Solving Quantum Mechanical Problems with Machine Learning

by

Vilde Moe Flugsrud

Thesis

for the degree of

Master of Science

Faculty of Mathematics and Natural Sciences
University of Oslo

June 2018

Abstract

In this work we have developed from scratch formalism and software for using unsupervised machine learning methods to study interacting many-particle systems. We employ so-called reduced Boltzmann machines to construct trial wave functions for systems of bosons and fermions confined to move in various trapping potentials. Our results from machine learning agree excellently with standard variational Monte Carlo calculations, with expectation values like energies and energy variance being of the same quality. This opens up for several exciting explorations, in particular since the construction of trial wave functions used in Monte Carlo calculations is often complicated, both due to specific correlations and/or the fact that the analytical form of the trial wave function is difficult to obtain. Our Boltzmann machine trial wave function can easily be used as starting points in Green's function Monte Carlo calculations. The latter allow for in principle exact solutions of Schrödinger's equation.

Acknowledgements

I would like to thank Morten Hjort-Jensen for being an inspiring and helpful supervisor. Thank you for including me in your adventurous research spirit - I think I will remember one of your comments when considering this project for a long time "It will be a bit like throwing yourself into deep water. Personally, I love that."

I want to thank computational physics and Anders Malthe-Sørensen, who provided me and others with exciting summer research projects as first years students and subsequently allowed us to have an office space in your group. I want to thank my office (the Ministry of Silly Imports) mates of four years, Magnus, Alocias and Øyvind. It feels like we have grown up together, from wide-eyed second-year bachelor students to multiple degree- and job-juggling master students. I am lucky to have been around your knowledge, intelligence, enthusiasm, humor and warmth during this time.

I want to thank "FAMinistene" for being such good friends these past five years, demonstrating that a physics degree should not be feared for the lack of excellent girlfriends. Thank you for dinner nights, concerts, trips abroad, evenings at Realistforeningen, coffees at Deilig and Georg and so much more. I hope that some of these traditions will be kept even as we leave Blindern and that new ones will be created. Thank you Helene and Elisabeth for finding as much joy in Llamas as me - this is meant to summarize a lot more than meets the eye. Thank you Ingrid for your never-ending capacity for fun experiences, lots of humor and intense discussions be it about life or physics.

Thank you Sebastian for filling these past few years with memories worth a life-time, be it road tripping in Western Australia, crossing the Mediterranean Sea or climbing blue mountains, to mention a few. Thank you for all your love and support.

Finally, I would like to thank my family in general and in particular my mom, dad and brother Marius for always being there.

Contents

Chapter 1

Introduction

Quantum Computing and Machine Learning are two of the most promising approaches for studying complex physical systems where several length and energy scales are involved. Traditional many-particle methods, either quantum mechanical or classical ones, face huge dimensionality problems when applied to studies of systems with many interacting particles. To be able to define properly effective potentials for realistic Molecular Dynamics simulations of billions or more particles, requires both precise quantum mechanical studies as well as algorithms that allow for parametrizations and simplifications of quantum mechanical results. Quantum Computing offers now an interesting avenue, together with traditional algorithms, for studying complex quantum mechanical systems. Machine Learning on the other hand allows us to parametrize these results in terms of classical interactions. These interactions are in turn suitable for large scale Molecular Dynamics simulations of complicated systems spanning from subatomic physics to materials science and life science.

In addition, Machine Learning plays nowadays a central role in the analysis of large data sets in order to extract information about complicated correlations. This information is often difficult to obtain with traditional methods. For example, there are about one trillion web pages; more than one hour of video is uploaded to YouTube every second, amounting to 10 years of content every day; the genomes of 1000s of people, each of which has a length of 3.8×10^9 base pairs, have been sequenced by various labs and so on. This deluge of data calls for automated methods of data analysis, which is exactly what machine learning provides. Developing activities in these frontier computational technologies is thus of strategic importance for our capability to address future science problems.

Enabling simulations of large-scale many-body systems is a long-standing problem in scientific computing. Quantum many-body interactions define the structure of the universe, from nucleons and nuclei, to atoms, molecules, and even stars. Since the discovery of quantum mechanics, a lot of progress has been made in understanding the dynamics of certain many-body systems. While some of our insight comes from a small set of analytically solvable models, numerical simula-

tions have become a mainstay in our understanding of many-body dynamics. The progress in numerical simulations has accelerated in the last few decades with the advent of modern high performance computing and clever developments in classical simulation algorithms such as, quantum Monte Carlo,large-scale diagonalization approaches, Coupled-Cluster theory and other renormalization schemes. Despite the monumental advances, classical simulation techniques are reaching fundamental limits in terms of the size of the quantum systems that can be processed. Fortunately, new developments in the fields of quantum simulations and machine learning have emerged, promising to enable simulations far beyond those which are classically tractable.

The approaches to machine learning are many, but are often split into two main categories. In *supervised learning* we know the answer to a problem, and let the computer deduce the logic behind it. On the other hand, *unsupervised learning* is a method for finding patterns and relationship in data sets without any prior knowledge of the system. Some authours also operate with a third category, namely *reinforcement learning*. This is a paradigm of learning inspired by behavioural psychology, where learning is achieved by trial-and-error, solely from rewards and punishment. In this thesis, the aim is to explore new developments in the field of machine learning, with an emphasis on unsupervided learning. Much of the work here and its implementations is motivated by the recent article of Carleo and Troyer [1]. In particular we have extended their work, which focused on spin-like quantum mechanical systems, to systems of interacting bosons and fermions confined to move in trapping potentials, with the harmonic oscillator as one of the foremost examples. In this work, we will start with quantum Monte Carlo methods, with an emphasis on Variational Monte Carlo methods. This approach to studies of complicated interacting many-particle systems has been widely used in almost all fields of physics where first principle calculations are employed. It provides an important starting point for almost exact solutions to Schrödinger's equation for many interacting particles using so-called Green's function Monte Carlo methods [2].

A variational Monte Carlo (VMC) calculation is based on an ansatz for say the ground state wave function. For fermionic systems this ansatz is often composed of a single-particle part (via a so-called Slater determinant which acoounts for the anti-symmetry) and a correlated part, normally called the Jastrow factor. For bosonic systems there may also be a product function of single-particle functions and a Jastrow factor that aims at incorporating correlations beyond a mean field. These trial wave functions are thereafter used in an optimization procedure where various variational parameters are optmized in order to find a minimum for expectation values like the energy and the variance.

Constructing both the single-particle part and the Jastrow part can often be complicated and tedious. In systems like interacting nucleons, the correlation part of the wave function contains often complicated two- and three-body oper-

ators that require dedicated code developments. Similalry, for systems of atoms and molecules (and nucleons as well), the single-particle part is often constructed using mean-field methods like Hartree-Fock theory.

The aim here is to see whether methods inspired from Machine Learning can do an equally good job as the standard approach to VMC calculations, this time however with trial wave functions determined by neural networks. These trial wave functions, to be described below, are based on what in the literature is called Boltzmann machines. These functions contain several parameters which are used to find an energy minimum and thereby the optimal solution for the energy.

In this work we have developed from scratch code and formalism which allow do to this, including various sampling algorithms and testing different optimization methods. As we will show in this thesis, the results we obtain agree excellently (for various expectation values) with standard Variational Monte Carlo approaches. This holds great promise for future studies and explorations. Here we focus mainly on systems of bosons and fermions confined to move in oscillator traps since this allows us to benchmark against exisiting calculations. For two particles we even have analytical solutions for specific oscillator frequencies in two and three dimensions [3, 4]. Furthermore, without a two-body interaction we have also analytical results for many-particle systems in two and three dimensions with the harmonic oscillator as trapping potential.

This work represents one of the very first explorations of the exciting research area of machine learning techniques applied to quantum mechanical problems. The hope is that these techniques can represent a way to circumvent the standard exponential growth of degrees of freedom encountered in typical first principle many-body calculations [5]. This thesis provides a proof of principle in the sense that we show that machine learning techniques give results that compare well with standard VMC calculations. Another reason for having chosen the VMC approach is that this method employs many of the standard optmization methods used in machine learning.

A typical machine learning algorithm consists of three basic ingredients, a dataset $\mathbf{x}$ (could be some observable quantity of the system we are studying), a model which is a function of a set of parameters α that relates to the dataset, say a likelihood function $p(\mathbf{x}|\alpha)$ or just a simple model $f(\alpha)$, and finally a so-called *cost* function $\mathcal{C}(\mathbf{x}, f(\alpha))$ which allows us to decide how well our model represents the dataset.

We seek to minimize the function $\mathcal{C}(\mathbf{x}, f(\alpha))$ by finding the parameter values which minimize $\mathcal{C}$. Thus, VMC calculations serve both as input to exact solutions via Green's functions methods and employ similar optimization approaches as employed in machine learning. A detailed discussion of various optmization methods is also given in this report.

After these introductory words, we present in the subsequent chapter some

of the basic ingredients of Variational Monte Carlo calculations. This chapter serves as a bridge between one of the standard many-body approaches (VMC in our case) and the machine learning algorithms discussed here. In the subsequent chapters we give a discussion of various machine learning algorithms with an emphasis on unsupervised learning. Thereafter, we present our implementation, tests we have developed and our final results. The results of this work have been submitted for publication. Our last chapter sums up our findings and presents several exciting perspectives for future work.

Part I

Theory

Chapter 2

The Quantum Many-Body Problem

2.1 Many-Body Quantum Mechanics

Quantum mechanics describes systems at the size of atoms and subatomic particles at speeds where relativistic effects are negligible. While in classical mechanics the state of a system of N particles is given by each particle's position and momentum, in quantum mechanics it is determined by the complex valued wavefunction Ψ, which is an element of an infinite dimensional Hilbert space. That is, a complete vector space with an inner product.[6]

Given a system's wavefunction we can calculate all physical quantities of interest[6]. We call these physical quantities observables and in quantum mechanics they are represented by hermitian operators. Observables are interesting because they are measurable in an experiment, hence the name "observable". The expectation value of an operator $\hat{\boldsymbol{O}}$ for a system of N particles is given as [7]

$$\langle \hat{\boldsymbol{O}} \rangle = \frac{\int \Psi^*(\boldsymbol{x}_1, ..., \boldsymbol{x}_N) \hat{\boldsymbol{O}}(\boldsymbol{x}_1, ..., \boldsymbol{x}_N) \Psi(\boldsymbol{x}_1, ..., \boldsymbol{x}_N) \, \mathrm{d}\boldsymbol{x}_1 ... \, \mathrm{d}\boldsymbol{x}_N}{\int \Psi^*(\boldsymbol{x}_1, ..., \boldsymbol{x}_N) \Psi(\boldsymbol{x}_1, ..., \boldsymbol{x}_N) \, \mathrm{d}\boldsymbol{x}_1 ... \, \mathrm{d}\boldsymbol{x}_N}. \tag{2.1}$$

For a system of N particles in three spatial dimensions we see that this becomes a $3N$-dimensional integral. Furthermore, in order to compute it, we first need to have Ψ. This requires us, if we are interested in the ground state energy, to solve the time-independent Schrödinger equation

$$\hat{\boldsymbol{H}} \Psi = E \Psi. \tag{2.2}$$

For a complicated many body problem this can turn out to be a problem consisting of millions of coupled second-order differential equations in $3N$ dimensions.

