WROCŁAW UNIVERSITY OF SCIENCE AND TECHNOLOGY
Faculty of Fundamental Problems of Technology
Big Data Analytics

MASTER'S THESIS

Machine learning-based solution of simple quantum problems

inż. Jakub Grabowski

Supervisor:
prof. dr hab. Maciej Maśka

Wrocław 2024

Abstract

This thesis explores the potential use of machine learning techniques to solve complex quantum mechanical problems, particularly in many-body spin systems. The exponential growth of the Hilbert space in such systems presents significant computational challenges. Traditional methods, such as exact diagonalization, often fail in efficiently solving these problems due to computational cost and memory storage. However, machine learning offers a promising alternative by leveraging data-driven models that can learn complex patterns and provide satisfactory approximations.

The research begins with a detailed discussion of the quantum mechanical background, exploring the concept of spin and various models describing their interactions, like the Ising, XY, Heisenberg and Kitaev models. These are fundamental in understanding quantum phenomena such as magnetism, superconductivity, and phase transitions. The thesis then introduces the concept of machine learning from the ground up, discussing various architectures of neural networks and regularization techniques. The topic of encoding a neural network as a wave function Ansatz is also covered, with the potential use in variational Monte Carlo, where the expectation value of the ground state energy is estimated based on samples generated by the Metropolis algorithm. Additionaly, the discussion covers also the regularization techniques based on quantum geometric tensor, which can reduce optimization time significantly. Although traditional methods allowed up to 30 spins to be simulated, these techniques have the potential to scale up to hundreds of spins, given optimal parameters and sufficient computational power at hand.

Furthermore, the thesis provides a comprehensive overview of the modern computational tools used to implement machine learning models, algorithms, and optimizations, particularly JAX, Flax, and NetKet. These libraries enable efficient and scalable computations, which are essential for handling large datasets and complex models in quantum simulations. Then it proceeds to benchmarking and simulations. First, the same computational tasks are given to both CPU and GPU to compare which ones will perform better. Then various wave function Ansatze are compared for different number of samples for expectation value estimation. After finding best configuration and most powerful machine learning models, the ground state energy search is performed. Additionally, the potential use case of Penalty-based VMC algorithm is utilized by finding excited states for spin models.

Contents

1 Introduction

Quantum mechanics is the theoretical framework describing the behavior of matter and energy at subatomic scales [1]. Developed in the early 20th centure, remains one of the most profound and successful scientific theories. Among the many revolutionary concepts it provides, the notion of an intrinsic form of angular momentum, called spin, is particularly important [2]. Spin is a purely quantum mechanical property, which has no classical counterpart. Although its notation is complex and abstract, it has practical implications in undestanding a wide range of physical phenomena, from the magnetic properties of materials to exotic phases of matter, like quantum spin liquids or frustrated states [3]. However, studying quantum spin systems, especially those involving many interacting particles, poses significant challenges due to the complex mathematics due to exponentially increasing dimensionality of the Hilbert space with scaling the spin system.

Potential solutions for challenging problems in the field might be provided by artificial intelligence, which sparked a revolution in the world of technology in recent years, mostly due to advance neural networks, like ChatGPT [4, 5]. Machine learning algorithms have shown great promise in approximating solutions to problems that are otherwise intractable using traditional methods. This thesis explores the integration of machine learning techniques, Monte Carlo simulations [6, 7] and quantum mechanics, with variational Monte Carlo as a major target to grasp during the comprehensive discussion. Complementing these chapters with leveraging modern computational frameworks such as JAX, Flax and NetKet [8, 9, 10], this research aims to provide new insights into the behavior of quantum systems and demonstrate the potential of machine learning in advanced quantum simulations.

Quantum mechanical background

The chapter provides a comprehensive overview of the quantum mechanical principles, theory of the spin, and models relevant to the study of spin systems. Spin is a purely quantum mechanical property, described by the Pauli matrices, spin operators characterized by their commutation relations [11]. The Pauli matrices are instrumental in describing spin-$\frac{1}{2}$ particles interacting with each other in a closed system. Together they exhibit a rich array of phenomena that form the basis for more complex models [12]. Then the chapter progresses to a detailed examination of various models used to describe spin interactions in many-body systems. The classical Ising model is introduced as a starting point, as the simplest model describing spin interactions [13]. It describes the particles as binary objects pointing up or down in a classical way, where each one interacts with its nearest neighbors on a 2D lattice. Despite its simplicity, the model is still able to exhibit phase transition at critical temperature, where it transitions from ferromagnetic to antiferromagnetic phase.

The further exploration of various spin models scales the degree of complexity up by introducing quantum Ising model [14], where spins are now defined on a Hilbert space as state vectors and we can act with operators on them. Quantum Ising model takes into consideration the Pauli σ_x, projecting the spin onto the x-axis, while at the

same time the transverse magnetic field needs to be accounted (acting on the z-axis). Because the two operators do not commute, the quantum phenomena emerge and can be simulated with numerical methods. This model serves as a getaway to understanding more sophisticated systems and highlights the computational challenges associated with high-dimensionality of state spaces.

The chapter then explores the XY and Heisenberg models [15, 16, 17], which extend the Ising model by considering spin interactions in additional dimensions. The XY model allows for rotations in the plane, introducing continuous deggrees of freedom, exhibiting new phenomena such as spin waves and Kosterlitz-Thouless transition, which describes the process of unbinding the vortex-antivortex pairs in a two-dimensional system [18]. The Heisenberg model considers full three-dimensional spin interactions and has a few variations that treat each axis in a slightly different way. The most basic one has only one parameter describing the coupling strength, while others, such as XXZ and XYZ models [12], introduce anisotropy into the system by providing additional parameters, which change the behavior of the system depending on the considered axis. Finally, the last system considered is the Kitaev model [3], which is highly anisotropic and particles are defined on a honeycomb lattice, where each of the three neighbors has an entirely different type of interaction. The model is particularly notable for its ability to host a quantum spin liquid ground state, where matter does not exhibit long-range magnetic order even at absolute zero temperature.

Machine learning for quantum systems

The goal of the second chapter is to present the potential enhancement of quantum many-body system simulations by incorporating machine learning models [19]. Firstly, the reader is introduced to the core concepts of machine learning. The whole field, especially deep learning [20], has revolutionized data analysis and pattern recognition by providing powerful tools with ability to learn and generalize from real world samples. This process is then useful for predicting future outcomes without being explicitly programmed. The whole chapter aims to bridge the gap between machine learning and quantum mechanics.

The true discussion starts by explaining the fundamentals of ML, including supervised and unsupervised learning. These represent different approaches to training neural networks [21]. The first is trained on labeled data by computing the cost function, where the goal is to reduce it in an iterative manner. The other learns from unlabeled data and finds its own way of storing its fingerprint based on energy minimization. The scope then shifts to neural networks, detailing their architecture, from simple perceptrons to more complex structures such as feedforward neural networks (FNN) [22]. One of the most promising models for quantum mechanics can be seen in neural networks called Restricted Boltzmann Machines (RBM) [19]. RBMs are energy-based models that can capture the probability distributions of quantum states and have been shown to be effective in representing the ground state wave functions in quantum spin models discussed in this work. In addition, the concept of RBM can be extended into structure with many hidden layers, forming a Deep Boltzmann Machines (DBM). Both types of neural networks can serve the purpose of wave function Ansatz, which

aims to represent a quantum state with fewer parameters that the conventional Hilbert space would require. Networks have proven to be successful in encoding long-range correlations in their set of parameters.

The key machine learning technique utilizing neural networks as wave functions is the variational Monte Carlo method, which is particularly useful for finding the ground state energy of quantum systems. VMC algorithm optimizes neural network models using gradient-based techniques to minimize the energy expectation value. Additionally, it incorporates complex regularization algorithm supporting the optimization by providing information about global energy landscape of the simulated system by utilizing the metric called quantum geometric tensor [23]. After a comprehensive discussion of the ground state search, a brief attempt to find excited states is presented. The specific algorithm used is called Penalty-based VMC and incorporates overlaps between different states to raise the energy minimum in the landscape [24].

Introduction to JAX, Flax and NetKet

The next chapter provides a practical guide to the computational tools and libraries used to implement variational Monte Carlo optimizing neural networks used as the quantum wave function Ansatz. The chapter is essential for understanding the technical aspects of the research as it details the software and frameworks enabling efficient computation. The first element of the discussion is JAX, a Python library designed for high-performance numerical computing [8, 25]. JAX provides automatic differentiation, which is necessary for gradient descent optimization algorithm. Another key feature offered by JAX is just-in-time (JIT) compilation, which translates instructions written in Python into machine code. Together with efficient storage, this approach to computation allows one to bypass Python's slow execution and achieve computational speeds compared to other compiled languages, such as C/C++.

Continuing the discussion, the chapter introduces Flax [9], a neural network library built on top of JAX that offers a flexible yet high-level interface for constructing and training neural networks. The framework utilizes functional programming paradigm, which is revolutionary compared to well-established ML tools, such as PyTorch [26], which focus on object-oriented style. The approach allows to separate the parameters of the model, which can be processed explicitly allowing for efficient work in the field of machine learning research.

The final section of the chapter focuses on NetKet [10], a specialized machine learning framework designed to study quantum many-body systems. It offers a comprehensive suite of tools for defining spin lattices, constructing Hilbert spaces and creating quantum operators. Additionally, the framework provides built-in neural networks on top of Flax, which makes it an ideal platform for quantum simulations. The chapter discusses how NetKet leverages variational Monte Carlo methods to optimize wave function Ansatz and perform large-scale quantum simulations. It provides detailed examples of using NetKet to simulate various quantum spin models, such as Ising, Heisenberg, and Kitaev models, and allows one to benchmark these simulations against traditional methods.

Purpose and scope of the thesis

The main objective of this diploma thesis is to explore the integration of machine learning techniques with quantum mechanics to address the computational challenges associated with solving quantum spin systems. To comprehensively grasp the whole field, several key areas need to be covered:

- *Studying foundational quantum mechanical concepts.* To understand the problem discussed, fundamental knowledge of quantum mechanics needs to be built. Then the whole topic of spin models is presented, such as the Ising, Heisenberg, and Kitaev models.
- *Studying the field of machine learning.* Secondly, the topic of ML needs to be introduce, with fundamental concepts such as neural networks, unsupervised learning and training models
- *Studying variational Monte Carlo method.* This part requires to comprehend every step of VMC optimization to perform a successful simulation.
- *Learning to use ML frameworks.* This part covers the study of JAX, Flax and NetKet - these tools need to be known to perform simulations.
- *Simulations.* Here, the colected knowledge is utilized to perform various ground state searches for system sizes beyond exact methods.

The author's own contribution:

- Acquiring knowledge in the field of quantum many-body physics, machine learning models and variational monte carlo methods.
- Learning to use frameworks for VMC optimization, specifically JAX, Flax, and NetKet.
- Implementing optimization loops using the tools mentioned above in Python.
- Performing various optimizations for different spin models.
- Custom implementation of Penalty-based VMC in NetKet for excited state search.
- Custom implementations of neural networks and Hamiltonians using NetKet.
- Maintaining the GitHub project with the code used for this thesis [27].
- Every figure and visualization was made by the author using MatPlotLib.

I would like to thank my supervisor for helping me understand the topic and sparking my interest in the field. I also would like to thank everyone who supported me in writing this work, especially my family, who stayed by my side during difficult moments.

2 Quantum mechanical background

Quantum mechanics is the fundamental theory developed at the beginning of the twentieth century. Up to this day it is considered the most successful theory ever formulated. It describes the behavior of particles at the smallest scales. One of the greatest innovations was the introduction of *spin* [2], an intrinsic form of angular momentum that is a key property of elementary particles. Its discovery marked a significant advancement in the understanding of quantum systems.

In this section we are going to provide a concise overview of the key quantum mechanical concepts and models relevant to the study of spin systems, we will delve into several models used to describe spin interactions in many-body systems: *the Ising model* [13], *the XY model* [15, 16], *the Heisenberg model* [17] and *the Kitaev model* [3]. By understanding these concepts, we gain a background for exploring advanced computational techniques used to calculate various properties of spin systems.

2.1 Spin systems in quantum mechanics

The spin was initially inferred from data acquired by *the Zeeman effect* [28]. This allowed us to explain the splitting of spectral lines observed in the presence of magnetic fields. This phenomenon could not be explained by orbital angular momentum alone. The term suggested a particle's rotation around its own axis, but now it is understood as an abstract quantum property without a classical analog. Spin was later mathematically formalized by Wolfgang Pauli [11]. We define two types of spin, integer and half integer [2]. The former are properties of bosons while the latter are fermions, and they are our main scope in this work.

2.1.1 Pauli matrices and commutation relations

Mathematically, spin description is very similar to the angular momentum but is intrinsic to the particle itself and cannot be defined by any spatial coordinates. However, we can describe operators projecting spin onto the spatial axis. We call these operators *Pauli matrices*.

For spin-$\frac{1}{2}$ particles (like electrons), the Pauli matrices are denoted as σ_x, σ_y and σ_z, which are $2x2$ Hermitian matrices of the form:

$$\sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad \sigma_y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \quad \sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \tag{2.1}$$

The commutation and anticommutation relations are as follows:

$$[\sigma_i, \sigma_j] = 2i\epsilon_{ijk}\sigma_k, \quad \{\sigma_i, \sigma_j\} = 2\delta_{ij}I \tag{2.2}$$

where $i, j, k \in \{x, y, z\}$ and ϵ_{ijk} is *the Levi-Civita symbol* and δ_{ij} is *the Kronecker delta* and I is the 2x2 identity matrix.

The commutation rules of the operators dictate the fundamental relationships between observables. Most importantly, they imply the *Heisenberg uncertainty principle* [29]. For two operators $\hat{A}$ and $\hat{B}$ that do not commute (that is, $[\hat{A}, \hat{B}] = i\hbar\hat{C}$), the

uncertainty principle states that it is impossible to simultaneously know the precise values of both observables:

$$\Delta A \Delta B \geq \frac{\hbar}{2} |\langle \hat{C} \rangle| \tag{2.3}$$

They also determine how operators act on quantum states and how different measurements relate to each other. In case of angular momentum, they ensure that the *eigenvalues* of the spin components are discrete, leading to their quantized nature. Anticommutators on the other hand, are equally important in the context of fermions and *spinors*. They are crucial for ensuring *Pauli exclusion principle* [30], which states that no two fermions can occupy the same quantum state simultaneously.

2.1.2 Spin operators and their eigenvalues

If we scale Pauli matrices by *the reduced Planck's constant* $\hbar$, we get a *spin operators* $\hat{S}_x$, $\hat{S}_y$, $\hat{S}_z$, which correspond to the components of the spin angular momentum:

$$\hat{S}_x = \frac{\hbar}{2}\sigma_x, \quad \hat{S}_y = \frac{\hbar}{2}\sigma_y, \quad \hat{S}_z = \frac{\hbar}{2}\sigma_z \tag{2.4}$$

Although the spin operators do not commute with each other, we can define *the total spin angular momentum operator* $\hat{S}^2$ that will commute with $\hat{S}_z$ [2]:

$$[\hat{S}_x, \hat{S}_y] = i\hbar\hat{S}_z, \quad [\hat{S}_y, \hat{S}_z] = i\hbar\hat{S}_x, \quad [\hat{S}_z, \hat{S}_x] = i\hbar\hat{S}_y \tag{2.5}$$

$$\hat{S}^2 = \hat{S}_x^2 + \hat{S}_y^2 + \hat{S}_z^2, \quad => \quad [\hat{S}_z, \hat{S}^2] = 0 \tag{2.6}$$

This indicates that both quantities can be measured simultaneously, they have a common set of eigenstates and both quantities are conserved. That is why it is a common practice to measure the total spin with its projection along the z-axis. For $\hat{S}_z$, the eigenvalue problem $\hat{S}_z|\psi\rangle = \lambda|\psi\rangle$ yields a measurable quantity $\lambda = \pm\frac{\hbar}{2}$. To describe the system with a single quantum number we denote $m_s = \pm 1$ and this is what we typically refer as *spin-up* or *spin-down*.

Now we can denote spin as another quantum number s that can have arbitrary half-integer values ($s = 0, \frac{1}{2}, 1, \frac{3}{2}, 2, \dots$). Formulating the eigenvalue problem for total spin $\hat{s}^2$ will give:

$$\hat{S}^2|s, m_s\rangle = s(s+1)\hbar^2|s, m_s\rangle \tag{2.7}$$

where for each half-integer value of s, m_s can have a set of values:

$$m_s = -s, -s+1, \dots, s-1, s \tag{2.8}$$

In the considered case of spin particles, we restrict m_s to take only half-integer values ($m_s = \pm\frac{1}{2}, \frac{3}{2}, \frac{5}{2}, \dots$) and $\hat{S}_z|s, m_s\rangle = m_s\hbar|s, m_s\rangle$. This is how we can fully describe the spin system with two quantum numbers.

2.1.3 Magnetic properties of spin

After introducing quantum numbers for the spin of the particle, it is time to briefly discuss other numbers, which are n and L. The former means the energy level of the system, which can be acquired by solving *stationary Schrödinger equation* $\hat{H}|\psi\rangle = E|\psi\rangle$ [31], where $\hat{H}$ is the *Hamiltonian* of the system and energy levels n are its eigenvalues. The number of energy levels corresponds to the dimensions of the Hamiltonian, which is the size of system's *Hilbert space* $\mathcal{H}$.

L is referred as *orbital angular momentum* [2], typically associated with the motion of a particle around a nucleus. It is quantized and characterized by the quantum number l and is dependent on $n \to l = 0, 1, 2, \ldots, n-1, n$. The angular momentum can have different orientations m_l, which gives us $2l+1$ possibilities for arrangement $(m_l = -l, -l+1, \ldots, 0, \ldots, l-1, l)$. Now we have four different quantum numbers that can describe the state of the particle $|\psi\rangle \equiv |n, l, m_l, m_s\rangle$.

All these different configurations of angular momenta lead to the concept of *degeneracy*, which means that different states can share the same eigenvalue. Taking the hydrogen atom as an example, we have a spherical symmetry that leads to different orbital momenta with the same energy (degeneracy in energy levels). Going further, we have orbital spin degeneracy, where we can have different values of m_l, where we can also have different spin orientations.

However, taking into consideration the interaction between the particle's spin and its motion around the nucleus leads to splitting these degeneracies. We call it *spin-orbit coupling*. When we include it, the total angular momentum J becomes the relevant quantum number. J can take values:

$$J = L + S, L + S - 1, \ldots, |L - S| \tag{2.9}$$

These different J states can have slightly different energies due to spin-orbit coupling. This interaction causes a splitting of spectral lines that were previously thought to be degenerate, which we call *the Fine Structure* [14, 1]. However, that energy split is much smaller compared to different levels of quantum number n. Going even further, we can take into consideration the interaction between nuclear magnetic moment and the magnetic field produced by the electron. This leads to a much finer split in the energy spectrum that we call *the hyperfine structure*.

This type of degeneracy arises from the symmetries of the considered system. However, they can be changed with various kinds of perturbations, such as magnetic field B [28, 1]. Each particle with spin also has a magnetic moment μ associated with it. The magnetic moment is a vector quantity proportional to the spin and orbital angular momentum:

$$\mu_L = -\frac{e}{2m_e}\bar{L}, \quad \mu_S = g\mu_B\bar{S}, \quad \mu_B = \frac{e\hbar}{2m_e}, \tag{2.10}$$

where μ_B is *Bohr magneton*, g is *the g-factor*, m_e is the mass of the particle (electron in this example) and $\bar{S}$, $\bar{L}$ are angular momenta vectors.

That leads us to *the Zeeman Effect* [1], which describes the splitting of spectral lines of atoms and molecules when they are placed in the magnetic field, due to the

interaction with magnetic momenta of the particle. The Hamiltonian with substituted μ_L and μ_S takes the form:

$$\hat{H}_{Zeeman} = \frac{e}{2m_e}(\bar{L} + g\bar{S}B), \tag{2.11}$$

where $\bar{B}$ is an external magnetic field. Finding the solutions for the $\hat{H}_{Zeeman}$ we get

$$E = -m_s g \mu_B B z, \tag{2.12}$$

where B_z is the z-axis component of $\bar{B}$. Due to the fact that $m_s = \pm\frac{1}{2}$, we can notice that with increasing B_z the Hyperfine structure splitting gets larger. This shows us that various factors can affect the energy spectra, which is important for next chapters.

2.1.4 Many-body spin systems

Now we move on to discuss the interaction between many spin-$\frac{1}{2}$ particles, which we call *Quantum many-body systems* [32]. Understanding them is one of central challenges in modern physics because they exhibit a variety of phenomena that cannot be explained by individual particles in isolation (like magnetism, superconductivity or phase transitions).

When we close many particles in a single system, they are all described by the single wave function $|\psi_N\rangle$, where N denotes their quantity. The Hilbert space $\mathcal{H}$ becomes the combination of all possible states, so it scales exponentially, leading to $dim(\mathcal{H}) = 2^N$. It means, that describing and simulating such systems becomes too complex very quickly - for $N = 50$, for example, the dimension of $\mathcal{H}$ becomes:

$$\dim(\mathcal{H}_{50}) = 2^{50} \approx 1.13 \times 10^{15}, \tag{2.13}$$

which is going to require more sophisticated methods of formulating such systems. Moreover, the exact numerical simulations for modern hardware become impossible, which indicates the need for different approaches, which will be discussed in future sections.

The Hamiltonian $\hat{H}$ of the quantum many-body system determines its energy levels and its eigenstates, which are equal to $dim(\mathcal{H})$. For a larger number of particles (the order of *Avogadro number*), we enter the regime of *Solid State Physics* and *Band Theory* [33], where the energy levels are so dense that effectively they form a continuous band of possible values. The energy spectrum of different many-body systems is the main scope of this work, where we will analyze its structure - most importantly, *the ground state*, which is the lowest energy eigenstate $|\psi_0\rangle$ with energy E_0 and *excited states* - the higher energy states $|\psi_n\rangle$ and energies $E_n > E_0$. In particular, we take the interest in the first excited state, which will allow us to define *the Energy Gap*:

$$\Delta E = E_1 - E_0 \tag{2.14}$$

The peculiar case is where $\Delta E = 0$. This means that the system exhibits *the Gapless Phase* which indicates high level of degeneracy. If we want to look for such cases, we need to analyze various spin lattice configurations:

- *Chain*, the lattice is formed from many spins aligned in ine dimension,
- *Square*, where spins form a 2D case of lattice chain,
- *Triangular*, each spin has six neighbours and lattice forms equilateral triangles
- *Hexagonal*, the lattice forms a hexagon and spins have three neighbors,
- *Kagome*, the special kind of the lattice where spins form hexagons mixed with triangles and each has four neighbors.

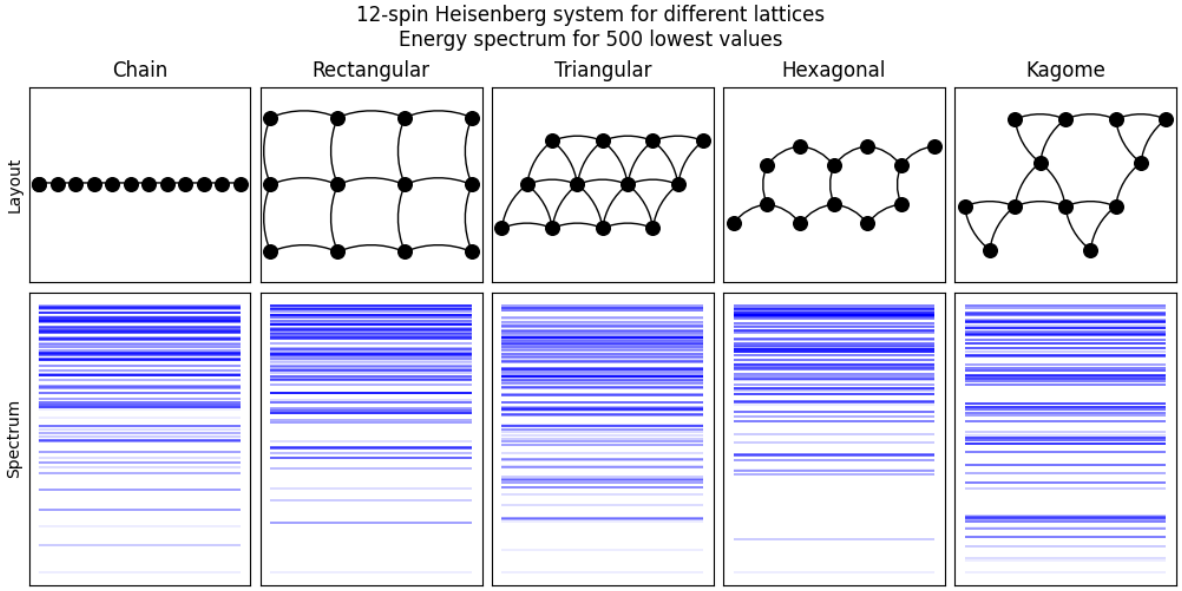

Figure 2.1.1: Comparison of different layouts of 12-spin Heisenberg ($J = 1$) system. The eigenvalues were calculated using *lanczos_ed* from *NetKet* [10]. The lattices infer open boundary conditions, but the spectrum is obtained for periodic boundary conditions for better symmetry. Lattices are printed with *NetKet's Lattice.draw()* method. The whole figure was created in *MatPlotLib* [34].

In figure 2.1.1 we can see how different layouts affect the energy spectrum. The results are calculated from an exact diagonalization of the Heisenberg interaction [35], which will be explained in Section 2.3. The geometry of the lattice can affect the energy gap itself by increasing it or decreasing it up to the point where we get a gapless phase. For example, 1D chains and some *frustrated* 2D lattices may result in gapless spectra, while its scale can depend on the degree of frustration due to competing interactions [36]. Additionally, different geometries promote different types of magnetic order: square lattices tend to exhibit *Neel antiferromagnetic order*, while frustrated lattices such as triangular, hexagonal, and kagome can host disordered states known as *spin liquids* [3]. Lower-dimensional states are also more susceptible to quantum fluctuations.

The geometry of the spin lattice can significantly influence the emergence of topological properties [37]. For example, honeycomb can lead to a band structure with *Dirac points* or topological band gaps, which leads to materials called *topological insulators*.

It can give new symmetries and global properties to the system, making it more robust against local perturbations. This makes them a good material for applications in *quantum computing*, where extending the *coherence time* is crucial for performing effective calculations [38]. These properties can also lead to *topological order*, where the quantum system is characterized by long-range quantum entanglement and cannot be described by local order parameters and excitations can have *anyonic statistics*, meaning that they can have properties that are neither bosonic nor fermionic [37, 12].

2.2 The Ising Model

After discussing the idea and possibilities of studying quantum spin systems, we can now proceed to get a better understanding of some actual models. The most fundamental and essential is *the Ising model* [13], introduced by the German physicist Ernst Ising in 1925. Initially developed to understand the behavior of ferromagnets, but its significance extended beyond that purpose. It also became a cornerstone in the study of phase transitions and critical parameters in condensed matter physics.

The need for the Ising model arises from the complex nature of systems with many interacting components. For example, in a ferromagnetic material, each atom has a magnetic moment, which arises from the behavior of spin. They all interact with their nearest neighbors and collectively lead to the quantity called *magnetization*, where the material as a whole exhibits a magnetic field. These interactions might be perturbed by external fields or changing the system's temperature, where for some critical values, we might observe the phase transition. In this section, we are going to explore the possibilities that arise from formulating *the Ising model*.

2.2.1 The classical Ising model

To start from the fundamentals, we first consider *the classical Ising model* [39]. Classical means that the state is a single static configuration, in contrast to the quantum case, where it would be a superposition of states, described by operators and exhibiting quantum fluctuations [2].

