easy in use, and make possible future expansions to the module more convenient. The implementation is done in Python with the Numpy library as basis for fast calculation. Since much of the Numpy library is written in the programming language C, it is faster than many of Pythons built in operations for arithmetic and loops [35]. The complete module used to produce the results found in this thesis can be found at github repository `https://github.com/Overskott/Master-Thesis-Project.git`

9 Hamiltonian classes

There are different Hamiltonians we use in this project that have different structures and also requires different method of calculation. These are random matrix Hamiltonian, Ising matrix Hamiltonian, and the Ising tensor product Hamiltonian, as presented in section 2.7. **Note**, the tensor product Hamiltonian is named *reduced Ising Hamiltonian* in the code. They are implemented as three classes that inherit from a parent `Hamiltonian` class. This parent class again inherits from the `ndarray` class found in Numpy's libraries to effectively handle array operations which are plentiful. This is done by not directly inheriting `ndarray`, but by inheriting from
`numpy.lib.mixins.NDArrayOperatorsMixin` instead. This "magic sentence" allows for an more relaxed code since it takes care of handling all array operations i.e. addition, multiplication, transposing etc. This would need to be implemented manually for a `ndarray` inheritance to work properly [36]. Still, we need to implement some array-like methods to handle iterations 9.0.1.

```python
    def __getitem__(self, key):
        return self.values[key]

    def __setitem__(self, key, value):
        self.values[key] = value

    def __array__(self):
        return self.values
```

Listing 9.0.1: Methods to make iteration operations available for the Hamiltonian class

The parent class `Hamiltonian` is intended as an abstract class, and is not of any use if instantiated. It merely provides the practicality for our other Hamiltonian's to be recognised as separate classes and still have (from our experiments) all of Numpy's array functionality. In the code the three sub-classes are `RandomHamiltonian`, `IsingHamiltonian` and, `ReducedIsingHamiltonian`.

9.1 Random Hamiltonian

To make sure that we are working with a valid Hamiltonian as discussed in 2.7, we made an method for creating and returning a Hamiltonian as `ndarray`. The returned array is made by combining two square matrices of elements from a normal distribution with mean 0 and variance 1. One with real values and the other with complex values. This gives us H in line 12 in listing 9.1.1. By adding this matrix with its own daggered i.e. complex conjugated and transposed as seen in line 13, we have a valid Hamiltonian since it is hermitian $\hat{H} \leftarrow H + H^{\dagger}$, which is then returned.

```
 1  import numpy as np
 2  import random
 3
 4
 5  def random_hamiltonian(n: int):
 6      """
 7      Generate and returns a random hamiltonian matrix with
 ↪      dimensions n^2 x n^2.
 8
 9      :param n: The number of qubits in the system of the
 ↪      Hamiltonian matrix.
10      :return H: Hamiltonian matrix with random elements.
```

Listing 9.1.1: Method for creating a valid Hamiltonian matrix with normal distributed random elements

9.2 Ising matrix Hamiltonian

This is the first of two ways we can represent the Ising model Hamiltonian. It represents the Hamiltonian as a matrix. To create an instance on the `IsingHamiltonian` you need to pass either the size of your system size `n` or an array of scaling values γ as described in (2.13). We will now look at code listing 9.2.1. If `n` is provided, the `gamma` array is randomly generated from a normal distribution in line 42. When the array of gamma values `gamma_array` is provided, these values are used for the construction of the matrix. <more when we are sure of which Ising model to use>

```python
    """
    return np.random.normal(size=n-1, loc=sigma, scale=mu)


def random_ising_hamiltonian(n: int = None, gamma_array:
↪ np.ndarray = None):
    """
    Generate a random Ising Hamiltonian matrix of size n^2 x n^2
↪ with random gamma values. Only provide one
    of the parameters n or gamma_array.
    :param n: Number of qubits in the system.
    :param gamma_array: The gamma values to use for the Ising
↪ Hamiltonian.

    :return: The Ising Hamiltonian matrix.
    """
    if gamma_array is None:
        n = n
        gamma = np.random.normal(0, 1, n - 1)
    else:
        n = len(gamma_array) + 1
        gamma = gamma_array

    # gamma = np.zeros(n-1) - 1
    I = np.array([[1, 0], [0, 1]])
    X = np.array([[0, 1], [1, 0]])
    XX = np.kron(X, X)

    H = 0
    for i in range(n - 1):
        h = None
        if i != 0:
            h = I

        for j in range(i - 1):
            h = np.kron(h, I)

        if h is None:
            h = XX
        else:
            h = np.kron(h, XX)

        for j in range(i + 2, n):
```

Listing 9.2.1: Code for creating a Ising Hamiltonian in matrix form.

9.3 Tensor product Hamiltonian

The last and in many ways most important Hamiltonian in this project is the tensor Ising Hamiltonian 2.7.3. It is mathematically identical with the Ising Hamiltonian, but in implementation it is represented just with the gamma-values γ. We can utilise this more compact representation to do calculations faster than the matrix representation as we will discuss in the implementation of the local energies of this Hamiltonian in section. 12.3.1 This `Hamiltonian` is an array of the $n-1$ gamma values needed to calculate the local energies as shown in (5.13). The class itself just contains the optional gamma array argument `gamma`. If no `gamma` is provided, it requires the argument `n` to be provided. This will instantiate the `ReducedIsingHamiltonian` with a gamma array of normal distributed values with mean 0 and variance 1. The length of this array will be $n - 1$.

10 | Building the RBM

The core of this project is to explore the RBM ansatz ability to express properties of some quantum systems described by a selection of Hamiltonians, more precisely the ground state of Ising-like systems. We implement this a s a class to be able to efficiently create multiple instances of different configurations of networks (layer size, parameters, distribution, etc.). We need to implement some, but in fact not every part of the mathematical description in section 4.3 is needed to be a part of the `RBM`.

Table 10.1: Parts of the RBM

Description	Symbol	Attribute
Number of visible neurons	N_σ	`visible_size`
Number of hidden neurons	N_h	`hidden_size`
Vector of the visible layer	$\sigma = \sigma_1, \sigma_2, \ldots, \sigma_{N_\sigma} \in \{0, 1\}$	`N/A`
Vector of the hidden layer	$h = h_1, h_2, \ldots, h_{N_h} \in \{0, 1\}$	`N/A`
The complete RBM state	$\chi = \{\sigma, h\}$	`N/A`
The visible layer biases	b	`b_r+b_i`
The hidden layer biases	c	`c_r+c_i`
The weights	$\mathcal{W}$	`W_r+W_i`
The complete parameter vector	$\theta = \{b, c, \mathcal{W}\}$	`params`

Table 10.2: Table showing the notation of different parts of the RBM with mathematical symbols and with attribute names. Note that the weights and biases are complex and therefore consist of two parts in the implementation

The RBM implementation does not actually need to contain neither the visible nor the hidden neurons. The hidden neurons are traced away, and the visible layer is only provided when calculating probabilities for a given state. On the other hand, the implementation also contains some parts that are not normally present in a RBM. A keen eye will notice while looking at the code that the `RBM` class has the attributes `hamiltonian`, `adam`, `walker_steps` in addition to the attributes we have been discussing earlier. The addition of these attributes to the `RBM` class lets us keep the code more compact and also contain all functions for calculating energies, probabilities, etc. inside the `RBM` class. When a instance of the `RBM` class is initialized, it's parameters is generated from a normal distribution as we can see in code listings 10.0.1. As discussed in section 7.4.1, the imaginary and the real parts of the parameters are stored as separate real valued arrays. In table 10.2, we have provided an easy access "translation" between the Neural Network (NN) jargon, the mathematical symbols, and the respective attribute name in the RBM class.

```
        self.b_r = np.random.normal(0, 1/scale,
        ↪  (self.visible_size, 1))  # Visible layer bias #
        self.b_i = np.random.normal(0, 1/scale,
        ↪  (self.visible_size, 1))  # Visible layer bias #

        self.c_r = np.random.normal(0, 1/scale, (1,
        ↪  self.hidden_size))  # Hidden layer bias
        self.c_i = np.random.normal(0, 1/scale, (1,
        ↪  self.hidden_size))  # Hidden layer bias

        self.W_r = np.random.normal(0, 1/scale,
        ↪  (self.visible_size, self.hidden_size))  # s - h
        ↪  weights
        self.W_i = np.random.normal(0, 1/scale,
        ↪  (self.visible_size, self.hidden_size))  # s - h
        ↪  weights

        self.params = [self.b_r, self.b_i, self.c_r, self.c_i,
        ↪  self.W_r, self.W_i]
```

Listing 10.0.1: The implementations of complex parameters as attributes in the RBM class

All the imaginary and real biases and weights are then stores in the `params` attribute. Note that the `params` is a list containing the different Numpy `ndarray` as elements. The arrays keep this dimensions so `params` is not a single vector array with all parameters, but have a more complex structure.

11 | Collecting probabilities from the RBM

We want to be able to test the RBM with both training on a target distribution which is exact and a distribution that is estimated. The exact distribution requires us to sample over all states σ and is an inefficient approach when the system grows since the number of states increases exponentially 2^n. This will not show us any speed up compared to diagonalization, but it is practical to benchmark using this as well. The estimated distribution is where the RBM shows its strengths. The MCMC sampling will hopefully give us a speed up with the cost of loss of accuracy.