The wavefunction Ψ must satisfy [7] the normalization condition

$$\int_{-\infty}^{\infty} P(x,t) \, \mathrm{d}x = \int_{-\infty}^{\infty} \Psi^*(x,t) \Psi(x,t) \, \mathrm{d}x = 1. \tag{2.3}$$

Furthermore $\Psi(x,t)$ and $\partial \Psi(x,t)/\partial x$ must be finite, continuous and single-valued.

2.2 Quantum many-body methods

A theoretical understanding of the behavior of quantum mechanical systems with many interacting particles, normally called many-body systems, is a great challenge and provides fundamental insights into systems governed by quantum mechanics, as well as offering potential areas of industrial applications, from semiconductor physics to the construction of quantum gates. The ability to simulate quantum mechanical systems with many interacting particles is crucial for advances in such rapidly developing fields like materials science.

However, most quantum mechanical systems of interest in physics consist of a large number of interacting particles. The total number of particles N is usually sufficiently large that an exact solution (i.e., in closed form) cannot be found. One needs therefore reliable numerical methods for studying quantum mechanical systems with many particles.

Studies of many-body systems span from examinations of the strong force with quarks and gluons as degrees of freedom, the spectacular macroscopic manifestations of quantal phenomena such as Bose-Einstein condensation with millions of atoms forming a single coherent state, to properties of new materials, with electrons as effective degrees of freedom. The length scales range from few micrometers and nanometers, typical scales met in materials science, to $10^{-15} - 10^{-18}$ m, a relevant length scale for the strong interaction. Energies can span from few meV to GeV or even TeV. In some cases the basic interaction between the interacting particles is well-known. A good example is the Coulomb force, familiar from studies of atoms, molecules and condensed matter physics. In other cases, such as for the strong interaction between neutrons and protons (commonly dubbed as nucleons) or dense quantum liquids or molecular dynamics simulations one has to resort to parameterizations of the underlying interparticle interactions. The system can also span over much larger dimensions as well, with neutron stars as one of the typical examples. A neutron star is the endpoint of massive stars which have used up their fuel. As the name suggests, a neutron star is composed mainly of neutrons, with a small fraction of protons and probably quarks in its inner parts. The star is extremely dense and compact, with a radius of approximately 10 km and a mass which is roughly 1.5 times that of our sun. The quantum mechanical pressure which is set up by the interacting particles counteracts the gravitational forces, hindering thus a gravitational collapse. To describe a neutron star one needs to solve Schrödinger's equation for approximately 10^{54} interacting particles!

With a given interparticle potential and the kinetic energy of the system, one can in turn define the so-called many-particle Hamiltonian $\hat{H}$ which enters the solution of Schrödinger's equation or Dirac's equation in case relativistic effects need to be included. For many particles, Schrödinger's equation is an integro-differential equation whose complexity increases exponentially with increasing

numbers of particles and states that the system can access. Unfortunately, apart from some few analytically solvable problems and one and two-particle systems that can be treated numerically exactly via the solution of sets of partial differential equations, the typical absence of an exactly solvable (on closed form) contribution to the many-particle Hamiltonian means that we need reliable numerical many-body methods. These methods should allow for controlled approximations and provide a computational scheme which accounts for successive many-body corrections in a systematic way. Typical examples of popular many-body methods are coupled-cluster methods, various types of Monte Carlo methods, perturbative many-body, Green's function methods, the density-matrix renormalization group, density functional theory and ab initio density functional theory, and large-scale diagonalization methods, just to mention a few. The physics of the system hints at which many-body methods to use. For systems with strong correlations among the constituents, methods based on mean-field theory such as Hartree-Fock theory and density functional theory are normally ruled out. This applies also to perturbative methods, unless one can renormalize the parts of the interaction which cause problems.

As previously noted, solving the Schrödinger equation (SE) exactly by hand is impossible in the overwhelming majority of interesting cases. However, methods which can get close to the exact solution exists. Full Configuration Interaction (FCI) or direct diagonalization of the Hamiltonian is exact in the limit of an in nite orbital basis set but suffers from an exponential complexity scaling (in system and basis size) [8]. The related Configuration Interaction (CI) and Coupled Cluster (CC) approaches both truncate the FCI expansion of Slater determinants, thus gaining speed but loosing some accuracy [6, 2].

Diffusion Monte Carlo (DMC) techniques can in principle provide the exact solution to the SE by imaginary-time evolution of an initial trial wave function [7, 2]. In practice, DMC methods are highly dependent on this ansatz and thus require as input the results of less accurate method but faster methods. One example may be the Variational Monte Carlo (VMC) method: conceptually simpler and faster than DMC, but not as accurate [2]. The Hartree-Fock (HF) framework—which provides an efficient but not enormously accurate result—has seen extensive use since its inception in 1930.

2.3 Variational Monte Carlo

2.3.1 The Variational Principle

The variational theorem states, see for example [6] and [8], that given a *trial state* $|\Psi_T\rangle$, the following inequality holds:

$$E_T = \frac{\langle\Psi_T|\hat{H}|\Psi_T\rangle}{\langle\Psi_T|\Psi_T\rangle} \geq E_0, \tag{2.4}$$

where E_0 is the ground state eigenvalue of the Hamiltonian $\hat{H}$, and E_T is the trial eigenvalue. The variational principle holds for the ground state, but also for excited states, provided that $|\Psi_T\rangle$ is orthogonal to all the eigenstates having eigenvalue lower than that of the state one wants to approximate [5]

2.3.2 Monte Carlo Integration

Our discussion of machine learning algorithms follows closely the philosophy of variational Monte Carlo (VMC) method. We review here some of the basic ingredients included in a VMC calculation. We give also a brief review of central elements from statistics and probability theory for the mere sake of completeness.

Variance and Covariance

For a continuous random variable X distributed according to the probability density $p(X)$ the nth **moment** of X is defined as

$$\langle X^n\rangle = \int p(x)x^n\,\mathrm{d}x, \tag{2.5}$$

where we notice that the zeroth moment recovers the **normalization condition** of the total probability,

$$1 = \int p(x)\,\mathrm{d}x, \tag{2.6}$$

and the first moment recovers the definition of the **expectation value** of X,

$$\mu = E(X) = \langle X\rangle = \int p(x)x\,\mathrm{d}x. \tag{2.7}$$

Furthermore we have that the **central moments** of X are defined

$$\langle(X-\mu)^n\rangle = \int p(x)(x-\mu)^n\,\mathrm{d}x. \tag{2.8}$$

Of particular interest here is the second central moment, which is what we know as the **variance** of X

$$\sigma^2 = var(X) = \langle (X-\mu)^2 \rangle = \int p(x)(x-\mu)^2 \, \mathrm{d}x \tag{2.9}$$

$$= \int p(x)(x^2 - 2x\langle X \rangle + \langle X \rangle^2) \, \mathrm{d}x \tag{2.10}$$

$$= \langle X^2 \rangle - 2\langle X \rangle \langle X \rangle + \langle X \rangle^2 \tag{2.11}$$

$$= \langle X^2 \rangle - \langle X \rangle^2. \tag{2.12}$$

The square root of the variance, σ, is called the **standard deviation** or **standard error**.

For two random variables X and Y we have that the expectation of a linear combination of them is equal to a linear combination of their expectations, $E(\lambda_1 X + \lambda_2 Y) = \lambda_1 E(X) + \lambda_2 E(Y)$. This is true whether X and Y are dependent or independent. In the variance however we see a difference between these two cases. We have in general that

$$var(\lambda_1 X + \lambda_2 Y) = \lambda_1^2 var(X) + \lambda_2^2 var(Y) + 2\lambda_1\lambda_2(\langle XY \rangle - \langle X \rangle \langle Y \rangle) \tag{2.13}$$

$$= \lambda_1^2 var(X) + \lambda_2^2 var(Y) + 2\lambda_1\lambda_2 cov(X,Y). \tag{2.14}$$

The **covariance** $cov(X,Y) = \langle XY \rangle - \langle X \rangle \langle Y \rangle$ measures the degree of independence between the two random variables. This is related to the probability of independent random events. If the two random variables are **independent**, we have that their joint probability can be written as a product of their respective probability distributions,

$$p(X,Y) = p(X)p(Y). \tag{2.15}$$

Hence the expectation of the product XY is

$$\langle XY \rangle = \int p(x,y)xy \, \mathrm{d}x \, \mathrm{d}y = \int p(x)x \, \mathrm{d}x \int p(y)y \, \mathrm{d}y = \langle X \rangle \langle Y \rangle. \tag{2.16}$$

We see from this that if the two random variables are independent, $cov(X,Y) = 0$. Zero covariance by itself does not, however, guarantee independence.

Estimators

Suppose the variables $X_1, X_2, ...$ are drawn randomly, but not necessarily independently, from the probability distribution function $p(X)$. Let g be a function of X and define the function G_N by

$$G_N = \frac{1}{N} \sum_{i=1}^{N} g(x_i). \tag{2.17}$$

The expected value of G_N is then

$$\langle G_N \rangle = \langle \frac{1}{N} \sum_i^N g(X_i) \rangle = \frac{1}{N} \sum_i^N \langle g(X) \rangle = \langle g(X) \rangle, \tag{2.18}$$

where G_N is the arithmetic average of the samples $g(X_i)$ and has the same expectation value as $g(X)$. G_N is said to be an **estimator** of $\langle g(X) \rangle$.

If all the X_i are independent, the variance of G_N is

$$var(G) = var(\frac{1}{N} \sum_i^N g(X_i)) = \sum_i^N \frac{1}{N^2} var(g(X)) = \frac{1}{N} var(g(X)). \tag{2.19}$$

The implication of this is that as the number of samples of X, N, increases, the variance of the mean value of G_N decreases as $\frac{1}{N}$. This is a core idea of Monte Carlo integration. That is, we may estimate an integral with a sum since

$$\langle g(X) \rangle = \int_{-\infty}^{\infty} p(x) g(x) \, \mathrm{d}x = \langle \frac{1}{N} \sum_{i=1}^N g(X_i) \rangle, \tag{2.20}$$

or stated differently, we can approximate the integral of a function g by

$$\int g(x) \, \mathrm{d}x = \int g(x) \frac{p(x)}{p(x)} \, \mathrm{d}x = \int \frac{g(x)}{p(x)} p(x) \, \mathrm{d}x = \langle \frac{g(X)}{p(X)} \rangle \tag{2.21}$$

$$= \langle \frac{1}{N} \sum_{i=1}^N \frac{g(X_i)}{p(X_i)} \rangle. \tag{2.22}$$

The next question that arises is how the sample average G_N approaches the expected value as N increases.