The Hamiltonian for classical model takes the following form:

$$H = -J \sum_{\langle i,j \rangle} \sigma_i \sigma_j, \tag{2.15}$$

It is important to point out, that H in this case is not an operator, in contrast to quantum case - it is a scalar value. J is the interaction constant that determines the behavior of the system. When $J > 0$, the interaction is ferromagnetic, while $J < 0$ indicates the antiferromagnetic system. σ_i represents the single spin, and just as with Hamiltonian, its values are a scalar. Moreover, they can be either $+1$ or -1 with no superposition involved. $\langle i, j \rangle$ in the summation means considering only neighboring spins. For example, on 1D chain it would be site on the left and on the right, while for the 2D case we need to sum up top, bottom, left, and right. The most common lattice to study is the 2D grid due to its ability to exhibit phase transitions. In some resources, one can also find $\frac{J}{2}$ instead of J, where the factor of 2 avoids double-counting

spin pairs. The goal is to find a spin configuration in the ground state, which minimizes the total energy H. For the ferromagnetic case ($J > 0$) this happens, when spins tend to align in the same direction, while for the antiferromagnetic case ($J < 0$) the energy is minimized for spins aligning opposite to each other.

The Hamiltonian can be extended by inducing an external magnetic field h. The hamiltonian gets an additional term accounting for interaction of each spin with the magnetic field:

$$H = -J \sum_{\langle i,j \rangle} \sigma_i \sigma_j - h \sum_i \sigma_i, \tag{2.16}$$

where the term $h \sum_i \sigma_i$ represents the energy contribution from the alignment of spins with the applied field. Another extension would be interaction with more neighbors, we call it *next-nearest neighbors interaction*, where we can include diagonal sites or ones, where the distance to the considered site is 2. The modified Hamiltonian takes the following form:

$$H = -J_1 \sum_{\langle i,j \rangle} \sigma_i \sigma_j - J_2 \sum_{\langle i',j' \rangle} \sigma_{i'} \sigma_{j'} - h \sum_i \sigma_i, \tag{2.17}$$

where J_2 is next-nearest neighbor interaction constant and $\langle i', j' \rangle$ means summation over different sites.

2.2.2 The Metropolis Algorithm

The Metropolis algorithm is one of the most popular *Monte Carlo* methods used in numerical simulations and computational physics [40]. Introduced by Nicholas Metropolis in 1953, the algorithm provides a formula to efficiently generate samples from complex, high-dimensional probability distributions (mostly *Boltzmann distribution* [41]) and is especially useful for exploring the equilibrium properties of systems with many interacting components.

At its core, the Metropolis algorithm is a *Markov Chain Monte Carlo* method. MCMC is a class of algorithms that are used to sample from complex distributions. The idea is to generate samples from a probability distribution by constructing a *Markov chain*, which is a sequence of random variables where the distribution of each variable depends only on the state of the previous one. After many iterations of chain evolution, the target distribution becomes stationary [42].

For the classical Ising model, the goal is to find a configuration of spins that minimizes the energy, starting from a completely random initial lattice grid. The algorithm can be summarized in the following steps:

1) Start with an initial state configuration $\in \{+1, -1\}$ chosen at random. The probability between the ± 1 spin can be arbitrary (probabilities other than $\frac{1}{2}$ can be used to predict the dominance of the spin sign. Initialization can be easily done using *numpy.random* package in *Python* [43].

2) Propose a new configuration by making a small change to the current configuration. In this case, choose an arbitrary site at random and calculate its local

energy E_1 by adding all neighbors considered ($E_1 = -\sum_{\langle i \rangle} \sigma_i$), then flip the sign of the chosen site.

3) Calculate the local energy E_2 again after the sign was flipped and calculate its difference $\Delta E = E_2 - E_1$. Now we make a choice based on the result of ΔE. If $\Delta E \leq 0$, accept the change. If $\Delta E > 0$, calculate the probability of acceptance according to:

$$P_{1\to 2} = \exp\left(-\frac{\Delta E}{k_B T}\right), \tag{2.18}$$

where k_B is *the Boltzmann constant* and T is the temperature of the system, introduced as a simulation parameter. Next, generate a random number $p_i \in (0, 1)$. If $p_i < P_{1\to 2}$, accept the change. If both conditions are not true, reject it.

4) Repeat steps 2) and 3) (a single Monte Carlo step) for a large number of iterations until the system converges to the equilibrium state for the provided temperature T. In practice, for the 150×150 lattice, it can take even a million repetitions.

5) Now, the state is prepared for calculating measurable quantities to study the system.

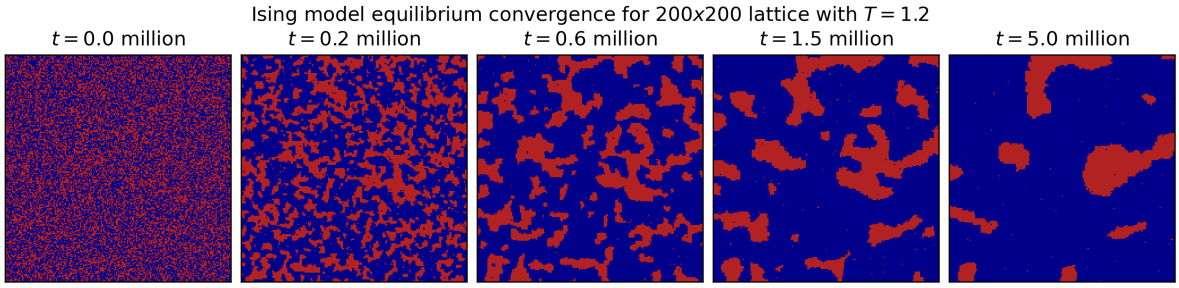

Figure 2.2.1: Ising Model numerical convergence to equilibrium state. The simulation was performed for 200×200 grid with initialization shifted to $p = 0.54$ for 5 million MC steps. Figure created using *MatPlotLib* [34].

The process of converging the system into an equilibrium state is shown in figure 2.2.1. We can see how from initially random configuration spins start to cluster. Smaller clusters tend to fade away, while larger ones smooth out, making them robust for future changes. The Metropolis algorithm will be discussed in a more detailed picture in section 3.4.

2.2.3 The quantum Ising model

Now we are going to extend the Ising model by incorporating quantum mechanical effects, such as superposition and entanglement [14]. This inclusion adds a layer of complexity to the model, making it a better tool for studying phenomena in condensed matter physics.

To start, we introduce the Hamiltonian for quantum model:

$$\hat{H} = -J_1 \sum_{\langle i,j \rangle} \sigma_i^z \sigma_j^z, \tag{2.19}$$

The first important change is $\hat{H}$, which is not a scalar value anymore, but rather an operator defined on the Hilbert space $\mathcal{H}$ consisting of all possible spin configurations, where each spin's basis is $\{|\uparrow\rangle, |\downarrow\rangle\}$. The spin state is now superposition of both spin-up and spin-down:

$$|\sigma_i\rangle = \alpha|\uparrow\rangle + \beta|\downarrow\rangle, \quad |\alpha|^2 + |\beta|^2 = 1, \tag{2.20}$$

If spin state $|\sigma_i\rangle$ is now a two-element vector, we need to act with an operator on it to get an actual value. That is why we now use *the Pauli z-matrix* as σ_i^z that was defined in 2.1, instead of $\sigma_i = \pm 1$. That makes the quantum Ising model a simple spin system, where we take into consideration only the z-axis vector component. For a system with N spins, the Hilbert space $\mathcal{H}$ is 2^N-dimensional and basis vectors are the tensor products of individual spin states. That makes the wave function $|\psi\rangle$ of the whole system take the following form:

$$|\psi\rangle = \sum_{i_1,i_2,\ldots,i_N} c_{i_1,i_2,\ldots,i_N} \bigotimes_{j=0}^{N} |i_j\rangle = \sum_{\{\sigma\}} c_{\{\sigma\}} |\sigma\rangle, \quad |\sigma\rangle = |\sigma_1, \sigma_2, \ldots, \sigma_N\rangle, \tag{2.21}$$

where $|\sigma\rangle$ is a single N-spin system and we make the sumation over whole Hilbert space.

However, the Hamiltonian in the form of eq. 2.19 is trivial, because we only have one operator that commutes with itself. That means we will not see any quantum effects and effectively we can still achieve similar results with the classical approach. One can add the external magnetic field term $-h\sum_i \sigma_i^z$ in the same way as was shown with the classical Ising model. Unfortunetly, that still will not make a significant change, because we still have only one operator σ_z acting on the system. To make the system more complex, we need to act with the magnetic field in different axis, choose x-axis [39]:

$$\hat{H} = -J \sum_{\langle i,j \rangle} \sigma_i^z \sigma_j^z - h \sum_i \sigma_i^x, \tag{2.22}$$

Now we have tho distinct operators σ_i^z and σ_i^x. From eq. 2.2 we know, that they don't commute. This subtle change makes all the difference and now we introduce quantum effects into the system with non-trivial quantum dynamics, which are governed by *time-dependent Shrödinger equation* [31]:

$$i\hbar \frac{\partial |\psi(t)\rangle}{\partial t} = \hat{H}|\psi(t)\rangle, \tag{2.23}$$

The equation allows to find the eigenstates and predict its evolution in time. That gives the possibility to study quantum coherence, superposition and entanglement [30]:

- The presence of the transverse field h allows spins to tunnel between the $+1$ and -1 states. *Quantum tunneling* is a purely quantum mechanical effect and is not present in the classical Ising model,

- The term $-h\sum_i \sigma_i^x$ introduces *quantum fluctuations* into the system. They tend to disrupt the order imposed by spin-spin interactions and can lead to a *quantum phase transition*,
- At zero temperature the system can exhibit a quantum phase transition as the transverse field h is varied,
- At a critical value of transverse field h_c, the system undergoes a transition from ferromagnetic to a paramagnetic phase (where spins are disordered due to quantum fluctuations)

All these phenomena lead to specific name for the system, which is *Transverse-field Ising Model* [44].

2.2.4 Thermodynamic properties

The thermodynamic properties of the Ising model provide deep insights into the macroscopic behavior of the system, especially near critical points where phase transitions occur. In this section, we explore some quantities that are important in studying the Ising model. To start, we need to introduce *the partition function Z*, which is a fundamental concept in statistical mechanics and plays a crucial role in determining the properties of the system in thermal equilibrium [45].

For a system in thermal equilibrium at a temperature T, Z is defined as the sum over all possible microstates of the system, where each microstate i is weighted by *the Boltzmann factor*:

$$Z = \sum_i \exp\left(-\frac{E_i}{k_B T}\right), \tag{2.24}$$

where summation runs over all possible states i, E_i is its energy and T is the absolute temperature of the system.

The partition function can be interpreted as the likelihood of all possible microstates at a given temperature. It also serves as a normalizing factor for *the Boltzmann distribution*, allowing to calculate the probability of the specific microstate for given T:

$$P_i = \frac{\exp\left(-\frac{E_i}{k_B T}\right)}{Z}, \tag{2.25}$$

which explains the formula used to calculate acceptance probability (eq. 2.18) for flipping the spin in Metropolis algorithm studied in section 2.2.2. This explains why the classical Ising model converged to the antiferromagnetic phase for higher temperatures - when it increases, accepting the higher energy configuration becomes more probable. This phenomenon is shown in figure 2.2.2. Thermal fluctuations caused by high acceptance probability lead to the phase transition from the ferromagnetic to the antiferromagnetic phase, where the quantity called *magnetization* drops to near-zero value when the temperature approaches the critical point $T_c = 2.27$.

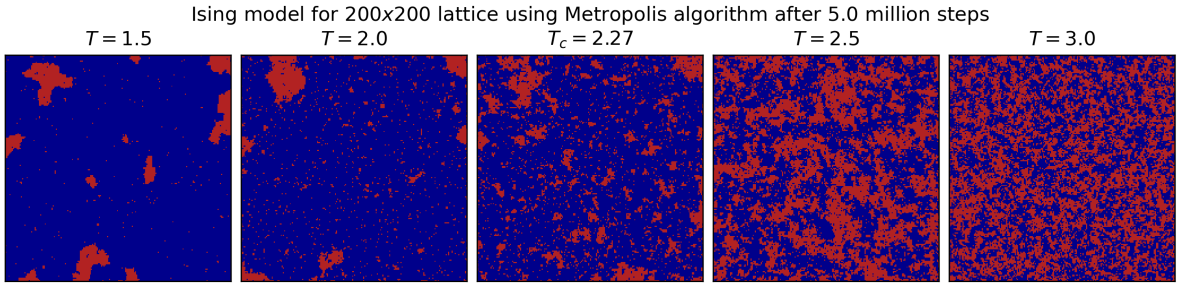

Figure 2.2.2: Comparison of converged 200×200 Ising model lattice for different temperatures T and critical point $T_c = 2.27$, where the phase transition occurs. The equilibrium state was achieved after 5 milion MC steps. Figure created in *MatPlotLib* [34].

The magnetization is an essential property studied in spin systems. Other ones worth mentioning are *the Helmholtz free energy* F, *specific heat* C and *susceptibility* χ. All of them are explicitly dependent on partition function Z.

2.3 Advanced spin models

The Ising model is particularly useful for understanding phase transitions and magnetic ordering in systems of interacting spins. However, its simplicity limits applicability to more complex and realistic simulations. The main contributing factor is that the model considers interactions between spins along z-axis. This limitation is adequate for modeling systems with strong uni-axial anisotropy, but it fails to capture the full complexity in many real materials. The Ising model can accurately describe a single classical phase transition (from ferromagnetic to antiferromagnetic), but it is unable to capture the range of transitions in quantum regime, like spin glasses, spin liquids and topological states.

To address these issues, we need to delve into more sophisticated models, which incorporate more quantum effects, consider interactions in multiple dimensions, and exhibit new types of order. In this section, we are going to discuss more spin models, with each one we add more levels of complexity. First we introduce *the XY model* and then we move on to all types of *Heisenberg models*. The most advanced one, *the Kitaev model*, will be presented at the end of the chapter [46].

2.3.1 The classical XY Model

The first stage of increasing the complexity of the spin models derived so far takes another dimension into account. For this section, we stay in the classical regime, neglecting all quantum effects, and we add another degree of freedom to the spin considered earlier in the classical Ising model in 2.2.1. Now spins have the ability to rotate in any direction (in this configuration, they may be referred as *rotors*). The system describing sites in this way is called *the classic XY model* [18].

In this model, each spin $\bar{S}_i$ is a two-dimensional vector represented in polar coordinates with length1:

$$\bar{S}_i = (\cos\theta_i, \sin\theta_i), \tag{2.26}$$

where θ_i is the angle of the spin $\bar{S}_i$ on the XY-plane and can take any value $\theta_i \in [0, 2\pi)$. The Hamiltonian is a scalar value and can be expressed as:

$$H = -J\sum_{\langle i,j\rangle} \bar{S}_i \cdot \bar{S}_j = -J\sum_{\langle i,j\rangle} \cos(\theta_i - \theta_j), \tag{2.27}$$

J, the coupling constant, is exactly the same, as mentioned in previous sections. $J > 0$ indicates ferromagnetic interaction ($\theta_i = \theta_j$), while $J < 0$ is antiferromagnetic ($\theta_i = \theta_j + \pi$). $\langle i, j\rangle$ means summation over nearest neighbors, and the number of them depends on the lattice structure, just as before. The new phenomenon emerging because of the increased degree of freedom is that now we have the possibility that neighboring rotors do not interact at all. If we consider the case where $\theta_i = \theta_j + \frac{\pi}{2}$, the term $cos(\theta_i - \theta_j)$ is zero.

The introduction of the XY-plane raised the level of symmetry from $\mathbb{Z}_2$ (spin up or down) to $U(1)$, corresponding to the continuous rotation in two dimensions [47]. This leads to entirely new phenomena, which were unavailable for the Ising model. One of them is *the Goldstone Mode*, which are low-energy excitations that occur in a system when a continuous symmetry is spontaneously broken (introduced by Jeffrey Goldstone in the context of *quantum field theory* [48]).

Spontaneous Symmetry Braking occurs when the ground state of a system has symmetry different from its symmetric Hamiltonian. In the context of the XY model we can observe this, when for high-temperature spins point in random directions, which makes it symmetric from any angle (rotational symmetry). For the lowest-energy state, on the other hand, spins prefer an aligned configuration, which makes the system different under rotations. Under these conditions, the small perturbation $\delta\theta_i$ in the angle of arbitrary site $\theta_i = \theta_i^0 + \delta\theta_i$ will interact with neighbors according to:

$$\cos(\theta_i - \theta_j) = \cos(\delta\theta_i - \delta\theta_j) \approx 1 - \frac{(\delta\theta_i - \delta\theta_j)^2}{2}, \tag{2.28}$$

where we used first term of *Taylor expansion*. The Hamiltonian now changes to:

$$H \approx H_0 + \frac{J}{2}\sum_{\langle i,j\rangle} (\delta\theta_i - \delta\theta_j)^2, \tag{2.29}$$

where we can infer that the energy cost due to the perturbation is proportional to the square gradient of the phase angle $\delta\theta$ across the lattice. The excitation caused by these deviations has wave-like properties, which is exactly what we call *the Goldstone mode*. It can be described by *the wave vector* k (inversely proportional to wavelength) and its energy $E(k)$ is proportional to k. However, these modes are unable to change the phase of the system.

The classical model also introduces an entirely new class of transitions called *the Kosterlitz-Thouless* transition [18]. It is a phenomenon that occurs only in 2D systems

with continuous symmetry, fundamentally different from the conventional phase transitions described by *Landau* [49]. It shows the unbinding of topological defects, in this case *vortices and antivortices pairs*. A vortex is a point where the spin direction winds around the point in a circular fashion, while the antivortex has the opposite direction. For high temperatures $T > T_{KT}$, the thermal fluctuations are strong enough to unbind the vortex pairs. There are many of them proliferating in the system independently. For $T < T_{KT}$ vortices are bounded in pairs and they do not disrupt the long-range order.

The XY model can also be simulated by using Metropolis algorithm, just like in section 2.2.2. The only thing that needs to be changed is the formula for calculating ΔE. Because all sites are able to rotate on a plane, we need to upgrade the system from a spin flip to a small random rotation by $\Delta\theta$. Then we can calculate the interaction energy with its neighbors according to:

$$\Delta E = J \sum_{\langle i,j \rangle} \left[\cos(\theta_i - \theta_j) - \cos(\theta_i + \Delta\theta - \theta_j) \right], \tag{2.30}$$

And then we calculate the acceptance probability in the same way as in eq. 2.18.

2.3.2 The quantum XY Model

Now leave the classical model and enter the quantum regime - we consider *the quantum XY model* [44]. Instead of considering the spin as an arrow pointing in any direction on the circle, we act with spin operators on it, particularly the $\hat{S}_x$ and $\hat{S}_y$. As we know from eq. 2.5, these operators do not commute with each other, which means that they will exhibit quantum fluctuations in the system.

The Hamiltonian takes the following form:

$$\hat{H} = -J \sum_{\langle i,j \rangle} \left(\hat{S}_i^x \hat{S}_j^x + \hat{S}_i^y \hat{S}_j^y \right) - h \sum_i \hat{S}_i^z, \tag{2.31}$$

where we now consider the spin as a quantum object with exchange coupling J in two dimensions: $\hat{S}_i^x$ and $\hat{S}_i^y$, acquiring continuous symmetry, which will allow the spins for the possibility of superposition and entanglement, leading to rich quantum phenomena. There is also an additional term with a magnetic field $-h \sum_i \hat{S}_i^z$, which we have already seen before. Now it is particularly important, because it becomes a major contributor in the quantum phase transition and quantum fluctuations in the system.

The quantum phase transition occurs as the magnetic field h is varied and approaches the critical value h_c [50]. Now we consider the quantum system only in the ground state, at $T = 0$. For $h < h_c$ we enter the ferromagnetic phase. In this case, spin-spin interactions J dominate over magnetic field h and spins tend to align with each other on the XY-plane. The spins exhibit long-range order (large distance correlations), leading to non-zero magnetization, which can be considered as the order parameter of the system. The direction of the spins on the plane can be affected by introducing the *anisotropy factor* δ, which forces the spin to prefer specific direction in

the following way:

$$\hat{H} = -J \sum_{\langle i,j \rangle} \left[(1+\delta) \hat{S}_i^x \hat{S}_j^x + (1-\delta) \hat{S}_i^y \hat{S}_j^y \right] - h \sum_i \hat{S}_i^z, \tag{2.32}$$

where terms $1+\delta$ and $1-\delta$ account for changing the anisotropy of the system, favoring one axis over the other.

Now we move to the regime of exceeded critical field $h > h_c$, where spin-spin interaction J is overwhelmed by the force of magnetic field h and spins tend to align along the z-axis, allowing one to minimize the contribution to energy by the term $-h \sum_i \hat{S}_i^z$. The system enters *paramagneticphase*, which might be misleading at the first site, because we said before that the spins are aligned. However, now the term *paramagnetic* refers to the XY-plane, where the spins now exhibit complete disorder. This disordered state means that there is no long-range order in the plane and magnetization in these directions drops to zero. The system no longer exhibits the correlated spin alignment characteristic of the ferromagnetic phase.

At the critical point $h = h_c$ we can observe a phenomenon called *energy gap closing*. As h approaches h_c (from either side), the energy gap between the ground state and the first excited state closes. It indicates a change in nature of the ground state from one that is dominated by spin-spin interactions to one dominated by an external field, and system exhibits critical behavior by divergent correlation lengths.

In analogy to other systems, such as *the Hubbard model* [51, 46], the transitions discussed can be compared to transition from *the Mott insulator* phase into *superfluid* phase [52]. In a Mott insulator, the particles are localized due to strong repulsive interactions. As a result, the system does not conduct electricity. It represents a state of matter where, despite the presence of available energy states that electrons could occupy, the system behaves as an insulator due to strong electron-electron repulsion. *Superfluidity*, in this model, is a phase of matter where the particles have the ability to jump freely between different states because of gapless excitations - they can move freely across the lattice with minimal energy cost.

2.3.3 The Heisenberg Model

The Heisenberg model is a cornerstone in the study of statistical mechanics and plays a crucial role in understanding the magnetic properties of materials [46]. Named after Werner Heisenberg, who introduced the model in 1920. It is another extension of the earlier Ising and XY models discussed above. The major change that has been incorporated is taking the $\hat{S}_z$ component into consideration. The Hamiltonian is formulated in the following way:

$$\hat{H} = -J \sum_{\langle i,j \rangle} \bar{S}_i \cdot \bar{S}_j - h \sum_i S_i^z, \tag{2.33}$$

where J is the interaction constant and, just like in models before, $J > 0$ means ferromagnetic phase and $J < 0$ is antiferromagnetic phase. Now we act with operators on every spatial axis, which makes the spin a vector operator $\bar{S}_i = (S_i^x, S_i^y, S_i^z)$. Analogically, h means a magnetic field pointed in z-axis and is an optional parameter that

introduces more quantum behavior. Lastly, the expression $\bar{S}_i \cdot \bar{S}_j$ is a dot product of two neighboring spins denoted by $\langle i, j \rangle$. If we evaluate the product, we get:

$$\hat{H} = -J \sum_{\langle i,j \rangle} \bar{S}_i \cdot \bar{S}_j = -J \sum_{\langle i,j \rangle} \left(\hat{S}_i^x \hat{S}_j^x + \hat{S}_i^y \hat{S}_j^y + \hat{S}_i^z \hat{S}_j^z \right), \tag{2.34}$$

for simplicity the h term was skipped. Now we have reached the generalized spin model, which has a spherical symmetry. This allows one to observe and simulate whole new range of phenomena like *quantum spin liquids* [53], which are exotic states of matter where spins remain disordered even at zero temperature due to strong quantum fluctuations and they do not exhibit any conventional long-range magnetic order. For Heisenberg model, they might be described on frustraded laticces, like triangular or kagome. Another phenomena emerging from spherical symmetry are *spin waves*, which are collective excitations on a magnetically ordered system, where spins precess coherently around their equilibrium positions - their quantized units are called *magnons* [54].

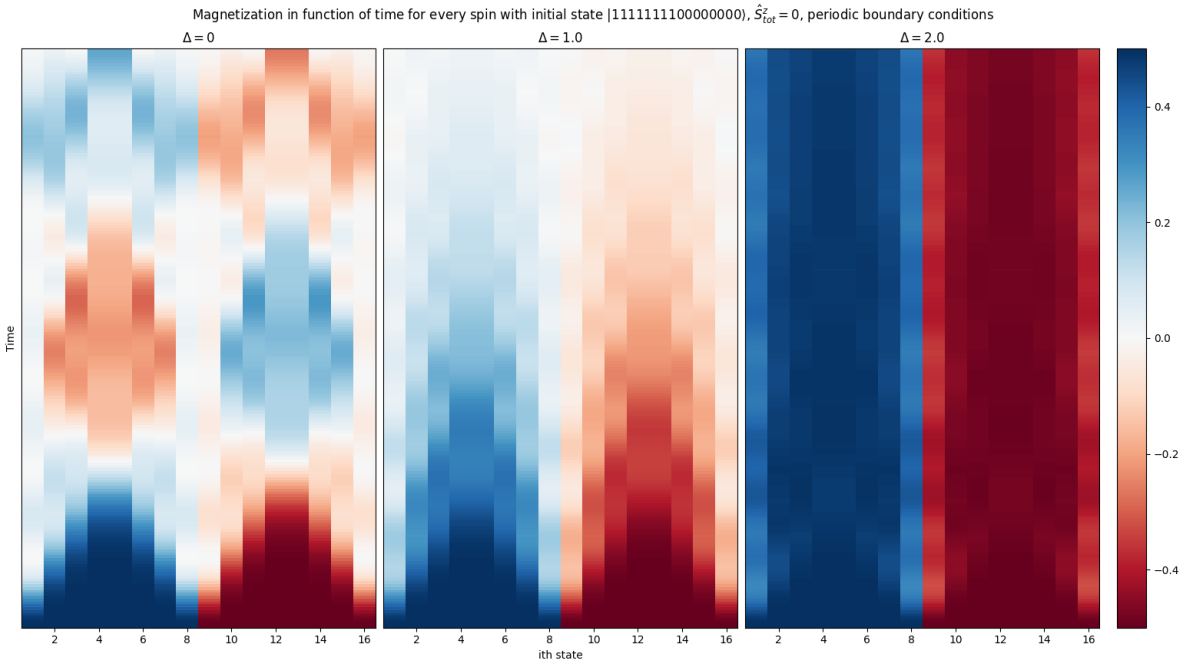

Figure 2.3.1: Time evolution of $\hat{S}_i^z$ on each site for the Heisenberg chain with $L = 14$. We can observe wave-like behavior as the effect of three-dimensional spin interactions. In the figure we see three different simulations for different parameters of Δ (Heisenberg XXZ model). Figure created in *MatPlotLib* and simulation was performed using *NumPy* [34, 43]

Although the Heisenberg model is a generalized model, so far we have considered only the isotropic case, where all spins interact equally, ignoring the directions. To include this, we need to introduce some *anisotropy* into the model, which will allow alteration of the strength of the interaction for different spatial components. All basic alterations of the Heisenberg model are listed below:

- *XXZ Heisenberg Model* - to begin, lets modify only the $\hat{S}_i^z$ component of the spin vector. It is a common practice to alter it first, because σ_z Pauli matrix is composed only of diagonal elements. The hamiltonian has then a following formula:

$$\hat{H} = -J \sum_{\langle i,j \rangle} \left(\hat{S}_i^x \hat{S}_j^x + \hat{S}_i^y \hat{S}_j^y + \Delta \hat{S}_i^z \hat{S}_j^z \right), \tag{2.35}$$

  where Δ is *the anisotropic factor* changing the behavior of the spin in z-axis. Increasing it leads to stronger resistance of the spin to force driving it off the axis. We can see how it affects the system in the figure 2.3.1. For $\Delta = 2$ the spins almost do not change their orientations at all.

- *XYZ Heisenberg Model* takes the concept of anisotropy even further and allows one to modify the strength of interaction for each component of the spin. Mathematically we can present it as follows:

$$\hat{H} = -\sum_{\langle i,j \rangle} \left(J_x \hat{S}_i^x \hat{S}_j^x + J_y \hat{S}_i^y \hat{S}_j^y + J_z \hat{S}_i^z \hat{S}_j^z \right), \tag{2.36}$$

  where $\bar{J} = (J_x, J_y, J_z)$ are interaction factors for corresponding spin operators. This model allows one to set the anisotropy in an arbitrary way. Furthermore, we can see that the configuration $J_1 = (0, 0, 1)$ reduces the Hamiltonian to the XXZ model and $J_2 = (1, 1, 0)$ makes the transition to the Hamiltonian of the XY model.