11.1 Probabilities with the estimated distribution

As we've seen, the scaling factor $\frac{1}{z}$ require us to normalize over all states. In huge Hilbert spaces this is a problem. We therefore want to calculate the *unnormalized* amplitude for any state. We now will present method `unnormalized_amplitude` 11.1.1 that realizes equation (4.12) and is without the computational heavy scaling factor Z.

```python
137 def unnormalized_amplitude(self, state):
138     Wstate = np.matmul(state, self.W_r) + 1j * np.matmul(state,
     ↪   self.W_i)
139     exponent = Wstate + self.c_r + 1j * self.c_i
140     A = np.exp(-exponent)
141     A = np.prod(1 + A, axis=1, keepdims=True)
142     A = A * np.exp(-np.matmul(state, self.b_r) - 1j *
     ↪   np.matmul(state, self.b_i))
143     return A
```

Listing 11.1.1: Methods to make iteration operations available for the Hamiltonian class

The method takes a `ndarray` as argument, and returns the $\mathcal{P}$ for that state. In line 138 we calculate the $\mathcal{W}^{[i]}$ vector, i.e. extracting the i-th row of $\mathcal{W}$, by using Numpys matrix multiplication function `matmul()`. Note that we need to make the new matrix complex by creating two separate matrices for the real and imaginary part, `W_r` and `W_i`, and add them together. In line 139-40 we create the exponential expression. In line 141 we utilise Numpy's `prod` function to find the column-wise product. This "probability" by itself is not giving us any information but as will be shown later has an important use. The "probability" is then simply calculated by taking the absolute value squared of the unnormalized amplitude 11.1.2.

```python
def unnormalized_probability(self, state):
    return np.abs(self.unnormalized_amplitude(state))**2
```

Listing 11.1.2: Merhod for returning the unnormalised probability

11.2 Probabilities with the exact distribution

To calculate the exact distribution of the RBM we need to calculate the probability of *every single possible state* the RBM can take i.e. the entire operational basis from 2.1 into calculation. This is done by creating a list of `numpy.array` by converting all integers from 0 to 2^{N_σ} to binary numbers represented as an array with the method `get_all_states()` 11.2.1.

```python
def get_all_states(self):
    """
    Generates all possible states of the RBM and returns them as a numpy array.
    :return: Distribution of all possible states as numpy array.
    """
    all_states_list = []

    for i in range(2 ** self.visible_size):
        state = utils.numberToBase(i, 2, self.visible_size)
        all_states_list.append(state)

    return np.array(all_states_list)
```

Listing 11.2.1: Methods to make iteration operations available for the Hamiltonian class

When this exact distribution is created, we can use the returned values for the `unnormalized_amplitude` method presented in 11.1.1 to calculate the normalization factor Z as shown in 11.2.2.

```python
def normalized_amplitude(self, state):
    """

    :param state:
    :return:
    """
    # Normalized amplitude_old
    Z =
    ↪   np.sqrt(np.sum(np.abs(self.unnormalized_amplitude(self.all_states))
    ↪   ** 2))
    return self.unnormalized_amplitude(state) / Z
```

Listing 11.2.2: Tthe method for calculating the normalized amplitude

The actual probability for this state to appear in the RBM is then simply the absolute value squared of the amplitude as shown in listings 11.2.3.

```python
def probability(self, state: np.ndarray) -> float:
    """
    Calculates and returns the probability of finding the RBM in
↪   the given state
    """
    return np.abs(self.normalized_amplitude(state)) ** 2
```

Listing 11.2.3: Methods to make iteration operations available for the Hamiltonian class

For future reference we will show the `wave_function()` method that returns the normalized amplitude for every state in the RBM distribution in listings 11.2.4.

```python
def wave_function(self):
    """
    Calculates the wave function of the RBM by sampling over all
↪   states.
    :return: the wave function of the RBM.
    """
    return self.normalized_amplitude(self.all_states)
```

Listing 11.2.4: Method for returning the RBM wave function i.e. all the amplitudes for all the states

12 Local energies

12.1 Matrix Hamiltonians

One of the benefits of using the matrix Hamiltonian representation in implementation, is the natural interaction with Numpy. We can benefit from the fast calculations by using matrix multiplication instead of the slower approach of utilising loops to calculate sums. In listing 12.1.1 we present how the important local energy is calculated when the Hamiltonian is defined as a matrix as presented in 5.1. The method `local_energy` takes an array of states, `state`, typically from a MCMC distribution, but the full state space of the system is an option as well (although inefficient). The for loop iterates over all the $\mathcal{W}[i]$ i.e. the columns of the Hamiltonian.

```python
def local_energy(self, state):
    batch_size = state.shape[0]
    E = np.zeros((batch_size, 1), dtype=np.complex128)
    a1 = self.unnormalized_amplitude(state)

    powers = np.array([2 ** i for i in
     ↪  reversed(range(self.visible_size))]).reshape(1, -1)
    state_indices = np.sum(state * powers, axis=1)
    for i in range(2 ** self.visible_size):
        state_prime = np.array(utils.numberToBase(i, 2,
         ↪  self.visible_size)).reshape(1, -1)
        a2 = self.unnormalized_amplitude(state_prime)

        h_slice = (self.hamiltonian[state_indices,
         ↪  i]).reshape(-1, 1)
        E += (h_slice / a1) * a2

    return E
```

Listing 12.1.1: The method for calculating local energies with a matrix Hamiltonian

It then returns the energy `E` as an array of all the energies associated with each of the states in the argument `state`. An important drawback with this method can be seen in line 155 in listing 12.1.1. We need to iterate over all the 2^{N_σ} states to achieve the local energy. This is the big weakness of calculating local energies for generic Hamiltonians. Even though we use a relative low number of local states, will each evaluation be exponentially expensive because of this.

12.2 Tensor Product Hamiltonian

12.3 Study objective

As discussed in 2.7.3, the implementation of the tensor product Hamiltonian in `IsingHamiltonianReduced` class, allows us to abuse the structure of the Ising model to make more efficient calculations of the local energies and circumvent the exponential growth of the generic Hamiltonian. This thus require another implementation than the methods constructed for solving matrix Hamiltonians that implements the knowledge of said structure. We want to realize equation (5.13) to find the local energies. The code for this can be found in listings 12.3.1. The method `ising_local_energy()` takes an array of states `states` as input argument. The subroutine `_create_mu` generates the

```python
def ising_local_energy(self, states: np.ndarray):
    gamma = self.hamiltonian
    p_i = self.unnormalized_amplitude(states)

    def _create_mu(state, index):
        mu = state.copy()
        mu[:, index] = 1 - state[:, index]

        if index == self.visible_size-1:
            pass
        else:
            mu[:, index+1] = 1 - state[:, index+1]
        return mu

    def _local_index_energy(gamma_values, index):
        mu_i = _create_mu(states, index)

        p_j = self.unnormalized_amplitude(mu_i)

        return gamma_values * p_j / p_i

    local_energy = sum([_local_index_energy(g, j) for (j, g) in
    ↪  enumerate(gamma)])

    return np.asarray(local_energy)
```

Listing 12.3.1: The method for calculating local energies with a tensor Hamiltonian

13 Energy estimation

13.1 Exact energy

To extract the exact energy (and thus be able to get the exact probability (4.2)), we need to brute force calculate amplitudes of all states with `wave_function()` from 11.2.4 as shown in listings 13.1.1 to get the expectation value $\langle \hat{H} \rangle$. The benefit is that the implementation can then be done by calculation the expectation value of the energy with expression (2.16).

```python
def exact_energy(self):
    wave_function = self.wave_function()
    E = wave_function.conj().T @ self.hamiltonian @ wave_function
    return E.real
```

Listing 13.1.1: Calculating the exact energy for the RBM

Even though this approach offers no computational improvements of doing this compared to e.g. diagonalization, it is important to access for comparison and error estimation.

13.2 Estimated energy

Contrary to the previous section, here he estimation of the energy is done by estimating the probability distribution by using a MCMC algorithm 6.1. We utilize this representation of the underlying distribution as the `state` argument for the local energy calculations as presented in 12. This estimation will have an error, as discussed in section 6.5.

```python
def estimate_energy(self):
    walker = Walker(self.visible_size, self.walker_steps)
    sampled_states = walker(self.probability, self.walker_steps)

    if type(self.hamiltonian) is IsingHamiltonianReduced:
        return
         ↪  np.mean(self.ising_local_energy(sampled_states)).real
    else:
        return np.mean(self.local_energy(sampled_states)).real
```

Listing 13.2.1: Method for estimating the RBM energy with the use of MCMC sampling

The `estimate_energy` method in listing 13.2.1 initialises a `Walker` which is used to estimate the distribution. It hen checks the type of `Hamiltonian` class the `RBM` is initialized with, and chooses the correct solver for local energy.

14 Optimization

14.1 Analytical expression for the gradients

In this section we want to explain how the expression to find the analytical gradients using MCMC sampling (7.37) is implemented. If we dissect the expression and look at the gradient matrix Ω defined in (7.21). It is a matrix with all the gradient for each state on the diagonal. To create it we will need to calculate the gradients for visible layer biases, hidden layer biases and weights separately.

```python
242 def b_grad(self, state):
243     return -state
244 
245 def c_grad(self, state):
246     exponent = np.matmul(state, self.W_r) + 1j * np.matmul(state,
      ↪   self.W_i)
247     exponent += self.c_r + 1j * self.c_i
248     A = -np.exp(-exponent) / (1 + np.exp(-exponent))
249     return A
250 
251 def W_grad(self, state):
252     batch_size = state.shape[0]
253     A = self.c_grad(state)
254     # batch-wise outer product between c_grad and state
255     A = np.einsum('ij,ik->ijk', state, A).reshape(batch_size, -1)
256     return A
```

Listing 14.1.1: Methods for finding gradients for b, c, and $\mathcal{W}$

The code in listings 14.1.1 is a quite straightforward implementation of the equations (7.40), (7.41), and (7.42) with the use of Numpys matrix multiplication. There is some not-so-clear code on Line 255. There the `np.einsum('ij,ik->ijk', state, A)` is the Einstein summation of the two matrices `A` and `state`. Each of being two-dimensional but with differ in the size of the dimensions. This piece of code forces Numpy to correctly construct the $\mathcal{W}$-gradient matrix.

When the gradients are calculated, we combine them go get our Ω. this is done in two ways. The `omega` method in listing 14.1.2 creates Ω as an diagonal matrix that can be used with matrix operations in `exact_distribution gradients` 14.1.5.

```python
def omega(self, states):
    omega_list = []

    b_grad = self.b_grad(states).T
    c_grad = self.c_grad(states).T
    W_grad = self.W_grad(states).T

    A = self._diag(b_grad)
    omega_list.extend([A, 1j * A])

    A = self._diag(c_grad)
    omega_list.extend([A, 1j * A])

    A = self._diag(W_grad)
    omega_list.extend([A, 1j * A])

    return omega_list
```

Listing 14.1.2: Omega

The `omega_estimate` method 14.1.3 return the Ω as a list of gradients instead of a diagonal matrix. The `omega` returned by this method is used in the `Estimated distribution` method to find the gradients.

```python
def omega_estimate(self, states):
    omega_list = []

    b_grad = self.b_grad(states)
    c_grad = self.c_grad(states)
    W_grad = self.W_grad(states)

    omega_list.extend([b_grad, 1j * b_grad])
    omega_list.extend([c_grad, 1j * c_grad])
    omega_list.extend([W_grad, 1j * W_grad])

    return omega_list
```

Listing 14.1.3: Omega estimated values

14.1.1 Estimated distribution

```python
def estimate_distribution_grad(self):
    walker = Walker(self.visible_size, self.walker_steps)
    states = walker(self.unnormalized_probability,
     ↪  self.walker_steps)
    omega_list = self.omega_estimate(states)

    if type(self.hamiltonian) is IsingHamiltonianReduced:
        local_energies = self.ising_local_energy(states)
    else:
        local_energies = self.local_energy(states)

    grad_list = []

    for omega in omega_list:
        omega_bar = np.mean(omega, axis=0)
        grad = np.mean(np.conj(local_energies)*(omega -
         ↪  omega_bar), axis=0).real * 2
        grad_list.append(grad)

    return grad_list
```