Convergence of the Estimator

We here present three results in statistics which provide information about the convergence of the estimator G_N.[9]

1. **The Law of Large Numbers**: Suppose we have independent, identically distributed (i.i.d.) random variables $X_1, ..., X_N$. The expectation of each X is then μ. As $N \to \infty$, the mean value of the $\{X_i\}$,

$$\bar{X}_N = \frac{1}{N} \sum_{i=1}^N X_i \tag{2.23}$$

is *almost sure to converge* to μ, meaning

$$p(\lim_{N \to \infty} \bar{X}_N = \mu) = 1. \tag{2.24}$$

2. **The Chebychev Inequality**: To estimate the *speed* of convergence we must make stronger assumptions. We assume an estimator G_N, its mean $\langle G_N \rangle$ and variance $var(G_N)$ all exist. The Chebychev inequality then is

$$p\left(|G_N - \langle G_N \rangle| \geq \sqrt{\frac{var(G_N)}{\delta}} = \sqrt{\frac{var(g)}{\delta N}} \right) \leq \delta, \tag{2.25}$$

with δ any postive number. By making N big, we can make the variance of G_N as small as we want, and the probability of the estimate differing from the true value by a large deviation relative to δ becomes small. This is at the core of the Monte Carlo method for evaluating integrals.

3. **The Central Limit Theorem**: This theorem makes a much stronger statement about the possible values of G_N than the Chebychev inequality. For any fixed value of N, the values of G_N are described by some probability distribution function. The central limit theorem shows that as $N \to \infty$ there is a specific limit distribution. That is, the **normal distribution**, specified by

$$p(G_N) = \frac{1}{\sqrt{2\pi \cdot var(G_N)}} e^{\frac{(G_N - \langle G_N \rangle)^2}{2 \cdot var(G_N)}} = \frac{1}{\sqrt{2\pi \frac{\sigma^2}{N}}} e^{\frac{N(G_N - \langle g \rangle)^2}{2\sigma^2}}, \tag{2.26}$$

where $\sigma^2 = var(g)$, the variance of g. As $N \to \infty$ the values of G_N occurs ever closer to $\langle g \rangle$. Given the standard deviation $\sigma/\sqrt{N}$, we have that the values of G_N are within one standard deviation 68.3% of the time, within two standard deviations 95.4% of the time, and within three standard deviations 99.7% of the time. The central limit theorem is very powerful in that it gives a specific distribution for the values of G_N, but it applies only asymptotically. How large N must be before the central limit theorem applies depends on the problem.

From these results it is clear that the estimator converges as N grows. While it is problem-dependent at what N the central limit theorem applies, one may always use the weaker upper bound of the Chebychev Inequality to suggest how much the estimator deviates from the true mean.

Estimating the variance and the standard error

We may estimate the variance $\sigma^2 = var(g(X))$ using independent values of $g(X_i)$ by

$$s^2 = \frac{1}{1-N} \sum_i^N (g(X_i) - G_N)^2 = \frac{N}{N-1} (\frac{1}{N} \sum_{i=1}^N g^2(X_i) - G_N^2), \tag{2.27}$$

where we have that $E(s^2) = \sigma^2$, which means that s^2 is an unbiased estimator for σ^2. From this we also have an estimator of the variance of the estimated mean

$$var(G_N) = \frac{1}{N}\sigma^2 \approx \frac{1}{N}s^2. \tag{2.28}$$

From these we may also estimate the standard error as $\sigma \approx s$ and $std(G_N) \approx s/\sqrt{N}$ respectively. While $E(s^2) = \sigma^2$ it does not imply $E(s) = \sigma$, hence s is not an unbiased estimator for σ. However it is a good approximation when the sample size N is large.

Local Energy

The remaining question is how to relate a quantum mechanical observable such as the energy of the system to a Monte Carlo estimator. If Ψ represents the unnormalized wavefunction, we have that [10]

$$\begin{aligned}
E =& \frac{\langle\Psi|\hat{H}|\Psi\rangle}{\langle\Psi|\Psi\rangle} && (2.29)\\
=& \frac{\int \Psi^*(\boldsymbol{R})\hat{H}\Psi(\boldsymbol{R})\,\mathrm{d}\boldsymbol{R}}{\int \Psi^*(\boldsymbol{R})\Psi(\boldsymbol{R})\,\mathrm{d}\boldsymbol{R}} && (2.30)\\
=& \frac{\int \Psi^*(\boldsymbol{R})(\Psi(\boldsymbol{R})\frac{1}{\Psi(\boldsymbol{R})})\hat{H}\Psi(\boldsymbol{R})\,\mathrm{d}\boldsymbol{R}}{\int |\Psi(\boldsymbol{R})|^2\,\mathrm{d}\boldsymbol{R}} && (2.31)\\
=& \frac{\int \Psi^*(\boldsymbol{R})\Psi(\boldsymbol{R})\frac{\hat{H}\Psi(\boldsymbol{R})}{\Psi(\boldsymbol{R})}\,\mathrm{d}\boldsymbol{R}}{\int |\Psi(\boldsymbol{R})|^2\,\mathrm{d}\boldsymbol{R}} && (2.32)\\
=& \frac{\int |\Psi(\boldsymbol{R})|^2 E_L(\boldsymbol{R})\,\mathrm{d}\boldsymbol{R}}{\int |\Psi(\boldsymbol{R})|^2\,\mathrm{d}\boldsymbol{R}} && (2.33)\\
=& \int p(\boldsymbol{R})E_L(\boldsymbol{R})\,\mathrm{d}\boldsymbol{R}, && (2.34)
\end{aligned}$$

where the **local energy** is defined $E_L = \frac{\hat{H}\Psi(\boldsymbol{R})}{\Psi(\boldsymbol{R})}$ and $p(\boldsymbol{R}) = \frac{|\Psi(\boldsymbol{R})|^2}{\int |\Psi(\boldsymbol{R})|^2\,\mathrm{d}\boldsymbol{R}}$ is the normalized probability density function given by the squared absolute wave function.

We then recognize that we can use the following quantity as our Monte Carlo estimator

$$\frac{1}{N}\sum_{i=1}^{N} E_L(\boldsymbol{R}_i), \tag{2.35}$$

with $\boldsymbol{R_i}$ sampled from $p(\boldsymbol{R})$ to approximate, as $N \to \infty$,

$$\langle\frac{1}{N}\sum_{i=1}^{N} E_L(\boldsymbol{R}_i)\rangle = \int p(\boldsymbol{R})E_L(\boldsymbol{R})\,\mathrm{d}\boldsymbol{R} = E. \tag{2.36}$$

2.3.3 Markov Chains

Monte Carlo integration requires us to sample from the probability distribution of interest. However we are not able to do this directly when the normalization constant is intractable, as is usually the case with the distribution $p(\boldsymbol{R}) = \frac{|\Psi(\boldsymbol{R})|^2}{\int |\Psi(\boldsymbol{R})|^2 \, \mathrm{d}\boldsymbol{R}}$. In order to sample an intractable distribution we use Markov Chain Monte Carlo (MCMC) methods. This section introduces Markov chains. A Markov chain is a type of Markov process, which in turn is a stochastic process.

Stochastic Processes

A **stochastic process** is a stochastic quantity Y that can be mapped from a stochastic variable X with a function f, and that also depends on another "normal" variable t, which usually represents time. That is,

$$Y_X(t) = f(X, t). \tag{2.37}$$

If X is described by the distribution p_X, the probability density for $Y_X(t)$ to take any value y at time t is then

$$p(y, t) = \int \delta(y - Y_X(t)) p_X(x) \, \mathrm{d}x. \tag{2.38}$$

Similarly, the joint probability density that Y has the value y_1 at t_1, and also y_2 at t_2 and so on till y_n at t_n is

$$p(y_1, y_1; y_2, t_2; ...; y_n, t_n) = \int \delta(y_1 - Y_X(t_1)) ... \delta(y_n - Y_X(t_n)) p_X(x) \, \mathrm{d}x. \tag{2.39}$$

This way a hierarchy of probability densities is defined. Any set of probability density functions p that obey the following four **consistency conditions** determine a stochastic process:

1. $p \geq 0$,
2. p does not change on interchanging two pairs (y_k, t_k) and (y_l, t_l),
3. $\int p(y_1, t_1; ...; y_n, t_n) \, \mathrm{d}y_n = p(y_1, t_1; ...; y_{n-1}, t_{n-1})$,
4. $\int p(y_1, t_1) \, \mathrm{d}y_1 = 1$.

Markov Processes

A **Markov process** is a stochastic process that has the **Markov property**. The Markov property is that the next state is dependent only on the current state. That is, for any set of n successive times (i.e. $t_1 < t_2 < t_3$) one has

$$p(y_n, t_n | y_1, t_1; ...; y_{n-1}, t_{n-1}) = p(y_n, t_n | y_{n-1}, t_{n-1}), \tag{2.40}$$

meaning that the conditional probability density at t_n, given the value y_{n-1} at t_{n-1}, is uniquely determined and not affected by any knowledge of the values at earlier times. $p(y_n, t_n|y_{n-1}, t_{n-1})$ is then known as the **transition probability**.

A Markov Process is thus fully determined by two functions: the initial probability distribution $p(y_1, t_1)$ and the transition probability $p(y_2, t_2|y_1, t_1)$. From this one can use the definition of conditional probability to successively construct the probability of all states, for example

$$p(y_2, t_2; y_1, t_1) = p(y_2, t_2|y_1, t_1)p(y_1, t_1), \tag{2.41}$$

and

$$\begin{aligned} p(y_3, t_3; y_2, t_2; y_1, t_1) =& p(y_3, t_3|y_2, t_2; y_1, t_1)p(y_2, t_2; y_1, t_1) && (2.42)\\ =& p(y_3, t_3|y_2, t_2)p(y_2, t_2; y_1, t_1), && (2.43) \end{aligned}$$

and so on. The two functions cannot be chosen arbitrarily, but have to obey two consistency conditions. Any two non-negative functions that obey these conditions uniquely determines a Markov Process. The conditions are

1. The **Chapman-Kolmogorov equation** which, given $t_1 < t_2 < t_3$, is the identity

$$p(y_3, t_3|y_1, t_1) = \int p(y_3, t_3|y_2, t_2)p(y_2, t_2|y_1, t_1)\,\mathrm{d}y_2, \tag{2.44}$$

2. $p(y_2, t_2) = \int p(y_2, t_2|y_1, t_1)p(y_1, t_1)\,\mathrm{d}y_1$.