- *Heisenberg with external magnetic field h* - so far we have considered only transverse magnetic field, now we can extend it to whole new magnetic field vector $\bar{h} = (h_x, h_y, h_z)$:

$$\hat{H} = -J \sum_{\langle i,j \rangle} \bar{S}_i \cdot \bar{S}_j - \sum_i \bar{h} \cdot \bar{S}_i, \tag{2.37}$$

  where evaluated dot product takes the form:

$$\hat{H} = -J \sum_{\langle i,j \rangle} \left(\hat{S}_i^x \hat{S}_j^x + \hat{S}_i^y \hat{S}_j^y + \hat{S}_i^z \hat{S}_j^z \right) - \sum_i \left(h_x \hat{S}_i^x + h_y \hat{S}_i^y + h_z \hat{S}_i^z \right), \tag{2.38}$$

  here, substituting $\bar{h} = (0, 0, 1)$ will reduce the model to transverse field considered earlier. Now, with this model, the magnetization of the system can be induced along a particular direction, competing with the spin-spin interactions.

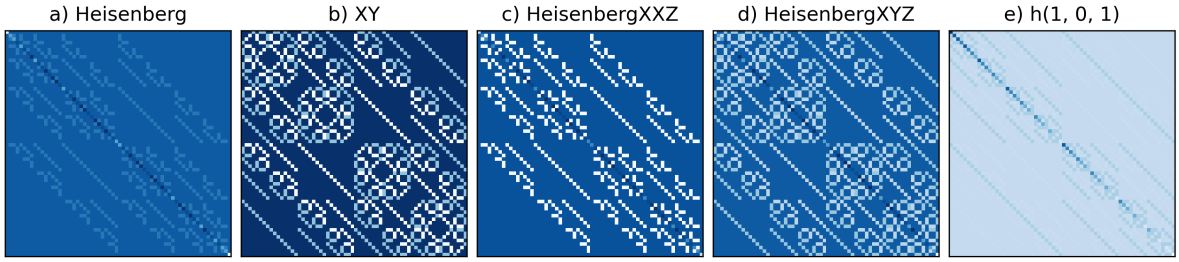

Figure 2.3.2: Comparison between five discussed Hamiltonians. a) the Heisenberg model for $J = 1$, b) the XY model with $J = 1$ and anisotropy $\Delta = (0.8, 0.2)$, c) the Heisenberg XXZ model for $J = 1$ and $\Delta = 0.1$, d) the Heisenberg XYZ model with $J = (0.9, 0.1, 0.1)$ and e) the Heisenberg model with $J = 1$ and magnetic field $h = (-5, 0, -5)$. All Hamiltonians are for the triangular lattice with 6 spins which makes the Hilbert space of size 128×128. All Hamiltonians were constructed using *NetKet* and visualization was made in *MatPlotLib* [10, 34]

All the five Hamiltonians presented so far are shown in the figure 2.3.2. The triangular lattice of the system made many connections between the sites, which made the complex patterns to emerge. We can also notice how acting with external field of the system makes the diagonal elements to stand out due to acting on each site. By comparing figure a) and d) we can see how anisotropy affects the system and far-diagonal Hamiltonian elements start to contribute to the state.

2.4 The Kitaev Model

In this section we raise the bar for the last time with spin model complexity by introducing *the Kitaev model* [3]. Alexei Kitaev formulated it in 2006 and his model is groundbreaking in the field of quantum magnetism. It is also known for its ability to host a quantum spin liquid phase with *non-Abelian anyons*, which are of great interest for topological quantum computation. This model is one of the simplest to be able to solve exactly with a quantum spin liquid ground state. In this phase, the spins are disordered at zero temperature, leading to strong quantum fluctuations and entanglement characterized by topological order. In this section we are going to briefly discuss the Kitaev model and properties emerging with it.

2.4.1 The Kitaev Hamiltonian

The Kitaev model is a highly anisotropic quantum spin model defined on a two-dimensional honeycomb lattice. This constraint emerges from the fact that we have three distinct types of interactions assigned to each neighboring spin. The Honeycomb lattice has a special case geometry, where each spin has exactly three neighbors. We can clearly see this in the figure 2.4.1. The J_x coupling is marked with red color, J_y is blue and J_z is green. In this way we achieve bond-dependent interactions and strong anisotropy of the system, which is crucial for the emergence of exotic phases, such as quantum spin liquids.

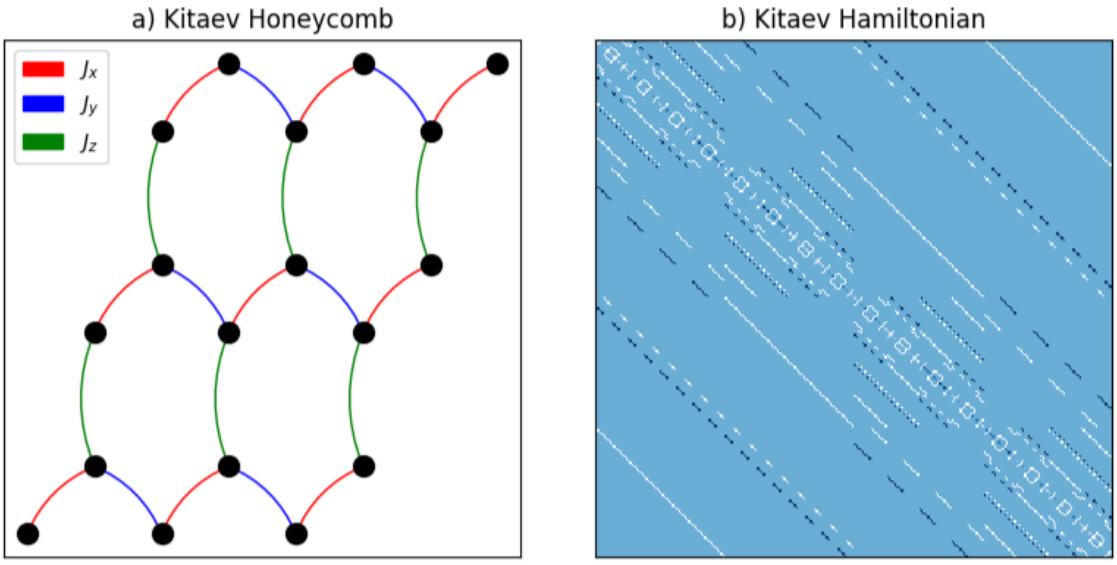

Figure 2.4.1: a) the Kitaev model with sites arranged in a honeycomb lattice. b) the Kitaev Hamiltonian for 8 spins, so the matrix size is 256×256. The Hamiltonian was created using *NetKet* and the figure was created in *MatPlotLib* [10, 34]

The Hamiltonian for the Kitaev model is defined as follows [3]:

$$\hat{H} = -J_x \sum_{\langle i,j \rangle_x} \hat{S}_i^x \hat{S}_i^j - J_y \sum_{\langle i,j \rangle_y} \hat{S}_i^y \hat{S}_j^y - J_z \sum_{\langle i,j \rangle_z} \hat{S}_i^z \hat{S}_j^z, \tag{2.39}$$

at first site we can see that it is similar to the Heisenberg XYZ model in eq. 2.36. However, the key difference is the sumation - $\langle i,j \rangle_\alpha$ means sumation over bond type α, where $\alpha \in \{x, y, z\}$ and means summing over sites with red, blue, and green bonds, respectively. For each of them, we assign anisotropic coupling constants J_x, J_y, J_z and $\hat{S}_i^x$, $\hat{S}_i^y$, $\hat{S}_i^z$ are Pauli matrices projecting spin component onto one of the axes. The solution for the model requires the application of a few transformations. The first of them is to express plaquettes from the honeycomb as a conserved quantity.

The plaquette operators are crucial elements in the Kitaev model, serving as a conserved quantity and contributing to the exact solvability. There are associated with the hexagons of the honeycomb lattice on which the model is defined. Understanding them is an important element for further study of the model.

A single plaquette in the honeycomb is defined by six sites that form a connected bond. For each hexagon p, the plaquette operator W_p is defined as:

$$W_p = S_1^x S_2^y S_3^z S_4^x S_5^y S_6^z \tag{2.40}$$

where indices $1, 2, \ldots, 6$ label the sites forming a single plaquette of the honeycomb. The operator has a few important properties. Firstly, it is defined on a $\mathcal{Z}_2$ gauge field, which means that plaquettes have eigenvalues restricted to two quantities equal to ± 1. This value can be interpreted as a flux of the plaquette. A key property is that they

commute with the Kitaev Hamiltonian, which implies that W_p is a conserved quantity and the flux does not change over time. Another important commutativity rule is with other plaquette, which allows for simultaneous diagonalization of all W_p, making the model exactly solvable. Two commutation rules take a following form:

$$[\hat{H}, \hat{W}_p] = 0; \quad [\hat{W}_p, \hat{W}_{p'}] = 0 \tag{2.41}$$

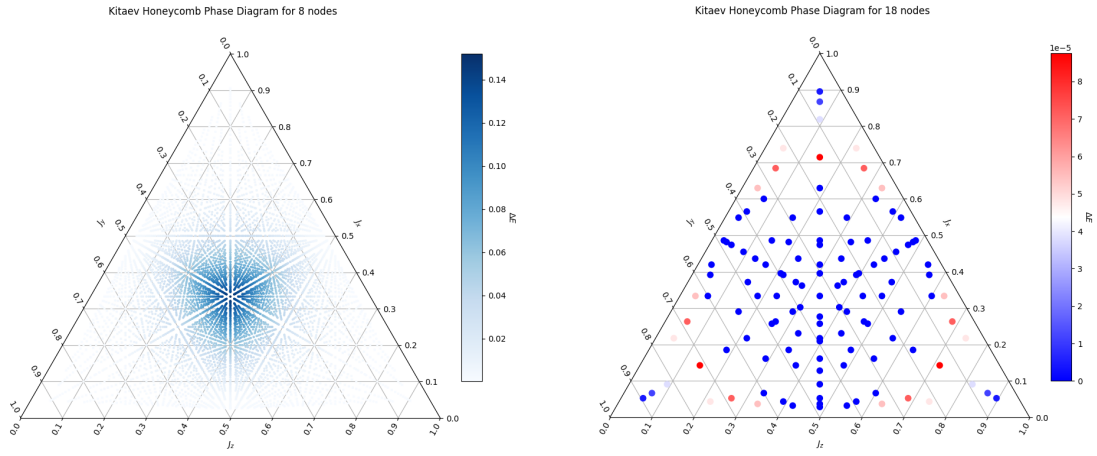

Figure 2.4.2: The exact diagonalization of Kitaev Hamiltonian for various J_α factors. The first figure reflects the mirror image of theoretical diagram, where gapped phase is in the middle. However, when honeycomb was increased from one to 4 cells, it started to approach theoretical values. Diagonalization was performed in *NetKet*, while the figure was created using *MatPlotLib* and *mpltern* [10, 34, 55]

2.4.2 Majorana fermions

The second crucial element is mapping the Kitaev model into Majorana fermions. This transformation allows to recast the degrees of freedom in terms of fermionic operators, which can simplify the model significantly.

A Majorana fermion is a particle that is its own antiparticle. In the context of quantum many-body physics, they are self-conjugate fermionic operators, satisfying the relation $\gamma^\dagger = \gamma$. To map the particle, we need to convert the for of spin operators:

$$\hat{S}_i^x = \frac{i}{2}\gamma_i^0\gamma_i^x, \quad \hat{S}_i^y = \frac{i}{2}\gamma_i^0\gamma_i^y, \quad \hat{S}_i^z = \frac{i}{2}\gamma_i^0\gamma_i^z \tag{2.42}$$

where γ^α refers to a Majorana fermion associated with axes α, the γ^0 is called the dynamical fermion, because this one is responsible for quasiparticle transitions between the plaquettes. The four different Majorana fermions introduce a redundancy that reflects a local $\mathcal{Z}_2$ gauge.

Once the spins are mapped to Majorana fermions, the Kitaev Hamiltonian takes the following form:

$$\hat{H} = \frac{i}{4}\sum_{\langle i,j\rangle_\alpha} J_\alpha u_{ij}^\alpha \gamma_i^0\gamma_j^0; \quad u_{ij}^\alpha = i\gamma_i^\alpha\gamma_j^\alpha \tag{2.43}$$

where u_{ij}^{α} is the link operator for bonds $\langle ij \rangle_{\alpha}$. They take values ± 1 and represent the $\mathcal{Z}_2$ gauge field on the lattice.

The mapping to Majorana fermions makes the Kitaev model exactly solvable. After the transformation, the problem is reduced to diagonalizing a quadratic fermion Hamiltonian, which can be done analytically. The Majorana representation also leads to the discovery of topological phases in the model, where the ground state and excitations are characterized by topological invariants.

After providing the solution, three different anisotropic coupling constants, J_x, J_y, and J_z, lead to the *phase diagram*, which is a triangle with three axes that describes each factor. When all J_i are proportional, the model is in *gapless phase*. However, when one of them differs from the rest, the model enters *the gapped phase*.

3 Machine learning for quantum systems

Machine Learning (ML) is a branch of *artificial intelligence (AI)*, which is fundamentally about enabling computers to learn from experience and data. The goal is to use this information for finding patterns, making predictions and decisions without being explicitly programmed for each task, which is different from traditional programming, where a developer writes instructions for every possible scenario. Instead, ML allows systems to discover them on their own by analyzing data [56].

The idea of self-learning machines dates back to the second half of the twentieth century. The term itself was named by Arthur Samuel in 1958 [57], who was the pioneer in the field back in the day. However, it was not until the past two decades that ML began to be widely used in real applications, largely due to advances in computational power (which doubles every two years according to **Moore's law** [58]).

The popularity of AI has spiked even more in recent years and it has sparked a revolution in computer science. As an effect, we can see that ML and AI algorithms are being implemented in most modern systems, and its possible applications also grew to scale unheard before. The advancements were driven by both theoretical work and practical applications. It started with basic models, such as linear regression or decision trees [59], that were effective for simple problems, but were unable to handle complex high-dimensional data. To tackle this problem, more advanced and sophisticated solutions were developed, like *support vector machines (SVM)*, *random forest* (and the whole branch of *ensemble models*) [60, 61]. However, the greatest breakthrough in the field was the invention of *neural networks* [62], which were inspired by the architecture of the human brain, which caused the creation of a new branch of ML called *Deep Learning* [20]. In essence, they are also neural networks, but in a more advanced version with many layers containing a large number of neurons. It would not be possible without the vast development of *Graphical Processing Units (GPU)* [63]. They turned out to be a very powerful tool for parallel computations, which massively enhanced the training processes in ML.

The last milestone to mention (and probably the biggest one, considering its worldwide popularity) is the development of *large language models (LLM)* [64], which are a class of deep learning models that are trained on vast amounts of text data, often sourced from the Internet. At their core, the mechanism responsible for their success is the type of neural network called *transformers*, developed in 2017 [5]. They leverage mechanisms like self-attention and multihead attention that enables us to capture the context of the text in a mathematical way. As an example, it is necessary to mention the release of *OpenAI's ChatGPT* [4], which is essentially an online virtual assistant, but with massively enhanced capabilities. Now it is possible to ask a bot about anything, from inspiring to organize a trip around Europe or solving a homework task, to explaining complex mathematical concepts, quantum mechanics, and writing a piece of code in any programming language. It also helped me a lot to write this particular work, enhancing my workflow.

In this section, we will walk through the basics of machine learning to find a way to use its potential to enhance the possibilities in quantum many-body systems discussed in the previous chapter.

3.1 The fundamental concepts of ML

The goal of this section is to provide a comprehensive understanding of basic machine learning models, how they scale to more advanced architectures, and explore their use cases in practice. We are going to walk through concepts like overfitting, regularization, linear regression, and classifiers.

3.1.1 Supervised learning

Supervised learning is the most widely used ML paradigm and is the basis for many applications of everyday use (*ChatGPT is a great example*). We will now explore this concept in detail.

It is a type of ML in which the model is trained on a labeled dataset [21]. This means that for each example in the training data, the input comes with a corresponding output, called *target*. The model tries to guess its own output based on the data it got and then makes corrections in its parameters by comparing the result with the actual data label. We will now discuss all the elements needed for successful supervised training of an ML model.

The model itself is the mathematical function implemented in practice as an algorithm that maps inputs, called *features*, to the desired result. It is made up of many parameters that alter the output. At first, they are generated randomly, and during the training they are tweaked accordingly to fit the labeled data as accurately as possible. As input, the shape of the features needs to have been defined, which are the input variables used to make predictions. In the dataset, features are attributes of samples that the model will use to learn patterns. For example, to learn the shape of the function taking the coordinates as input, we define the features to have a size of their dimensionality. The labels are of the size of the output computed by the model, which for a simple function to approximate would be of size 1.

The training process consists of repeated feeding the dataset samples into the model, compering its output with real labels, and then adjusts its parameters to reduce the error rate. The way of adjusting those parameters is a complex procedure and will be discussed in the next sections.

3.1.2 The cost function

So far, we have established that in the training process the output computed by the model needs to be compared with real data to make adjustments. To do so, we need a way to quantify their difference. *The cost function* is designed exactly to solve this kind of problems [21].

The cost function, also known as *loss*, is a mathematical function that measures the error or difference between the predicted output of the model and the actual target values in the training data by mapping the predictions to a nonnegative real number, called *cost*. In essence, it quantifies how well a model is performing. Its value indicates how much the prediction deviates from the ground truth. During training, the model updates its parameters in a specific way to minimize this cost, effectively improving accuracy.

Over the years, many different types of cost function have been developed. The task of choosing the right one for the problem can be non-trivial, because it might strongly depend on the model and the purpose it serves. We now try to briefly describe some of them [65, 20].

- For regression, the goal is to predict a continuous output variable based on the features. The most widely used function for this task is *the mean squared error (MSE)*. Measures the average square difference between the target and prediction:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - \hat{y}_i\right)^2, \tag{3.1}$$

  where n is the number of data points in the dataset, y_i is the true value and $\hat{y}_i$ is a predicted value for the ith element. Squaring ensures that every difference between the point is positive, but gives more weight to distant predicions, making the model sensitive to outliers (elements deviating largely from the mean). To avoid this, the cost can be changed to *mean absolute error (MAE)*, where the term in the sum changes from $\left(y_i - \hat{y}_i\right)^2$ to $|y_i - \hat{y}_i|$. Sometimes, there might be a need to have the cost of the same unit as the targets. For this purpose, *root mean squared error (RMSE)* can be used, which is $RMSE = \sqrt{MSE}$. MSE and RMSE are also called *L1* and *L2* norm, respectively,

- Another type of problems is classification, where the ML model outputs a set of probabilities for the particular class of the target, where the highest one is the prediction and true elements are labeld using integers. As a solution, the common choice is *the Cross-Entropy Loss*:

$$\text{Cross-Entropy Loss} = -\frac{1}{n}\sum_{i=1}^{n}\left[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)\right], \tag{3.2}$$

  again, n is the number of samples, y_i as the true binary label and $\hat{y}_i$ is the probability of the target predicted by the model. However, in this form, the cross-entropy can predict only binary data. To generalize, we need to change the formula to:

$$\text{Cross-Entropy Loss} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{c=1}^{C} y_{i,c}\log(\hat{y}_{i,c}), \tag{3.3}$$

  where C is the number of classes and the summation is over all of them and $y_{i,c}$ indicates the ith value for a particular class c. The cross-entropy loss penalizes the model when it predicts high probabilities for incorrect values; a smaller value indicates that the probabilities are close to the correct class.

These two functions barely scratch the surface of the topic, but it is sufficient to get a basic understanding of why we need them. The choice of the function comes down to the task that it is supposed to do. This is also one of many responsibilities of a machine learning engineer working on optimization of the model.

3.1.3 The gradient descent algorithm

We have learned that ML models in supervised learning predict the outcome, compare it with real values in the dataset and quantifies the difference using a cost function. Based on that information we want to change the parameters of the model to make it perform better. But how do we know, how to change them to make the prediction better? The solution for this problem is provide by *optimization algorithms*, which exactly quantify how much and in which direction the parameters should be updated. The most common one is *the gradient descent* [20].

The goal here is to use the derivative of the cost function with respect to model parameters to acquire its slope (mostly, it has many dimensions, so the term is abstract, but still the analogy is correct). Having that information, we can find the correct direction to minimize the cost (the gradient), and we update the parameters multiplied by some factor, which we call *the learning rate*. Mathematically, we can formulate this as follows:

$$\theta_{i+1} = \theta_i - \eta \nabla f(\theta_i), \quad \nabla f(\theta_i) = \begin{bmatrix} \frac{\partial \theta_i}{\partial p_i^1} \\ \frac{\partial \theta_i}{\partial p_i^2} \\ \vdots \\ \frac{\partial \theta_i}{\partial p_i^n} \end{bmatrix} \tag{3.4}$$

where θ_i represents the model as an ensemble of parameters for iteration i, f is the cost, $\nabla f(\theta_i)$ represents the gradient, and η is the learning rate, also called a *hypermarameter*, as a value provided by the user, not a learnable value. The gradient essentially is a vector pointing in the direction of higher value of the function, which is why we subtract it to get a new state, decreasing the cost. We continue to upgrade the parameters until the cost converges and stops changing. Setting the correct value of η is a non-trivial task. Too large will overshoot the critical points and will not descent. η too low can have to small momentum and get stuck in the local minimum of the loss and will not get any lower.

When the cost function is calculated for every element in the dataset, we call it *batch gradient descent*. Although it is the most accurate approach, for large datasets it becomes too computationally expensive and turns out to be unnecessary. To solve this, for each iteration we choose a small fraction of the dataset picked randomly. It makes the convergence process noisy, but for a sufficient size of the subset, it is not an issue, and it serves a perfect trade-off between computational complexity and accuracy. SGD in this configuration is called *stochastic gradient descent*.

The major disadvantage of using SGD is the choice of η, to properly train the model, it must be chosen wisely to be sure that it will find a global minimum. However, it is often the case that optimization gets stuck at the local minimum or saddle point and stops learning. To solve this problem, another hypermarameter was introduced, mainly *the momentum* γ. It serves as a coefficient of the velocity term added to the formula, which accumulates gradients from previous iterations. This approach allows to accelerate the convergence by maintaining the direction of previous iterations and dampening the oscillations caused by variance in current gradients. It reduces the risk

of getting stuck, but if the model initializes on the plateau, it will still not be able to escape it. As an alternative to SGD, it is common to use the *ADAM algorithm*, which allows dynamic adaptation of learning rates with increased computational cost. However, for the scope of this work, we will remain with SGD to keep it simpler.

3.1.4 The training process

In this section we can revise all new ML concepts discussed earlier and combine them to create a training loop step by step. As an example, we will start with a simple *linear regression* using SGD [59], where we infer $y = ax+b$ and try to adjust parameters a and b to acurately predict the new points based on the noisy dataset that is expected to exhibit a linear-like behavior.

1) First, we need to prepare the data. For our example, we assume a noisy set of points from an unknown linear function. The acquired set needs to be split into three parts - training, validation and test sets. The first will be used for updating the parameters, the second one will monitor the performance of the model during training, and the third will be used at the end to check the final result. Usually training data has $70-80\%$ of datapoint while both validation and test are around 15%.

2) Now we need to initialize the model. In our example, it is $y = ax + b$ and for now a and b are chosen randomly. It is also important to choose the values for the hyperparameters used in optimization.

3) Another step is to choose a correct cost function for the problem discussed. For regression, the common choice is mean squared error (MSE, eq. 3.1), which in this case takes the form:

$$f(a, b) = \frac{1}{n} \sum_{i=1}^{n} \left[y_i - (ax_i + b) \right]^2 , \tag{3.5}$$

where we substituted $\hat{y}_i = ax_i+b$. Now, it is important to formulate the derivative of the cost, which will be calculated repeatedly in SGD:

$$\begin{cases} \dfrac{\partial f(a,b)}{\partial a} = -\dfrac{2}{n} \sum_{i=1}^{n} x_i \left[y_i - (ax_i + b) \right] , \\ \dfrac{\partial f(a,b)}{\partial b} = -\dfrac{2}{n} \sum_{i=1}^{n} \left[y_i - (ax_i + b) \right] , \end{cases} \tag{3.6}$$

as we can see, now we have two distinct derivatives, each for one parameter of the model. Now we can proceed with the training loop.

4) Each iteration of parameter adjustment passes through all datapoints and is called an *epoch*. The number of them can be also defined as another training hyperparameter. Each epoch looks as follows:

(a) shuffle the data points to ensure that the model will not learn in any specific order.

(b) make a prediction (forward pass): $\hat{y}_i = ax_i + b$.

(c) calculate the cost $f(a, b)$

(d) update parameters: $a_{n+1} = a_n - \eta \frac{\partial f(a,b)}{\partial a}$ and $b_{n+1} = b_n - \eta \frac{\partial f(a,b)}{\partial b}$. n refers to the epoch number.

(e) calculate the accuracy of the model by evaluating it on validation data.

5) After training, evaluate model's performance on the test set to obtain an unbiased estimate of its accuracy on unseen data.

Now we have established the core of ML by evaluating the whole learning workflow and we can proceed and discuss more advanced models. In the scope of this work we that will be *neural networks*.

3.2 Feedforward Neural Networks (FNN)

Neural networks are a major class in machine learning, designed to recognize patterns and relationships within the data. to be more precise, they approximate complex functions, classify objects and make predictions based on the input. Their architecture is a composition of layers of interconnected units called *neurons*, which based on their input can activate or not. The information is processed in similar manner as biological neurons in the human brain (although it is incomparably more complex) [20]. They have dominated the world of ML and now are used in a wide range of applications, but were particularly useful in image recognition, speech recognition, natural language processing, physics and finances.

Neural networks are revolutionary and essential in modern machine learning for several reasons. Firstly, we have to emphasize the fact that they are able to capture nonlinear relationships thanks to the use of *activation functions*, which was a disadvantage of many other simpler ML techniques. Neural networks are also very good at feature learning. This ability is very evident for image recognition, where each layer learns to recognize more detailed parts of the image. Due to the complex NN architecture, they can scale for large and high-dimensional data, making them suitable for the most advanced ML tasks. It is also theoretically proved that NN can act as a *universal function approximator*, which means that for a sufficient number of neurons a network can approximate any continuous function to the desired precision, which makes it extremely versatile [66].

Neural networks also allow for very efficient computations during training. Firstly, thanks to their architecture, which mathematically is mostly matrix-vector multiplications. This enables one to use them for *parallel processing*, splitting the computation for many smaller problems, and distributing them to different processing units. That is why we train neural networks on GPUs or TPUs ([63, 67]). Modern GPUs have thousands of CUDA cores, which are small processing units designed for parallel computation used in image and video editing. The TPU (Tensor Processing Unit), on the other hand,

is specifically optimized for tensor operations, allowing to perform large-scale matrix multiplications more efficiently than other computing units.

3.2.1 Perceptron

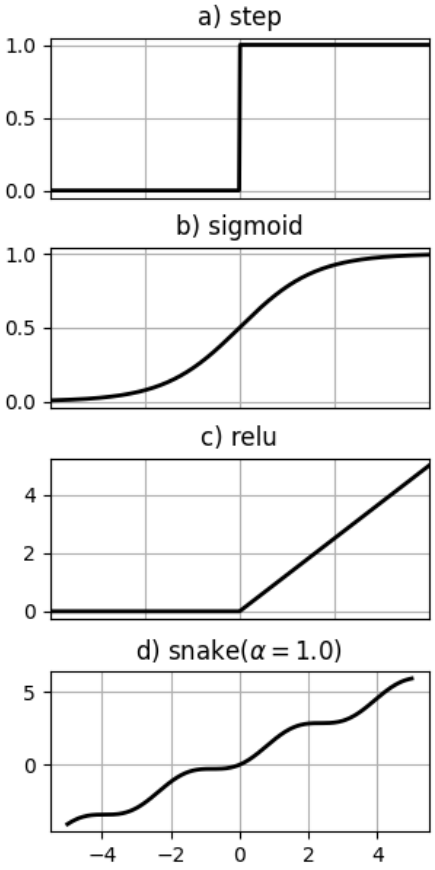

Figure 3.2.1: Comparison between common activation functions. a) step function shown in equation 3.8, b) sigmoid function $f = 1/(1 + exp(-x))$, c) Most common choice for neural networks, ReLU (Rectified Linear Unit) $f = max(0, x)$, d) More specific example, *the Snake Function*, a good activation for approximating rapid sine waves [68] $f = 1 + \sin^2(ax)/a$. The figure was created using MatplotLib and NumPy [34, 43]

We delve now into neural networks from theoretical way. To start, *perceptron* needs to be discussed, which is the simplest type of artificial neural network and serves as the building block for more complex architectures (introduced by Frank Rosenblatt in 1958 as a binary classifier) [22].