Listing 14.1.4: Estimation of gradients with estimated distribution

14.1.2 Gradients of the exact distribution

Since we have derived the expression for the analytical gradient (7.36), we can calculate it directly when we utilize the complete distribution of the system. We can access the `omega` value Ω from the RBM as seen in line 295 in listing 14.1.5

```python
def exact_distribution_grad(self):
    grad_list = []
    H = self.hamiltonian
    omega = self.omega(self.all_states)
    wf = self.wave_function()

    # loop over b, c and W
    for i, O in enumerate(omega):
        EO = wf.conj().T @ H @ O @ wf
        E = wf.conj().T @ H @ wf
        O = wf.conj().T @ O @ wf
        grad = 2 * (EO - E * O)
        # reshape according to b, c or W
        if i == 0 or i == 1:
            grad = grad.reshape(-1, 1)
        elif i == 2 or i == 3:
            grad = grad.reshape(1, -1)
        else:
            grad = grad.reshape(self.visible_size,
             ↪  self.hidden_size)

        grad_list.append(grad.real)
    return grad_list
```

Listing 14.1.5: Estimation of gradients with exact distribution

14.2 Finite Difference

The finite difference method is implemented by nested for-loops. Looking at listing 14.2.1 we see that it first checks if it is going to use an exact or estimated probability distribution in line 260-265. If `walker_steps` for the instance of `RBM` is set to 0, the FD will utilise the exact distribution and initialise a small `h` value on line 265. for the exact distribution, the value of `h` is not so crucial since the error in the probability distribution is 0. If the `RBM` instance is initialised with a number of `walker_steps`>0, the estimated distribution is used, and the `h` value is set relative to the number of steps as discussed in section 6.5. This makes the code more reusable for several function we want to find the gradients to i.e. the exact energy calculation or the estimated energy calculation 13.1.

```
258 def finite_grad(self):
259 
260     if self.walker_steps == 0:
261         func = self.exact_energy
262         h = 10e-4
263     else:
264         func = self.estimate_energy
265         h = 3 / np.sqrt(self.walker_steps)
266 
267     grad_list = []
268     for param in self.params:
269         grad_array = np.zeros(param.shape)
270         for i in range(param.shape[0]):
271             for j in range(param.shape[1]):
272                 param[i, j] += h
273                 E1 = func()
274                 param[i, j] -= 2 * h
275                 E2 = func()
276                 param[i, j] += h
277                 grad = (E1 - E2) / (2 * h)
278                 grad_array[i, j] = grad
279 
280         grad_list.append(grad_array)
281 
282     return grad_list
```

Listing 14.2.1: The implementation of the FD method

We now look at the for-loop at line 268. Recall that the `params` attribute is a list of `ndarray` of different dimensions i.e. weights and biases of the RBM. The first loop iterates over each of the `ndarray` and creates a new, empty `ndarray` of same dimensions, `grad_array` on line 269. The next loop in line 270 runs through every row of the `grad_array`

14.3 Adam optimiser class

Here we present the implementation of the Adam optimizer introduced in section 7.2 The Adam optimizer algorithm keeps the running average of the gradient, and stores information about all previous steps. We also want to use Adan on both Finite Difference (FD) and Analytical Gradient (AG). This motivates the Adam optimiser to be implemented as an separate class where the steps are stored and can take any gradient as input. The `step()` method takes a list of gradients received either from a finite difference gradient or an analytical gradient. As seen in listing 14.3.1 at line 11 and 12 the time step `t` is increased for each call of `Adam.step()`. The time step is used in line 20 and 21 to average the movements before returning the optimized list of gradients in line 24.

```
1  class Adam:
2      def __init__(self, beta1=0.9, beta2=0.999, eps=1e-8):
3          self.beta1 = beta1
4          self.beta2 = beta2
5          self.eps = eps
6          self.t = 0
7          self.m = None
8          self.v = None
9
10     def step(self, grad_list):
11         self.t += 1
12         if self.t == 1:
13             self.m = [np.zeros_like(grad) for grad in grad_list]
14             self.v = [np.zeros_like(grad) for grad in grad_list]
15
16         mod_grad_list = []
17         for i, grad in enumerate(grad_list):
18             self.m[i] = self.beta1 * self.m[i] + (1 - self.beta1)
                ↪  * grad
19             self.v[i] = self.beta2 * self.v[i] + (1 - self.beta2)
                ↪  * grad ** 2
20             m_hat = self.m[i] / (1 - self.beta1 ** self.t)
21             v_hat = self.v[i] / (1 - self.beta2 ** self.t)
22             mod_grad_list.append(m_hat / (np.sqrt(v_hat) +
                ↪  self.eps))
23
24         return mod_grad_list
```

Listing 14.3.1: The code realizing the Adam optimizer 1

15 | MCMC

Here we will explain the implementation of the Metropolis-hasting algorithm from section 6.4. We created a `Walker` class to handle the collection and storage of the MCMC algorithm. This lets us keep all relevant attributes together and not clutter our `NQS` class. As seen in listing 15.0.1, the `Walker` requires two arguments, the visible layer size `visible_size`, and the number of samples to be collected or steps in the random walk, `steps`. The other attributes are generated on initialization. `current_state`, and `next_state` represents $|i\rangle$ and $|i'\rangle$ respectively, from section 6.4. The `walk_result` list is where the distribution of states collected by the random walk is stored.

```python
class Walker(object):

    def __init__(self,
                 visible_size: int,
                 steps: int,
                 ):

        self.steps = steps
        self.burn_in = self.steps // 10
        self.current_state = np.random.randint(0, 2,
         ↪  visible_size)
        self.next_state = copy.deepcopy(self.current_state)
        self.walk_results = []
```

Listing 15.0.1: The init method for the Walker class

```python
    def __call__(self, function, num_steps):

        self.estimate_distribution(function)
        return self.get_history()

    def get_history(self):
        return np.asarray(self.walk_results)

    def clear_history(self):
        self.walk_results = []
```

Listing 15.0.2: The init method for the Walker class

The method for creating and returning a MCMC distribution is `__call__` method of the `Walker` class. It takes `function` and `num_steps` to be passed on to it's sub-routines. It executes and accesses the methods and attributes for populating and returning the `walk_results` list. Details can been seen in listing 15.0.2.

```python
    def estimate_distribution(self, function, burn_in=True) ->
    ↪   None:
        self.clear_history()

        if burn_in:
            self.burn_in_walk(function)

        self.random_walk(function)
```

Listing 15.0.3: The init method for the Walker class

The `estimate_distribution` 15.0.3 method called by `__call__` clears the `walk_result` list and adds the optional argument `burn_in` for executing the algorithm with or without the warm up period. In this project we always use the warm up period.

```python
    def random_walk(self, function):

        for i in range(self.steps):
            self.next_state =
             ↪  utils.hamming_steps(self.current_state)
            self.walk_results.append(self.current_state)

            if self.acceptance_criterion(function):
                self.current_state =
                 ↪  copy.deepcopy(self.next_state)

            else:
                self.next_state =
                 ↪  copy.deepcopy(self.current_state)
```

Listing 15.0.4: The init method for the Walker class

The `walk_result` list is populated in the `random_walk` method 15.0.4. It iterates for the number of `steps` given in the initialization of the `Walker`, and accepts either `current_state` or `next_state`. This is done in the `acceptance_criterion`15.0.5 which realises the Metropolis-Hastings algorithm (6.2) .

```python
54      def acceptance_criterion(self, function) -> bool:
55          u = np.random.uniform(0, 1)
56          new_score = function(self.next_state)
57          old_score = function(self.current_state)
58
59          score = new_score / old_score > u
60
61          return score
```

Listing 15.0.5: The method realizing the Metropolis-Hastings algorithm.

It takes an argument `function` which is the function that calculates the probability for a given state of the RBM. In this module the functions will be either `probability`, or `unnormalized_probability`. First we draw a random number between 0 and 1 from an uniform distribution `u`. Then we calculate the probability for each state on line 56-57. Lastly we see if the ratio `new_score` and `old_score` is bigger than `u`, if it is we return boolean value `True`.

16 Tools for measurements

Here is an brief presentation of the tools we have utilized for error and time measurements in the experiments presented in 17.

16.1 Timing

To collect the processing time of different functions, we created the decorator method `timing`. It takes a function as an argument.

```python
def timing(f):
    """
    Decorator for timing functions based on the following
↪   example:
    https://stackoverflow.com/questions/1622943/time
↪   it-versus-timing-decorator.

    Also adds a run_time attribute to the function decorated.
↪   run_time can be
    accessed as f.run_time.

    :param f: The function to time
    :return:
    """

    from functools import wraps
    from time import time

    @wraps(f)
    def wrap(*args, **kw):
        ts = time()
        result = f(*args, **kw)
        te = time()
        wrap.run_time = te - ts # Add the run_time attribute to
        ↪  the function decorated.
        print(f"func:{f.__name__} args:[{args}, {kw}] took:
        ↪  {te-ts} sec")
        return result
    return wrap
```

Listing 16.1.1: Decorator for timing functions

It calculates the time the function `f` takes to run in milliseconds. When the process finished it prints to console the time taken together with the process id and its arguments. It also creates an attribute `run_time` and attaches it to the function `f`. This attribute can then later be accessed by calling `<f>.run_time`, where `<f>` is the name of the function passed as argument.

16.2 Error measures

This section present the implementation of the error measures from section 7.5.

16.2.1 Relative error

```python
def relative_error(true_value, approx_value):
    """
    Calculate the relative error between the true and approximate
↪   value.

    :param true_value: The true value
    :param approx_value: The approximate value

    :return: The relative error
    """
    return np.abs((true_value - approx_value) / true_value)
```

Listing 16.2.1: Implementation of relative error measure presented in 7.5.3

16.2.2 State error

```python
def prob_error(true_value, approx_value):
    """
    Calculate the relative error between the true and approximate
↪   value.

    :param true_value: The true value
    :param approx_value: The approximate value

    :return: The relative error
    """
    return np.sum(abs(true_value - approx_value))/2
```

Listing 16.2.2: Implementation of state error measure presented in 7.5.3

16.2.3 Probability error

```python
def state_error(true_state, approx_state):
    """
    Calculate the 1-fidelity error between the true and
    approximate state.

    :param true_state: The true state
    :param approx_state: The approximate state

    :return: The state error
    """
    return 1 - (np.abs(true_state.T.conj() @ approx_state))
```

Listing 16.2.3: Implementation of probability error measure presented in 7.5.3

Part III

Results and Discussion

17 | Results

As described in the project objective, investigating how the implementation performs and how its different components affect the accuracy is the main part on this thesis. In this chapter we will present the experiments done with the Python module, the arguments and their results along with some commentary and discussion. The main arguments for each experiment will be presented in the figure caption. In the end of each section of this chapter we will have a more in depth discussion of the findings. First, we look at the two gradient methods presented in section 7.3 and 7.4. Second, we will investigate the Markov Chain Monte Carlo (MCMC) sampling algorithm from section 6.4. Then, the impact of the visible and hidden layer of the Restricted Boltzmann Machine (RBM) 4.1 is tested. Lastly the implementation's accuracy to find ground states and ground state energies will be presented. The overall goal is to find evidence supporting that the RBM has the flexibility to estimate larger quantum systems, that it is a potential faster approach than brute force solving i.e. diagonalization for bigger systems, and find patterns that can motivate the argument setting for optimal performance and results. All experiments in this thesis are performed using an Windows 10 desktop computer equipped with a AMD Ryzen 7 3700X and 16 GB RAM.