Markov Chains

Markov chains are an especially simple class of Markov processes where the range of Y is a discrete set of states and the time variable only takes integer values. The initial and transition probabilities can then be written

$$\begin{aligned} p(y_1, t_1) =& p(y_1), && (2.45)\\ p(y_n, t_n|y_{n-1}, t_{n-1}) =& p(y_n|y_{n-1}), && (2.46) \end{aligned}$$

such that the time is now indicated by the index n.

If the Markov chain is **finite** the range of Y consists of a finite number of states N. In this case the initial probability distribution $p(y_1)$ is an N-component vector and the transition probability $p(y_n|y_{n-1})$ is an $N \times N$ matrix. It is a **stochastic matrix** (also called a **Markov matrix**), that is

1. Its elements are non-negative $p(y_n|y_{n-1}) \geq 0$,
2. Each column adds up to unity, that is $\sum_{y_n} p(y_n|y_{n-1}) = 1$.

The eigenvalues of a stochastic matrix range between 0 and 1.

The interested reader will find a comprehensive introduction to stochastic processes and Markov chains, and their use in physics and chemistry, in reference [11].

2.3.4 The Metropolis-Hastings Algorithm

The previous section gave an introduction to Markov chains, but we have not explained how to use them to generate samples $\boldsymbol{R}$ from an intractable distribution $p(\boldsymbol{R}) = \frac{|\Psi(\boldsymbol{R})|^2}{\int |\Psi(\boldsymbol{R})|^2 \, \mathrm{d}\boldsymbol{R}}$ in order to carry out Monte Carlo integration. The Metropolis algorithm provides a practical way of using Markov chains for this purpose and has been named one of the ten most influential algorithms for science and engineering in the 20th century. We will introduce it in this section.

The central idea is the fact that it is possible to construct a Markov chain which converges to a given probability distribution. If the aim is to generate samples $\boldsymbol{R}$ according to a probability distribution $\rho(\boldsymbol{R})$, we construct a Markov chain which converges to this distribution. One achieves this by imposing several conditions on the transition probability matrix $p(\boldsymbol{R}_f|\boldsymbol{R}_i)$ (i denotes initial state and f denotes final state after a transition) of the Markov chain.

Ergodicity and Random Walks

First, we must require that if we sample from the desired distribution $\rho(\boldsymbol{R})$, we will continue to sample from it. This is called the **stationary condition** and is expressed as

$$\sum_i p(\boldsymbol{R}_f|\boldsymbol{R}_i)\rho(\boldsymbol{R}_i) = \rho(\boldsymbol{R}_f) = \sum_i p(\boldsymbol{R}_i|\boldsymbol{R}_f)\rho(\boldsymbol{R}_f) \tag{2.47}$$

for all states $\boldsymbol{R}_f$ (the second equality simply follows from the column normalization that was required for a stochastic matrix). This condition is fulfilled if the desired distribution $\rho(\boldsymbol{R})$ is a (right) eigenvector of the transition matrix with eigenvalue 1 (recall that a stochastic matrix has eigenvalues in the range 0 to 1).

Next, we require that *any* initial distribution $p(\boldsymbol{R}_1)$ should evolve to the target distribution $\rho(\boldsymbol{R})$ after repeated application of the transition probability matrix, i.e.

$$\lim_{M\to\infty} \sum_{\boldsymbol{R}_1} p^M(\boldsymbol{R}|\boldsymbol{R}_1)p(\boldsymbol{R}_1) = \rho(\boldsymbol{R}). \tag{2.48}$$

This means that $\rho(\boldsymbol{R})$ must be the **dominant** eigenvector of the transition matrix. When the stationary condition is fulfilled, this is the case except if there are other eigenvectors with eigenvalue 1. The transition matrix only has one eigenvector of eigenvalue 1 if it is **primitive**. That is, there is an integer $n \geq 1$ such that $p^n(\boldsymbol{R}_f|\boldsymbol{R}_i) > 0, \forall \boldsymbol{R}_f, \boldsymbol{R}_i$. The Markov chain is then said to be **ergodic** [10]. This property ensures that the chain is aperiodic (it does not return to the same state at fixed intervals) and positive recurrent (the expected number of steps for returning to the same state is finite). It means it is possible to move between any pair of states $\boldsymbol{R}_i$ and $\boldsymbol{R}_f$ in n steps.

We realize the Markov chain by a **random walk**. Starting from an initial point $\boldsymbol{R}_1$ the initial probability is given by $p(\boldsymbol{R}_1) = \delta(\boldsymbol{R} - \boldsymbol{R}_1)$. We sample the second point $\boldsymbol{R}_2$ by drawing from the probability distribution $P(\boldsymbol{R}_2|\boldsymbol{R}_1)$, the third point $\boldsymbol{R}_3$ by drawing from $P(\boldsymbol{R}_3|\boldsymbol{R}_2)$, and so on. After a number of convergence steps referred to as the **equilibration** time, the random walk samples the desired distribution $\rho(\boldsymbol{R})$. That is,

$$\rho(\boldsymbol{R}) = E[\delta(\boldsymbol{R} - \boldsymbol{R}_k)] \approx \frac{1}{M}\sum_{k=1}^{M} \delta(\boldsymbol{R} - \boldsymbol{R}_k). \tag{2.49}$$

Thus, one commonly makes an estimation of the equlibration time, that is the number of steps before the stationary distribution is reached, and exclude these first samples from the computation of averages of the estimators of interest.

The Metropolis-Hastings Algorithm

In the derivation of the Metropolis algorithm we begin by applying a condition that is stronger than the stationary condition, called the **detailed balance condition**. It is sufficient, but not necessary. The Markov chain is then called **reversible** because we require that for every pair of states $\boldsymbol{R}_i$, $\boldsymbol{R}_f$ the probability of transitioning from one to the other must be the same as for the reverse transition. That is, the detailed balance condition [12] requires

$$p(\boldsymbol{R}_f|\boldsymbol{R}_i)\rho(\boldsymbol{R}_i) = p(\boldsymbol{R}_i|\boldsymbol{R}_f)\rho(\boldsymbol{R}_f). \tag{2.50}$$

We write the transition probability matrix as a product of a proposal (attempted transition) matrix $\boldsymbol{Q}$ and an acceptance matrix $\boldsymbol{A}$,

$$p(\boldsymbol{R}_f|\boldsymbol{R}_i) = \boldsymbol{A}(\boldsymbol{R}_f|\boldsymbol{R}_i)\boldsymbol{Q}(\boldsymbol{R}_f|\boldsymbol{R}_i), \tag{2.51}$$

where $\boldsymbol{Q}$ is a stochastic matrix like the transition probability. Inserting this back into the detailed balance condition yields

$$\boldsymbol{A}(\boldsymbol{R}_f|\boldsymbol{R}_i)\boldsymbol{Q}(\boldsymbol{R}_f|\boldsymbol{R}_i)\rho(\boldsymbol{R}_i) = \boldsymbol{A}(\boldsymbol{R}_i|\boldsymbol{R}_f)\boldsymbol{Q}(\boldsymbol{R}_i|\boldsymbol{R}_f)\rho(\boldsymbol{R}_f) \tag{2.52}$$

$$\frac{\boldsymbol{A}(\boldsymbol{R}_f|\boldsymbol{R}_i)}{\boldsymbol{A}(\boldsymbol{R}_i|\boldsymbol{R}_f)} = \frac{\boldsymbol{Q}(\boldsymbol{R}_i|\boldsymbol{R}_f)\rho(\boldsymbol{R}_f)}{\boldsymbol{Q}(\boldsymbol{R}_f|\boldsymbol{R}_i)\rho(\boldsymbol{R}_i)} \tag{2.53}$$

The choice of $\boldsymbol{A}$ which maximize the acceptance probability is the choice proposed by Metropolis *et al.* [13], given by

$$\boldsymbol{A}(\boldsymbol{R}_f|\boldsymbol{R}_i) = \min\Big(1, \frac{\boldsymbol{Q}(\boldsymbol{R}_i|\boldsymbol{R}_f)\rho(\boldsymbol{R}_f)}{\boldsymbol{Q}(\boldsymbol{R}_f|\boldsymbol{R}_i)\rho(\boldsymbol{R}_i)}\Big). \tag{2.54}$$

The Metropolis algorithm for moving a random walk to a new point is then carried out in two steps.

1. A temporary point $\boldsymbol{R}'_f$ is proposed with the probability $\boldsymbol{Q}(\boldsymbol{R}'_f|\boldsymbol{R}_i)$,
2. The point $\boldsymbol{R}'_f$ is accepted (i.e., $\boldsymbol{R}_f = \boldsymbol{R}'_f$) with probability $\boldsymbol{A}(\boldsymbol{R}'_f|\boldsymbol{R}_i)$ or rejected (i.e., $\boldsymbol{R}_f = \boldsymbol{R}_i$) with probability $1 - \boldsymbol{A}(\boldsymbol{R}'_f|\boldsymbol{R}_i)$.

Since only the ratio $\rho(\boldsymbol{R}_f)/\rho(\boldsymbol{R}_i)$ is involved in the above form of the accpetance probability, it is not necessary to calculate the normalization constait of the probability density $\rho(\boldsymbol{R})$.

The proposal probability $\boldsymbol{Q}(\boldsymbol{R}_f|\boldsymbol{R}_i)$ should be chosen in order to achieve a small autocorrelation time [10]. The simplest choice is a brute force approach where the probability is constant within a hyper-cube in the configuration space such that any configuration with $|\boldsymbol{R}_f - \boldsymbol{R}_i| \leq \Delta x$ is equiprobable. In practice this means a new configuration is proposed according to the expression

$$\boldsymbol{R}_f = \boldsymbol{R}_i + \Delta x \cdot \boldsymbol{\chi}, \tag{2.55}$$

where Δx defines the size of the hyper-cube and $\boldsymbol{\chi}$ is a vector of $3N$ uniform random numbers in the range -0.5 to 0.5. In this case the proposal distribution $\boldsymbol{Q}$ is symmetric, so that it cancels out in the ratio of the acceptance probability, leading to the simplified expression

$$\boldsymbol{A}(\boldsymbol{R}_f|\boldsymbol{R}_i) = \min\Big(1, \frac{\rho(\boldsymbol{R}_f)}{\rho(\boldsymbol{R}_i)}\Big). \tag{2.56}$$

While Metropolis *et al* used a symmetric proposal distribution of this kind, more general forms of the proposal distribution that were not symmetric were eventually used, among the first known were Hastings in 1970 [14]. Choosing a more sophisticated proposal distribution which takes the probability distribution into account in order to increase sampling from the more important parts of the function domain leads to more efficient sampling and can help reduce the variance of the estimators. This is called **importance sampling** and one such method commonly applied in VMC methods is discussed in the next section.