The perceptron in its basic configuration simply takes an input of arbitrary size, applies weights to it, sums it up, and adds bias. We can assume that the input is a vector $\bar{x} = [x_1, x_2, \ldots, x_n]$ and each x_i is a feature of the data set (can be a binary number or a continuous variable, the choice is arbitrary). The perceptron must be designed to accept such an input, to do so, its weights must be $\bar{w} = [w_1, w_2, \ldots, w_3]$. They determine the importance of each input in the decision-making process. Initially, they are small random values and gain real meaning after training the model. Mathematically, the perceptron performs a following computation:

$$z = \sum_{i=1}^{n} w_i x_i + b, \tag{3.7}$$

where b as a bias term, which makes a final adjustment to the model by shifting the decision boundary. The last thing to complete the model is to apply an activation function to the outcome. There are many different activations which vary in their form depending on its purpose (for approximations the common choice is *sigmoid*, for example). For our case, we will use *the step function*, as was used in the original paper.

$$step(z) = \begin{cases} 1 & \text{if} \quad z \geq 0, \\ 0 & \text{if} \quad z < 0. \end{cases} \tag{3.8}$$

Some common activation functions and their comparisons are shown in figure 3.2.1. To train a perceptron, we simply repeat the training process discussed in section 3.1.4. In essence, during learning, it tries to find a linear decision boundary that separates the classes in the output space (it is a *hyperplane*).

The model in this configuration is only capable of solving problems that are linearly separable, which makes it unsuitable for practical use. Also, using a step function restricted the output only to a binary number, so it lacks the indication of confidence, which is quantified as the probability of the answer. Although the problem of probability as an output can easily be solved by changing the activation sigmoid, the linearity issue can be solved only by increasing the level of complexity of the perceptron. To do so, the concept of *hidden layers* is introduced, leading to the *multi-layer perceptron*. The MLP is typically what we refer to as a neural network, or a *feed-forward neural network*.

Effectively, adding a hidden layer of size n makes n new perceptrons (each element of the hidden layer acts as an SLP we discussed earlier). We can keep repeating this process to add even more leyers, the choice is arbitrary. The key note is that, at the end, the output must be equal to the dimensionality of the target from the dataset.

For each hidden layer l, we have j perceptrons, and we can now update the equation 3.7 accordingly:

$$x_j^{(l)} = f(z_j^{(l)}), \quad z_j^{(l)} = \sum_{i=1}^{n^{l-1}} w_{ji}^{(l)} x_i^{(l-1)} + b_j^{(l)}, \tag{3.9}$$

where f is the activation, $z_j^{(l)}$ is an output generated by neuron summing the elements from layer $l-1$ and $x_j^{(l)}$ is the output of the network or a potential input for another layer $l+1$.

Modern neural networks can have tens of layers and their size can reach thousands of perceptrons. This complex structure started a whole new branch of ML, called *deep learning* [20].

3.2.2 Backward pass

Now, it is clear, how the data flows through neural network - each perceptrons computes its own values and then passes them further. The process of computing the output of the network is called *the forward pass* [20]. According to section 3.1.4, after acquiring predictions, we must calculate the loss. The cost function itself is not a problem, but things get complicated when the gradients must be retained. This is where the algorithm called *backpropagation* comes into play.

The backpropagation, also known as backward pass, is the process of updating models parameters in response to a cost function calculated after forward pass. The trick is that gradients must be calculated with respect to every parameter in the network, which seems complicated and computationally expensive. However, the algorithm uses *the chain rule* to do this in the most efficient way [69].

The goal of the backward pass is to minimize the loss function $L(\mathbf{W}, \mathbf{b})$, where $\mathbf{W}$ is represented as the matrix with network's weights, while $\mathbf{b}$ stands for the vector with biases. To achieve this, we make use of the chain rule:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}. \tag{3.10}$$

After the forward pass, our results take the form written in equation 3.9. The backward pass itself requires a few steps. Let's assume, that the cost after the forward pass was calculated, the first step of backpropagation is to acquire the derivative of the activation function:

$$\delta_j^{(l)} = \frac{\partial L}{\partial a_j^{(l)}} f'(z_j^{(l)}). \tag{3.11}$$

The quantity $\delta_j^{(l)}$ is called the error signal for the jth neuron in the last layer, $\frac{\partial L}{\partial a_j^{(l)}}$ is the derivative of cost with respect to activation and $f'(z_j^{(l)}$ is the derivative of the activation with respect to the weighted input of the last neuron $z_j^{(l)}$. At this point, the first layer was backpropagated and $\delta_j^{(l)}$ was computed. Now we have many neurons in another layer and we need to compute the same error for each one of them, effectively evaluating the following expression:

$$\delta_j^{(l)} = \left(\sum_{k=1}^{n^{(l+1)}} \delta_k^{(l+1)} w_{kj}^{(l+1)} \right) f'(z_j^{(l)}), \tag{3.12}$$

which now allows to transfer the error values from layer $l+1$ to another layer l (in the backward direction). After computing all necessary $\delta_j^{(l)}$ we can achieve the gradients of the cost:

$$\frac{\partial L}{\partial w_{ji}^{(l)}} = \delta_j^{(l)} a_i^{(l-1)}, \quad \frac{\partial L}{\partial b_j^{(l)}} = \delta_j^{(l)}, \tag{3.13}$$

for weights $w_{ji}^{(l)}$ and biases $b_j^{(l)}$ respectively. Now the final step is to update the parameters an using optimization algorithm, like the gradient descent discussed in previous section 3.1.3, but in this case it is slightly more complicated:

$$w_{ji}^{(l)} := w_{ji}^{(l)} - \eta \frac{\partial L}{\partial w_{ji}^{(l)}}, \quad b_j^{(l)} := b_j^{(l)} - \eta \frac{\partial L}{\partial b_j^{(l)}}, \tag{3.14}$$

where η is the learning rate and $:=$ denotes that we replace previous parameters with new ones, updating them.

That completes the whole backpropagation step. Together with the forward pass, this whole sequence is a single iteration of the optimization, which we refer to as the epoch. The backward pass is probably the most complicated part of network training. Luckily, in practice, it is all automated in ML frameworks (like PyTorch [26]) using a concept called *computational graph*.

3.2.3 Regularization

Knowing exactly what the training of neural nets looks like, we can now refer to another problem - when the network learns the dataset too well. If the training loop has too many epochs, at some point the model starts learning the noise and details in the training data to such an extent that it negatively impacts the model's performance on unseen data and it becomes too complex. This results in excellent performance on the

training data but poor generalization on the test data. We call this problem *overfitting* [20].

The concept of preventing overfitting is called regularization. The main idea is to impose additional constraints on a model's parameters during training, and effectively it limits the model's capacity, encouraging it to learn only the most important patterns and stop focusing on detailed, localized features.

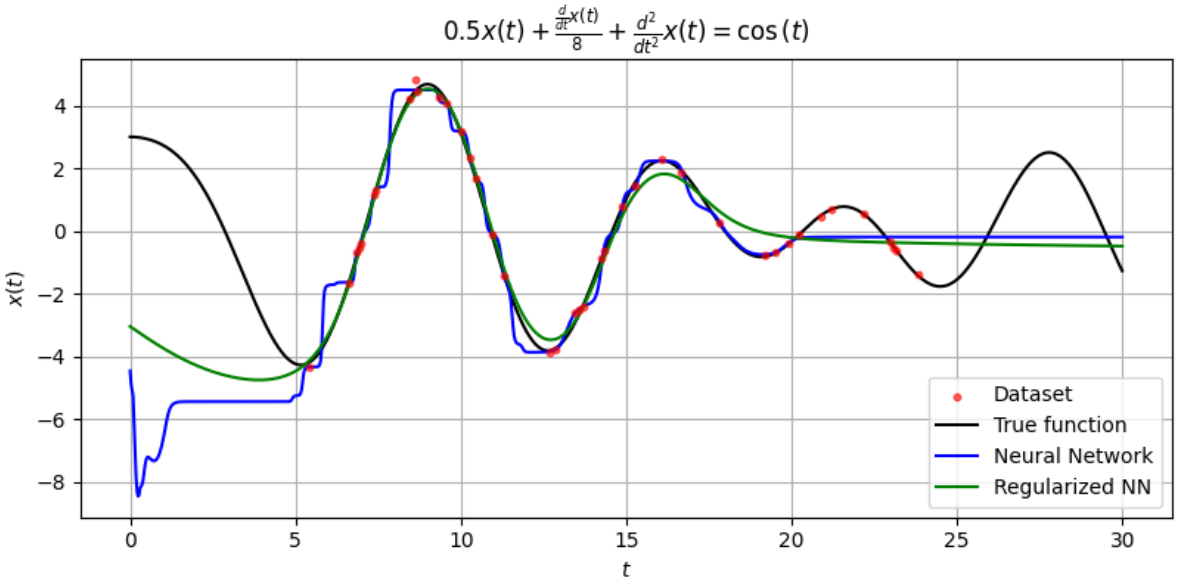

Figure 3.2.2: Regularization in action. Black color shows the true solution to differential equation, red points are the dataset and we have two neural networks learning on it - one without regularization and one with L2 regularization. Both models had three hidden layers with 200 neurons and were trained through 30 thousand epochs. Neural networks were trained in PyTorch and the figure was created in MatPlotLib [26, 34]

In practice, it works by adding a penalty term to the loss function, which has a broader effect during backpropagation and effectively prevents some parameters from increasing to high values. We can write this in general form:

$$L_{reg}(\theta) = L(\theta) + \lambda\Omega(\theta), \tag{3.15}$$

where L_{reg} is the loss after regularization, L is the original loss (like MSE), θ are network's parameters and finally, Ω is the regularization term, added to the loss as penalty. It is also multiplied by a factor λ that controls its strength (and in practice is defined as another hyperparameter).

The most common type of regularization is *L2 Regularizadtion* [70], which adds a penalty proportional to the square of the magnitude of the model parameters. The formula looks as follows:

$$L_{reg}(\theta) = L(\theta) + \frac{\lambda}{2}\sum_{i=1}^{n}\theta_j^2. \tag{3.16}$$

The L2 regularization encourages the model to distribute the weights more evenly across all features, preventing the possibility of dominating by some values. Another technique

is L1 regularization, although it is less common, and some frameworks do not have the built-in option to add them. The formula is very similar to the first case:

$$L_{reg}(\theta) = L(\theta) + \frac{\lambda}{2} \sum_{i=1}^{n} |\theta_j|. \tag{3.17}$$

The only difference here is that we take the magnitude of the weights, not the square. In this form, the L1 regularization encourages sparsity (some parameters are reduced to zero). It can be particularly important for feature selection in high-dimensional data.

The result of regularization was visualized in the figure 3.2.2, where we can see that the model is overfitted to the solution of a differential equation. The regularized model, with green color, did not fit perfectly to the solution, but it is smoothed out, which over all will lead to smaller error on wider range of test samples.

3.3 Restricted Boltzmann Machines (RBM)

As we have discussed in previous section, the FNNs are the simplest and most commonly used types of neural networks, consisting of an input layer and an arbitrary amount of hidden layers. The data is flowing in one direction, giving as an answer of the shape we designed in the first place. The output was compared with ground truth, through supervised learning and, based on the accuracy, the network received a feedback to correct itself. However, while they were a powerful tool for tasks involved labeled data, they had their limitations when it comes to discovering underlying patterns, due to their demanding of large amount of data and the ability to generalize from small datasets was limited (we could even notice this on the toy-example in the figure 3.2.2, where it was not able to reconstruct the pattern well enough). This is where *Restricted Boltzmann Machines (RBM)* come into play, which can be used for an entirely different approach to learning neural networks [71].

RBMs represent a shift from a purely supervised paradigm to a more flexible, unsupervised learning framework, where we feed the data into the network, which are unlabeled. The motivation was to learn hidden structures within the data and generate new samples that are similar to training examples. These capabilities are particularly valuable in situations where the labeled dataset is unable to be obtained, and we want to learn its distribution, rather than mapping inputs to outputs. RBMs accomplish this by implementing an entirely different approach to learning, where they calculate an *energy function* and change its curvature by creating minima for different data samples. The samples that look similar are going to drop into one area of the function, while other samples will have their own distinct minima to converge into. The concept is similar to emerging equilibrium systems in a thermodynamical system, where its energy tends to drop to the lowest value in a probabilistic fashion [12]. The purpose of this section is to discuss and understand this mechanism.

3.3.1 Recurrent neural networks

Due to the fact, that RBMs are a specific type of a larger branch of ML, called *Recurrent Neural Networks (RNN)* [72], it is important to understand this architecture

first. Unlike FNNs, which assume that inputs are independent of each other, RNNs are able to maintain a hidden state that captures information about previous inputs. This unique capability makes them well-suited for tasks where the order and context of inputs plays an important role.

The core ideas behind RNNs are recursion and recurrence, where the output of the network at one time step is fed back into the network as input for the next time step. It allows one to create a loop within the network, maintaining a hidden state that evolves over time while processing a sequential input. This hidden state can be interpreted as the memory of the network, which captures the information about all previous inputs and is able to make a prediction based on all of them relative to each other.

The architecture of RNNs consists of an input layer, a hidden layer with recurrent connections, and an output layer. Its operation can be described mathematically using the recurrence relation for the hidden state. Let us assume an input sequence $\{x_1, x_2, \ldots, x_T\}$. The hidden state h_t update rule for each time step t can be written as follows:

$$h_t = f\left(\mathbf{W}_{xh} x_t + \mathbf{W}_{hh} h_{t-1} + b_h\right), \tag{3.18}$$

where x_t is the input vector at time t, $\mathbf{W}_{xh}$ is the weight matrix connecting the input to the hidden layer and $\mathbf{W}_{hh}$ is the recurrent matrix which connects the hidden state from the previous time step to the stat in the current one. Lastly, we have b_h which is the bias vector for state h. The whole expression is an input for the activation function, which choice can be arbitrary, but it must bring non-linearity into the system.

Equation 3.18 shows how the state h_t depends both on the current input x_t and the precious hidden state h_{t-1}. To get the output at each time step we need to compute another expression:

$$y_t = g\left(\mathbf{W}_{hy} h_t + b_y\right), \tag{3.19}$$

where we have weights and biases for the output layer y_t and g is another activation function. This time it serves a different purpose, because it needs to scale the y_t to be a vector of probabilities, adding to 1 (example function can be *the Softmax*). The computed vector can be used for various purposes, depending on the task it is supposed to solve. For example, in a sequence-to-sequence model, the output at each time step might be combined to form a final prediction. A good example of this is the model translating the sentence from one language to another, where previous words need to be accounted for to capture the context.

Training such a network involves minimizing a loss function that measures the discrepancy between the predicted output y_t and the actual target for the same time step. It also uses a modified version of backward pass to account for recurrent elements. The process is called *backpropagation through time algorithm (BPTT)* [72], which takes into account temporal dependencies while computing gradients.

Despite their strengths, RNNs start to struggle for long sequences, this is mainly because of the *vanishing or exploding* gradient problem, where calculated derivatives stack onto each other and that leads to exponential growth or decay of the derivatives, effectively ruining learning. To address this limitation, more advanced networks were

developed, for example *Long Short-Term Memory (LSTM)* networks [73], which incorporate mechanisms allowing one to control the flow of information to the network. Another branch of RNNs are *Boltzmann machines*, which we will discuss in this chapter. Before that, it is worth presenting another type of RNN, called *Hopfield* networks [74], which laid the ground for more advanced development for BMs.

3.3.2 Hopfield networks

Before we proceed to Restricted Boltzmann Machines, the key model of this work, it is important to introduce a few other concepts to fully grasp the idea. To start, let us discuss the *Hopfield Networks*, introduced by John Hopfield in 1982, which are a class of recurrent neural networks that were revolutionary in the development of associative memory models [74]. These networks are designed to store patterns and retrieve them when presented with noisy versions of the original input.

Associative memory refers to a type of memory system, where information is stored in such a way that complete memory or a pattern based on incomplete or noisy input, in contradiction to other memory systems, which require a specific address or index to retrieve stored data. The techniques used in neural networks give them the ability to efficiently match patterns by converging into the closest possible configuration of remembered neurons.

The Hopfield network consists of a set of neurons $\mathbf{s}_i$, which are binary, either -1 or 1. All neurons are fully connected to each other, and the connection can be an arbitrary value. It is also symmetric, meaning that the weight of the neuron i to j is equal in the opposite way. This is a crucial constraint to ensure that the computation converges to some element in memory instead of looping infinitely.

The operation of the network is governed by an energy function. It is given by:

$$E(\mathbf{s}) = -\frac{1}{2}\sum_{i \neq j} w_{ij} s_i s_j, \tag{3.20}$$

where s_i are values of a neuron i ($s_i \in \{1, -1\}$) and w_{ij} is the weight assigned to the connection between neuron i and j. The energy function serves as a measure of the stability of the network, where lower-energy states are stable. Now, if the network is trained, we give it input data similar to the one on which it was trained on as a configuration of neurons. Then we iteratively start convergence by updating the neurons in the following way:

$$s_i^{new} = \text{sign}\left(\sum_j w_{ij} s_j\right), \tag{3.21}$$

where s_i is changed to either -1 or 1 depending on the sign of the expression in parenthesis, which is a sum over connected neurons multiplied by weight. This constraint let us iteratively minimize the energy, leading to convergence into a configuration which is stored in memory.

Network learning is performed using *the Hebbian rule*, which says that the strength of the connection between two neurons increases if they are activated simultaneously. To describe this mathematically, let us first assume the quantity P, which is the set of all patterns μ that are meant to be stored in the memory of the network. Then, the update rule can be defined as:

$$w_{ij} = \frac{1}{N} \sum_{\mu=1}^{P} s_i^\mu s_j^\mu, \tag{3.22}$$

where N is the number of neurons, s_i^μ and s_j^μ are neurons in a pattern μ. The expression allows encoding and remembering the patterns in the matrix $\mathbf{w}$, effectively creating an *energy landscape*, where the goal is to find the closest minimum while retrieving the pattern. We can see here the connections to statistical physics, where the energy of the system is also minimized to achieve an equilibrium state [12].

While Hopfield networks are a good starting point for learning models based on energy functions, they have limitations that restrict their applicability to more complex tasks. Mostly because of the limited storage capacity due to their binary nature. Over the years more advanced models were developed and one of them we are going to explore right now, discussing Boltzmann machines.

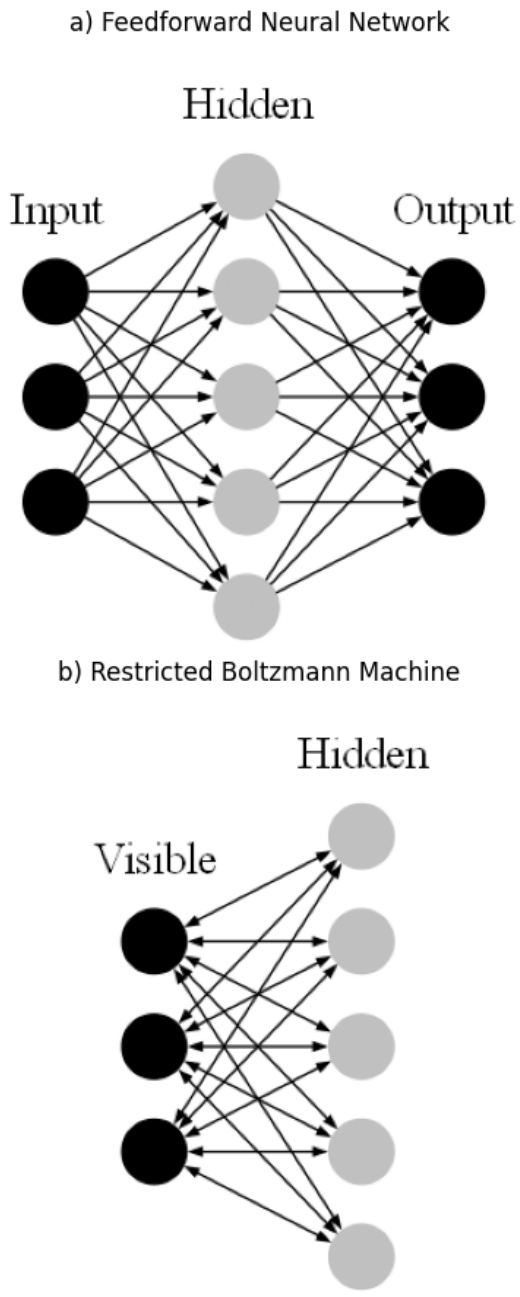

Figure 3.3.1: a) FFN, b) RBM. In RBM we can see the lack of output layer and connections in both ways. The graph visualization was created in *Python Graphviz*

3.3.3 Boltzmann machines

The evolution from Hopfield Networks to Boltzmann Machines, introduced in 1980s by Geoffrey Hinton and Terry Sejnowski as an extension for Hopfield Networks [75], marks significant advancements in the field, especially in the modeling of complex data distributions and generative models. Both types of network are energy-based RNN that utilize concepts from statistical mechanics to store and retrieve information. However, Boltzmann machines introduce key enhancements that make them more powerful by introducing the concept of probability.

The major change compared to Hopfield networks is the transition from fully deterministic binary neurons to stochastic ones, where state of each neuron is determined probabilistically. For given neuron i, the probability that it takes on the state $s_i = 1$ is given by the sigmoid activation function σ:

$$P(s_i = 1) = \sigma\left(\sum_j w_{ij} s_j + b_i\right) = \frac{1}{1 + \exp(-\sum_j w_{ij} s_j - b_i)}. \tag{3.23}$$

Here, W_{ij} are weights for the connection $i - j$, s_j is the state of the neuron j and b_i is the bias for the neuron i (which can increase the network capabilities by an additional probability shift). The use of the sigmoid function ensures that the state is determined in a probabilistic way. It is also worth pointing out that connections in the network are also symmetric, with no difference, depending on direction (the same as in Hopfield Network). The stochastic nature of neurons updates means that the network can explore different states even when starting from the same initial conditions, which was impossible for Hopfield NNs.

Another important thing about Bolztmann machines is that its neurons are divided into visible and hidden layers to increase the capacity of the model. The visible layer is exposed to data, while the hidden layer interacts only when data is being processed by the model, but still all neurons are fully connected. Knowing this, the energy function will take the form:

$$E(\mathbf{v}, \mathbf{h}) = -\sum_{i,j} v_i w_{ij} h_j - \sum_i b_i v_i - \sum_j c_j h_j, \tag{3.24}$$

where $E(\mathbf{v}, \mathbf{h})$ is the energy of the network, v_i is a visible layer neuron i and h_j is a hidden layer neuron j and c_j is its bias. When a new data sample is fed into the model, the energy tends to minimize, like before. However, this time, the uncertainty due to the probabilistic updates of neurons brings a range of probable outputs. That range is described by Boltzmann distribution:

$$P(\mathbf{v}) = -\frac{e^{-E(\mathbf{v},\mathbf{h})}}{Z}, \quad Z = \sum_{v,h} e^{-E(v,h)}, \tag{3.25}$$

where Z is the partition function, which scales the probabilities to sum up to 1. $P(\mathbf{v})$ tells the probability of retrieving state $\mathbf{v}$. It is worth to point out that in parentheses we have only $\mathbf{v}$, due to the fact that $\mathbf{h}$ is hidden. Overall, this equation means that this time we will not get exactly the same answer after the same input, but it can deviate.

To train a Boltzmann Machine, first you need to compute the log-likelihood of the entire dataset $\mathcal{D} = \{\mathbf{v}^{(1)}, \mathbf{v}^{(2)}, \ldots, \mathbf{v}^{(M)}\}$:

$$\mathcal{L}(\mathbf{w}, \mathbf{b}, \mathbf{c}) = \sum_{m=1}^{M} \log P(\mathbf{v}^{(m)}) \tag{3.26}$$

where $\mathcal{L}$ is the log-likelihood and M is the size of the dataset. In order to minimize system energy, you need to maximize $\mathcal{L}$. Its derivative is:

$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = \langle v_i h_j \rangle_{\text{data}} - \langle v_i h_j \rangle_{\text{model}}, \tag{3.27}$$

where$\langle v_i h_j \rangle_{\text{data}}$ is the expectation of the product of v_i and h_j under the distribution of the data (sampling over hidden units). $\langle v_i h_j \rangle_{\text{model}}$ is the expectation of the entire model (freely sampled from both layers).

This is a computationally expensive task because of the partition function Z, which is the sum over all states. It grows exponentially with the number of neurons. To solve this issue, a walkaround had to be developed using stochastic methods, which in this case we call *stochastic divergence* to efficiently approximate gradients [71]. Instead of summing every possible state, we sample it using *Gibbs sampling chain* [76]. The total derivation is beyond the scope of this work, but the equation at the end takes the form:

$$\Delta w_{ij} = \eta \left(\langle v_i h_j \rangle_{\text{data}} - \langle v_i h_j \rangle_{\text{model}} \right), \tag{3.28}$$

where Δw_{ij} is the computed change for considering connection and η is the parameter that allows to scale the process (equivalent of the learning rate from Gradient Descent).

Although contrast divergence significantly decreased computational cost, it was still a demanding process and some further simplifications needed to be made. Another solution was to put a constraint on the Boltzmann machine in such a way that connections were able to form only from the visible layer to the hidden layer (the connections of neurons within the layer are forbidden). It turns out that it massively increased the efficiency compared to the drop in performance, which was insignificant. With this constraint, the model is now called *restricted Boltzmann machine* and its comparison with feedforward Neural Network was shown in figure 3.3.1. The main advantage of RMBs is that they connect in both ways, achieving similar performance to FNN with a smaller number of nodes.

RBMs are going to be the main model used in *Variational Monte Carlo* method, which will be discussed in detail in the next section.

3.4 Variational Monte Carlo

Now we have built a solid foundation to proceed with the key concept of this work. We started from discussing various models used to simulate quantum many-body spin systems, such as Heisenberg or Kitaev. We have established that they are challenging to simulate (to find exact diagonalization of their Hamiltonians) due to their computational cost increasing exponentially with size of the system. The alternative was to explore other methods, and the rapidly growing field of machine learning could show the answer. To find out if that is the case, we had to develop an understanding of basic ML concepts, like neural networks and energy-based optimization.

Variational Monte Carlo is an optimization method used frequently in quantum mechanics, mainly because of the fact, that it has a large potential to find solutions for many-body system with an amount of spins beyond exact methods [7]. In this section

we are going to walk through main concepts of VMC and we will explain why it might one of the most prospering approaches in the field.

3.4.1 Variational principle

The core of the entire VMC algorithm is *the variational principle*, which is a fundamental concept in quantum mechanics, providing a powerful method to approximate the ground state energy of a many-body system, and serves as the theoretical foundation for the field of energy-minimizing techniques [77]. The principle has its roots in the broader concepts of variational methods, which involved finding extreme values of functionals (quantities that depend on functions). In classical mechanics, we can take as an example Hamilton's principle, which stated that the path taken by a physical system is the one that minimizes the *action*, an integral over *Lagrangian* (the difference between kinetic and potential energy) [78].

The variational principle states, that for any trial wave function ψ_t, the expectation value of the Hamiltonian with respect to this function provides an upper bound to the true ground state energy of the system:

$$E_{var} = \frac{\langle \psi_t | \hat{H} | \psi_t \rangle}{\langle \psi_t | \psi_t \rangle} \geq E_0, \tag{3.29}$$

in the equation above, $\hat{H}$ is the Hamiltonian of the system and $|\psi_t\rangle$ refers to the trial wave function which is expected to be as close to the ground state as possible. E_{var} is the variational energy, which is the expectation value of $\hat{H}$ with respect to $|\psi_t\rangle$ and lastly, E_0 is the ground energy, the quantity we want to approximate. Due to the fact that coefficients of $|\psi_t\rangle$ can vary, we divide them by the dot product $\langle \psi_t | \psi_t \rangle$ to ensure that $|\psi_t\rangle$ normalizes and E_{var} reflects a meaningful value.