17.1 Comparison of the gradient methods

Here, we want to investigate our Analytical Gradient (AG) (7.37). It is interesting to see if it benefits us in terms of timing, accuracy and tuning of the arguments. To do this we compare it with the Finite Difference (FD) scheme (7.4). Since we know FD with the right tuning of the h parameter, will achieve good results it serves as a excellent benchmark to see if the Analytical Gradient (AG) performs as good or hopefully better. This section presents some findings when training the RBM with FD and AG. We will have a look at how the two gradient schemes compare in accuracy, the impact of the MCMC sampling, and the time differences.

17.1.1 Accuracy with exact distribution

First we want to see the schemes' performance on an exact distribution i.e. calculating the probability for all states of the RBM as described in section 11.2. By using the exact distribution we will not introduce any error generated by the MCMC estimation. The Hamiltonian used is the generic random Hamiltonian 9.1.1.

In figure 17.1 we see the effect of the FD parameter h as presented in (7.4). We start by setting h to a big value, 1, and then we plot the results of training. We decrease h several times and see how the accuracy of the energy converges to the analytical gradient (purple dashed curve). As we can see, the finite difference is very close when $h = 0.1$ (green curve) for this system configuration. For $h = 0.01$ (red curve) there is no visual difference in this plot.

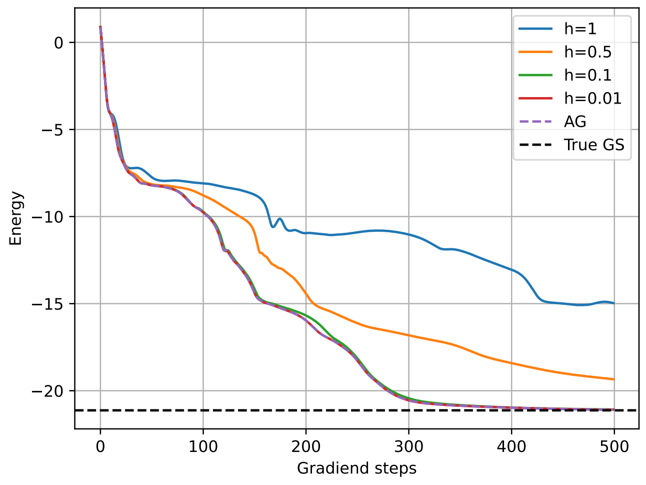


Figure 17.1: Energy during training with gradient decent done by both finite difference and analytical against the exact ground state energy. The FD scheme is run with a decreasing size of h value. Visible nodes: 5, hidden nodes: 10, gradient decent steps: 500, Hamiltonian: random.

17.1.2 Accuracy with estimated distribution

Next we want to compare the two gradients while sampling from the RBM with the MCMC algorithm. In contrast to experiment 17.1.1, the size of parameter h in FD is dependent on the accuracy of the sampled distribution. In MCMC, the error of the estimated quantities goes as $1/\sqrt{n_\lambda}$ (6.5). If the true value of the energy is E_{gs}, the sampled value is likely somewhere in the interval $[(1-1/\sqrt{n_\lambda})E_{gs}, (1+1/\sqrt{n_\lambda})E_{gs}]$. When using FD, we calculate the difference between energies very close to each other in parameter space, to find the slope. The smaller h is, the smaller the difference between E_{gs}'s associated with sampling. The estimated gradient could be influenced mainly by noise if the h is too small i.e. smaller than the interval $[(1-1/\sqrt{n_\lambda})E_{gs}, (1+1/\sqrt{n_\lambda})E_{gs}]$. Another way to look at this is that the number of samples are not high enough. The exact point where this crossover happens is hard to define, but we have achieved good results with the use of $h = 1/\sqrt{n_\lambda}$ as a default size for h in our experiments.

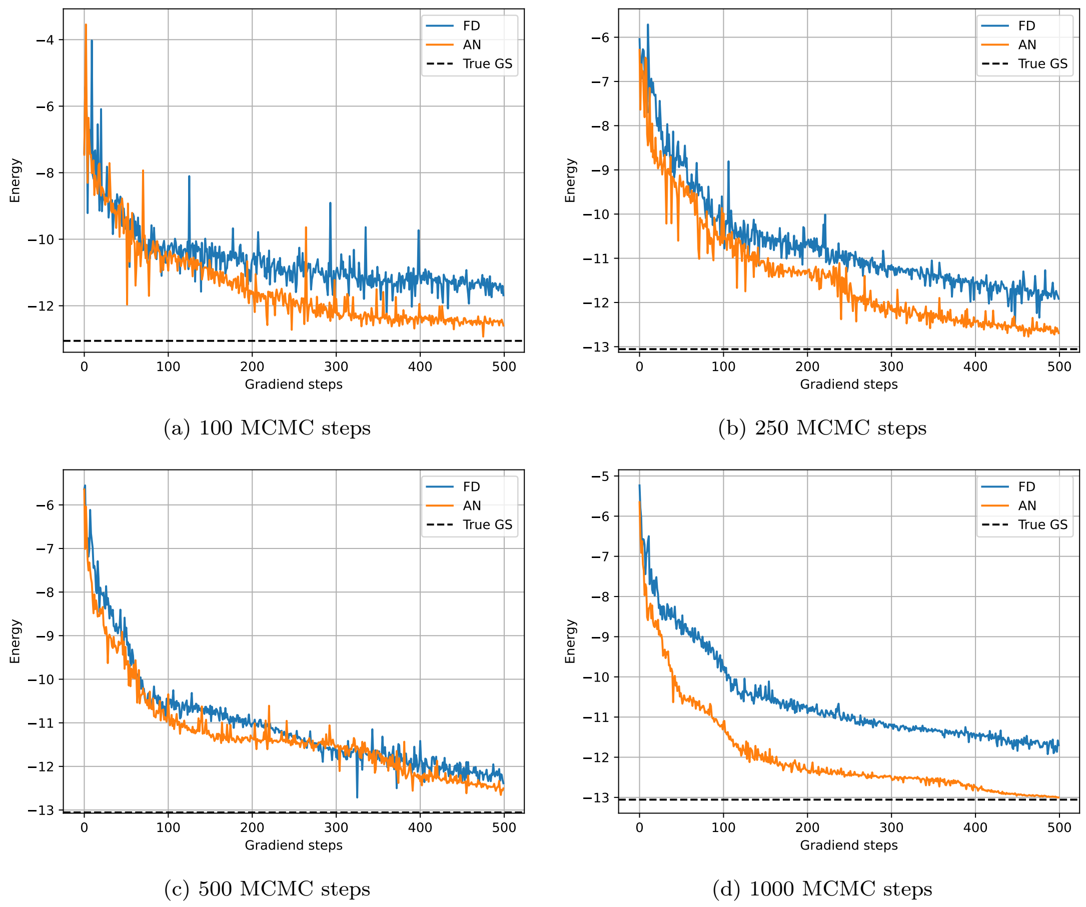
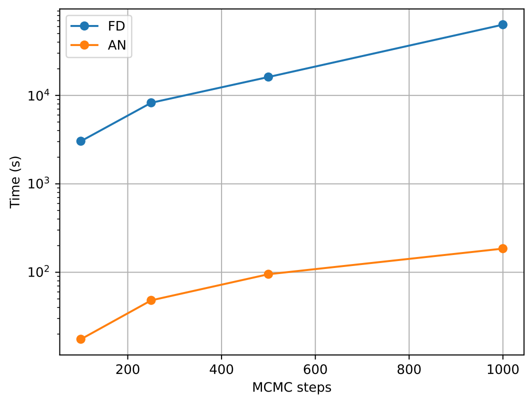


Figure 17.2: Graphs showing the energy during training with finite difference (FD) and analytic gradient (AN) of an RBM with 4 visible nodes, 8 hidden nodes, learning rate 0.01, and a random Hamiltonian as the target ground state energy. The number of MCMC steps are increased: Figure 17.2a has the lowest, and 17.2d has the most.

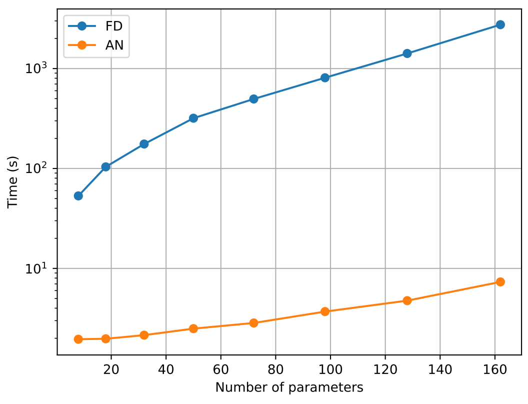
We can see that both the gradient methods struggle to find the ground state energy for lower numbers of steps collected by the MCMC algorithm in 17.2a, 17.2b, and 17.2c. In figure 17.2d the AG gets desirably close to the ground state. When the number of samples are low, we can also see the fluctuations in that the trajectory are much larger for both gradient methods . With a 1000 collected samples in each step, the curve are much smoother, especially for the AG.

17.1.3 Time comparison

Last in this section of experiments, we wanted to see how the two gradient schemes compared in timing. It is the same type of setup as in 17.1.2, with the same increasing numbers of MCMC steps, where each training is timed.



(a) Training time with increasing MCMC steps



(b) Training time with increasing parameteres

Figure 17.3: Timing of the two gradient approaches. 17.3a is run with an fixed set of parameters (visible node = 5, hidden nodes = 5), and increasing the number of collected MCMC steps. 17.3b is run with . both experiments is run with a random Hamiltonian

In figure 17.3a we can see a difference between the two methods on the scale of 10^3 when we increase the sample size. The same happens when we increase the sizes of the visible layer and hidden layer as shown in figure 17.3b. The FD took 1000 times longer than Analytical Gradient to train the RBM. Still, it seems that they are somewhat proportional.

17.1.4 Discussion

After running several experiments we can confidently say that the AG is beneficial over FD for training the RBM. We have confirmed that it indeed finds the true gradient from the experiment in section 17.1.1. By construction, we know FD approximates the gradient of the energy when using a small enough h-value. Since FD converges toward the AG when we decrease h, we know that the AG finds the correct gradient. The experiment in section 17.1.2 shows us that AG achieves better results with fewer samples from the MCMC algorithm. In figure

17.2d we see that the analytical gradient is converging much better than the FD using the same amount of samples. This increase in performance can be explained by a weakness in the FD method as explained in 17.1.2. When calculating the difference between two nearly equal quantities, the difference may be dominated by mainly the error or noise of these quantities. This results in the FD finding gradients based on the noise, not the actual gradient, and poor optimization. By looking at the mathematical form of the AG, 7.36, we see that the equation is not affected by the problem of difference between to nearly equal quantities. As for computational cost i.e. timing, we can see in section 17.1.3 that although we have exponential cost for both gradient schemes, the AG is close to 10^3 times faster. The FD needs to evaluate the RBM and produce MCMC samples two times for each parameter, while the AG evaluates the RBM simply once, and the whole gradient can be derived from the results of this single batch of MCMC samples. This gives the AG a superior scaling property.