Importance sampling

A simple, isotropic diffusion process characterized by a time-dependent density $f(\boldsymbol{x}, t)$ obey the following Fokker-Planck equation, [2]

$$\frac{\partial f}{\partial t} = \sum_i D \frac{\partial}{\partial x_i}\left(\frac{\partial}{\partial x_i} - F_i(\boldsymbol{x})\right) f. \tag{2.57}$$

Here D is the diffusion constant and F_i is the i-th component of a drift velocity $\boldsymbol{F}$ caused by an external potential. We wish to converge to the stationary density $f = \Psi^2 / \int \Psi^2 \, \mathrm{d}\boldsymbol{x}$. An unchanging state, for which $\partial f / \partial t = 0$, may be obtained by setting the left-hand side of the above equation to zero, that is

$$\sum_i D \left(\frac{\partial^2 f}{\partial x_i^2} - \frac{\partial}{\partial x_i}(F_i f) \right) = 0. \tag{2.58}$$

This equation is satisfied if each term of the sum vanishes, yielding

$$\frac{\partial^2 f}{\partial x_i^2} = f \frac{\partial}{\partial x_i} F_i + F_i \frac{\partial}{\partial x_i} f. \tag{2.59}$$

This means the drift velocity $\boldsymbol{F}$ must have the form $F_i = g(f) \partial f / \partial x_i$ in order to obtain a second derivatie of f on the right hand side. Substituting this $\boldsymbol{F}$ into the previous equation gives us

$$\frac{\partial^2 f}{\partial x_i^2} = f \frac{\partial g}{\partial f} \left(\frac{\partial f}{\partial x_i} \right)^2 + f g \frac{\partial^2 f}{\partial x_i^2} + g \left(\frac{\partial f}{\partial x_i} \right)^2. \tag{2.60}$$

For the second derivative terms to cancel we must have $g = 1/f$, which also leads to cancellation of the first derivative terms. Therefore we obtain the stationary density $f = \Psi^2 / \int \Psi^2 \, \mathrm{d}\boldsymbol{x}$ by choosing the drift vector to be

$$\boldsymbol{F} = \frac{1}{f} \nabla f = 2 \frac{1}{\Psi} \nabla \Psi. \tag{2.61}$$

This drift causes the move to be biased by Ψ. This biased diffusion process incorporates importance sampling.

The next question is how to implement this diffusion process using Monte Carlo sampling. In statistical mechanics, Fokker-Planck trajectories are generated by means of a Langevin equation. The Langevin equation corresponding to the Fokker-Planck equation given above is [2]

$$\frac{\partial \boldsymbol{x}(t)}{\partial t} = D \boldsymbol{F}(\boldsymbol{x}(t)) + \eta \tag{2.62}$$

where η is a randomly fluctuating force which is distributed according to a multidimensional Gaussian distribution with a mean of zero and a variance of $2D$. By integrating the equation over a short time interval, δt, we obtain a discretized form which moves the particle from point $\boldsymbol{x}$ to $\boldsymbol{y}$ according to

$$\boldsymbol{y} = \boldsymbol{x} + D \boldsymbol{F}(\boldsymbol{x}) \delta t + \chi, \tag{2.63}$$

where χ is a Gaussian random variable with a mean value of zero and a variance of $2D\delta t$. By using the discretized form rather than the continuous form, we have

introduced a bias into the dynamics for any $\delta t \geq 0$. The distributed trajectories, and therefore the measured energy, will deviate increasingly from the exact as δt increases. However, this error may be corrected by the Metropolis step.

We have that $G(\boldsymbol{y}, \boldsymbol{x}; \delta t)$ must be a solution of the Fokker-Planck equation with the added condition that $G(\boldsymbol{y}, \boldsymbol{x}; \delta t = 0) = \delta(\boldsymbol{x} - \boldsymbol{y})$. In order to solve the Fokker-Planck equation, we rewrite it as

$$\frac{\partial f}{\partial t} = \mathcal{L} f, \tag{2.64}$$

where $\mathcal{L} = D\nabla\cdot(\nabla - \boldsymbol{F})$. Then $G(\boldsymbol{y}, \boldsymbol{x}; \delta t)$ is the spatial resolution of the operator $e^{-\mathcal{L}\delta t}$. In operator form, $G(\boldsymbol{y}, \boldsymbol{x}; \delta t)$ is given by

$$G(\boldsymbol{y}, \boldsymbol{x}; \delta t) = e^{D\delta t(\nabla^2 - \nabla\cdot\boldsymbol{F} - \boldsymbol{F}\cdot\nabla)}. \tag{2.65}$$

If we now assume that the force $\boldsymbol{F}$ remains constant between $\boldsymbol{x}$ and $\boldsymbol{y}$, we can integrate the previous expression over a small time interval (which we still call δt). This makes the expression a function of $\boldsymbol{x}$ and $\boldsymbol{y}$. When normalized it becomes

$$G(\boldsymbol{y}, \boldsymbol{x}; \delta t) = \frac{1}{(4\pi D\delta t)^{\frac{3N}{2}}} e^{-(\boldsymbol{y} - \boldsymbol{x} - D\delta t \boldsymbol{F}(\boldsymbol{x}))^2 / 4D\delta t}. \tag{2.66}$$

We have that $G(\boldsymbol{y}, \boldsymbol{x}; \delta t)$ gives the probability of a walker moving from $\boldsymbol{x}$ to $\boldsymbol{y}$. Thus, the total density at point $\boldsymbol{y}$ is given by the integral over all space of the transition probabilities multiplied by f at each point, that is

$$f(\boldsymbol{y}, t + \delta t) = \int G(\boldsymbol{y}, \boldsymbol{x}; \delta t) f(\boldsymbol{x}, t)\, \mathrm{d}\boldsymbol{x} \tag{2.67}$$

Repeated iterations of this expression by means of the given update move and the Metropolis acceptance step will therefore produce $f(\boldsymbol{y}, t \to \infty) = \Psi^2$ [2].

2.3.5 Uncertainty Estimates and Correlation

In section 2.3.2 on Monte Carlo integration we discussed the convergence and variance of the estimate of the integral. However, these results assumed that samples were **independent**. In a Markov proces, two consecutive samples are correlated by construction. This is referred to as autocorrelation or serial correlation. We will now look into how this affects the estimation and what ammends can be made.

We approach the problem by looking at the variance of the estimator G_N of

the function $g(X)$ now that samples X_i are correlated. We find [5]

$$\begin{aligned} var(G_N) =& \langle G_N^2 \rangle - \langle G_N \rangle^2 && (2.68)\\ =& \langle \frac{1}{N^2} \sum_{i=1}^{N} g(X_i) \sum_{j=1}^{N} g(X_j) \rangle - \langle g(X) \rangle^2 && (2.69)\\ =& \frac{1}{N^2} \sum_{i,j=1}^{N} \int p(X_1, ..., X_N) g(X_i) g(X_j) \, \mathrm{d}X_1 ... \, \mathrm{d}X_N - \langle g(X) \rangle^2 && (2.70)\\ =& \frac{1}{N^2} \sum_{i,j}^{N} \langle g(X_i) g(X_j) \rangle - \langle g(X) \rangle^2 && (2.71)\\ =& \frac{1}{N^2} var(g) \sum_{\tau=1}^{N} c(g)_\tau + \langle g(X) \rangle^2 - \langle g(X) \rangle^2 && (2.72)\\ =& \frac{1}{N} var(g) \sum_{\tau=1}^{N} c(g)_\tau && (2.73) \end{aligned}$$

where in line 2.72 we introduced the **autocorrelation coefficient**

$$c(g)_\tau = \frac{\langle g(X_i) g(X_{i+\tau}) \rangle - \langle g \rangle^2}{var(g)} \tag{2.74}$$

using that a stationary Markov chain does not depend on the times i and j, only the time interval $\tau = j - i$.

We can compare equation 2.73 to the expression given in section 2.3.2, $var(G_N) = \frac{1}{N} var(g)$. Clearly, this latter expression will underestimate the variance and standard error of the estimator when samples are correlated and the autocorrelation coefficients add up to more than one.

Usually the autocorrelation coefficients $c(g)_\tau$ have an exponential decay and can be approximated by $c(g)_\tau \sim exp(-\tau/\bar{\tau})$. The sum of coefficients can then be approximated as

$$\sum_{\tau=1}^{N} c(g)_\tau \sim \int_0^\infty e^{-\frac{\tau}{\bar{\tau}}} = \bar{\tau}, \tag{2.75}$$

which yields an estimate of the characteristic **autocorrelation time** that corrects the estimate of the error in the integral. It is the number of time steps between two samples for the samples not to be correlated at the given significance level. Inserting the autocorrelation time $\bar{\tau}$, the exression for the variance is

$$var(G_N) = \frac{\bar{\tau}}{N} var(g). \tag{2.76}$$

This provides a simple interpretation of the effect of correlation on our estimates. We are not generating N independent samples of X during a Markov process, but rather $N/\bar{\tau}$ of them. The latter is the number that must be used as the correct count for error estimation.

Thus, the main effect of the correlation between samples is that we need to generate even more samples in order to get satisfactory estimates. This can be a drawback when the sampling is a computationally intensive process, as is often the case in VMC programs. Thus, parameters in the Metropolis sampling algortihm, such as Δx in the brute force case and δt in the importance sampling case presented in the previous section, are chosen to minimize the autocorrelation time. This means tuning the proposal distribution to suggest moves which change the sample configuration as much as possible, but not so much that moves are rarely accepted.

Blocking

The blocking method provides a simple way of estimating variance in the presence of autocorrelation. The values of the quantity of interest are summed up in blocks of size N_b each:

$$g_l^b = \sum_{i=1}^{N_b} g(X_i). \tag{2.77}$$

The g_l^b are then used as the data points from which the variance and standard error are computed. When $N_b \gtrsim \bar{\tau}$ the standard error is corrected by the effects of the autocorrelation in the original data. The procedure is usually that one starts with a small N_b, then increase it, while calculating the variance for each size. As the block size increases the estimate of the variance increases as it takes into account more of the correlation effects and becomes more accurate. Then, when the block size has reached the size of the autocorrelation time, one will see that the estimate of the variance flattens out because there are no more correlation effects to take into account. This signals that we have reached an accurate block size. An example of this behavior is shown in figure 2.1. The decision of which block size to use can either be made manually by looking at a graph like this, or it can be automated. In this work we have used the automated method and code provided in [15] when applying the blocking technique to compute errors.