The Hamiltonian operator $\hat{H}$ is Hermitian, which means that it has real eigenvalues. Knowing this, we infer that its eigenfunctions form a complete orthonormal basis for quantum system state space. Given any function $|\psi_t\rangle$, it can be expanded as the linear combination consisting of eigenfunctions $|\psi_n\rangle$ of the Hamiltonian:

$$|\psi_t\rangle = \sum_n c_n |\psi_n\rangle \quad \rightarrow \quad E_{var} = \frac{\sum_n |c_n|^2 E_n}{\sum_n |c_n|^2}, \tag{3.30}$$

where c_n are expansion coefficients, which allow us to come up with another expression for E_{var}. Since E_n are ordered from smaller to larger one and $\sum_n |c_n|^2 = 1$, the expression of the weighed average for E_{var} cannot be less than E_0.

An important feature of the variational principle is that $|\psi_t\rangle$ can be systematically improved by refining it. It is possible to iteratively reduce E_{var}, bringing it closer to E_0. One of the greatest strengths is that the variational principle allows one to incorporate $|\psi_t\rangle$ as accurate approximations of the ground state with functions simpler than true $|\psi_0\rangle$. Here is where machine learning models come in. Our goal is to use FFN and RBM as $|\psi_t\rangle$ and use them in optimization algorithms to find the lowest possible energy E_{var}.

When we encode a simpler function into $|\psi_t\rangle$ with a set of parameters $\{\alpha\}$ (an RBM, for example [19]), we can calculate the $E_{var}(\alpha)$ in the following way:

$$E_{var}(\alpha) = \frac{\langle\psi_t(\alpha)|\hat{H}|\psi_t(\alpha)\rangle}{\langle\psi_t(\alpha)|\psi_t(\alpha)\rangle}. \tag{3.31}$$

However, calculating exactly the expectation values in this expression is infeasible. To solve this issue, stochastic methods had to be incorporated. We are going to briefly discuss it on the next topic.

3.4.2 Monte Carlo integration

To efficiently approach the problem of computing expectation values for large quantum spaces, we need to introduce the concept of *Monte Carlo integration* [6, 7, 79]. It is a powerful numerical technique used to evaluate high-dimensional integrals, which will be particularly useful in computing the variational energy $E_{var}(\alpha)$.

First, we need to introduce the concept of local energy $E_{loc}(r, \alpha)$, which is crucial in variational method with Monte Carlo sampling. It serves as an intermediate quantity that allows us to efficiently compute the variational energy $E_{var}(\alpha)$. It is defined as the ratio of the Hamiltonian acting on $|\psi_t(r, \alpha)\rangle$:

$$E_{loc}(r, \alpha) = \frac{\hat{H}|\psi_t(r, \alpha)\rangle}{|\psi_t(r, \alpha)\rangle}, \tag{3.32}$$

where $|\psi_t(r, \alpha)\rangle$ is a wave function for a local state r, which in case of considered in this work quantum spin states, is a single spin configuration from the whole space of all possible outcomes and the function $|\psi_t(\alpha)\rangle$ is a combination of all of them, but in a simplified function, like we discussed earlier. In terms of evaluating this more simply, local energy is the energy returned by a single possible spin arrangement from the whole state space.

The basic idea is that to compute $E_{var}(\alpha)$ we need to average the local energy $E_{loc}(r, \alpha)$ over a large number of configurations r (the larger the number, the better approximation of the energy). The configuration are sampled according to the probability distribution $P(r, \alpha)$, which according to basic quantum mechanics is [1]:

$$P(r, \alpha) = \frac{|\psi_t(r, \alpha)|^2}{\int |\psi_t(r, \alpha)|^2 dr}. \tag{3.33}$$

As we have already established, direct integration over all states r is infeasible, so we use Monte Carlo sampling to estimate it. To do it, we come back again to *the Metropolis* algorithm, which we talked about in the context of the classical Ising model in section 2.2.2. The algorithm generates a sequence of configurations $r_1, r_2, \ldots, r_N$ that approximately resembles the distribution $P(r, \alpha)$ (to be precise, this algorithm is called *Markov Chain Monte Carlo (MCMC)* algorithm.

Due to the fact, that we consider now entirely different task than we did with Ising model, steps to be taken in the algorithm take the new form:

1) Start with a random configuration r_0,

2) Find a new configuration r' by slightly perturbing the current one. In the case of a spin system, it is usually a flip of an arbitrary spin (similar to the Ising model).

3) Compare two generated state by computing the acceptance probability:

$$p = \min\left(1, \frac{|\psi_t(r', \alpha)|^2}{|\psi_t(r_t, \alpha)|^2}\right), \tag{3.34}$$

where r_t is the current configuration, before the change was applied,

4) Accept the change by generating a random number $x \in (0; 1)$. If $x \neq p$, accept the change and set r' as the current configuration r_{t+1}, reject it otherwise and come up with new proposed state,

5) Repeat the process for a large number of times (in Variational Monte Carlo optimization this number will be set as a hyperparameter). Store the whole sequence (or count the occurrences of each state), and discard the initial burn-in to make sure the sampling resembles the true probability.

After executing each step, we finish with a sequence of configurations, which will serve as the basis to approximate $E_{var}(\alpha)$ in the following way:

$$E_{var}(\alpha) \approx \frac{1}{N} \sum_{t=1}^{N} E_{loc}(r_t, \alpha), \tag{3.35}$$

where N is the number of sampled configurations after discarding the burn-in part and $E_{loc}(r_t, \alpha)$ is the local energy for configuration t.

The algorithm allows us to efficiently approximate the variational energy for the considered spin system. Now we need to discuss how we can propose a new trial state $|\psi_t\rangle$ in a specific way to minimize energy with gradient descent.

3.4.3 Gradient descent for variational energy

After estimating the energy using Monte Carlo sampling, the next crucial step is to optimize the variational parameters α to minimize $E_{var}(\alpha)$. The common procedure is to use gradient descent and it is the exact method, which we are going to use as an example in this explanation and walk through the necessary step to do it successfully [80].

As we have already known from section 3.1.3, to compute the descent we need to acquire the derivative of the cost function with respect to its every parameter, in this case, the variational energy $E_{var}(\alpha)$ [81]:

$$\frac{\partial E_{\text{var}}(\alpha)}{\partial \alpha_k} = 2\left\langle \left(\hat{H} - E_{\text{var}}(\alpha)\right) \frac{\partial \psi_t(\alpha)}{\partial \alpha_k} \right\rangle, \tag{3.36}$$

where all quantities are known from the previous section, so we will not explain them again here. The same problem still applies with evaluating functions in quantum many-body systems - it cannot be computed exactly. Again, we make use of the Monte Carlo sampling, and the derivative can be estimated with:

$$\frac{\partial E_{\text{var}}(\alpha)}{\partial \alpha_k} \approx 2 \left\langle \frac{E_{\text{loc}}(r,\alpha) - E_{\text{var}}(\alpha)}{\psi_t(r,\alpha)} \frac{\partial \psi_t(r,\alpha)}{\partial \alpha_k} \right\rangle, \tag{3.37}$$

where $\langle \cdot \rangle$ denotes averaging over MC sampling and the term $E_{\text{loc}}(r,\alpha) - E_{\text{var}}(\alpha)$ represents the deviation of the local energy from the average and $\frac{\partial \psi_t(r,\alpha)}{\partial \alpha_k}$ captures the rate of change of all parameters α_k. Now, the last step to compute is to update the parameters. The procedure is very similar to the update rule for the standard NN gradient descent. The main thing is replacing the cost function to energy:

$$\alpha_k^{(t+1)} = \alpha_k^{(t)} - \eta \frac{\partial E_{\text{var}}(\alpha)}{\partial \alpha_k}, \tag{3.38}$$

where t denotes the iteration and again we see the learning rate η adjusted as a hyperparameter during optimization. It is also a not trivial task - too low η will cause the algorithm to work slowly, while too high may deviate too much or even diverge entirely. To balance this issue, adaptive learning rates or momentum-based methods can be employed. However, this is beyond the scope of this work.

Now we have all necessary elements of the algorithm, so let us briefly discuss the whole process. To begin optimization, the variational parameters α are initiated as small random values. Then we proceed with Monte Carlo sampling with the Metropolis algorithm and acquire configurations $\{r_i\}$, which let us approximate the probability distribution $P(r,\alpha)$. Then we combine both elements and achieve a range of local energy values $E_{loc}(r_i,\alpha)$ that we then average to the variational energy $E_{var}(\alpha)$. The most complex part of the algorithm is to calculate the gradients of this energy with respect to every variational parameter α, which is also approximated by Monte Carlo sampling. The last step is to apply a gradient descent update rule to update α. This process is repeated until the convergence is evident.

Although this technique is very powerful, the exponential complexity of approximating the system has one major caveat. The update rule has no sense of the direction over whole state space and moves chaotically through energy landscape. To solve this, one last quantity needs to be provided in the algorithm, which is exactly what we are going to discuss in the next section.

3.4.4 Stochastic reconfiguration

The concept meant to enhance the optimization with gradient descent is *stochastic reconfiguration (SR)*, also known as *natural gradient method* [23]. SR improves upon standard gradient descent by taking into account the geometry of the parameter space, making it more effective for navigating complex energy landscapes in quantum many-body systems. The technique turned out to be successful and is now widely adopted because it provides a more robust way of optimization, especially with high-dimensional parameter space in the trial wave function.

The problem with parameter update in gradient descent from equation 3.38 is that it treats all directions in state space equally, without considering the underlying geometry. As a result, gradient descent can become inefficient, especially if the landscape is high-dimensional and complex. The reason behind the success of SR is the use of *the quantum geometric tensor (QGT)*, which provides a natural metric on the parameter space of the variational wave function. It characterizes the geometry by measuring the overlap (dot product, effectively) between the derivatives of the wave function with respect to different parameters [24]:

$$S_{kl} = \langle \partial_k \psi | \partial_l \psi \rangle - \langle \partial_k \psi | \psi \rangle \langle \psi | \partial_l \psi \rangle, \tag{3.39}$$

where, for simplicity, we denoted $\partial_k = \frac{\partial \ln}{\partial \alpha_k}$ and $\partial_l = \frac{\partial \ln}{\partial \alpha_l}$ and k with l both denote parameters of the trial wave function, but here we want to emphasize that it is relative to different ones (also, ln means, that we are taking the logarithmic derivative). QGT has both real and imaginary parts, where the imaginary part is called the *Berry curvature*, which also accounts for the geometric phase of the quantum state of the tensor. However, for optimization purposes, the focus is typically on the real part (it is more important when considering the curvature). In practice, the matrix is also approximated with Monte Carlo from probability distribution $P(r, \alpha)$:

$$\begin{aligned} S_{kl} \approx \frac{1}{N} \sum_{i=1}^{N} \left(\frac{1}{\psi_t(r,\alpha)} \frac{\partial \psi_t(r,\alpha)}{\partial \alpha_k} \right) \left(\frac{1}{\psi_t(r,\alpha)} \frac{\partial \psi_t(r,\alpha)}{\partial \alpha_l} \right) - \\ - \left(\frac{1}{N} \sum_{i=1}^{N} \frac{1}{\psi_t(r,\alpha)} \frac{\partial \psi_t(r,\alpha)}{\partial \alpha_k} \right) \left(\frac{1}{N} \sum_{i=1}^{N} \frac{1}{\psi_t(r,\alpha)} \frac{\partial \psi_t(r,\alpha)}{\partial \alpha_l} \right), \end{aligned} \tag{3.40}$$

where now we have evaluated the logarithmic derivative $\frac{\partial \ln \psi_t(r,\alpha)}{\partial \alpha_k} = \frac{1}{\psi_t(r,\alpha)} \frac{\partial \psi_t(r,\alpha)}{\partial \alpha_k}$. In optimization, the inverse of QGT needs to be calculated. To make this task easier, it is a convenient way to add a *diagonal shift* into the form of the matrix, as a form of regularization (set up as another hyperparameter). Applying it, the QGT takes the form:

$$S_{ij}^{\lambda} = S_{ik} + \lambda \mathbb{I}, \tag{3.41}$$

where $\lambda > 0$ denotes the diagonal shift. After applying it to regularize the stochastic gradient descent, the equation 3.38 transforms into:

$$\alpha_k^{(t+1)} = \alpha_k^{(t)} - \nu \sum_l (S^{\lambda})_{kl}^{-1} \mathcal{F}_l \tag{3.42}$$

and, from now on, $\mathcal{F}_k = \frac{\partial E_{\text{var}}(\alpha)}{\partial \alpha_k}$ we will denote as *the Force vector*, sticking to convenient terms. SR with QGT is a sophisticated optimization method leveraging the natural geometry of the quantum state space to achieve more efficient and robust variational Monte Carlo. This approach enhances the stability and efficiency of the process, making it particularly valuable for the scope of this work. Now, we have gained a comprehensive understanding of the whole algorithm, and we can now proceed to mathematically evaluate models for the trial wave functions.

3.4.5 Wave function Ansatz

Knowing the form of regularized VMC, we can now discuss in detail what can serve as the model for the optimization to begin with. As we can infer so far, providing the exact wave function $|\psi\rangle$ to solve the diagonalization problem $\hat{H}|\psi\rangle = E|\psi\rangle$ is not an option, and the reason is still the same - a too vast and exponentially complex Hilbert space $\mathcal{H}$ [31].

It turns out that there is a prominent potential to do the most straightforward thing possible - to guess, or speaking more scientifically, to come up with *an Ansatz*, which from German simply means '*an educated guess*'. The idea in VMC is to replace the true wave function $|\psi\rangle$ with the trial one $|\psi_t\rangle$, which must be of the simplest form, to make all necessary computations feasible. From now on, we will also refer to $|\psi_t\rangle$ as Ansatz.

The wave function Ansatz must be parameterized by a set of variational parameters α and there are three main motivations to make it functional and usable. First, the crucial condition must be computational feasibility, so the number of α needs to drop significantly compared to the true function. Second, it needs to be complex enough to capture significant physical insights about the system, such as correlations between electrons interacting with other electrons, for example. Lastly, the key thing is to be compatible with the variational principle (equation 3.29), which means that it needs to take such a form that for a certain combination of α it will return the lowest possible value for energy estimation, and it is not a trivial task for large systems.

One of the primary and simplest forms of a trial wave function is *the Jastrow Ansatz* [82]. It provides a powerful way to incorporate correlations between particles and significantly reduce the parameter space. In this Ansatz, the reference wave function ψ_{ref} is used and is enhanced by the coefficients called Jastrow factor. Its explicit form is:

$$\psi_{\text{Jastrow}}(r) = \exp\left(\sum_{i,j} u(r_{ij})\right) \psi_{\text{ref}}(r), \tag{3.43}$$

where $r = \{r_1, r_2, \ldots, r_N\}$ represents the coordinates of all N particles in the system and ψ_{ref} is the reference wave function, which for spin systems usually takes the form of the product state $\psi_{\text{ref}}(\sigma) = \prod_i \phi(\sigma_i)$ (which might be all spins up, for example). The key term responsible for the success of Ansatz is the Jastrow term $u(r_{ij})$, which introduces a correlation between particles i and j (for spin systems, it can be denoted as $u_{ij}\sigma_i\sigma_j$ where it is a factor describing the interaction). Its choice also depends on the purpose it is supposed to serve, and these are the parameters that are adjusted during VMC optimization.

Although the Jastrow ansatz provides an intuitive example for presenting the core idea, it is primarily limited to two-body correlations and may struggle with capturing complex, long-range entanglements. That means that further development for alternatives needed to be taken, and that is exactly what we will now focus on.

To look for an appropriate trial wave function, we now point our attention towards the machine learning approaches. Let us begin with its basic form, which is a Restricted Boltzmann Machine. It offers a more flexible and expressive framework for the approx-

imation of quantum states, particularly in systems with strong correlations [19]. An RBM is a type of stochastic neural network that consists of a visible and hidden layer, as previously explained in 3.3.3. The former represents the physical configuration of the quantum state, so the latter is supposed to capture more insight into the system. In contrast to deterministic Jastrow Ansatz, RBM uses a probabilistic approach to model the wave function. It can be written in the following way:

$$\psi_{\mathrm{RBM}}(r,\theta) = \sum_{\mathbf{h}} \exp\left(\sum_i a_i r_i + \sum_j b_j h_j + \sum_{i,j} W_{ij} r_i h_j\right). \tag{3.44}$$

In the expression, $r = (r_1, r_2, \ldots, r_N)$ are the visible units corresponding to the physical configuration (like spins on a lattice). $h = (h_1, h_2, \ldots, h_N)$ references to the hidden units, which store the information about the correlations in the system. Lastly, we have the entire set of parameters $\theta = \{a_i, b_j, W_{ij}\}$, which are visible biases, hidden biases, and the matrix that contains weights for the connections between all neurons. The summation over the hidden units $\mathbf{h}$ allows the RBM to learn information over many possible hidden configurations, capturing both local and non-local correlations. The expression for an RBM can be evaluated further due to the fact, that we do not have any additional layers:

$$\psi_{\mathrm{RBM}}(r,\theta) = \exp\left(\sum_i a_i r_i\right) \prod_j 2\cosh\left(b_j + \sum_i W_{ij} r_i\right). \tag{3.45}$$

where now we can see RBM as a product of factors between visible and hidden units. Hyperbolic cosine $cosh(x)$ is effective in capturing the effect of hidden unit activations. Is is also an additional non-linearity allowing higher capacity compared to Jastrow. RBMs can be easily scaled to larger systems by increasing the number of hidden units, allowing for a more detailed representation of the quantum state.

The transition from the Jastrow Ansatz to RBMs represents a significant advancement in the field. However, as the complexity and dimensionality of quantum systems increase, there might be a need for more advanced models that will capture even more insights into the system. We arrive at the intersection of RBMs and feedforward neural networks by introducing many hidden layers into the model to enhance the representational power and this type of NN is called a *Deep Boltzmann Machine (DBM)* [83]. Their description can be presented as generalized version of equation 3.44:

$$\psi_{\mathrm{DBM}}(r,\theta) = \sum_{\{\mathbf{h}^{(l)}\}_{l=1}^{L}} \exp\left(\sum_i a_i r_i + \sum_{l=1}^{L}\sum_j b_j^{(l)} h_j^{(l)} + \sum_{l=0}^{L-1}\sum_{i,j} W_{ij}^{(l)} h_i^{(l)} h_j^{(l+1)}\right), \tag{3.46}$$

where most parameters are the same as before, but now we have additional summations over all hidden layers labeled as l and we have L of them. The visible units are directly connected to the first hidden layer, and each subsequent layer is connected to another repeatedly L times. The final output is obtained by summing over all possible configuration of the hidden units, increasing the capabilities of representing the wave function. The deep structure of DBM enables hierarchical learning, where each layer captures

insights of the previous one, which has potential to discover more hidden information. Training DBMs involves optimizing the increased set of parameters $\theta = \{a_i, b_j^{(l)}, W_{ij}^{(l)}\}$, where (l) denotes hidden layers, so we can infer that they become much more computationally expensive compared to RBM. To end the topic, we can briefly discuss a few other types of neural quantum states. The whole field is developing rapidly, with many papers being published every year. Three of them are listed below:

- *Autoregressive Neural Networks (ARNN)* [84]: a class of NN that model the probability distribution of a quantum state in a sequential manner. The wave function is decomposed into a product of conditional probabilities, where each probability is conditioned on preceding variables in a given order. This approach allows to model complex correlations as sequential decomposition. ARNNs allow for exact sampling, which eliminates the need for approximations using MC methods.

- *Convolutional Neural Networks (CNN)* [85]: their abilities were proven particularly in computer vision. The idea is to use the convolution kernel, which makes it suitable for quantum systems, where local correlations are dominant. The CNN Ansatz for quantum states is recommended for two-dimensional lattice systems, where spatial local structures play a crucial role.

- *Graph Neural Networks (GNN)* [86]: this type of NN is specifically designed to work on graph-structured data, which makes it a good choice for various lattice layouts - in general they are a well-suited option for systems with complex connectivity. They operate by iteratively exchanging parameters between neighboring nodes, passing information one to another, which can be used to capture both local and non-local correlations. That makes them also a good choice for states with complex entanglements and frustrated states.

- *Psiformers (Transformer-Based Neural Networks for Quantum States)* [87]: originally developed for natural language processing, revolutionary in the field. Here, the idea is to transform transformers into psiformers by leveraging self-attention mechanisms, followed by a feedforward layer to compute the final wave function amplitude. The attention mechanism allows transformers to dynamically weigh the importance of different parts of the input data, making them well-suited for complex, long range correlations in quantum systems.

3.4.6 VMC algorithm overview

Now we have gained a comprehensive understanding of every detail of variational Monte Carlo to be able to perform a successful ground state optimization. However, as we can see, this process is a non-trivial task to utilize with many intersecting modifications and different options to choose, depending on the purpose. The purpose of this part is to briefly review the steps and choices that must be taken to complete a well-designed VMC algorithm.

1) The first thing to do is selecting the appropriate trial wave function $\psi_t(r, \theta)$. It is important to choose a correct Ansatz for the quantum system. Otherwise it will

not converge to the true ground state. The main scope of this work is to utilize various ML models to find the ground states of different spin systems, so we leave as an example the Restricted Boltzmann Machine.

2) Second step in initialization is to choose all hyperparameters, which are: learning rate η, diagonal shift λ, number of MC samples to compute average values n and t, which is the number of iterations.

3) As the Ansatz was chosen and initialized, it is the time to perform first Monte Carlo sampling based on it. The most popular choice of algorithm to sample is the Metropolis algorithm, which was comprehensively presented. Here, we make use of first hyperparameter - n, the number of samples.

4) Having acquired an acceptable amount of samples, we act with the Hamiltonian on it to get many local energies, which will be further evaluated to approximate the expected energy value for the current Ansatz. It is also important to normalizing this value by the wave function itself.

5) In this step we evaluate the gradients with respect to all parameters θ of the model. Combining this with all MC samples the derivative of variational energy can be estimated, the goal is to update the parameters θ in a way to make energy decrease its value in next iteration. This method suggests the estimated direction to do so.

6) As a form of regularization, it is recommended to perform a stochastic reconfiguration to further enhance the process of updating the parameters. To do so, the quantum geometric tensor is estimated, which allows to point more accurately the direction of global minimum in the energy landscape. The inverse of it is needed while updating - to optimize this task, we add a diagonal shift λ to it, another hyperparameter.

7) The last step of the iteration, updating θ. Everything computed in previous steps cross in this process. In one equation (3.38) we combine the derivative, quantum geometric tensor, stochastic reconfiguration. All of this is scaled by another hyper parameter, the learning rate η.

8) Repeat steps 3-7 t amount of times, like was declared at initialization. It is recommended to store estimated energy for every iteration.

9) Evaluate the algorithm by plotting the energy values. Strong convergence suggests that everything was successful. Otherwise, some major changes of hyperparameters needs to be done, or the Ansatz needs to be replaced.

3.4.7 Penalty-based VMC for excited states

The whole subject of the ground state is now completed, and the last subsection is about estimating excited states. The choice for this work was *Penalty-based excited-state variational Monte Carlo algorithm* [24, 88]. It enhances the traditional VMC

approach to converge into excited states of the quantum system, which is crucial for understanding a wider spectrum of quantum properties, such as dynamics or response functions. In this section, we provide a detailed overview of the algorithm, focusing on the modifications to the standard VMC framework.

The approach effectively incorporates an additional term to the Hamiltonian $\hat{H}$ of the system by penalizing previously computed states, where now the new excited state is its ground state. We can define this as follows.

$$\hat{H}_n = \hat{H}_0 + \sum_{j=0}^{n-1} \beta_j \frac{|\psi_j\rangle\langle\psi_j|}{\langle\psi_j|\psi_j\rangle}, \tag{3.47}$$

where $\hat{H}_n$ is the Hamiltonian for nth excited state, and we sum over all previously computed states j from zero to $n-1$. The wave function $|\psi_j\rangle$ is an Ansatz optimized for the state j and β_j was the penalty coefficient used to calculate it. The value β effectively pushes the state up in the spectrum, forcing it to find a higher local minimum of the energy landscape.

However, as we mentioned before, this is what happens to the Hamiltonian of the system effectively. The core of the computation lies in redefining the energy estimation formula:

$$\tilde{E}_n(W) = \frac{\langle W|\hat{H}|W\rangle}{\langle W|W\rangle} + \sum_{j=0}^{n-1} \beta_j \frac{\langle W|\psi_j\rangle\langle\psi_j|W\rangle}{\langle W|W\rangle\langle\psi_j|\psi_j\rangle}, \tag{3.48}$$

where $\tilde{E}_n(W)$ is a redefined energy function and W represents the parameters of the currently optimized Ansatz. The geometric quantum tensor remains in the same form, while another major change is to redefine the force vector $\mathcal{F}_n$, which was the derivative with respect to all parameters W. By recalling the form of the primary derivative in equation 3.37 and rewriting in with the convention of the paper [24]:

$$\mathcal{F}_i = \frac{\partial E}{\partial W_i} = \langle E_{\text{local}} \mathcal{O}_i^\dagger \rangle_\sigma - \langle E_{\text{local}} \rangle_\sigma \langle \mathcal{O}_i^\dagger \rangle_\sigma, \tag{3.49}$$

where averages $\langle\cdot\rangle_\sigma$ are computed over samples σ and, for clarity, $\mathcal{O}_i$ denotes the logarithmic derivative of the model $\mathcal{O}_i = \frac{1}{\psi_W(\sigma)} \frac{\partial \psi_W(\sigma)}{\partial W_i}$. Now we can evaluate the function further to write the form for jth eigenstate:

$$\mathcal{F}_i^n = \frac{\partial E_n}{\partial W_i} = \mathcal{F}_i + \sum_{j=0}^{n-1} \beta_j \left\langle \left(\frac{\psi_{W^j}}{\psi_W} - \left\langle \frac{\psi_{W^j}}{\psi_W} \right\rangle_\sigma \mathcal{O}_i^\dagger \right) \right\rangle \left\langle \frac{\psi_W}{\psi_{W^j}} \right\rangle_{\sigma_j} \tag{3.50}$$

Although the form of the force vector is complex, we will try to briefly explain it. There are two W in the expression. The clear W defines the currently trained Ansatz, while W^j denotes parameters for state j. The term $\mathcal{F}_i^n$ refers to the force vector calculated over the standard VMC for the true ground state.

Having presented and briefly understood the math, we can describe what the id does. The idea is to begin with standard VMC and compute the ground state. The result must be stored, because while calculating the first excited state requires adding

the ground state to energy estimation. Together with the penalty coefficient β_j, it effectively penalizes the ground state by computing the overlap with the current state - this increases the convergence energy for the excited state. This can be computed iteratively for an arbitrary number of states, by sorting from the lowest one.

4 Introduction to JAX, Flax and NetKet

After a fairly elaborate theoretical introduction, it is time to discuss the frameworks used in this work. The most important one is *NetKet: a machine learning toolbox for many-body quantum systems* [10]. In this chapter, we discuss the foundational technologies of NetKet. We begin by exploring *JAX*, the numerical computing library that provides the automatic differentiation and GPU acceleration capabilities [8]. The next part will be presenting *Flax* [9], which is a neural network library built on top of JAX. It offers the flexibility and modularity needed to design complex neural network architectures for the scope of this work. The last part will be getting a basic knowledge about NetKet itself, highlighting the capabilities necessary for performing optimizations and defining Hamiltonians.

4.1 Introduction to JAX

JAX is a high-performance numerical computing library built in Python and developed by *Google Research* [8]. Its creation was motivated by the growing need for a flexible and powerful framework that could leverage the computational possibilities of modern hardware architecture and, at the same time, providing a high-level interface for scientific programming.

One of the strongest advantages over traditional computing libraries (NumPy, for example [43]) is that it has built-in support for automatic differentiation, allowing one to effortlessly compute the gradients during training. JAX also provides support for hardware acceleration on GPUs and TPUs, in contrast to NumPy, which was only CPU-bound. In addition, the API for JAX looks almost the same as that for NumPy, because it was specifically designed to reflect the same convention. In this section, we will briefly discuss all possibilities that come with JAX.

4.1.1 Just-in-time compilation

Python is one of the most popular languages in programming and is used in a variety of different fields [25]. The development in machine learning is mainly driven by Python-based frameworks, such as Pytorch [26]. It is so successful because of its clear syntax and high-level operation. However, in scientific computing, its biggest flaw is that it is a dynamically typed language. Before explaining what that is, let us describe statistically typed languages.