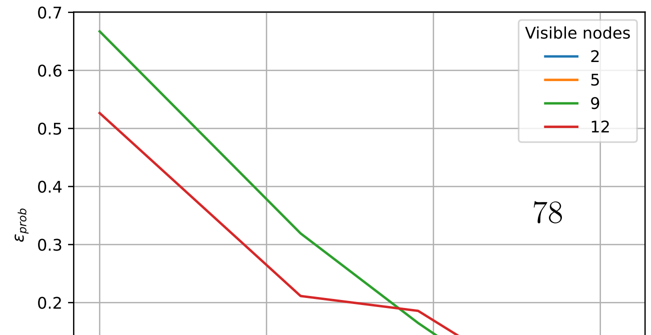
17.2 The MCMC algorithm

A very important piece of the NQS is the MCMC sampling algorithm. It is used to extract the important probability distributions from the RBM. The probability distributions are used to calculate the local energies, that in turn are used in the gradients and training. The error of the sampled distributions will influence the results, so it is important to have enough steps as described in 6.5. Although important, it is not clear from literature what the number of samples should be to achieve desired accuracy for a given system size. First, we present some result testing how different numbers of samples, N_σ, impact state error 7.5.2 on a random RBM. Second, we propose a formula for calculating the needed number of steps for increasing system sizes that seems to keep the error somewhat constant for different system sizes. Last in the section, we will have a quick look at how the warm up step affects the MCMC distribution ϵ_{prob}

17.2.1 Error for increasing system size

We have seen that the MCMC algorithms achieves good results given enough samples. Here we want to investigate how many samples are required in relation to the target distribution size.

In figure 17.4 we can see the probability error presented in section 6.5. The plot show very similar trends for all sys-



tem sized. When the sample size is low, we have a huge error between the estimated and target probability. When the number of samples is $\sim 10^3$, the error is very low for all of the system sizes tested. This is not surprising, but it is obvious that bigger systems require a larger amount of samples.

17.2.2 Formula for finding N_λ

To test this further, we experimented with different formulas for deciding the number of samples to be included based on the system size. Through trial and error (as literature suggests), we landed on this formula for calculating necessary MCMC samples that seems to scale well with an increasing system size. $N_\lambda = K \cdot N_\sigma^p$. Figure 17.5 presents the result of testing this formula with two different choices of K and p. Each parameter configuration is run 10 times.

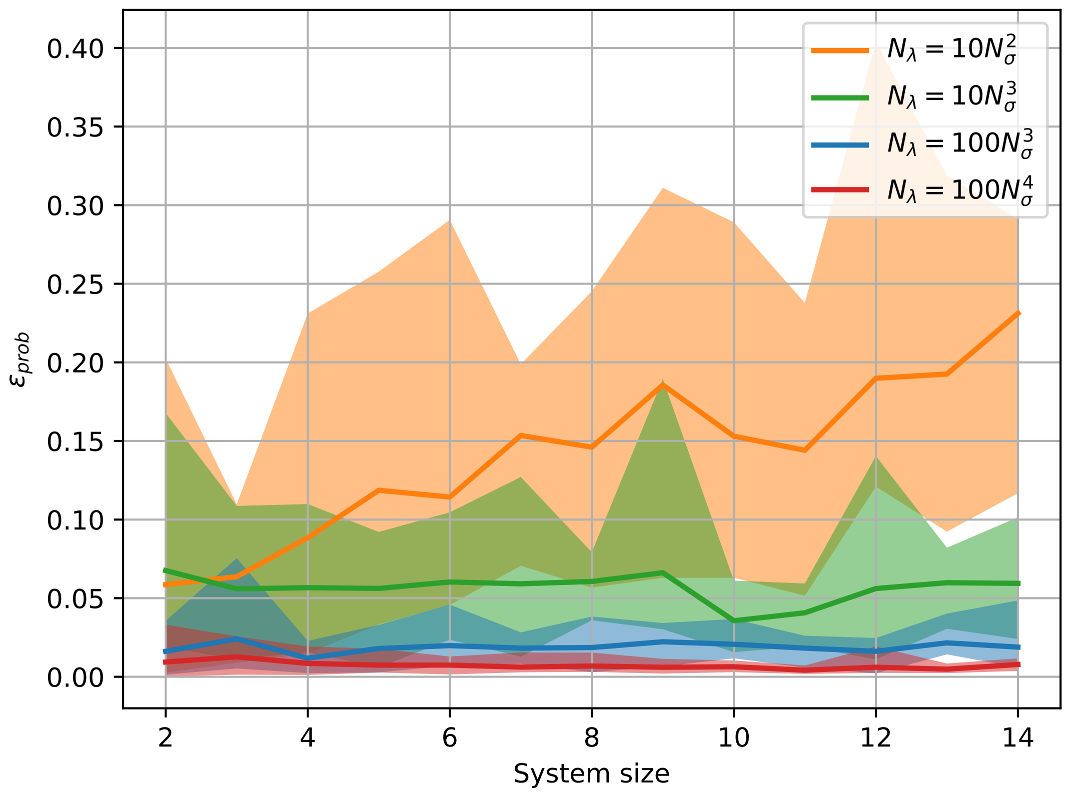


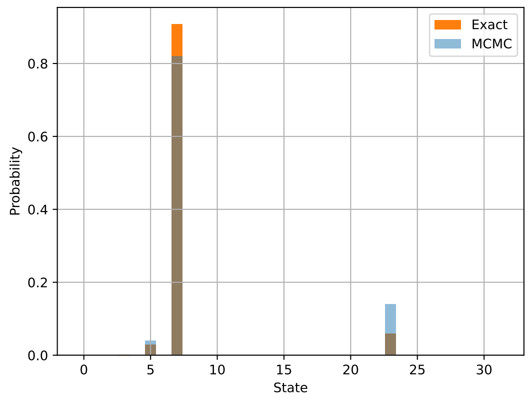
Figure 17.5: ϵ_{prob} averages over 10 runs with increasing system size with polynomial increase in MCMC samples N_λ related to the system size N_σ.

We can see that the polynomial increase in MCMC steps gives us a quite consistent accuracy for system sizes between 2 and 14 qubits. The ϵ_{prob} does not increase in any significant manner.

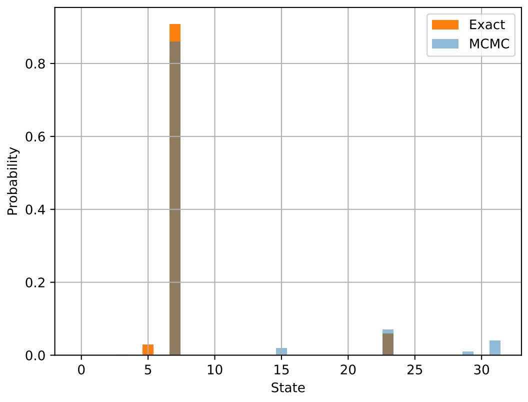
17.2.3 Warm up steps

In this section we look at how the number of collected steps, and the warm up steps affect the accuracy of the estimated distribution as discussed in 6.6. We will see how the inclusion of warm up steps affect the accuracy of the MCMC probability. Again, literature does not provide a definite number for this, but leaves it to trial and error.

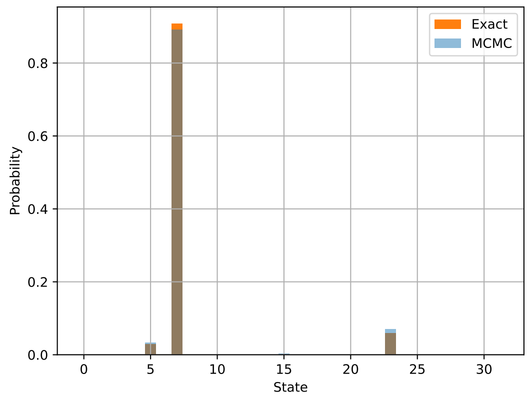
We sample the same random RBM with increasing number of collected samples, both with and without the warm up steps. In figure 17.2.3 we visualize this with a toy setup. A RBM estimating a 5 qubit system (visible size 5) with a hidden layer size of 10 for a Ising model Hamiltonian. We run the MCMC algorithm 6.4 with 100 steps: One time with warm up steps 17.6a, and one time without warm up steps 17.6b. After that we do the same setup but with 1000 collected steps (still 10 warm up steps).



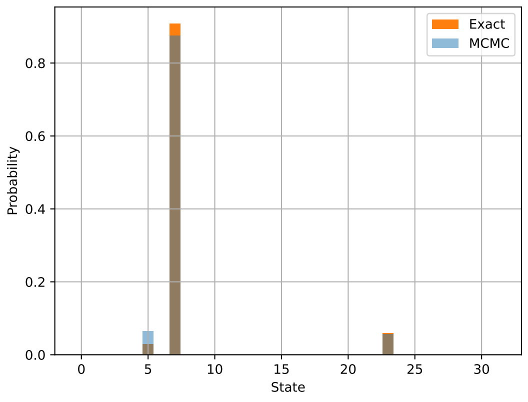
(a) 100 MCMC steps, 10 warm up steps



(b) 100 MCMC steps, no warm up



(c) 1000 MCMC steps, 10 warm up steps



(d) 1000 MCMC steps, no warm up

Figure 17.6: Histograms showing how the probability distribution for the RBM and the one found by the MCMC algorithms overlap. Parameters: visible nodes = 5, hidden nodes = 5, walker and warm up steps below each figure.

The left columns shows the sampling with warm up, and the right column without warm up steps. We can see in figure 17.6a that the inclusion of warm up steps aids the sampled distribution to follow the target distribution compared to figure 17.6b which has no warm up steps. We can see that some samples (i.e. 15, 29 and 30) are included although they are not a part of the target distribution. When we increase the number of samples, the estimated distributions, both with our without warm up steps, look more alike. At 1000 samples, the impact of the warm up steps has little to no impact.

To further investigate the effect of the warm up steps, we did several runs sampling different sized systems and calculated the average error over multiple runs. The number of warm up samples is $N_\lambda/10$. In figure 17.7 we can see that there is a difference in accuracy between MCMC estimated distributions with or without warm up steps.