2.4 Systems of Bosons and Fermions

2.4.1 Quantum dots

Electrons confined to move in oscillator-like potentials form a set of widely studied many-fermion systems. References [16, 17] contain more details about the

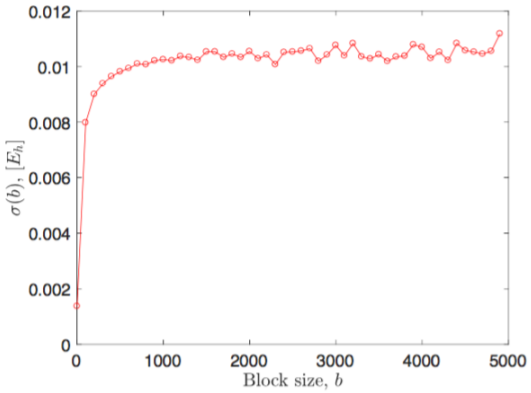

Figure 2.1: The figure illustrates the effect of applying the blocking method to compute the variance of samples in order to accurately account for autocorrelation. Figure taken from [16].

systems and the VMC calculations. For two electrons there are even analytical solutions in two and three dimensions. For the two-dimensional case, we have considered a system of electrons confined in a pure two-dimensional isotropic harmonic oscillator potential, with an idealized total Hamiltonian given by

$$\hat{H} = \sum_{i=1}^{N} \left(-\frac{1}{2}\nabla_i^2 + \frac{1}{2}\omega^2 r_i^2 \right) + \sum_{i<j} \frac{1}{r_{ij}}, \tag{2.78}$$

where natural units ($\hbar = c = e = m_e = 1$) are used and all energies are in so-called atomic units a.u. We will study systems of many electrons N as functions of the oscillator frequency ω using the above Hamiltonian. The Hamiltonian includes a standard harmonic oscillator part

$$\hat{H}_0 = \sum_{i=1}^{N} \left(-\frac{1}{2}\nabla_i^2 + \frac{1}{2}\omega^2 r_i^2 \right),$$

and the repulsive interaction between two electrons given by

$$\hat{H}_1 = \sum_{i<j} \frac{1}{r_{ij}},$$

with the distance between electrons given by $r_{ij} = |\boldsymbol{r}_1 - \boldsymbol{r}_2|$. We define the norm of the positions of the electrons (for a given electron i) as $r_i = \sqrt{r_{i_x}^2 + r_{i_y}^2}$. For electrons in three dimensions, it suffices to add the additional dimension, the equations are essentially the same. For the trial wave function used to compute the local energy, we employ a standard Slater determinant with a Jastrow factor,

see Refs. [16, 17]. For the two-electron case this reduces to (when we only account for the spatial degrees of freedom since the Hamiltonian is spin independent)

$$\psi_T(\boldsymbol{r}_1, \boldsymbol{r}_2) = C \exp\left(-\alpha\omega(r_1^2 + r_2^2)/2\right) \exp\left(\frac{ar_{12}}{(1+\beta r_{12})}\right), \tag{2.79}$$

where a is equal to one when the two electrons have anti-parallel spins and 1/3 when the spins are parallel in two dimensions. In three dimensions the corresponding values are 1/2 and 1/4, respectively. Finally, α and β are our variational parameters. Note well the dependence on α for the single-particle part of the trial function. We will compare our machine learning calculations with VMC calculations obtained using the above trial wave functions.

2.4.2 Confined Bosons

The spectacular demonstration of Bose-Einstein condensation (BEC) in gases of alkali atoms ^{87}Rb, ^{23}Na, ^{7}Li confined in magnetic traps has led to an explosion of interest in confined Bose systems. Of interest is the fraction of condensed atoms, the nature of the condensate, the excitations above the condensate, the atomic density in the trap as a function of Temperature and the critical temperature of BEC, T_c. [18] [19]

A key feature of the trapped alkali and atomic hydrogen systems is that they are dilute. The characteristic dimensions of a typical trap for ^{87}Rb is $a_{h0} = (\hbar/m\omega_\perp)^{\frac{1}{2}} = 1 - 2 \times 10^4$ Å . The interaction between ^{87}Rb atoms can be well represented by its s-wave scattering length, a_{Rb}. This scattering length lies in the range $85 < a_{Rb} < 140a_0$ where $a_0 = 0.5292$ Å is the Bohr radius. The definite value $a_{Rb} = 100a_0$ is usually selected and for calculations the definite ratio of atom size to trap size $a_{Rb}/a_{h0} = 4.33 \times 10^{-3}$ is usually chosen. A typical ^{87}Rb atom density in the trap is $n \simeq 10^{12} - 10^{14}$ atoms per cubic cm, giving an inter-atom spacing $\ell \simeq 10^4$ Å. Thus the effective atom size is small compared to both the trap size and the inter-atom spacing, the condition for diluteness ($na_{Rb}^3 \simeq 10^{-6}$ where $n = N/V$ is the number density).

Many theoretical studies of Bose-Einstein condensates (BEC) in gases of alkali atoms confined in magnetic or optical traps have been conducted in the framework of the Gross-Pitaevskii (GP) equation. The key point for the validity of this description is the dilute condition of these systems, that is, the average distance between the atoms is much larger than the range of the inter-atomic interaction. In this situation the physics is dominated by two-body collisions, well described in terms of the s-wave scattering length a. The crucial parameter defining the condition for diluteness is the gas parameter $x(\mathbf{r}) = n(\mathbf{r})a^3$, where $n(\mathbf{r})$ is the local density of the system. For low values of the average gas parameter $x_{av} \leq 10^{-3}$, the mean field Gross-Pitaevskii equation does an excellent job. However, in recent experiments, the local gas parameter may well exceed this value due to the

possibility of tuning the scattering length in the presence of a so-called Feshbach resonance.

Here, for the sake of simplicity we will use the same Hamiltonian as for the quantum dot case. The main difference now is that the single-particle part of the wave function ansatz is given by a symmetric function which is mainly given by a product of single-particle functions. Although the repulsive Coulomb is less realistic here compared with standard hard sphere potentials used in studies of Bose-Einstein condensation, we stay with with this potential in order to simplify our calculations. The only thing which changes thus is the symmetry of the single-particle part.

Chapter 3

Machine Learning

3.1 Overview

3.1.1 Historical background

Machine learning is a branch of **artifical intelligence**. Artificial intelligence became an academic discipline during the 1950s when the Dartmouth Summer Research Project on Artificial Intelligence assembled leading researchers in related fields including Marvin Minsky, John Nash, and Claude Shannon ("the father of information theory"). The aim was to explore ideas based on the conjecture *every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it.* The discussion covered characteristics of intelligence, such as self-improvement, abstractions, intuition ("controlled randomness") and creativity, as well as computer-related considerations such as speeds and memory capacities and the size of required calculations.

Soon after, an IBM computer scientist named Arthur L. Samuel coined the term machine learning in a breakthrough paper. In it, he trained a computer to play the game of checkers using a combination of **search trees** and the **minmax strategy** (minimizing the possible loss of a worst case scenario). The program is now considered one of the first successful learning programs. He defined machine learning as programming computers to **learn from experience**, eliminating the need for the time-consuming procedure of a programmer specifying problem solution in exact detail.

The interest in artificial intelligence had already become evident in the preceding decades, ever since Alan Turing concieved the modern computer when he developed an abstract, mathematical model of computing machines, the **Turing machine**, in 1936. Later he also proposed the famous **Turing Test** for determining if a machine can think. The Turing machine demonstrated that a machine could simulate mathematical deduction and formal reasoning through

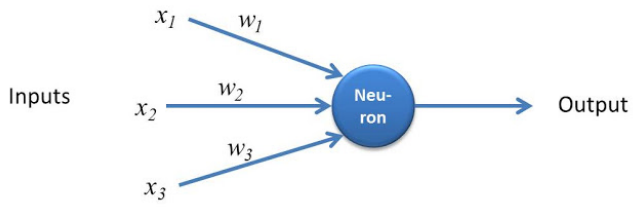

Figure 3.1: The perceptron was one of the first forms of neural networks. Inspired by human brain neurons and Hebbian learning it takes weighted inputs. Figure taken from http://kindsonthegenius.blogspot.com/2018/01/what-is-perceptron-how-perceptron-works.html.

the use of zeros and ones. This inspired W. McCulloch and W. Pitts to treat the human brain as a Turing machine by attempting to implement one using a network of neurons. In the process they developed some of the first, formal models of neurons, the **threshold logic unit**. Soon after, psychologist Donald Hebb connected the biological function of the brain to the psychological function of the mind by proposing **Hebb's law**, most often paraphrased as "neurons that fire together wire together". It means that when two neurons fire together the connection between them is strengthened. He proposed that this operation is one of the fundamental mechanisms underlying learning and memory.

In 1958 Frank Rosenblatt developed the **perceptron** which combined McCulloch and Pitts' idea of mathematical neurons with Hebb's learning theory. The perceptron could learn in the Hebbian sense through the *weighting of inputs*, see figure 3.1. However, the single layer perceptron could only learn linearly separable patterns and the multilayer perceptrons (later to be known as **feedforward neural networks**, see section 3.2.2) were difficult to train. Thus, these methods became abandoned until the 1980s when they re-emerged in connection to approaches known as parallel distributed processing (PDP) and connectionism. Researchers became aware that the **backpropagation algorithm**, with origins in control theory, made it possible to efficiently train neural networks.

In the meantime the AI community had focused on symbolic, or high-level, models. Neural networks, PDP and connectionsim are subsymbolic or low-level because they embody knowledge in model *parameters*. Symbolic methods, on the other hand, store knowledge in high-level, human-readable representations, *rules* which they use to perform formal operations on symbols. Examples are **decision trees** and **expert systems**, the latter which were very successful in the 1970s and 80s. Symbolic and subsymbolic methods reflect two different views on how human mental activity is best explained. The high-level models resemble human logical thinking, while low-level models are closer to neurophysiological models. While subsymbolic methods such as neural networks are often considered black-boxes, symbolic methods are transparant about what reasons lead to their

conclusion. The subsymbolic methods have advantages like better performance, scaling and adaptability to big data and have, perhaps unsurprisingly given these attributes, been the most popular over the last few decades.

The shift that happened in the 1980s from symbolic to subsymbolic AI was partly driven by the machine learning field becoming more seperate from AI, as they were interested in pursuing other areas than symbolic methods. They started the journal Machine Learning and their alternative focus, including an emphasis on computational approaches, attracted people from the pattern recognition community, leading to an increase of breadth by the inclusion of probabilistic and instance-based representations [20]. A more formal approach to evaluating algorithms was developed, based on the idea that the purpose of learning is to improve performance on some class of tasks, whether the metric be accuracy, efficiency or something else. This lead to a more empirical approach, with the majority of articles soon reporting experimental results about performance improvement on well-defined tasks. This effect was strenghtened by the development of repositories of data sets used for supervised classification tasks, making it easy to compare results to previous findings on such problems.

These changes narrowed down the focus from larger-scale intelligent systems carrying out multi-step reasoning to simpler tasks like regression and classification. One of the researchers behind the Machine Learning journal states that "the early volumes of Machine Learning included a variety of papers on problem solving, reasoning, and language, but by the mid-1990s they had nearly disappeared from the literature, having been displaced by work on less audacious tasks." There was a continued growth in the use of statistical and probabilistic approaches due to the focus on performance metrics, further encouraged by the advent of **big data**. This development explains the degree of overlap between statistics and machine learning that some people can be puzzled by when introduced to modern ML. The field **statistical learning** also reflects the many similarities.