In statistically typed languages, the type of each variable is determined at compile time. This means that every variable needs to have a type declared explicitly by a programmer. This condition is necessary because the code is interpreted by a compiler, which translates the written code into a bytecode, that can be interpreted by the machine.

In dynamically typed languages, like Python, each variable is determined at runtime and there is no need to define it explicitly. The interpreter does not enforce any constraints, and the variable's types can be changed freely. This makes the language flexible and easy to use, but it comes with a drawback. The dynamic interpretation of

the code is extremely slow compared to that compiled into bytecode. It is a common problem, especially with Python, because of its wide popularity.

The solution to this problem was utilizing the *Just-In-Time* compilation [89]. Enhances the Python code by dynamically compiling the code into optimized machine code at runtime. JIT analyzes the code, identifies sections that can be optimized, and stores the machine code compiled in cache memory, so that it could be used efficiently again. This solution leads to speed-up, which can be compared to C/C++ in numerical computations.

In the context of JAX, this type of compilation is particularly beneficial for machine learning, which often involves repeated execution of the same code. By compiling it, the execution time is significantly reduced, making it feasible to run and train large models.

4.1.2 Accelerated Linear Algebra

Accelerated Linear Algebra (XLA) is a compiler that optimizes the execution of numerical computations in JAX, especially linear algebra operations by converting them into efficient machine code. At its core, XLA takes a high-level computational graph which represents the sequence of operations needed to perform a computation and applies a series of transformations, creating a set of instructions optimized for the target hardware (like operator fusion, dead code elimination, or algebraic simplifications). The resulting machine code is efficiently designed for CPU, GPU, or TPU [63, 67], allowing one to leverage the full potential of hardware accelerators.

One of the key advantages to mention is its ability to perform operator fusion, which allows one to combine multiple operations into a single kernel [90]. To bring an example that utilizes this feature, a sequence of matrix multiplications can be fused into a single operation that performs every computation in a single pass. It allows reducing the number of intermediate results that need to be stored and retrieved from memory, effectively reducing memory overhead and improving performance. This capability is particularly beneficial in deep learning applications, where memory access can be a bottleneck.

XLA also provides a way to distribute computations across different hardware components - to choose specifically the one that will perform best for given task. For example, on TPUs utilize specialized matrix multiplication units designed for deep learning model training, while for parallel processing it will incorporate GPU. This feature makes JAX a flexible and scalable framework.

4.1.3 Automatic differentiation

We now explain the most crucial feature of JAX, which is the *automatic differentiation*, which allows us to calculate the derivatives of functions with respect to their inputs automatically. This is exactly what makes this framework stand out above other, like NumPy, where other methods approximate derivatives, which can suffer from numerical instability (finite differences, for example). Automatic differentiation

computes exact derivatives by systematically applying the chain rule to break down complex functions into a series of elementary operations.

JAX implements two primary modes of automatic differentiation. We begin with the first one, which is the *forward-mode* differentiation. It is the most effective for functions with a small number of inputs and a wide range of outputs. It works by utilizing the computational graph, where the inputs at each step of the computation are augmented with an additional derivative. Both quantities propagate through computations, where the chain rule is applied at each step. It allows one to compute the derivative in a single forward pass. The mode is effective for small-scale problems.

The second one is the *reverse-mode* differentiation, which is what we refer to as backpropagation in the field of neural networks. It is more efficient for complex functions with a large number of inputs and a few outputs, which is exactly the opposite to the forward-mode differentiation. To compute it, JAX performs first a forward pass through the computational graph to record intermediate results, and then performs a back pass by applying the chain rule in reverse direction. This approach is memory-intensive because it requires one to store all values along the way. The benefit is that it allows for a highly efficient function evaluation with high-dimensional inputs.

In the framework itself, automatic differentiation is performed by calling the function *'grad'*. It allows specifying which inputs to differentiate with additional support for higher-order computations. This provides a simple, flexible interface, which is very valuable in the field of machine learning.

The combination of XLA and automatic differentiation in JAX allows us to optimize the execution of the computational graph generated by differentiating. When the function *'grad'* is called, the computational graph passes value-derivative pairs into XLA for optimization. Then, XLA converts these variables to a machine code to ensure the most efficient operation.

4.1.4 Random number generation

At the core of modern computational tasks lies the process called *random number generation (RNG)* [91], which turns out to be a non-trivial topic in computer science. In the macroscopic, classical world, all processes are deterministic, and pure randomness is possible only in quantum mechanics, achieved by the collapse of wave function during a measurement. Many different approaches were taken to develop the best RNG possible. It is important to point out that all algorithms (purely computational) created are based on deterministic mechanisms and are not truly random. This is why they are called *pseudo-random number generators*. However, modern techniques are so advanced and complex that effectively the difference between true randomness fades away with each new development.

In standard Python [25] all random number generators are handled by an algorithm called *Mersenne Twister* [92]. It works by maintaining a large internal state that consists of an array of integers. The state is initialized with an arbitrary value, provided by the user (we call it *the seed*). Providing the seed sets the starting point of the random sequence. The flow of random numbers is achieved by repeatedly applying a series of transformations which mix the bits in the internal state in a way so complex

that the output is effectively random. The drawback of Mersenne Twister is sequential generation, where generating random numbers in parallel requires careful management of the internal state to avoid correlations between sequences. An additional problem is that, in parallel distribution, it is impossible to reproduce the internal state, which makes debugging more challenging. These problems make Mersenne Twister unsuitable for hardware accelerators, such as a TPU or CPU.

This is why JAX comes with a new RNG mechanism, where the goal now is to make it suitable for high-performance computing environments and to overcome the Mersenne Twister limitations. For this reason, JAX's RNG is stateless and functional, which makes it more suitable for parallel computations for accelerators. The mechanism of the system consists of a splittable counter-based PRNG, which operates by treating random number generation as a deterministic function of a key and a counter. The key is a tuple of two 32-bit integers, which serves as the seed for an RNG. The counter is used to generate different random numbers from the same key without modification. This allows the gegneration process to be deterministically divided into many distributions.

The RNG system in JAX is designed to support efficient parallel computation. The ability to split keys allows one to generate random numbers independently on multiple cores, without synchronizing with the initial state. In the scope of this work, this is a crucial feature used in estimating energies in variational Monte Carlo, where sampling the basis states can be split into many individual processes, where each has its own independent random number generation. Overall, the solution provided by JAX allows for better and quicker convergence to the eigenstate.

4.2 Brief overview of Flax

Having briefly discussed the JAX environment, we could grasp the underlying potential of the framework. A successful attempt to put that potential to use was to build a machine learning library on top of it - this is exactly what offers the Flax framework [9]. Provides the necessary tools to build, train, and deploy neural networks with an emphasis on research and experimentation. Flax is designed to efficiently leverage the features of JAX, including automatic differentiation, just-in-time compilation, and hardware acceleration. In addition, it continues the idea of offering a user-friendly interface to build complex neural networks.

It is worth to point out that JAX is not only an advanced numerical library - it is a well-optimized foundation for building larger libraries and frameworks, where Flax is only one of many examples. In contradiction to PyTorch [26], where we have all the necessary elements to create and train a model in one framework, JAX is a core of a larger ecosystem, where many different libraries are merged into single machine learning projects. For example, Flax can be combined with Optax, a library used for gradient-based optimization algorithms, to implement a sophisticated training method that is both efficient and easy to use.

In this section, we will briefly discuss what Flax has to offer in the field of machine learning and why it might be a better solution compared to other well-established frameworks.

4.2.1 Functional programming

In programming we have two distinct and most prominent paradigms, one of them is *object-oriented programming*, and the second is *functional programming* [93]. they represent fundamentally different approaches to structure code, each with its own set of principles and use cases. It is important to understand the basic differences.

Object-oriented programming is a paradigm centered on the concept of *classes*. A class defines a blueprint for *objects*, which are their instances. It encapsulates data and their behaviors within a single entity, which we call *attributes* and *methods*.

OOP is based on four main principles, and we will briefly describe each of them. The first is *encapsulation*, which is the idea of storing all necessary data and functions that describe its behavior in a single entity to make it concise. Another rule is *Inheritance*. It is a concept that allows to inherit all features and methods from another class into a new one, allowing to reuse the code and create a hierarchy structure. Enhances the development of complex systems by creating new classes based on existing ones, with modifications and extensions. The third principle is *polymorphism*, which is a concept that provides a way to define a common behavior for different data types. Allows for the use of a single function or method to operate on different types of objects, enabling a more generic code. Lastly, we have *abstraction*, which is about hiding complex implementation details and expose only essential features of an object. This allows one to focus on the representation of the object rather than on how they work.

The concept of OOP is widely used in the majority of ML frameworks. This promotes a clear organizational structure, making it easier to model relationships between different component parts of the application. This approach is used in PyTorch [26], for example, where neural networks are defined as classes, where the model architecture, parameters, and logic are encapsulated within the object. The concept makes the code flexible and concise.

However, encapsulation can lead to problems in complex systems, where mutable states and side effects become difficult to manage and debug. Here we approach the alternative, which is *functional programming*. This paradigm is built on the concept of pure functions, which are the functions that always produce the same output for the same input with no additional side effects, such as changing the data outside their scope. It emphasizes *immutability*, where data structures are not modified after creation and functions can be passed and manipulated into other functions like any other data type (they are called *first-class citizens*).

In the context of ML and Flax, which incorporates the functional paradigm, this means that neural networks and other computational entities are defined as pure functions that take inputs and produce outputs without altering any internal state. The functional style focuses on separating model definition and its state management. Because its functions are pure, they can easily be composed or reused independently without affecting any other parts of the system. This approach is beneficial in research, where models are often made up of different types of layers. Another aspect of the functional style is compatibility with advanced optimization techniques, like just-in-time compilation. It allows one to efficiently use JIT functions, where one can be passed to another and they are all compiled to machine code together. Traditional frameworks

often mix computation and state management, making it harder to implement JIT effectively.

Overall, the adoption of the functional programming style in Flax represents a revolutionary change in how neural networks are defined. The approach of composing small JIT-optimized functions into a broader composition makes it particularly useful in the area of research and development in deep learning.

4.2.2 Constructing neural networks

After discussing the primary programming paradigm used in Flax, it is time to introduce some key components used to construct a whole neural network with its training procedure. The functional style allows one to separate the model definition from its management, which we will try to present in this section.

To begin with, it is worth to introduce the concept of a module in Flax. It is a self-contained and reusable component that represents a single operation in a neural network. Modules can be linear layers, activation functions, convolutions, even self-attention mechanisms. Following the principle of functional programming, each module defines a specific computation that transforms its inputs into outputs. The developer can create its own modules out of built-in components and then merge custom ones to form a complete single structure. This design allows to scale up the level of separation, where each module is responsible for a specific part of the computation, making it easier to understand, test, and modify.

When defining a new neural network module in Flax, it must inherit from a class *'flax.linen.Module'*, which forms a basis for encapsulating in a single entity all functions that should be applied to inputs. Describes the structure of the neural network in a declarative manner, specifying the sequence of layers and operations. Because, according to functional style, Flax treats networks as compositions of pure functions, it allows for greater modularity, which is especially beneficial for researchers, where experimenting with many architectures in a reasonable amount of time is crucial. Here we can describe a new feature, which is the *hierarchical composition*. Its main idea is to form a tree-like structure in which modules can contain other modules. It allows one to build complex models by nesting simpler components and better organize the code. It also provides a way to manage and modify complex models as changes can be made at any level of the hierarchy without affecting other components. Here we arrive at the concept of *subnetworking*, where entire neural networks can be encapsulated in one major entity. The approach is particularly useful in transfer learning, where pre-trained models are fine-tuned for new tasks [94].

Overall, the functional approach in Flax promotes a more declarative style of programming, where the focus is on the computations, rather than how they are implemented and structured. It makes it more straightforward to reason about network's behavior and identify problems or space for improvements.

4.2.3 State and parameter management

For the last section on Flax, the concept of state and parameter management needs to be discussed, because that makes the framework different from the other ones. While traditional ML libraries treat managing weights, biases, and model's definition as a merged entity, the idea in Flax is to separate these concepts. This allows for a more robust and efficient handling of parameters, which is valuable for complex neural networks and advanced research. Understanding how Flax manages the state and parameters is beneficial for maximizing its full potential and purpose. This knowledge impacts everything from model initialization, to training, and deployment.

In neural networks, the word *parameter* refers to the learnable components of the model, which in most cases are weights and biases, but incorporates also the filters in convolutional layers, for example [85]. The term *'learnable'* refers to quantities that are being optimized, where automatic differentiation tracks their gradients.

The variables that are not directly optimized during training but are still essential for model operation are called *state*. They are often automated automatically during a forward pass and are critical for ensuring the correct behavior of certain layers or operations. The examples might include hidden state in RNNs ore statistics of layer normalization or batch normalization.

Flax also provides its own way to initialize parameters. While other frameworks do it automatically and changing them requires an extra effort, in Flax the initialization process is separated from the model definition, ensuring that they are not embedded within the model and can be passed into functions explicitly. It is done by calling *'init'* function, which takes a random key (following a random number generation convention in JAX) and sample input that defines the shape and structure of the data processed by the model. The function traverses the architecture layer by layer, initializing them according to requirements specified by the developer (such as the normal distribution). Handling initialization in this way allows one to control the process, enabling different strategies to experiment to control its impact on the model's performance. At the same time, it ensures that the model definition remains purely functional, which simplifies processes like transfer learning or hyperparameter tuning. Once initialized, the parameters are explicitly passed to the model during the forward pass. This contrasts with traditional libraries, where parameters are accessed implicitly.

The state of the model, on the other hand, is tread as a mutable object, which can be updated along the way. When a model is executed in Flax, the forward pass returns not only the output but also an updated version of the state. This approach ensures that all state updates are handled explicitly. This design allows for precise control over state updates, where they need to be carefully tracked. The in-place update is revolutionary, compared to older frameworks, where state was treated as immutable, which required constant reassigning of the variable. Flax also provides an automatic hierarchical state update, which also detects and computes the states that inherit from the major one.

4.3 VMC with NetKet

After a brief introduction into a vast environment of JAX and Flax, we can finally present the main framework of this work, which is *NetKet* [10]. NetKet is fundamentally a machine learning framework specified for quantum many-body physics. It provides a clean and understandable interface that allows easy implementation of neural quantum states [83]. NQS are neural network models that can efficiently represent a quantum wave function, which can be parameterized to approximate the quantum state of the system.

The use of QNS for simulating a quantum system defined on a Hilbert space can significantly increase the number of particles to be simulated. Traditional exact diagonalization methods fail between $20 - 30$ objects, The other, more advanced methods, like *density matrix renormalization group (DMRG)* [95], can increase that number to higher tens, but only in one dimension (or quasi-one dimension, which is a honeycomb stretching in one direction, for example). Encoding quantum states into neural networks has a goal to provide a universal tool for approximating a larger amount of spin. The whole community and framework itself is still developing, so it is impossible to provide the exact number, but they can be expected in hundreds of spins. For highly powerful machine utilizing TPUs it might even scale up to a 4-digit numbers. The advantage of NQS is that it is less dependent on Hilbert space and the bottleneck can be hidden in networks complexity or a high number of Monte Carlo samples for accurate approximations. Let us now briefly walk through other possibilities offered by NetKet:

- Built-in quantum models. NetKet provides a whole Python module *'Operator'*. We can use their implemented spin models, like Heisenberg or Ising, but the same module contains all components to create custom Hamiltonians with ease. We will explore this capability in detail in section 4.3.3.

- NetKet provides an efficient representation of quantum states. It begins with exact implementation of the wave function and Jastrow Ansatz [82], ends on advanced neural quantum states, like range of RBMs, ARNNs, MLPs and even *group convolutional neural networks (G-CNN)* [96].

- Built-in different types of Hilbert spaces and lattices. NetKet provides two modules: *Hilbert* and *Graph*. The first allows to implement Hilbert spaces lazily, which means that there are evaluated only when there are needed in the computation. It allows for defining large space, which would not be possible to store in memory. There are implementations for both discrete and continuous Hilbert spaces. The second module is used for defining lattices and neighboring spins. NetKet provides built-in lattices (like triangular, honeycomb or kagome), but also provides a possibility to use 3D cells, like cubes or diamond-like structures.

- NetKet has built-in samplers that utilize Monte Carlo sampling with Metropolis algorithm. There are a few variations depending on the use case. For example, there is a *'MetropolisLocal'* sampler, which flips a random spin, or *'MetropolisExchange'*, which swaps positions of two spins. This one is more specific, because it allows to preserve conserved quantities, like total spin. This allows to perform

block-diagonalization, because there is a guarantee, that sampler will not fall out from simulated regime.

- Built-in variational states and their optimizations. Variational states are prepared for both pure and mixed states, which allows for simulating real case scenarios. The optimization algorithms, also refered as drivers. Can optimize for the lowest value returned by an operator (VMC), or can even simulate their time evolution (T-DVP algorithm).

Throughout this section we will take an attempt to understand the whole process to create a variational driver in NetKet, as this is important part in next chapter, where we will be discussing the results.

4.3.1 Constructing lattices

The fundamental aspect of modeling quantum systems in NetKet is the construction of appropriate lattices. They provide a spatial framework on which particles or spins reside and interact with each other. Understanding how to construct a layout for particles is a starting point of the simulation and is essential for accurately modeling the system in search of frustrated states, for example.

NetKet supports a variety of lattice structures that are common in quantum many-body system simulations. Let us now discuss what the framework has to offer:

- The most fundamental lattice constructor is the class *'Hypercube'*, where we can specify L as a number of spins and define the dimensionality with *n_dim* parameter. In this way, it is a convenient way to construct chains, squares, and cubes. Additionally, there is also the *'Grid'* class that allows one to create a cell of an arbitrary dimensionality. As an example, specifying the parameter *'extent'* equal to $[2, 3, 3]$ will create a cuboid of size $2 \times 3 \times 3$, while $[4, 2]$ creates a 4×2 rectangle.

- Various types of 2D lattice, where the most common ones are triangular, honeycomb, and kagome. Effectively, their main difference is the number of spins with which they interact. With the examples above, the numbers are 6, 3 and 4, respectively. Each of them is also specified by an argument *'extent'* and in this case it needs to be a tuple of two integers, where each characterizes the number of closed cells in given direction. NetKet also provides visualizations of the created lattices, which are shown in the figure 2.1.1 in chapter 2.

- There is also a support for 3D lattices, where now the *'extent'* argument is a 3-valued integer tuple. The built-in layouts are primitive, face-centered and body-centered cubic. The first is equivalent to specifying a *Hypercube* with *n_dim* equal to 3. The other ones are Bravais lattice cells, where the face-centered (FCC) means a cube with additional spin in the middle of each wall, while the body-centered (BCC) has additional spin in the middle of the cubic itself. Two other lattices are diamond, where spins form a tetrahedral structure, and hexagonal close-packed (HCP), where many layers of honeycomb 2D lattice are stacked on top of each other.

- When some custom lattice layout is needed, it can easily be implemented in various ways. The first option would be to define a *'Lattice'* class, where *'basis_vectors* need to be specified, as a coordinates for a vector forming a cell in Bravais lattice. There is also an option to load a custom graph with defined nodes and edges as an object in the *NetworkX* library [97].

It is important to mention that a flexible way of boundary conditions is implemented for each type of lattice. We can specify different types for boundary conditions for each dimension in 2D and 3D lattices. NetKet also provides an API for creating custom classes which can inherit for the ones that are already created, where possibilities of upgrading are boundless.

4.3.2 Hilbert space

After constructing lattices, the next step is to define the Hilbert space. The exact Hilbert space is the complete set of all possible states the system can occupy. It is obvious that directly storing or enumerating all possible states of large spaces is impractical and computationally infeasible. To overcome these challenges, NetKet employs sophisticated methods to represent Hilbert spaces without storing all states explicitly.

NetKet uses an implicit representation that relies on a few key strategies. First of all, the states are enumerated based on constraints and local rules determining which rules are valid. It allows one to manipulate only the states that are relevant to a specific calculation. Another important feature is that quantum state indexing is executed in the most efficient way, where basis states are mapped into individual bits allocated to an integer value [35], enabling a fast computation of observables and other quantities without full enumeration. Combining these features with Monte Carlo sampling makes a powerful tool which allows for defining much larger Hilbert spaces.

NetKet provides a flexible and comprehensive tools for creating various Hilbert spaces, depending on the type od a particle, properties and applications, here we briefly discuss some of them:

- Provides classes for creating bosonic Hilbert spaces, where particles like photons or Cooper pairs do not obey the Pauli exclusion principle [11, 1]. For this task, the class *'Fock'* can be used for defining the bosonic space, where each site can accommodate an arbitrary number of bosons. In addition, some constraints can be applied, such as the maximum number of particles per site. This flexibility provides a powerful tool for simulating Bose-Hubbard models, for example.

- Fermionic Hilbert spaces, where particles are characterized by the antisymmetric nature of their wave functions, reflect the Pauli exclusion principle, where no two fermions can occupy the same state simultaneously. NetKet provides a class *'Spin'* or *'SpinOrbitalFermions'*, where we can define the number of orbitals, spin of the fermions, and the number of them per single spin. Useful for simulating the Hubbard model, for example.

- The most important part of the scope of this work is the creation of spin systems. NetKet has a special class *'Spin'* for this task, which was also mentioned earlier. It is a foundation for all simulations performed in chapter 5. We can provide an exact spin value for a single particle as a parameter *'s'*. Applying $\frac{1}{2}$ to it will create two basis states, which are $\{-\frac{1}{2}, \frac{1}{2}\}$, but higher spin states can also be provided, such as $\frac{3}{2}$ or $\frac{5}{2}$. It also applies to integer values of spin and the number of them scales according to the rules discussed in section 2.1.2.

- The last example of use case are continuous Hilbert spaces. NetKet has a built-in class for this, called *'Particle'*. It allows for specifying the number of spins and providing details about continuous quantum numbers. However, at the time of writing, the features for continuous spaces are still in development and some capabilities might be added.

To end the topic of Hilbert spaces, some last key notes are worth to mention. All spaces discussed above can be combined using a single multiplying operator *'*'*. This powerful feature allows one to simulate systems, where many different particles interact with each other. Secondly, NetKet provides a variety of ways to reduce the real size of Hilbert spaces by applying constraints, such as symmetry reduction or block diagonalization for total spin of the system.

4.3.3 Creating operators

After building a solid foundation to construct the Hilbert space and defining spin lattices that describe the interaction of particles, it is now time to discuss the most important element for estimating the energy of the system, which is the Hamiltonian. In quantum mechanics, it is a square matrix of the size of a Hilbert space. However, approaching this dimensionality comes first from Kronecker products between individual particle subspaces (which have only two elements for the spin). The same approach is taken by NetKet, but there is the same problem of memory storage, so they are stored implicitly instead [1, 12].

There are two main ways to create operators in NetKet: by specifying a class *'LocalOperator'* and *'GraphOperator'*. They both offer slightly different optionalities and capabilities. The most basic type is the LocalOperator. It is designed to represent operators that act locally on a small subset of sites within the Hilbert space, that is why it needs to have provided the Hilbert space object and the number of the site that it acts on. The key feature is that LocalOperators can be added and multiplied together to form a Hamiltonian, which intuitively reflects their explicit mathematical form. For example, it is possible to form a Heisenberg Hamiltonian (equation 2.33) in a few lines of Python code:

```
# define the Hamiltonian
hamiltonian = LocalOperator(hilbert, dtype=jnp.complex128)
for i, j in graph.edges(): # loop through interactions
    # add each Pauli matrix to the Hamiltonian
    hamiltonian += Jx * sigmax(hilbert, i) * sigmax(hilbert, j)
    hamiltonian += Jy * sigmay(hilbert, i) * sigmay(hilbert, j)
    hamiltonian += Jz * sigmaz(hilbert, i) * sigmaz(hilbert, j)
```

where *'hilbert'* is a previously defined Hilbert space and *'graph'* refers to the object containing the lattice layout (it contains information about number of sites and all interactions). The function *'graph.edges()'* returns a tuple of two interacting sites i and j. We can also predefine the interaction constants Jx, Jy, and Jz. And lastly, *'sigmax', 'sigmay', 'sigmaz'* are Pauli matrices predefined in NetKet module (equation 2.1). We can see the flexibility of the algebra operators $'*'$ and $'+'$, where they can be used in arbitrary order with integers, floats, and NetKet LocalOperator.

Now we move to discussing the *'GraphOperator'* class. Provides a higher level of abstraction that is particularly useful for defining operators that act on multiple sites or have complex interaction patterns due to the lattice structure. While LocalOperators are suitable for defining local interactions, GraphOperators leverage the connectivity and symmetry of the underlying lattice or graph, which makes them suitable for Hamiltonians with interactions that extend beyond simple local terms.

Constructing GraphOperator at first seems to be more complex than LocalOperator; however, this approach is more powerful for creating Hamiltonians with complicated interactions. In addition to providing the lattice and Hilbart space objects, it needs to have two important parameters specified: *'site_ops'* and *'bond_ops'*. The first needs to be a 2×2 array, which specifies an operator acting on the particular site only. The second defines an operator acting between two sites, which explains why its size must be 4×4. Having all parameters provided, the GraphOperator determines the dimensionality based on the Hilbert space object and applies interaction operators specifically to the sites described by the lattice object. Below we can see an example of creating the same Heisenberg Hamiltonian as before:

```
# interaction operators
bond_ops = [Jx * SX_SX + Jy * SY_SY + Jz * SZ_SZ]
# define the Hamiltonian
hamiltonian = GraphOperator(
    hilbert=hilbert,
    graph=graph,
    site_ops=[],
    bond_ops=bond_ops)
```

In the example, all variables with the same name as before refer to the same quantity. The new term is *site_ops* which takes a list of operators acting on a single site. The basic form of Heisenberg model has no such terms, which is why we leave it empty. However, it can be used to add other elements acting on particular sites, like magnetic field. The most important parameter is *bond_ops*, where we specified the matrix acting

on the edges of the graph. *SX_SX*, *SY_SY*, *SZ_SZ* are Pauli matrices scaled up to two spins, mathematically they are:

$$\text{SX\_SX} = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \quad \text{SY\_SY} = \begin{pmatrix} 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \end{pmatrix} \quad \text{SZ\_SZ} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The last important feature of GraphOperator and Lattice is *bond color*. There are specific types of lattices, where each edge can be specified by unique number, called color. GraphOperator class can interpret them and it allows to specify different operators for each bond, filtered by color. This feature is ideal for defining the Kitaev Model. Below we can see the comparison between two classes. First with LocalOperator:

```
# specifying color for each Ji
Ji = {0: Jx, 1: Jy, 2: Jz}
# specifying color for Pauli matrix
operators = {0: sigmax, 1: sigmay, 2: sigmaz}
#defining Hamiltonian
hamiltonian = LocalOperator(hilbert, dtype=jnp.complex128)
for (i, j), color in zip(graph.edges(), graph.edge_colors):
    operator = operators[color]
    edge_operator = operator(hilbert, i) * operator(hilbert, j)
    hamiltonian += -Ji[color] * edge_operator
```

and GraphOperator:

```
# specifyng 3 operators
bond_ops = [-Jx * SX_SX, -Jy * SY_SY, -Jz * SZ_SZ]
# specifying colors
bond_ops_colors = [0, 1, 2]
#defining the Hamiltonian
hamiltonian = GraphOperator(
    hilbert=hilbert,
    graph=graph,
    site_ops=[],
    bond_ops=bond_ops,
    bond_ops_colors=bond_ops_colors)
```

It is clear to see that implementation of LocalOperator is more complex, because dictionaries need to be specified. GraphOperator takes *bond_ops* and *bond_ops_colors*, which then assign corresponding operator to the specific value defined in the lattice, which makes it way more straightforward. Additionally, it can be implemented as float instead of imaginary number, because $\sigma_i^y \otimes \sigma_{i+1}^y$ is a real matrix.

4.3.4 Optimization loop

Having discussed all important classes in NetKet, we can walk through an example VMC optimization written in Python. Let us briefly enumerate all steps that need to be taken:

1) Construct the lattice. This step determines the number of spins to be simulated and assigns them to a particular layout. A class defining the graph object also takes boundary conditions as an argument.

2) Define the Hilbert space. Here, more details need to be specified - in our case the particle is a Spin class. In this step, we also provide information about the spin value (choose $\frac{1}{2}$). Additionally, we can force a block diagonalization by putting a constraint for total magnetization of the system. The most common choice is $S^z_{total} = 0$.