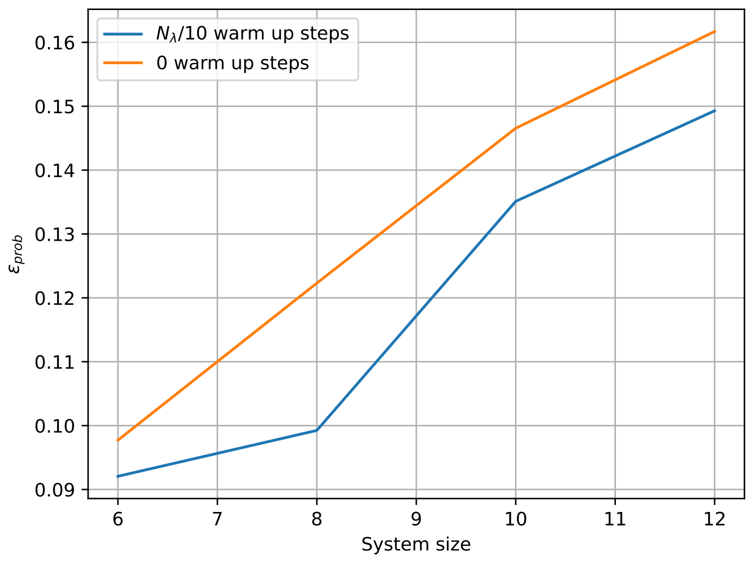


Figure 17.7: ϵ_{prob} averages over 50 runs with increasing system size with polynomial increase in MCMC samples N_λ related to the system size N_σ as $100N_\sigma^3$, and warm up steps being $N_\sigma/10$.

The distribution collected with $n_\lambda/10$ warm up steps has a slightly better $\epsilon_{prob} \sim 0.01$ error than the distributions collected without warm up steps. This is significant in cases where low error is important.

17.2.4 Discussion

The systems we sample grows exponentially in size, and we want to beat this complexity. It is generally agreed upon in literature that the number of samples required is a art of trial and error. We propose a formula for calculating the number of required steps

$$N_\lambda = K \cdot N_\sigma^p \tag{17.1}$$

where K and p are constants to be tuned for desired accuracy, and N_σ is the system size i.e. number of qubits. As figure 17.5 shows, this approach yields good results for increasing system sizes. The size of the variable K is more important for achieving low errors for small systems, but becomes negligible for larger systems. For $p = 3$, we cannot see the error increase over the span of a system of 2 to 14 qubits. Due to the increased number of N_λ, the same stability applies to $p = 4$, but with an overall lower error. We should probably see even lower errors with higher p values, but higher p values will also add a computational cost. Tuning p is a matter of desired accuracy, but we see that $p = 3$ is the lowest value that keeps the error consistent across different system sizes. We suggest keeping $p = 3$ and instead increasing K to achieve desired accuracy, thus avoiding an overestimating of the number of N_λ for larger systems.

Formula (17.1) can provide a good staring ground that scales well with the operational basis. It grows polynomial $O(N^p)$, which beats the exponential $O(2^N)$ for the system. We have to keep in mind that these experiments are done on a random RBM, that is an RBM with randomly generated biases and weights. It is not obvious that this formula generalizes for the case where the RBM is trained on any Hamiltonian. Still, experiments run with this formula like the one in 17.4, shows good results in training.

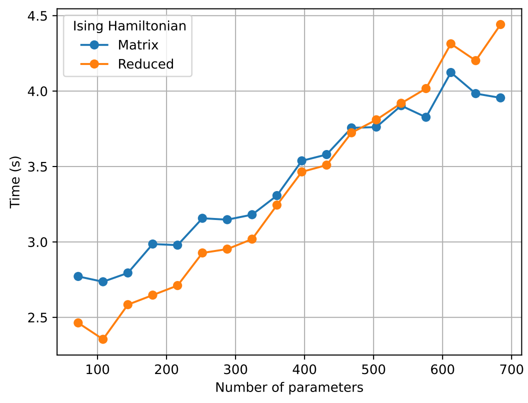
The warm up steps experiments in 17.2.3 shows that the warm up steps do matter, although on a small scale. Used in combination with the system size scaling (17.1), we see that when choosing $N_\lambda/10$ as the number of warm up steps, the distribution beats the error of the non-warm up distribution by $\epsilon_{prob} \sim 0.01$.

17.3 Investigating the hidden node parameter

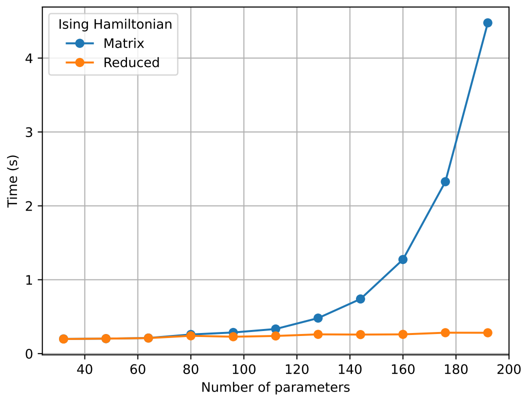
In this section we will see how the main parameter of the RBM, the hidden layer size N_h, is affecting the RBM's ability to grasp quantum states. One of the settings that are some what brushed over in literature, is the effect of the size of the hidden layer on the models accuracy i.e. how many hidden nodes are required to achieve a good approximation of the ground state energy. The hidden layer is what gives the RBM its expressiveness and flexibility to grasp the intricate quantum systems. As we have shown in 17.3.1, the hidden layer size does contribute to computational time in a linear fashion. We want to investigate how small hidden layer size we can use to reduce computation time while still maintaining good results for the random Hamiltonian, that has very many degrees of freedom, and the Ising model which has less freedom and more structure. We also will investigate the computational impact for both the hidden and visible layer on the matrix Hamiltonian 2.7.1 and the tensor product Hamiltonian 2.7.3

17.3.1 Timing with increasing system size

As discussed earlier, one of the main ideas behind using the RBM as an ansatz for a complex Many Body Problem (MBP) is its ability to extract structures from the problem. Here we will compare the Ising model Hamiltonian represented as a sum of tensor products and general Hamiltonian as a matrix. We have measured training time of both Hamiltonians in two settings. The first where we increase the number of hidden nodes in the RBM while maintaining the same number of visible nodes, and the second where we increase the visible nodes and keep the hidden nodes fixed.



(a) 20 to 200 Hidden nodes



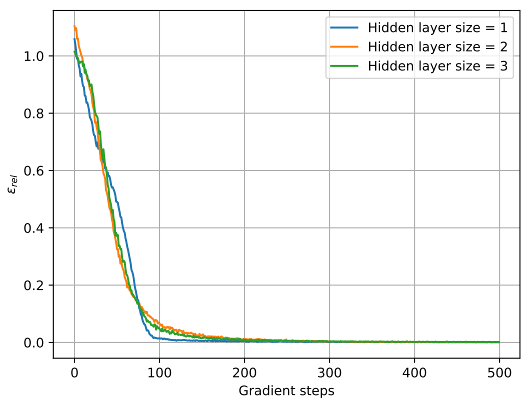
(b) 2 to 10 visible nodes

Figure 17.8: Speed comparison of training a RBM with the two Ising model representation. Training is done with 100 gradient decent steps and with 1000 MCMC steps. In figure 17.8a the visible layer is fixed to 6 nodes and the hidden layer size is increased from 20 to 200. In figure 17.8b the hidden layer is fixed to 16 nodes, and the visible layer is increased from 2 to 10 nodes.

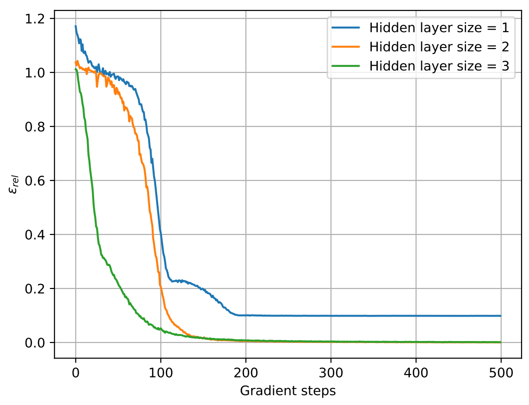
In figure 17.8a we see that the time for the two Hamiltonians increase in a proportional and linear manner. In figure 17.8b we can see that the two representations are very similar for small (nodes < 6) systems. For bigger ($6 <$ nodes) systems we see the exponential increase in time for the matrix representation, while the tensor product Hamiltonian is linear, and barely increases at all on this scale.

17.3.2 How many hidden nodes do we require?

Here we want to restrict the RBM to have as few hidden nodes as possible, but still reach the ground state energy after training. We ran the experiments first with a visible layer size of 4 then a visible layer size of 5 representing a system size of 4 and 5 respectively. We then ran three training runs of the RBM with a hidden layer size of 1, 2 and 3. The plots in figure 17.9a and 17.9b show the relative error 7.44 vs. steps in the gradient descent.



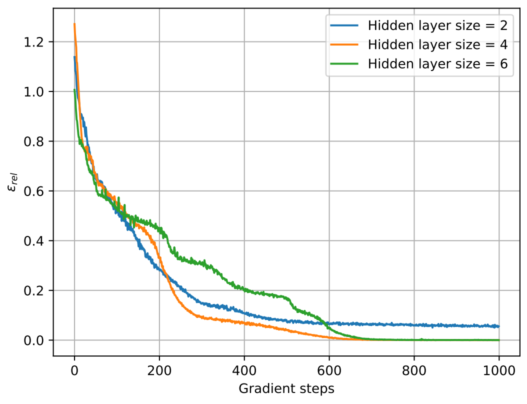
(a) System size 4



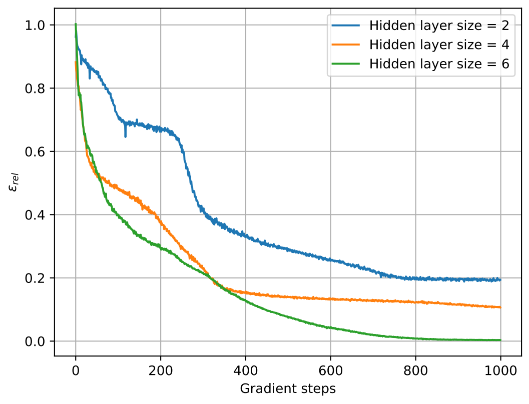
(b) System size 5

Figure 17.9: Ising tensor product Hamiltonian with increasing hidden layer size. MCMC steps: $100N_\sigma^3$

For a system size of 4, only 1 hidden node is required to find an energy with an error $\epsilon_{rel} < 0.01$ to the true ground state. For a system of 5 qubits, 2 hidden nodes are needed for the same accuracy. Then we repeated the experiment for a generic matrix Hamiltonian of size 4 and 5. Here we needed to increase the hidden nodes to 2, 4 and 6 to be able to find the ground state energy as seen in figures 17.10a and 17.10b.