While the 1990s had favored methods such as **support vector machines (SVMs)**, **Gaussian mixture models** and **hidden Markov models** to neural networks, the latter saw a resurgence in the 2000s. In 2006, Geoffrey Hinton *et al* (the same researcher who helped rediscover backpropagation and introduced the **Boltzmann machine** (section 4.1) while working in the PDP field in 1986) introduced the term **deep learning** when publishing several papers on how to efficiently train deep networks [21] [22] [23]. He noted that that while backpropagation was the first computationally efficient method for learning multiple layers of representation it still did not work well enough for deep networks to make their performance comparable to e.g. SVMs, prartly due to the **vanishing gradients problem** (section 3.2.2). Instead Hinton proposed **deep belief networks (DBNs)**, using several layers of stacked **restricted Boltzmann machines (RBMs)** (an unsupervised, generative neural network with a single hid-

den layer, see section 4.1.1), or alternatively layers of **autoencoders**. In addition to working independently the DBNs could be used as feature detectors for pre-training the traditional feedforward neural networks, with each layer in the DBN serving as an initialization of the corresponding layer in the FNN. Afterwards the FNN could be fine-tuned with backpropagation. These methods showed excellent performance on tasks such as recognition of hand-written digits and was quickly applied to problems in speech recognition, denoising images, retrieving documents and predictions of movie preferences and words in sentences.

This revival of neural networks came around the same time as advances in hardware capacities, in partiuclar the use of **graphical processing units (GPUs)** which greatly surpassed training speed on multi-core CPUs. Experiments which would take weeks on CPUs were reduced to a few hours on GPUs, the latter being particularly well-suited for the matrix/vector operations prominant in neural networks training. This enabled researchers to train deep FNNs with many hidden nodes in each layer. Using only backpropagation without pretraining the share size of the model combined with the GPU speedup proved enough for the deep FNNs to break new image recognition records, without the use of complicated pre-training [24]. All of these advances have lead to neural networks and deep learning dominating different machine learning contests over the past years, including variants such as **convolutional neural networks (CNNs)**, popular for image recognition, and **recurrent neural networks (RNNs)**, popular for speech recognition. This has been termed the deep learning revolution.

Nvidia, which originally invented GPUs and the CUDA platform for programming them, eventually developed the library cuDNN for efficient implementations of deep neural networks. Today several accessible deep learning libraries have been developed in Python, like SciKit Learn, Keras, TensorFlow and PyTorch, and most of them are cuDNN accelerated.

Energy-based, generative models like the Boltzmann machines require different training procedures from their discriminative counterparts (FNNs, CNNs, RNNs). Specifically, the gradient of the cost function must be computed using physics-inspired MCMC methods, rather than backpropagation (the reason is explained in section 3.3.1). These training procedures have not been as widely implemented in deep learning libraries (while SciKit Learn offers an implementation of RBMs one can only choose a simple version with binary nodes), which partly explains why they are less known to many people. This is changing, however, for example an open-source package called Paysage has been built on top of PyTorch to allow for such methods.

3.1.2 Present day machine learning

When defining the aims of modern machine learning, it is natural to distinguish between **supervised** and **unsupervised** learning. In supervised learning the

goal is usually to make **predictions**, either of cateogires (classification) or of continuous values (regression). In unsupervised learning some goals include discovering groups of similar examples within data (**clustering**), determining the distribution of data (**density estimation**), or projecting data onto a lower dimensional or latent space (**dimensionality reduction** or **feature extraction**).

Modern machine learning problems vary greatly, but they typically share four common ingredients:

1. Model

2. Dataset

3. Cost function (also known as objective function or loss function)

4. Optimization algorithm

In supervised learning, the model should produce some output $\boldsymbol{y}$ given some input $\boldsymbol{x}$. From a probabilistic perspective, one wishes either to model the conditional distribution $p(\boldsymbol{y}|\boldsymbol{x})$ (discriminative model) or the joint distribution $p(\boldsymbol{y}, \boldsymbol{x})$ (generative model). Unsupervised learning is more varied, but in a number of problems the goal is to model a probability distribution $p(\boldsymbol{x})$. The subsequent three ingredients relate to the **training** of the model. The cost function should be a measure of the error in the model, commonly based on how it performs on the dataset. The optimization algorithm is a method for updating the model in order to minimize the cost function.

With models of the above form, this procedure is a form of function estimation, which is also a problem in classical statistics. One way of differing between the two fields is that while the goal in statistics is usually inference, in machine learning it is prediction. This leads to slightly different approaches to the problem. When the goal is prediction, it is more acceptable to treat the model as a black-box. One cares more about the accuracy of the model's output than its exact form. In inference, on the other hand, the focus is on understanding the relationship between $\boldsymbol{x}$ and $\boldsymbol{y}$, how the latter change as a function of the former, and what features or predictors (components) of $\boldsymbol{x}$ are important. This difference in motives might influence the choice of model, for example in the degree of flexibility. A linear regression model is restrictive because it only allows linear relationships, while a support vector machine is much more flexible. A flexible model may provide better predictions (though not necessarily), however it may be hard or impossible to interpret. Hence restrictive models may be preferred when the goal is inference.

A practical approach to function estimation is assuming the function has a particular form and procede with estimating the parameters that characterize it.

3.2 Supervised learning

In supervised learning, training is performed with **labeled** data. This means we have a dataset with n samples $\mathcal{D} = \{(\boldsymbol{y}_i, \boldsymbol{x}_i)\}_{i=1}^n$, where $\boldsymbol{x}_i$ is the i-th observation and $\boldsymbol{y}_i$ is its corresponding response, or label. In this section we will briefly discuss two supervised learning methods, linear regression, which is the simplest one, and feedforward neural networks, which is the most well-known neural networks method.

3.2.1 Linear regression

Let f be a true, unkown function and the response variabel y be given by

$$y = f(\boldsymbol{x}; \boldsymbol{w}_{true}) + \epsilon. \tag{3.1}$$

The independent variable $\boldsymbol{x}$ has p features, $\boldsymbol{x} \in \mathbb{R}^p$. $\boldsymbol{w}_{true} \in \mathbb{R}^p$ is a parameter vector and ϵ is some i.i.d. error term with zero mean and finite variance.

Model

In linear regression we assume the functional form of f to be linear, so that

$$y = f(\boldsymbol{x}; \boldsymbol{w}_{true}) + \epsilon = \boldsymbol{x}^T \boldsymbol{w}_{true} + \epsilon \tag{3.2}$$

for some unkown but fixed $\boldsymbol{w}_{true}$. We wish to find parameters $\hat{\boldsymbol{w}}$ such that $g(\boldsymbol{x}; \hat{\boldsymbol{w}})$ yields our best estimate of f. We can then use this g to make predictions about the response y_0 for a new data point $\boldsymbol{x}_0$.

Cost function

In order to train the model we need to define a cost function which measures how well it works. Since this is a supervised learning problem the cost function should take into account the training data $\mathcal{D} = \{(y_i, \boldsymbol{x}_i)\}_{i=1}^n$, which is assumed to have been generated by the true function f.

The L_t norm of a vector $\boldsymbol{x} = (x_1, ..., x_d) \in \mathbb{R}^d$ may be defined

$$||\boldsymbol{x}||_t = (|x_1|^t + ... + |x_d|^t)^{\frac{1}{t}} \tag{3.3}$$

for any real number $p \geq 1$. The cost function is then chosen to be the L_2 norm of the difference between the response y_i and the predictor $g(\boldsymbol{x}_i; \boldsymbol{w}) = \boldsymbol{x}^T \boldsymbol{w}$. The samples may be represented by a $n \times p$ matrix, $X \in \mathbb{R}^{n \times p}$ with the rows $\boldsymbol{x}_1, ..., \boldsymbol{x}_n$ being observations and the columns $X_1, ..., X_p$ being measured features. The cost function can then be defined as the sum of squared errors

$$\mathcal{C}_{SSE} = ||X\mathbf{w} - \mathbf{y}||_2^2 = \sum_{i=1}^{n} (\mathbf{x}_i^T \mathbf{w} - y_i)^2. \tag{3.4}$$

The trained model is the model with parameters $\boldsymbol{w}$ which minimize the cost function. This method for estimating the linear regression parameters is known as ordinary **least squares**, and the minimization becomes a linear algebra problem:

$$\hat{\mathbf{w}}_{LS} = \text{argmin}_{\mathbf{w}\in\mathbb{R}^p} ||X\mathbf{w} - \mathbf{y}||_2^2 \tag{3.5}$$

$$\text{(after differentiating)} \quad \Rightarrow \hat{\mathbf{w}}_{LS} = (X^T X)^{-1} X^T \mathbf{y} \tag{3.6}$$

where $X^T X$ is assumed invertible.

Sometimes the training might be improved by adding **regularizers** to the loss function. Two common examples are the L_2 norm of the parameter vector $\boldsymbol{w}$, called Ridge regression, and the L_1 norm of the parameters, called LASSO regression. This leads to the following cost functions respectively:

$$\mathcal{C}_{Ridge} = ||X\mathbf{w} - \mathbf{y}||_2^2 + \lambda ||\boldsymbol{w}||_2^2 \tag{3.7}$$

$$\mathcal{C}_{LASSO} = ||X\mathbf{w} - \mathbf{y}||_2^2 + \lambda ||\boldsymbol{w}||_1. \tag{3.8}$$

$$\tag{3.9}$$

We see that this introduces the extra parameter $\lambda \geq 0$ which must be chosen manually. Adding the regularizers is equivalent to performing *constrained* optimization of the simple least square cost function. Ridge regression corresponds to imposing a constraint $||\boldsymbol{w}||_2^2 \leq t$, $t \geq 0$, and LASSO corresponds to the constraint $||\boldsymbol{w}||_1 \leq t$ on the parameters. Effectively, Ridge regression puts a constraint on the magnitude of the parameter vector, shrinking the size of the coefficients, while LASSO enforces sparse solutions, forcing several of the coefficients to be zero. Both these teqhniques reduce the expressivity of the model, which is a way of preventing overfitting (for the model to generalize well it must not be overfitted to the training data). Additionally LASSO may increase the interpretability of the model since it selects certain features, giving others zero weight.