3) Construct the Hamiltonian matrix. At this point, the specific choice of the spin model must be made (choose Heisenberg model), and all parameters must be passed into the operator's constructor. The arguments for this step are usually the graph, the Hilbert space, and the interaction constants J_i.

4) Choose a suitable sampler. NetKet offers a variety of samplers, where at their core all are Metropolis algorithms, but each have a slightly different rule. For example, class *'MetropolisLocal'* is general class that should work for all cases, while *'MetropolisExchanges'* swaps two random spins and that guarantees that the total magnetization will be conserved - it makes the sampler suitable for block-concatenated systems.

5) Choose an optimization algorithm and its learning rate. The most common choice is gradient descent, in NetKet, denoted as *'Sgd'*.

6) In addition, add stochastic reconfiguration and determine its diagonal shift λ. Usually, the lambda ranges between $0.01 - 0.1$.

7) Determine the wave function Ansatz. The optimal model is restricted Boltzmann machine (which also is a main model used in simulations). The hidden layer coefficient α can give satisfactory results when it takes values ranging from $2 - 8$. Higher values start to be computationally intensive for larger systems.

8) Create a variational state. In NetKet, this is a class that encapsulates previously defined objects. It controls the workflow of all classes, maintains parameters, and computes gradients. The arguments for this class are the sampler, Ansatz, and the number of samples for each expected value. In NetKet, the common choice is class *'MCState'*.

9) Create the driver, which is a class that controls the optimization and drives the iteration process. Additionally, it can print out and store carious types of logs, the history of expectation values for each iteration, for example. In NetKet it is a class *'VMC'*, which takes into its constructor the Hamiltonian, optimizer, stochastic reconfiguration, and variational state.

10) If everything was correct, the function *'VMC.run()'* performs VMC iterations. Below we can see the whole process written in Python.

```
# create graph with nodes and edges
graph = nk.graph.Chain(length=20, pbc=True)

# create Hilbert space
hilbert = nk.hilbert.Spin(s=1/2, N=graph.n_nodes, total_sz=0)

# create Hamiltonian
hamiltonian = nk.operator.Heisenberg(
    hilbert=hilbert, graph=graph, J=1)

# create sampler, n_chains defines the number of independent
   Markov chains
sampler = nk.sampler.MetropolisExchange(
    hilbert=hilbert, graph=chain, n_chains=n_chains)

# use stochastic gradient descent
optimizer = nk.optimizer.Sgd(learning_rate=lr)

# define stochastic reconfiguration
sr = nk.optimizer.SR(diag_shift=ds, holomorphic=True)

# define the Ansatz
rbm = RestrictedBoltzmannMachine(
    alpha=alpha, param_dtype=jnp.complex64)

# variational state encapsulates all necessary objects
vstate = nk.vqs.MCState(sampler, rbm, n_samples=2000)

# define the driver
gs = nk.VMC(hamiltonian=hamiltonian, optimizer=optimizer,
            preconditioner=sr, variational_state=vstate)

# run optimization
start = time()
gs.run(n_iter=n_iter, show_progress=True)
```

The algorithm shown above is the core for all optimizations of this work.

5 Simulations and benchmarking with NetKet

This section presents the results of all calculations performed by the author. We start by simple benchmarking with different numbers of independent Markov chains. The scope then proceeds into ground state search for different models with different complexities. Another section shows the attempt to converge into excited states of the spin system. All simulations were performed on GPU in JAX.

Hardware used for simulations:

- CPU: AMD Ryzen 5 3600, Windows 10 with WSL.
- GPU: Nvidia Gigabyte RTX 2060
- RAM: 16 GB

5.1 Benchmarks

5.1.1 Number of chains

The sampler implemented in NetKet has an option to provide the parameter called *n_chains* or *n_chains_per_rank*, they both specify the number of independent Markov Chain Monte Carlo sampling local states from Hilbert space. Theoretically, a larger number strongly supports parallelism to some extent. The goal of this benchmark was to find the optimal number of independent chains to minimize the optimization time of VMC.

Figure 5.1.1 shows the results. Each datapoint in the grid on the x -axis was simulated with a different number of chains. Starting from eight, incremented by eight, with the largest number equal to 112. The y-axis is for the execution time in seconds. The example spin system was the Heisenberg model on a 5×5 honeycomb lattice with 50 sites and a coupling constant equal to one. The Ansatz used was the restricted Boltzmann machine with parameter α equal to two and three. Lastly, two different plots were generated, with different numbers of samples used to estimate the expectation energy value.

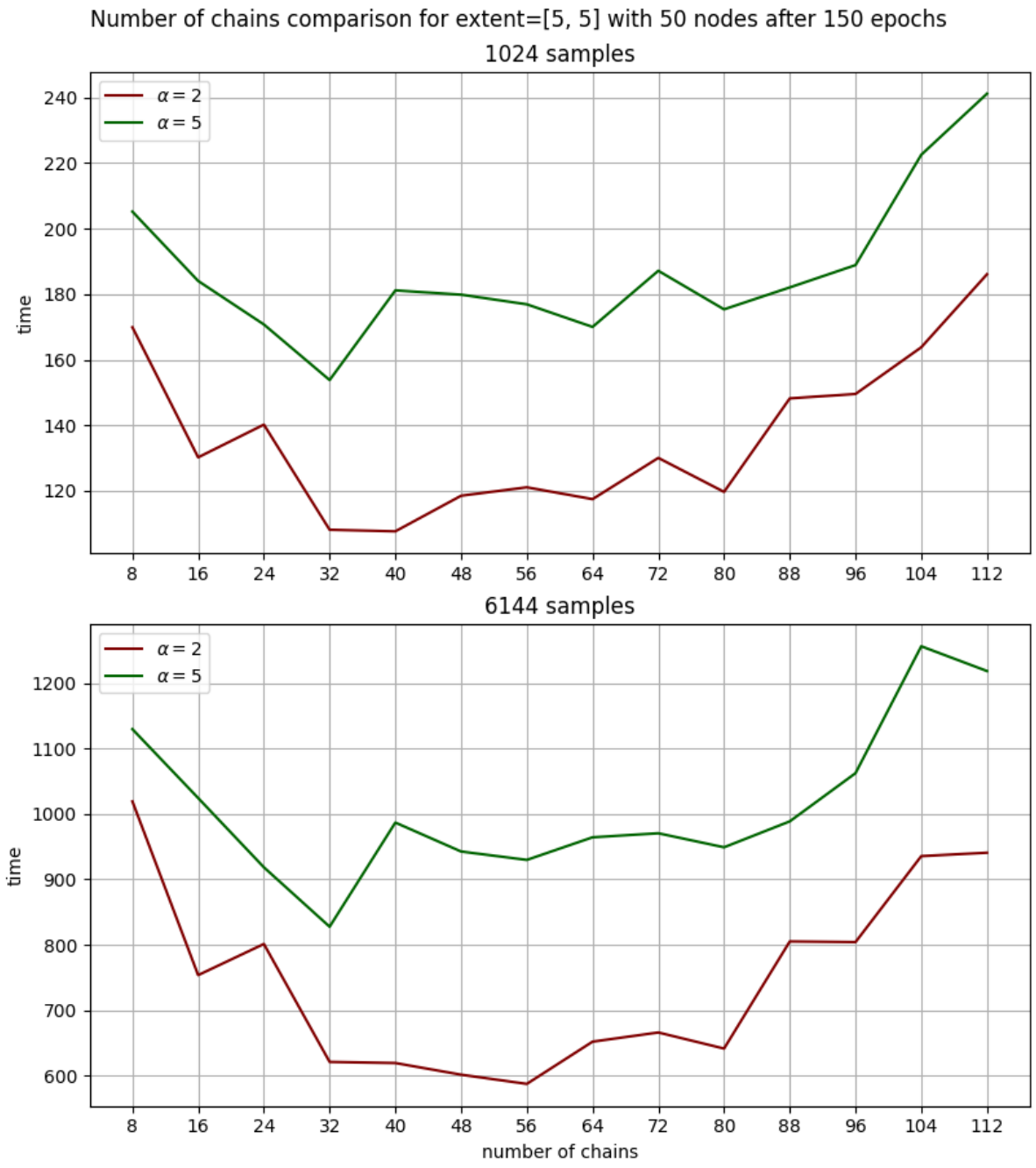
Number of chains comparison for extent=[5, 5] with 50 nodes after 150 epochs
1024 samples
α = 2
α = 5
time
240
220
200
180
160
140
120
8
16
24
32
40
48
56
64
72
80
88
96
104
112
6144 samples
α = 2
α = 5
time
1200
1100
1000
900
800
700
600
8
16
24
32
40
48
56
64
72
80
88
96
104
112
number of chains

Figure 5.1.1

5.1.2 Largest possible model

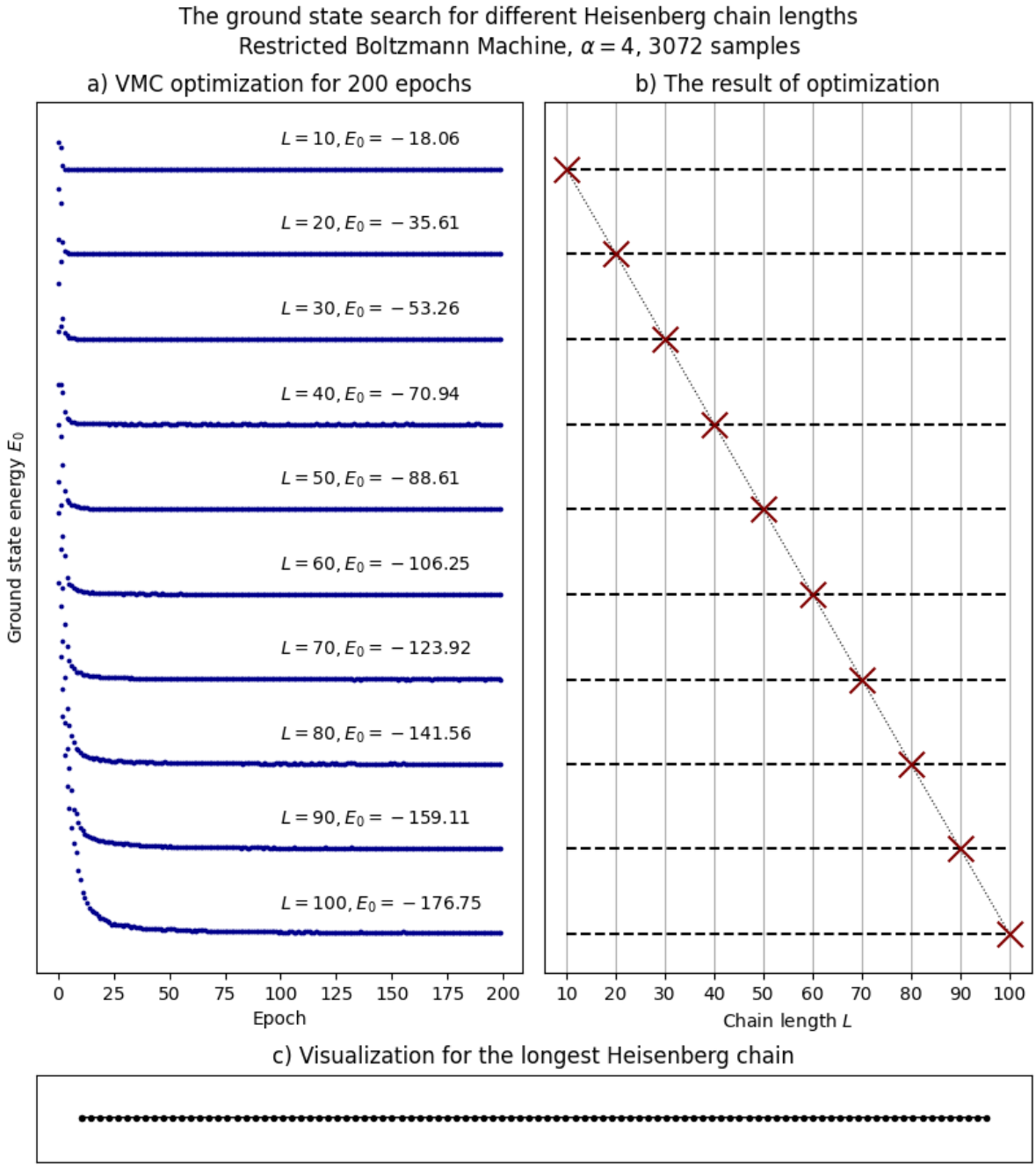

Figure 5.1.2: The figure presents the attempt to simulate the largest model possible. The number of spins larger than 100 caused a memory error. a) VMC run for each L b) The linear scaling of energy with system size is clear. This indicates that the values were correctly estimated. c) Visualization of the largest model to present the spin system

5.2 Ground state search

5.2.1 Restricted Boltzmann Machine

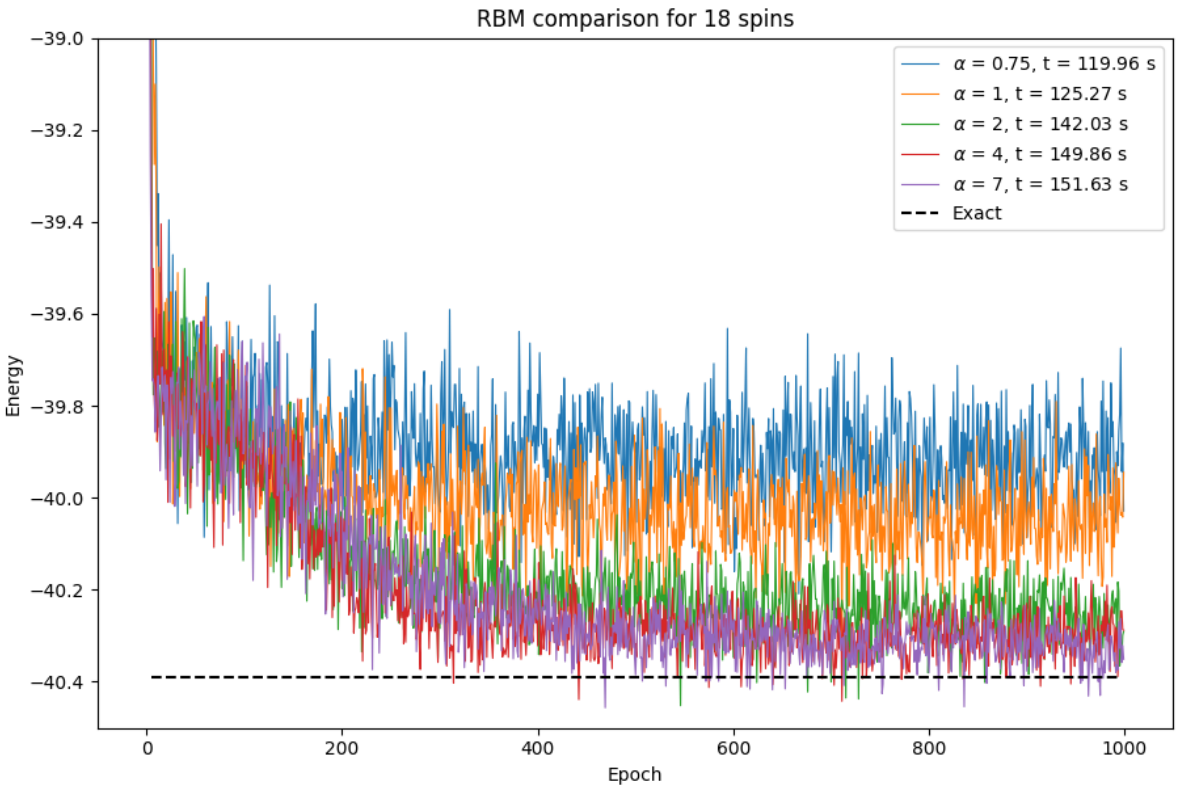

Figure 5.2.1

The figure 5.2.1 shows the VMC optimization for restricted Boltzmann machine for various α parameters. The simulated model was Heisenberg embedded in the honeycomb lattice with extent 3×3. These simulations were performed on CPU. The relatively low number of spins allowed to calculate the exact ground state energy. the expectation values were performed for 800 samples, and optimization took 1000 epochs.

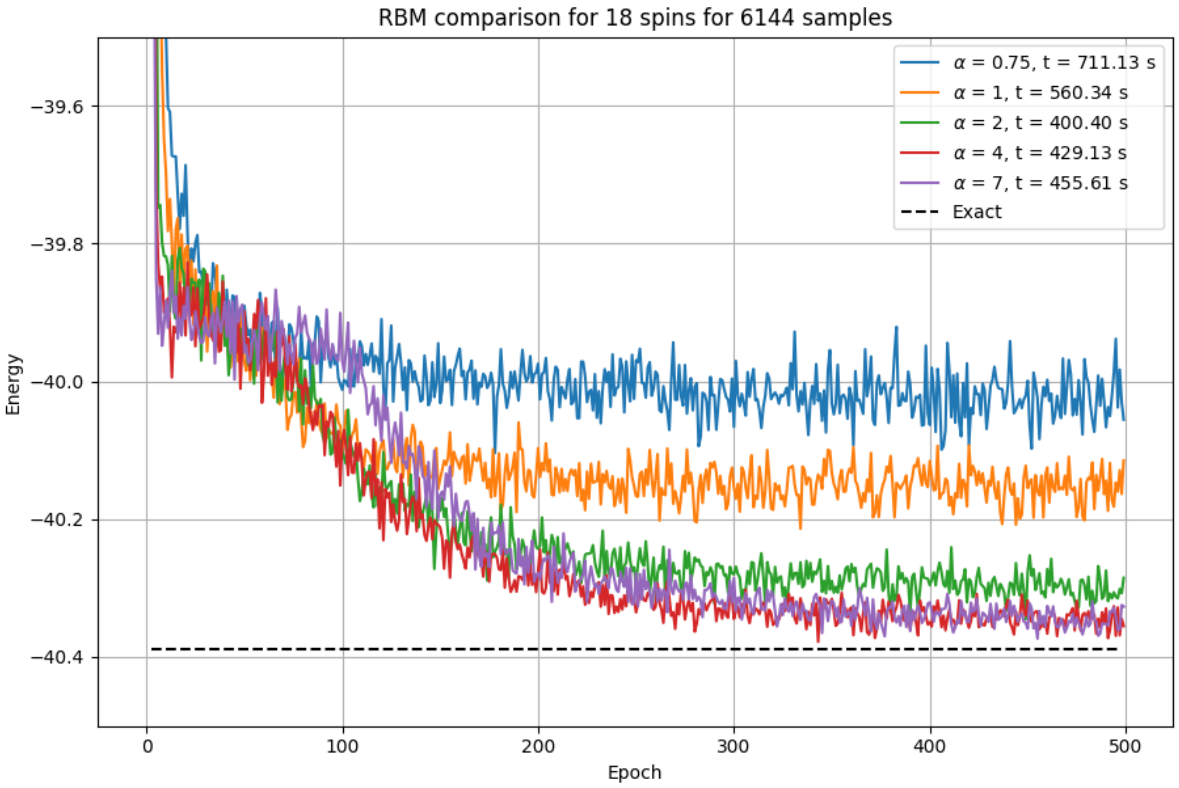

Figure 5.2.2

The figure 5.2.2 was performed in similar way to 5.2.1. The main difference is that this time it was executed on GPU, with increased the number of samples from 800 to 6144 and epochs were reduced to 500. We can see that even for small models GPU allowed to massively increase the number of samples with relatively low computational time increase.

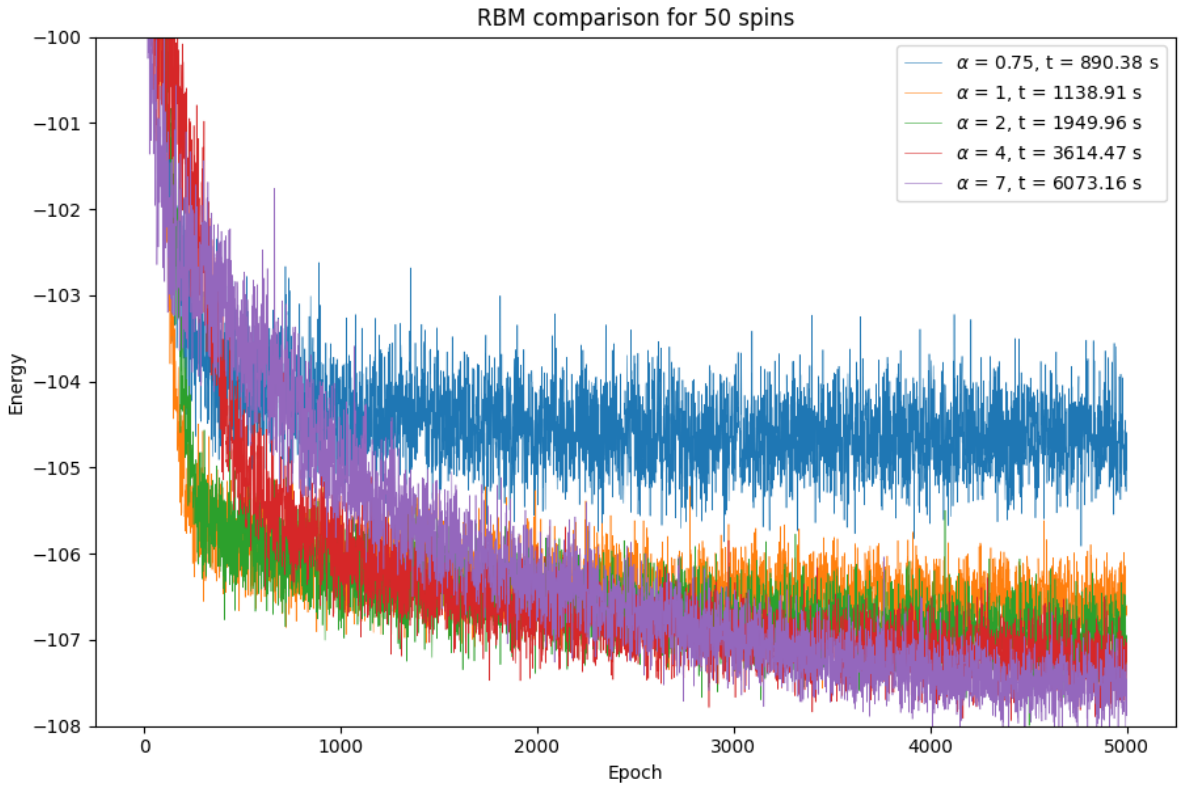

Figure 5.2.3

The figure 5.2.3 presents another attempt to run VMC on CPU. This time for larger model with Heisenberg honeycomb lattice with extent 5×5, which now spans to 50 sites. This is beyond possibilities of exact methods, so only VMC optimization was performed. The configuration used 800 samples and 5000 epochs. We can see how smaller number of samples leads to slower convergence. Running the model on CPU has a massive computational cost.

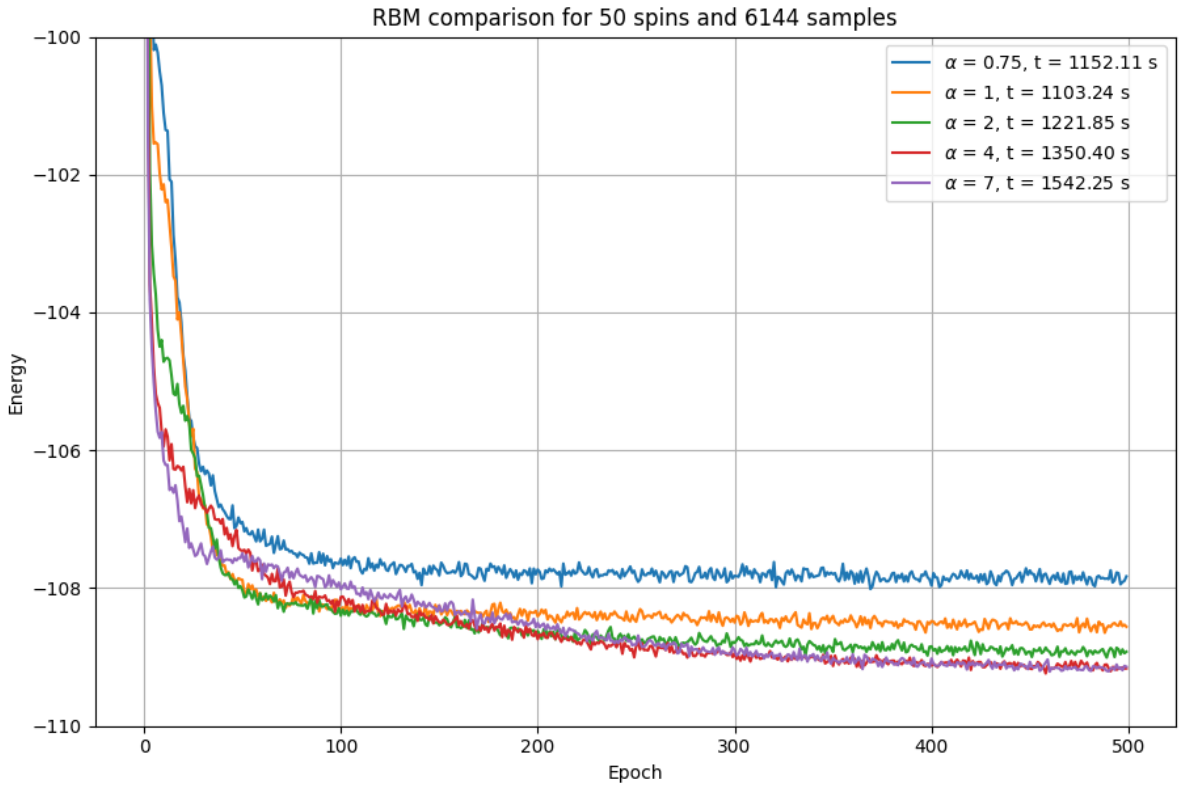

Figure 5.2.4

The last simulation run for an RBM was performed on GPU for the same model that was used in 5.2.3. Now the number of samples is equal to 6144 again, with the number of epochs dropping to 500. Now we can see the benefit of using GPU. For massively larger number of samples and fewer iterations led to even better result, with significantly lower computational time compared to the CPU.

5.2.2 Deep Boltzmann Machine

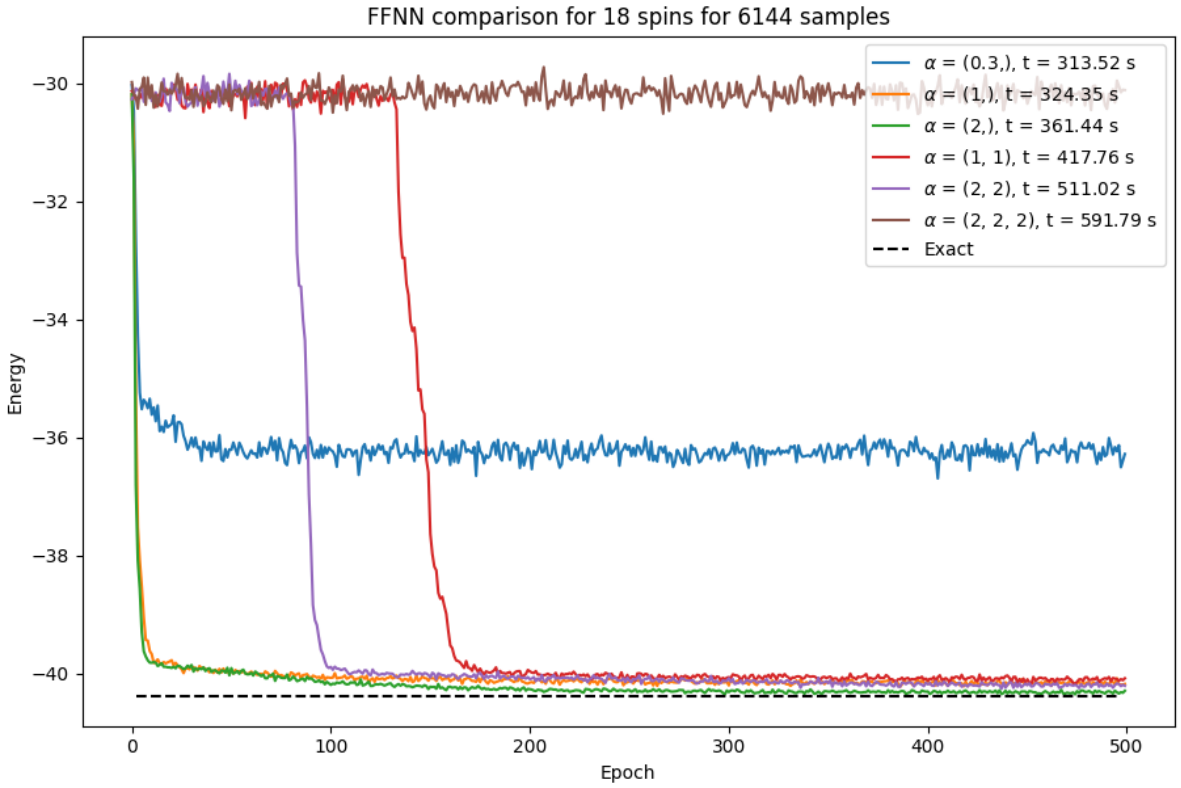

Figure 5.2.5

This attempt of running a deep Boltzmann machine was performed using 6144 samples, running on GPU for 500 epochs. We can see a different behavior compared to RBM, where in some cases the energy drops eventually, but two networks stay at the same level and do not converge. Again, simulation was run for Heisenberg model on 3×3 honeycomb lattice, with exact solution provided.