(a) System size 4



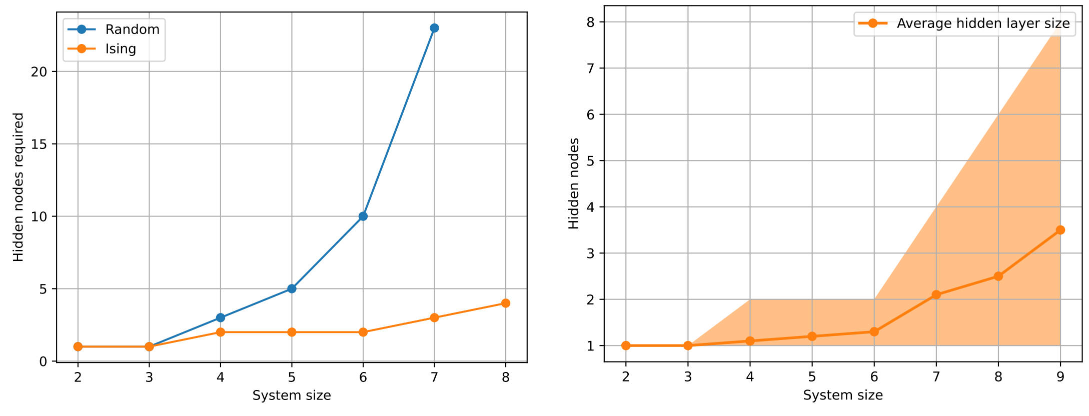
(b) System size 5

Figure 17.10: Random Hamiltonian with increasing hidden layer size. MCMC steps: $100N_\sigma^3$

We can see that even with twice as many hidden nodes, the RBM has more difficulties expressing the ground state energy. In both figure 17.10a and 17.10b we can see that the ground state is found at 6 hidden nodes. Still, the bigger system in 17.10b needs almost twice as many training steps to achieve the ground state.

Last in this section we want to see how the hidden layer size scale with different Hamiltonians representations. This is done by creating a tensor product

Hamiltonian (Ising) and generic Hamiltonian (Random) with visible size 2. Then wee see if it can solve it with 1 hidden node with 0.01 relative error, ϵ_{rel}. If 1 hidden node is enough, we increase the visible size to 2 nodes. Then we again try to solve it with 2 hidden nodes. If we can't get a $\epsilon_{rel} < 0.01$, we increase the number of hidden nodes by 1 and try to solve again. If we achieve the target accuracy, we increase the system size by 1. This is repeated for a range system sizes from 2 to 8.



(a) Hidden nodes required for $\epsilon_{rel} < 0.01$ with increasing system size.

(b) The average hidden layer size needed to achieve $\epsilon_{rel} < 0.01$ plotted with the spread over 10 runs.

Figure 17.11: Plots showing how many hidden nodes are required for $\epsilon_{rel} < 0.01$ for random (blue) and Ising (orange) Hamiltonian

In figure 17.11a we can again see the exponential growth for the generic Hamiltonian, as we have seen in figure 17.8b. Although they do not use the same measure, we can see the quickly escalating computational complexity if we don't abuse structure. Here we see that the required hidden size to get an $\epsilon_{rel} < 0.01$ is growing exponentially for the random Hamiltonian.

17.3.3 Discussion

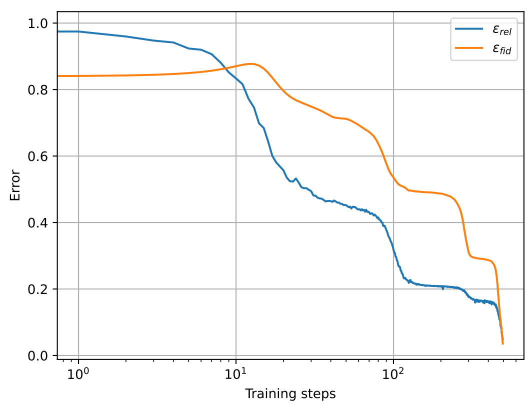
The true power of the RBM can be seen in section 17.3.1, from figure 17.8b where we increase the system size the RBM. If it used with a matrix Hamiltonian i.e. a non-compressible Hamiltonian, we see the exponential growth that is expected for quantum systems. When we look at the results in figure 17.8b, we see a huge difference in growth between the matrix Hamiltonian and tensor product Hamiltonian (reduced Hamiltonian). While the matrix Hamiltonian training time grows exponentially, the tensor product Hamiltonian training grows sub-exponentially for the system size range we present here. This is strong evidence for a huge speed when using the tensor product structure of the Ising Hamiltonian, especially for larger systems.

In section 17.3.2 we see that the number of hidden nodes required to train

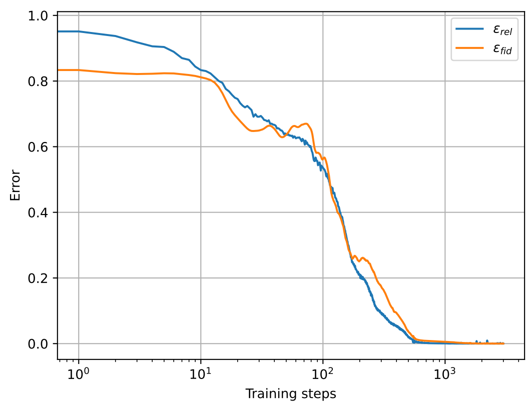
RBM well, i.e. get close to the true ground state, is scaling much slower for the random Hamiltonian than that for the Ising Hamiltonian. This is to be expected since the random Hamiltonian has more degrees of freedom and thus require a greater expressiveness from the RBM. We know that an Ising Hamiltonian has only linearly many degrees of freedom as opposed to exponentially many for the random Hamiltonian. However, it is not immediately obvious that a RBM can use this structure to find more efficient representations of the ground state. Even so, from the results we see that the RBM requires much fewer hidden nodes to learn the Ising Hamiltonian than the random Hamiltonian. This shows that the RBM is able to discover smarter ways to compress and represent the Ising model.

17.4 The accuracy of ground state and ground state energy

With all the computational arguments investigated, it is natural to check the results they give when finding the ground state energy of a system. This benchmark will tell us if our implementation holds up to the claim of being able to approximate quantum systems. The training is done by using the variational principle to find the lowest energy. The measure of accuracy in this case is the difference between the RBM energy and the actual Hamiltonian ground state, the relative error ϵ_{rel} 7.5.1. This is an obvious measure to check how our model perform, given the importance of ground state energies in quantum physics. Another aspect we want to investigate is that although a solution is good in terms of energy, which was optimized, we do not have a guarantee that the state we find is close to the actual ground state energy. Two states can have very close energies and not be the same state i.e. $E_\theta \simeq E_{gs} \;\not\Rightarrow\; \psi_\theta \simeq \psi_{gs}$. Given that the state configuration is often as important as the energy, it is interesting to see how the RBM state compares to that of the ground state. By using a *state error* as described in equation (7.45) for the random Hamiltonian, and (7.46) for the Ising Hamiltonian. we can compare states to get a second measure of error.



(a) Ising Hamiltonian. Visible nodes: 6, hidden Nodes: 6, MCMC steps: 21600, warm up step: 2160. terminated ad $\epsilon_{rel} < 0.05$



(b) Random Hamiltonian, Visible nodes: 6, hidden Nodes: 18, MCMC steps: 21600, warm up step: 2160. No termination, gradient steps: 300

Figure 17.12: Relationship between the relative error of the energy and state error while training for two different Hamiltonians

The energy converges very well for both the Ising Hamiltonian, and the random Hamiltonian. This means our implementation works and that the parameter setting we suggest yields good results. Also, we see that the energy is "guiding" the state fidelity error towards zero. The ϵ_{fid} is weaving around the ϵ_{rel} toward the optimal solution.

17.4.1 Discussion

The RBM initialised with parameters found after experiments 17.2 and 17.3.2 and trained with the gradient that showed best performance in 17.1 manages to find the ground state energy with ~ 0 relative error. This shows that our implementation can indeed find a very good approximation of the quantum system described by the Hamiltonian, both for the more structured Ising model and for the generic Hamiltonian. Another interesting find is that the ground state seems to converge as well as the energy. This means that the error of the energy is a good proxy for the error of the state, at least for the discrete spin system we have studied. This is a very desirable result, although not obvious from training the RBM with the target of minimizing energy only. This is not the case when using a RBM to approximate the ground state of electron systems in space [37]. Another observation is that the optimizer hits plateaus during training in figure 17.9a and 17.12a, but manages to break free from these. This is a desired property, since the training is less prone to get stuck. Our theory is that the Adam optimizer is aiding the gradients when plateaus are met, and the momentum this algorithm adds lets the optimiser escape the plateaus.

Part IV

Conclusion and future work

18 | Conclusion

18.1 Conclusion

This thesis has resulted in a Python framework that implements Neural Quantum State (NQS) as a way to approximate ground state energies for discrete quantum spin systems. We have shown that our implementation of a Restricted Boltzmann Machine (RBM), trained by the use of a quantum Monte Carlo algorithm can achieve promising results with sub-exponential growth of computational cost. The Analytical Gradient derived in this thesis is shown to give the true gradient and is 10^3 faster to use than a traditional Finite Difference scheme. We have investigated several of the parameters for the RBM and the Markov Chain Monte Carlo (MCMC) algorithm, and their impact on accuracy and speed. We propose a function for finding good values for the number of samples in the MCMC distribution based on system size. The experiments done in this thesis show that the RBM manages to find the ground state energy of structured Hamiltonian with sub-exponential number of parameters. The experiments show also that the quantum Monte Carlo algorithm used, has a linear growth in computational time when solving a structured Hamiltonian.

The analytical gradient

The analytical gradient for the RBM in the operational basis 7.36, derived and used in this thesis, is shown to be correct and to outperform a numerical gradient scheme (Finite Difference). Compared to the FD, it converges better with the same amount of MCMC samples and runs 10^3 times faster. It also achieves to train the RBM for the random Hamiltonian, and the Ising Hamiltonian for all cases we tested.

A proposed formula for the number MCMC steps

As a solution to find the required number sampling steps, we propose a formula as an aid for a good staring point. Recognised as an art of trial and error, finding the desired number of MCMC steps can be time-consuming. Our experiments show that the formula $N_\lambda = K \cdot N_\sigma^p$ gives a consistent accuracy across different system sizes when $p = 3$. The warm up steps used in combination with the system size scaling formula, can be set to $N\sigma/10$ and gives an improvement of ~ 0.01 accuracy.

Importance of hidden layer size

We found that our Ising model (2.14) has a non-exponential growth in the number of hidden nodes required to find the true ground state a Hamiltonian. This shows that the RBM effectively manages to compress the system space, and find its structure. This is not the case with the generic Hamiltonian that requires an exponential growth in hidden nodes. It cannot be different, otherwise all quantum states can be effectively "squeezed" into classical states, and this is not possible. This knowledge of the required hidden nodes can be used as a well-educated guess for initializing much bigger systems.