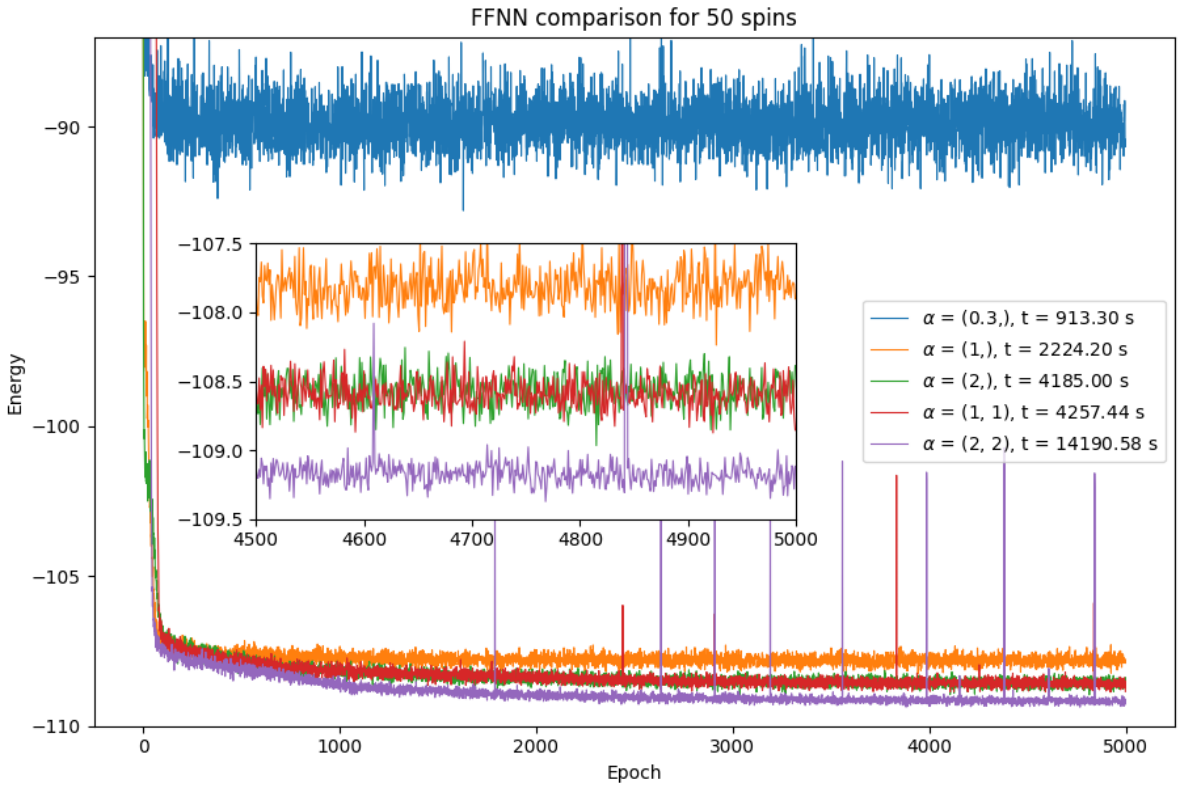

Figure 5.2.6

In the figure 5.2.6 we can see how simulation run on CPU slows down the VMC optimization. The conclusion is that deep networks are executed much more efficiently on a GPU. The simulation was performed for 800 samples, 50 spins and 5000 epochs. In the zoomed part of the figure we can see the ending result and differences in convergence between deeper models.

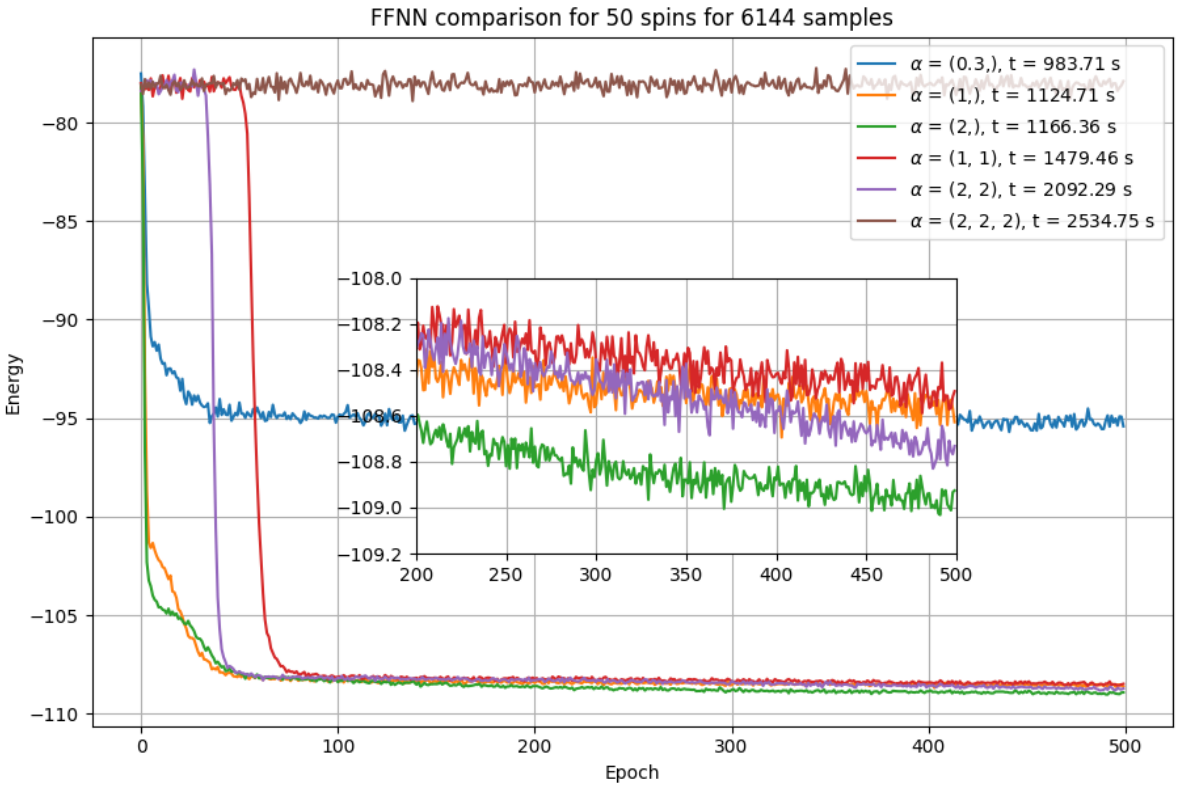

Figure 5.2.7

This time the simulation was moved to GPU again. We can see that deeper networks have massive computational cost with each layer. The simulation was run for Heisenberg 5×5 lattice, with 6144 samples and 500 epochs. In the zoomed part of the figure we can see that there is still more potential to converge. Based on the figure 5.2.4 and the current one we can assume, that the ground state energy resides below value -109.

5.3 Excited state search

5.3.1 Heisenberg chain

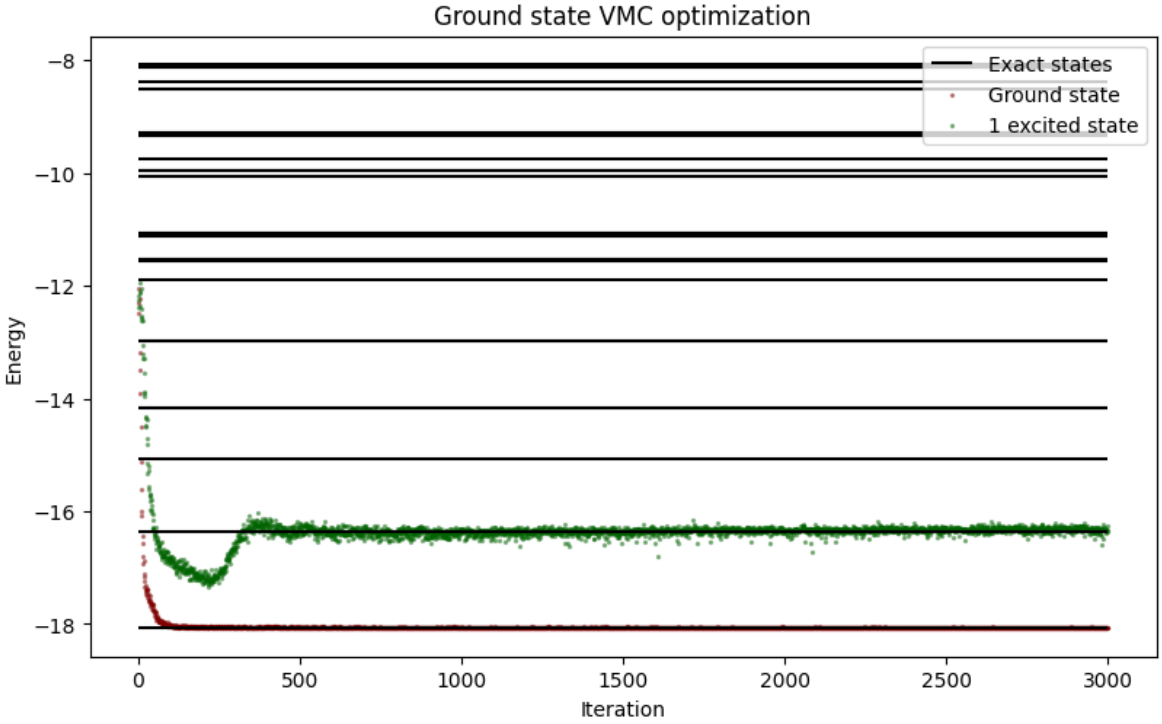

Figure 5.3.1

the figure 5.3.1 represents an attempt to find a first excited state using Penalty-based VMC method. The simulation was performed for Heisenberg chain with interaction strength equal to 1. The Ansatz used was restricted Boltzmann machine with expectation value calculated from 3000 samples. The optimization took 3000 epochs. Additionally, the penalty coefficient was provided, to push the expectation energy above the ground state.

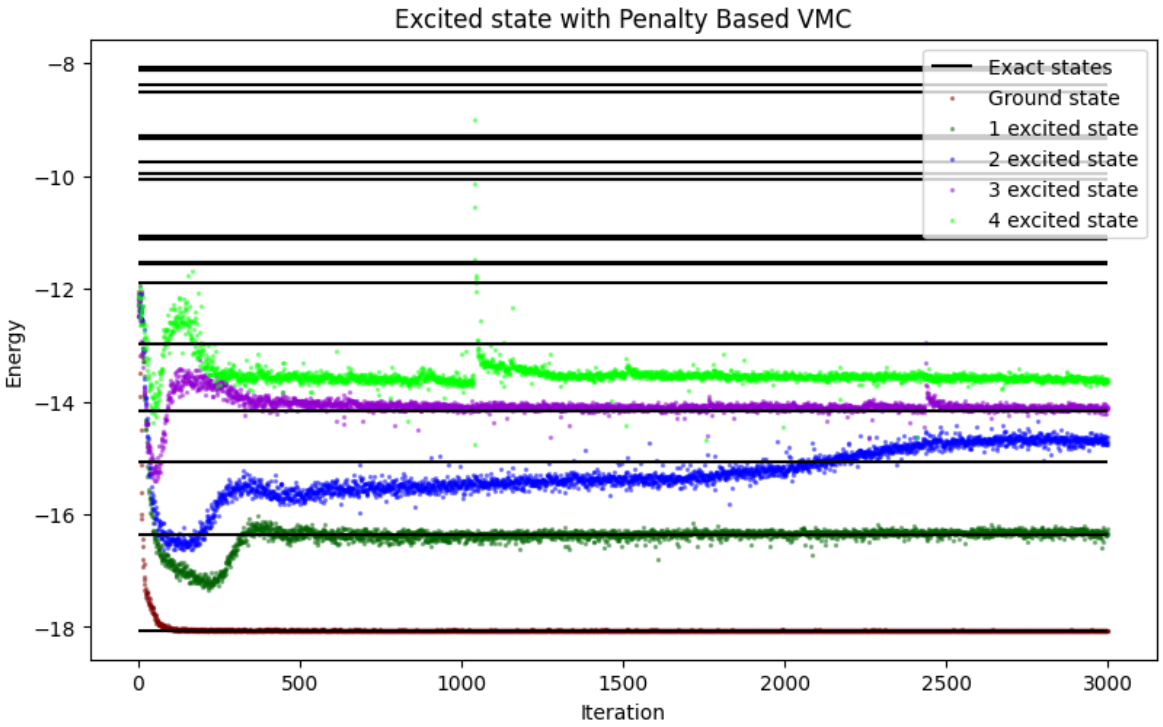

Figure 5.3.2

The figure shows continuation of the same model discussed in 5.3.1. Theoretically, each excited state can be calculated based on the eigenstates with lower energies. However, in reality the Penalty-based VMC has its limitations. The penalty coefficients have to be carefully adjusted, otherwise the expectation value will not converge in desired range.

6 Discussion

The results in this thesis provide significant insight into the application of machine learning techniques, particularly variational Monte Carlo methods, by solving complex quantum mechanical problems involving many-body spin systems. Using advanced computational tools such as NetKet, this research has demonstrated the potential of VMC to perform efficient computations. In this section we briefly discuss the results and potential further development.

The benchmarking process in figure 5.1.1 for extracting information on the most efficient number of independent Markov chains turns out to be ambiguous. We can clearly notice that for a very small number of chains the efficiency drops as well as for higher values. The best performance was run for chains of length 32, 48, 64. We can also notice that it is not dependent on the number of samples for estimating the energy. Overall, some further investigation should be done to exactly determine the most optimal values, such as ground state search for other models than Heisenberg.

Finding ground states for the longest possible Heisenberg chain yielded interesting results. In figure 5.1.2 we can see that each increase in the size of the system leads to a linear correlation between the energies of the ground state, where each value is lower by about a value of $\Delta E = 17$. This proves the accuracy of the VMC optimization; otherwise, the calculated energies would be chaotic and uncorrelated. The largest possible value to compute was $L = 100$, above this value the error occurred informing about too little resources.

Moving to RBM and DBM optimizations we can infer, that choosing the best performing model is non-trivial. Firstly, none of the models manage to perfectly converge and variate around true value. Each model found the value that was slightly above. However, deeper models with more parameters allowed us to get closer to the true energy. The best performing models were computed for $\alpha = 4$. Although wider and deeper networks offered similar performance, the additional computational time is not worth the effort. The peculiar case we can see in the figure 5.2.7, where for the model with three hidden layers the performance dropped and the optimization did not yield any direction to reduce the energy, so the model oscillated close to an incorrect value throughout the simulation. To choose the best hyperparameters, we can deduce that using GPU leads overall to a better performance, while for smaller networks they have comparable speeds. The GPU, however, allows us to estimate energy from more samples, which leads to a more robust convergence in a smaller number of epochs.

The Penalty-based VMC turns out to work well for some scenarios, while in some situations it led to wrong results. The problem is manual adjustment of the penalty coefficients, which have to be very well fine-tuned to yield correct results. We can deduce that in the formula discussed in this work, the penalty allows us to push the energy up in the landscape, but if other minima are not close enough, the VMC algorithm will find the balance between force pushing by penalty and natural gradient into the true ground state energy. To improve this, the algorithm in NetKet requires advanced development. The most important upgrade would be setting up the penalty coefficients to automatically adjust based on another metric.

In conclusion, this thesis has demonstrated the transformative potential of ma-

chine learning in quantum mechanics, providing a foundation for further advancements in quantum simulations and opening up new possibilities for practical applications.

References

[1] Mark Fox. *Quantum Optics. An Introduction*. Oxford University Press, 2006.

[2] David J. Griffiths. *Introduction to Quantum Mechanics*. Pearson Education, 2nd edition, 2005.

[3] Alexei Kitaev. Anyons in an exactly solved model and beyond. *Annals of Physics*, 321(1), 2006.

[4] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[5] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

[6] Kurt Binder and Dieter W. Heermann. *Monte Carlo Simulation in Statistical Physics: An Introduction*. Springer, 5th edition, 2010.

[7] W. M. C. Foulkes, L. Mitas, R. J. Needs, and G. Rajagopal. Quantum monte carlo simulations of solids. *Rev. Mod. Phys.*, 73:33–83, Jan 2001.

[8] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[9] Jonathan Heek, Anselm Levskaya, Avital Oliver, Marvin Ritter, Bertrand Rondepierre, Andreas Steiner, and Marc van Zee. Flax: A neural network library and ecosystem for JAX, 2023.

[10] Filippo Vicentini, Damian Hofmann, Attila Szabó, Dian Wu, Christopher Roth, Clemens Giuliani, Gabriel Pescia, Jannes Nys, Vladimir Vargas-Calderón, Nikita Astrakhantsev, and Giuseppe Carleo. NetKet 3: Machine Learning Toolbox for Many-Body Quantum Systems. *SciPost Phys. Codebases*, page 7, 2022.

[11] Wolfgang Pauli. On the connexion between the completion of electron groups in an atom with the complex structure of spectra. *Zeitschrift für Physik*, 31:765, 1925.

[12] Lev D. Landau and Evgeny M. Lifshitz. *Statistical Physics: Volume 5, Part 1*. Course of Theoretical Physics. Elsevier, 3rd edition, 2013.

[13] Ernst Ising. Contribution to the Theory of Ferromagnetism. *Z. Phys.*, 31:253–258, 1925.

[14] Igor W. Sawieliew. *Kurs Fizyki*. Państwowe Wydawnictwo Naukowe, 1989.

[15] H. E. Stanley. Dependence of critical properties on dimensionality of spins. *Phys. Rev. Lett.*, 20, Mar 1968.

[16] Xiaoguang Wang. Entanglement in the quantum heisenberg XY model. *Phys. Rev. A*, 64, Jun 2001.

[17] Werner Heisenberg. Quantum-theoretical re-interpretation of kinematic and mechanical relations. *Z. Phys*, 33:879–893, 1925.

[18] Paul M. Chaikin and Tom C. Lubensky. *Principles of Condensed Matter Physics*. Cambridge University Press, Cambridge, 1995.

[19] Giuseppe Carleo and Matthias Troyer. Solving the quantum many-body problem with artificial neural networks. *Science*, 355(6325):602–606, February 2017.

[20] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, Cambridge, MA, USA, 2016.

[21] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning: with Applications in R*. Springer, 2013.

[22] Frank Rosenblatt. *The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain*, volume 65 6. American Psychological Association, 1958.

[23] Sandro Sorella. Generalized lanczos algorithm for variational quantum monte carlo. *Physical Review B*, 64(2), June 2001.

[24] Victor Wei, Alev Orfi, Felix Fehse, and William A. Coish. Finding the dynamics of an integrable quantum many-body system via machine learning. *Advanced Physics Research*, 3(1), September 2023.

[25] Python Software Foundation. Python 3 documentation. `https://docs.python.org/3/`, 2024.

[26] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

[27] Jakub Grabowski. Machine learning-based solution of simple quantum problems, 2024. https://github.com/Yolonek/MLSolutionsForQuantumProblems.

[28] Herbert M. Reese. The zeeman effect. *Science*, 12, 1900.

[29] Masanao Ozawa. Heisenberg's original derivation of the uncertainty principle and its universally valid reformulations, 2020.

[30] R. Shankar. *Principles of Quantum Mechanics*. Springer, 2011.

[31] E. Schrödinger. An undulatory theory of the mechanics of atoms and molecules. *Physical Review*, 28:1049–1070, 1926.

[32] Hal Tasaki. Physics and mathematics of quantum many-body systems. *Graduate Texts in Physics*, 2020.

[33] Charles Kittel. *Introduction to Solid State Physics*. Wiley, 8th edition, 2004.

[34] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[35] Anders W. Sandvik, Adolfo Avella, and Ferdinando Mancini. Computational studies of quantum spin systems. In *AIP Conference Proceedings*. AIP, 2010.

[36] U. Schollwöck, J. Richter, D.J.J. Farnell, and R.F. Bishop. *Quantum Magnetism*, volume 645 of *Lecture Notes in Physics*. Springer, 2004.

[37] B. Andrei Bernevig and Taylor L. Hughes. *Topological Insulators and Topological Superconductors*. Princeton University Press, 2013.

[38] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2000.

[39] Rodney J. Baxter. *Exactly Solved Models in Statistical Mechanics*. Academic Press, London, 1982.

[40] David P. Landau and Kurt Binder. *A Guide to Monte Carlo Simulations in Statistical Physics*. Cambridge University Press, Cambridge, 3rd edition, 2009.

[41] R.K. Pathria and Paul D. Beale. *Statistical Mechanics*. Elsevier, Amsterdam, 3rd edition, 2011.

[42] W.R. Gilks, Sylvia Richardson, and D.J. Spiegelhalter. *Markov Chain Monte Carlo in Practice*. Chapman & Hall/CRC, London, 1995.

[43] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[44] Subir Sachdev. *Quantum Phase Transitions*. Cambridge University Press, Cambridge, 2nd edition, 2011.

[45] Julia Yeomans. *Statistical Mechanics of Phase Transitions*. Oxford University Press, Oxford, 1992.

[46] Assa Auerbach. *Interacting Electrons and Quantum Magnetism*. Springer, New York, 1994.

[47] Eduardo Fradkin. *Field Theories of Condensed Matter Physics*. Cambridge University Press, Cambridge, 2nd edition, 2013.

[48] Michael E. Peskin and Daniel V. Schroeder. *An Introduction to Quantum Field Theory*. Westview Press, Boulder, CO, 1995.

[49] L. D. Landau and E. M. Lifshitz. *Statistical Physics, Part 1*. Butterworth-Heinemann, Oxford, 3rd edition, 1980.

[50] F. Pzmndi and Z. Domaski. Quantum phase transitions in xy spin models. *Physical Review Letters*, 74:2363, 1995.

[51] K. Sengupta and N. Dupuis. Mott-insulator–to–superfluid transition in the bose-hubbard model: A strong-coupling approach. *Physical Review A*, 71(3), March 2005.

[52] Amit Dutta, Gabriel Aeppli, Bikas K. Chakrabarti, Uma Divakaran, Thomas F. Rosenbaum, and Diptiman Sen. Quantum phase transitions in transverse field spin models: from statistical physics to quantum information, 2015.

[53] Lucile Savary and Leon Balents. *Quantum Spin Liquids: A Review*, volume 80. IOP Publishing, 2017.

[54] Vladimir Cherepanov, Igor Kolokolov, and Victor S L'vov. *Saga of YIG: Spectra, Thermodynamics, Interaction and Relaxation of Magnons in a Complex Magnet*, volume 229. Elsevier, 1993.

[55] Yuji Ikeda. yuzie007/mpltern: 1.0.4, April 2024.

[56] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, New York, NY, USA, 2006.

[57] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, 1959.

[58] G.E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

[59] Douglas C. Montgomery, Elizabeth A. Peck, and G. Geoffrey Vining. *Introduction to Linear Regression Analysis*. John Wiley & Sons, Hoboken, NJ, USA, 5th edition, 2012.

[60] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[61] Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, 1997.

[62] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 1998.

[63] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[64] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners, 2019.

[65] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.

[66] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[67] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. *SIGARCH Comput. Archit. News*, 45(2), 2017.

[68] Liu Ziyin, Tilman Hartwig, and Masahito Ueda. Neural networks fail to learn periodic functions and how to fix it, 2020.

[69] James Stewart. *Calculus: Early Transcendentals*. Cengage Learning, Boston, MA, USA, 8th edition, 2015.

[70] Robert Tibshirani. *Regression Shrinkage and Selection via the Lasso*, volume 58 1. Wiley Online Library, 1996.

[71] Geoffrey E. Hinton. Training products of experts by minimizing contrastive divergence. *Neural Computation*, 14(8):1771–1800, 2002.

[72] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. In *Nature*, volume 323 6088, pages 533–536. Springer, 1986.

[73] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9 8:1735–1780, 1997.

[74] John J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79(8):2554–2558, 1982.

[75] David H. Ackley, Geoffrey E. Hinton, and Terrence J. Sejnowski. A learning algorithm for boltzmann machines. *Cognitive Science*, 9 1:147–169, 1985.

[76] Stuart Geman and Donald Geman. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-6:721–741, 1984.

[77] Attila Szabo and Neil S. Ostlund. *Modern Quantum Chemistry: Introduction to Advanced Electronic Structure Theory*. Dover Publications, Mineola, NY, 1996.

[78] Lev D. Landau and Evgeny M. Lifshitz. *Mechanics*, volume 1 of *Course of Theoretical Physics*. Butterworth-Heinemann, Oxford, UK, 3rd edition, 1976.

[79] Mark E. J. Newman and G. T. Barkema. *Monte Carlo Methods in Statistical Physics*. Oxford University Press, Oxford, 1999.

[80] Sandro Sorella. Wave function optimization in the variational monte carlo method. *Phys. Rev. B*, 71:241103, Jun 2005.

[81] Julien Toulouse and C. J. Umrigar. Optimization of quantum monte carlo wave functions by energy minimization. *The Journal of chemical physics*, 126 8:084102, 2007.

[82] Robert Jastrow. Many-body problem with strong forces. *Phys. Rev.*, 98:1479–1484, Jun 1955.

[83] Xun Gao and Lu-Ming Duan. Efficient representation of quantum many-body states with deep neural networks. *Nature Communications*, 8(1), September 2017.

[84] Or Sharir, Yoav Levine, Noam Wies, Giuseppe Carleo, and Amnon Shashua. Deep autoregressive models for the efficient variational simulation of many-body quantum systems. *Physical Review Letters*, 124(2), January 2020.

[85] Kenny Choo, Titus Neupert, and Giuseppe Carleo. Two-dimensional frustrated model studied with neural network quantum states. *Physical Review B*, 100(12), September 2019.

[86] Dmitrii Kochkov, Tobias Pfaff, Alvaro Sanchez-Gonzalez, Peter Battaglia, and Bryan K. Clark. Learning ground states of quantum hamiltonians with graph networks, 2021.

[87] Ingrid von Glehn, James S. Spencer, and David Pfau. A self-attention ansatz for ab-initio quantum chemistry, 2023.

[88] P. Bernát Szabó, Zeno Schätzle, Mike T. Entwistle, and Frank Noé. An improved penalty-based excited-state variational monte carlo approach with deep-learning ansatzes, 2024.

[89] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, pages 1–6, 2015.

[90] Daniel Snider and Ruofan Liang. Operator fusion in xla: Analysis and evaluation, 2023.

[91] Donald E. Knuth. *The Art of Computer Programming: Random Numbers*. Addison-Wesley Professional, 1997.

[92] Stephen Marsland. *Machine Learning: An Algorithmic Perspective*. Chapman and Hall/CRC, 2009.

[93] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, Sunnyvale, CA, USA, 1st edition, 2008.

[94] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks?, 2014.

[95] U. Schollwöck. The density-matrix renormalization group. *Reviews of Modern Physics*, 77(1):259–315, April 2005.

[96] Taco S. Cohen and Max Welling. Group equivariant convolutional networks, 2016.

[97] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. `https://networkx.github.io/`, 2008. Presented at the 7th Python in Science Conference (SciPy2008).