Training and accuracy

Solving for the ground state with the use of the parameter setting we found, shows us that the method achieves very good results. It finds the ground state energy for both the random Hamiltonian and the Ising Hamiltonian. We also see that when trained to minimize energy, the RBM manages to find the ground state. Additionally, it is robust against plateaus and local minima during training.

18.2 Future research

A natural next step for future work is to use the parameter setting we have found to run a simulation of a much larger system: typically one that would be impossible to solve on a reasonable sized classical computer due to memory requirements i.e. ~ 20 qubits. Based on our test, our implementation should be able to find the ground state of such a system in reasonable time. Since the computers we had access to during this thesis is not of the high performing kind, this test was postponed.

Implement support for parallel processing would speed up the processing time for the module. Now all calculations are done on one thread, and this is not optimal on today's multi-core processors. This could be especially beneficial for the computation times when generating the samples with the MCMC algorithm. Instead of using one walker gathering the whole sample distribution, we could let several walkers collect a smaller sample size and then combine them.

Adding support for new Hamiltonians would be another way to expand the module. While the toy-like Hamiltonians used in this thesis are good for benchmarking, Hamiltonians that describe real-life problem would be interesting and useful. A low hanging fruit is a more interesting Ising model that can describe real-life spin systems e.g. ferromagnets.

The RBM is only one of many Neural Quantum States that are studied these days. Implementing other Neural Network ansatzes and more sophisticated ansatzes like Fermi-net [38], recurrent neural networks [39], and transformer networks [40] will make the module a platform for comparing ansatzes.

A | Appendix

A.1 Source code repository

The module created and used in this project can be found in the Github repository `https://github.com/Overskott/Master-Thesis-Project.git`.

A.2 List of figures

Bibliography

[1] Matteo Atzori and Roberta Sessoli. "The Second Quantum Revolution: Role and Challenges of Molecular Chemistry." In: *Journal of the American Chemical Society* (2019).

[2] Lov K. Grover. *A fast quantum mechanical algorithm for database search.* 1996. arXiv: `quant-ph/9605043 [quant-ph]`.

[3] Peter W. Shor. "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer". In: *SIAM Journal on Computing* 26.5 (Oct. 1997), pp. 1484–1509. DOI: `10.1137/s0097539795293172`. URL: `https://doi.org/10.1137%2Fs0097539795293172`.

[4] Roger Melko et al. "Restricted Boltzmann machines in quantum physics". In: *Nature Physics* 15 (June 2019). DOI: `10.1038/s41567-019-0545-1`.

[5] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition.* Cambridge University Press, 2010. DOI: `10.1017/CBO9780511976667`.

[6] Tom Siegfried. *In honor of his centennial, the Top 10 Feynman quotations.* (version: 2018-05-11). eprint: `https://www.sciencenews.org/blog/context/top-10-richard-feynman-quotations`. URL: `https://www.sciencenews.org/blog/context/top-10-richard-feynman-quotations` (visited on 07/26/2023).

[7] Frank Arute et al. "Quantum supremacy using a programmable superconducting processor". In: *Nature* 574.7779 (2019), pp. 505–510. DOI: `t`.

[8] Robin Harper and Steven T. Flammia. "Fault-Tolerant Logical Gates in the IBM Quantum Experience". In: *Phys. Rev. Lett.* 122 (8 Feb. 2019), p. 080504. DOI: `10.1103/PhysRevLett.122.080504`. URL: `https://link.aps.org/doi/10.1103/PhysRevLett.122.080504`.

[9] Jordan Sullivan and Matthew Beach. *Amazon Braket launches IonQ Aria with built-in error mitigation.* 2023. URL: `https://aws.amazon.com/blogs/quantum-computing/amazon-braket-launches-ionq-aria-with-built-in-error-mitigation/` (visited on 07/29/2023).

[10] Giuseppe Carleo. *Neural-Network Quantum States. A Lecture for the Machine Learning and Many-Body Physics workshop*. June 2017.

[11] Giuseppe Carleo and Matthias Troyer. "Solving the quantum many-body problem with artificial neural networks". In: *Science* 355.6325 (Feb. 2017), pp. 602–606. DOI: 10.1126/science.aag2302. URL: https://doi.org/10.1126%5C%2Fscience.aag2302.

[12] Ahmet Murat Ozbayoglu, Mehmet Ugur Gudelek, and Omer Berat Sezer. *Deep Learning for Financial Applications : A Survey*. 2020. arXiv: 2002.05786 [q-fin.ST].

[13] Prajoy Podder et al. *Artificial Neural Network for Cybersecurity: A Comprehensive Review*. 2021. arXiv: 2107.01185 [cs.CR].

[14] Jinghua Zhang et al. *Applications of Artificial Neural Networks in Microorganism Image Analysis: A Comprehensive Review from Conventional Multilayer Perceptron to Popular Convolutional Neural Network and Potential Visual Transformer*. 2022. arXiv: 2108.00358 [cs.CV].

[15] Marco Tavora. *Neural quantum states*. Aug. 2020. URL: https://towardsdatascience.com/neural-quantum-states-4793fdf67b13.

[16] D. Perez-Garcia et al. *Matrix Product State Representations*. 2007. arXiv: quant-ph/0608197 [quant-ph].

[17] Yusuke Nomura et al. "Restricted Boltzmann machine learning for solving strongly correlated quantum systems". In: *Physical Review B* 96.20 (Nov. 2017). DOI: 10.1103/physrevb.96.205152. URL: https://doi.org/10.1103%5C%2Fphysrevb.96.205152.

[18] Giacomo Torlai and Roger G. Melko. "Latent Space Purification via Neural Density Operators". In: *Phys. Rev. Lett.* 120 (24 June 2018), p. 240503. DOI: 10.1103/PhysRevLett.120.240503. URL: https://link.aps.org/doi/10.1103/PhysRevLett.120.240503.

[19] A. Decelle, G. Fissore, and C. Furtlehner. "Spectral dynamics of learning in restricted Boltzmann machines". In: *EPL (Europhysics Letters)* 119.6 (Sept. 2017), p. 60001. DOI: 10.1209/0295-5075/119/60001. URL: https://doi.org/10.1209%5C%2F0295-5075%5C%2F119%5C%2F60001.

[20] Luciano Loris Viteritti, Francesco Ferrari, and Federico Becca. "Accuracy of restricted Boltzmann machines for the one-dimensional J_1-J_2 Heisenberg model". In: *SciPost Physics* 12.5 (Apr. 2022). DOI: 10.21468/scipostphys.12.5.166. URL: https://doi.org/10.21468%5C%2Fscipostphys.12.5.166.

[21] Zhih-Ahn Jia et al. "Quantum Neural Network States: A Brief Review of Methods and Applications". In: *Advanced Quantum Technologies* 2.7-8 (Apr. 2019), p. 1800077. DOI: `10.1002/qute.201800077`. URL: `https://doi.org/10.1002%5C%2Fqute.201800077`.

[22] Dong-Ling Deng, Xiaopeng Li, and S. Das Sarma. "Quantum Entanglement in Neural Network States". In: *Phys. Rev. X* 7 (2 May 2017), p. 021021. DOI: `10.1103/PhysRevX.7.021021`. URL: `https://link.aps.org/doi/10.1103/PhysRevX.7.021021`.

[23] Kristian Wold. *Parameterized quantum circuits for machine learning.* Dec. 2021. URL: `http://urn.nb.no/URN:NBN:no-92142`.

[24] Ashkan Shekaari and Mahmoud Jafari. *Theory and Simulation of the Ising Model.* 2021. arXiv: `2105.00841 [cond-mat.stat-mech]`.

[25] Román Orús. "A practical introduction to tensor networks: Matrix product states and projected entangled pair states". In: *Annals of Physics* 349 (Oct. 2014), pp. 117–158. DOI: `10.1016/j.aop.2014.06.013`. URL: `https://doi.org/10.1016%5C%2Fj.aop.2014.06.013`.

[26] Donald D. Fitts. "Principles of Quantum Mechanics: As Applied to Chemistry and Chemical Physics". In: 1999.

[27] Guido Montufar. *Restricted Boltzmann Machines: Introduction and Review.* 2018. DOI: `10.48550/ARXIV.1806.07066`. URL: `https://arxiv.org/abs/1806.07066`.

[28] Asja Fischer and Christian Igel. "An Introduction to Restricted Boltzmann Machines". In: *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications.* Ed. by Luis Alvarez et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 14–36. ISBN: 978-3-642-33275-3.

[29] Joshua S. Speagle. *A Conceptual Introduction to Markov Chain Monte Carlo Methods.* 2020. arXiv: `1909.12313 [stat.OT]`.

[30] Giuseppe Carleo. *Machine learning methods for many body problems. Lectures for the Advanced School on Quantum Science and Quantum technology.* Sept. 2017.

[31] Tian-Cheng Yi, Richard T. Scalettar, and Rubem Mondaini. "Hamming distance and the onset of quantum criticality". In: *Physical Review B* 106.20 (Nov. 2022). DOI: `10.1103/physrevb.106.205113`. URL: `https://doi.org/10.1103%2Fphysrevb.106.205113`.

[32] Andrew Gelman. *Burn-in for MCMC, why we prefer the term warm-up.* 2017. URL: https://statmodeling.stat.columbia.edu/2017/12/15/burn-vs-warm-iterative-simulation-algorithms/ (visited on 07/19/2023).

[33] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization.* 2014. DOI: 10.48550/ARXIV.1412.6980. URL: https://arxiv.org/abs/1412.6980.

[34] J. W. Thomas. *Numerical Partial Differential Equations: Finite Difference Methods.* Springer New York, 1995. DOI: 10.1007/978-1-4899-7278-1. URL: https://doi.org/10.1007/978-1-4899-7278-1.

[35] Gajjar Deep. *Prove Numpy Is Faster Than Normal List (python).* 2022. URL: https://www.algorithmtech.tech/2022/01/prove-numpy-is-faster-than-normal-list.html (visited on 07/01/2023).

[36] Numpy.org. *numpy.lib.mixins.NDArrayOperatorsMixin.* 2023. URL: https://numpy.org/doc/stable/reference/generated/numpy.lib.mixins.NDArrayOperatorsMixin.html (visited on 06/28/2023).

[37] Even Marius Nordhagen. *Studies of Quantum Dots Using Machine Learning.* Mar. 2020. URL: https://www.duo.uio.no/handle/10852/73753.

[38] David Pfau et al. *Ferminet: Quantum Physics and chemistry from first principles.* Oct. 2020. URL: https://www.deepmind.com/blog/ferminet-quantum-physics-and-chemistry-from-first-principles.

[39] IBM. *What are recurrent neural networks?* URL: https://www.ibm.com/topics/recurrent-neural-networks.

[40] Yuan-Hang Zhang and Massimiliano Di Ventra. "Transformer quantum state: A multipurpose model for quantum many-body problems". In: *Physical Review B* 107.7 (Feb. 2023). DOI: 10.1103/physrevb.107.075147. URL: https://doi.org/10.1103%2Fphysrevb.107.075147.