The linear regression cost function can also be formulated from a probabilistic perspective. The linear regression model assumes the data $\mathcal{D}$ is generated through $y = \mathbf{x}^T\mathbf{w} + \epsilon$, where ϵ is a Gaussian noise with mean zero and variance σ^2. This model may be reformulated as

$$p(y|\mathbf{x}, \boldsymbol{\theta}) = \mathcal{N}(y|\mu(\boldsymbol{x}), \sigma^2(\boldsymbol{x})). \tag{3.10}$$

That is, the model is reformulated as a conditional probability that depends on the data $\boldsymbol{x}$ as well as some model parameters $\boldsymbol{\theta}$. For example, if the mean of the linear function of $\boldsymbol{x}$ is $\mu = \boldsymbol{x}^T\boldsymbol{w}$ and the variance is fixed $\sigma^2(\boldsymbol{x}) = \sigma^2$, then $\boldsymbol{\theta} = (\boldsymbol{w}, \sigma^2)$. From this we can define another cost function based on the **likelihood** function

$$p(\mathcal{D}|\boldsymbol{\theta}) = \prod_{i=1}^{n} p(y_i|\boldsymbol{x}_i, \boldsymbol{\theta}), \tag{3.11}$$

using the assumption that samples are i.i.d. It is common to take the logarithm, naming the result the log-likelihood. We wish to maximize the log-likelihood, thus this approach is called **maximum log-likelihood estimation** (MLE). That means our cost function is the negative log-likelihood:

$$\mathcal{C}_{LL} = -\log p(\mathcal{D}|\boldsymbol{\theta}) = -\sum_{i=1}^{n} \log p(y_i|\boldsymbol{x}_i, \boldsymbol{\theta}). \tag{3.12}$$

It is useful to make clear the distinction between the **frequentist** and **Bayesian** interpretation of the likelihood function. In freqeuntist statistics the likelihood is not a probability. In Bayesian statistics, on the other hand, it is seen as a conditional probability, as indicated by the notation above. The reason is that while frequentists consider model parameters to be estimated, in this case $\boldsymbol{\theta}$, as an unkown but *fixed* value, the Bayesians consider them to be *random variables*, like the observed data. Hence Bayesians see the likelihood as the conditional probability of the observed data $\mathcal{D}$ given the model parameters $\boldsymbol{\theta}$.

If we write out the MLE problem we realize that the same solution as in least squares is recovered:

$$\begin{aligned}
\hat{\boldsymbol{\theta}}_{MLE} &= \text{argmax}_\theta \log \sum_{i=1}^{n} \log p(y_i|\boldsymbol{x}_i, \boldsymbol{\theta}) && (3.13)\\
&= \text{argmax}_\theta \Big(-\frac{1}{2\sigma^2} \sum_{i=1}^{n} (y_i - \boldsymbol{x}_i^T \boldsymbol{w})^2 - \frac{n}{2} \log(2\pi\sigma^2) \Big) && (3.14)\\
&= \text{argmax}_\theta \Big(-\frac{1}{2\sigma^2} ||X\boldsymbol{w} - \boldsymbol{y}||_2^2 + \text{const} \Big) && (3.15)\\
&= \text{argmin}_\theta ||X\mathbf{w} - \mathbf{y}||_2^2. && (3.16)\\
& && (3.17)
\end{aligned}$$

How would we incorporate regularizers in the likelihood cost function? In fact, regularizers emerge as a natural part of the Bayesian inference procedure. Before we continue then we will quickly go through Bayesian inference. The starting point is Baye's rule of conditional probabilities between two events A and B, given by

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}. \tag{3.18}$$

In Bayesian inference the events in this formula are interpreted as a hypothesis

$\mathcal{H}$ and data $\mathcal{D}$ which may give evidence for or against $\mathcal{H}$.

$$\text{posterior} = \frac{\text{likelihood} \times \text{prior}}{\text{partition function}} \tag{3.19}$$

$$P(\mathcal{H}|\mathcal{D}) = \frac{P(\mathcal{D}|\mathcal{H}) \times P(\mathcal{H})}{P(\mathcal{D})} \tag{3.20}$$

$$p(\boldsymbol{\theta}|\mathcal{D}) = \frac{p(\mathcal{D}|\boldsymbol{\theta}) \times p(\boldsymbol{\theta})}{\int p(\mathcal{D}|\boldsymbol{\theta}) \times p(\boldsymbol{\theta}) \, \mathrm{d}\boldsymbol{\theta}}. \tag{3.21}$$

$$\tag{3.22}$$

In our case the hypothesis is about the value of the model parameters $\boldsymbol{\theta}$. While the likelihood function represents our knowledge about $\boldsymbol{\theta}$ given the data, the **prior** probability $p(\boldsymbol{\theta})$ represents any knowledge we might have about $\boldsymbol{\theta}$ before we look at the data. Priors are one of the most important features to distinguish Bayesian from frequentist statistics, because a prior is subjective in that it may vary from one investigator to another. The partition function is a normalization constant. Thus, Bayesian inference combines prior knowledge about $\boldsymbol{\theta}$ with knowledge from the data $\mathcal{D}$ to compute the **posterior distribution** $p(\boldsymbol{\theta}|\mathcal{D})$, that is, the probability that the hyopthesis is correct, given the evidence. We notice that the posterior distribution is proportional to the log-likelihood. Thus, we could choose the posterior distribution as the basis for the cost function, thereby including priors in addition to the likelihood. This is called **maximum a posteriori probability (MAP) estimate**. Excluding the normalization constant which will not affect the optimization, this leads to the cost function

$$\mathcal{C}_{MAP} = -\log p(\boldsymbol{\theta}|\mathcal{D}) = -(\log p(\mathcal{D}|\boldsymbol{\theta})). + \log p(\boldsymbol{\theta}) \tag{3.23}$$

The MAP estimator becomes

$$\hat{\boldsymbol{\theta}}_{MAP} \equiv \text{argmax}_\theta \log p(\mathcal{D}|\boldsymbol{\theta}) + \log p(\boldsymbol{\theta}). \tag{3.24}$$

Suppose we use the Gaussian prior with zero mean and variance τ^2, namely $p(\boldsymbol{w}) = \prod_j \mathcal{N}(w_j|0, \tau^2)$. We can then rewrite the MAP estimator, dropping constant terms that don't depend on the parameters, as

$$\hat{\boldsymbol{\theta}}_{MAP} = \text{argmax}_\theta[-\frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i - \boldsymbol{x}_i^T\boldsymbol{w})^2 - \frac{1}{2\tau^2}\sum_{j=1}^{n} w_j^2] \tag{3.25}$$

$$= \text{argmax}_\theta[-\frac{1}{2\sigma^2}||X\boldsymbol{w} - \boldsymbol{y}||_2^2 - \frac{1}{2\tau^2}||\boldsymbol{w}||_2^2]. \tag{3.26}$$

We see then that the equivalence between MAP estimation with a Gaussian prior and Ridge regression may be established by comparing this expression to the one for $\hat{\boldsymbol{w}}_{Ridge}(t)$ with $\equiv \sigma^2/\tau^2$. There's an analogous derivation for LASSO.

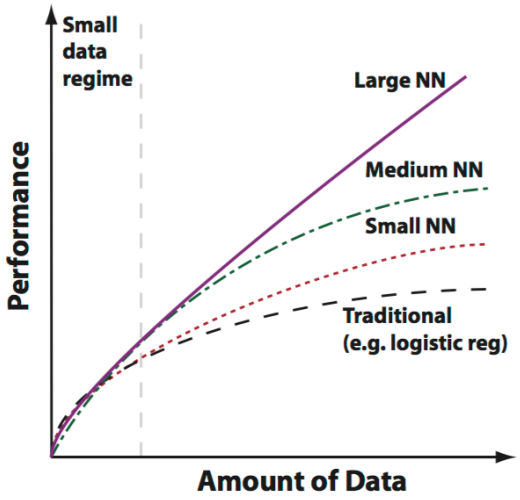

Figure 3.2: The figure shows the performance of feed forward neural networks of different sizes compared to traditional prediction methods as a function of the amount of data. Taken from [25].

On a final note, MAP estimation is not very representative of Bayesian methods in general. The reason is that it estimates a *point estimate* $\hat{\boldsymbol{\theta}}_{MAP}$, returning the value where the posterior $p(\boldsymbol{\theta}|\mathcal{D})$ is a maximum. In general, Bayesian methods calculate fully the posterior distribution. One then chooses the value of $\boldsymbol{\theta}$ one considers the best based on this. It could for example be the expected value. It also allows for calculating the variance of $\boldsymbol{\theta}$, which is used as a measure of the confidence in the value used as the estimate. In order to fully calculate the posterior $p(\boldsymbol{\theta}|\mathcal{D})$ one can no longer ignore the partition function, and the need to compute this integral is what makes Bayesian methods computationally intensive. As we know MCMC methods allows for efficient sampling from complicated probability distributions like this, and have therefore become an important part of Bayesian inference.

3.2.2 Feedforward Neural Networks

As perhaps became clear in section 3.1.1, feedforward neural netowkrs (FNNs) are the simplest and most well known neural networks. We will discuss them here as an example of more complex supervised machine learning models and as an introduction to neural networks before we move on to a generative, unsupervised variant in the following section. Figure 3.2 shows how the performance of deep FNNs improve with the amount of data compared to other supervised learning algorithms such as SVMs and ensemble methods. The deep FNNs are better able to exploit additional amounts of data. This is likely due to a combination of expressivity (ability to represent complicated functions) and trainability (ability

to learn millions of parameters) thanks to their simple architecture of layer-wise connections.

Model

In general, linear models for regression and classification are based on linear combinations of fixed nonlinear basis functions $\phi_j(\boldsymbol{x})$ and take the form

$$y(\boldsymbol{x}, \boldsymbol{w}) = f\Big(\sum_{j=1}^{M} w_j \phi_j(\boldsymbol{x})\Big) \tag{3.27}$$

where $f(\cdot)$ is a nonlinear activation function in the case of classification and the identity in the case of regression. We wish to extend this model by making the basis functions $\phi_j(\boldsymbol{x})$ depend on parameters and then allow these parameters to be adjusted, along with the coefficients $\boldsymbol{w}$ during training. There are many ways to construct parametric nonlinear basis functions. Neural networks use basis functions that follow the same form as the above expression, so that each basis function is itself a nonlinear function of a linear combination of the inputs, where the coefficients in the linear combination are adaptive parameters.

This leads to the basic neural network model, which can be described as a series of functional transformations. We first construct M linear combinations of the input variables $x_1, ..., x_D$ in the form

$$a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \tag{3.28}$$

where $j = 1, ..., M$, and the superscript (1) indicates that the corresponding parameters belong to the first 'layer' of the network. The parameters $w_{ji}^{(1)}$ are called **weights** and the parameters $w_{j0}^{(1)}$ **biases**. The quantities a_j are known as **activations**. Each of them is then transformed using a differentiable, nonlinear **activation function** $h(\cdot)$ to give

$$z_j = h(a_j). \tag{3.29}$$

These quantities correspond to the outputs of the basis functions in equation 3.27 that are called **hidden units**. The nonlinear functions $h(\cdot)$ are generally chosen to be sigmoidal functions such as the logistic sigmoid or the hyperbolic tangent, or less commonly step-functions (perceptrons). Recently it has become popular to use rectified linear functions (ReLUs), leaky rectified linear units (leaky ReLUs), and exponential linear units (ELUs) [25], see figure 3.3. The reason for the change is that when training with gradient descent based methods one needs the derivatives of the cost function with respect to the weights and biases. Firstly, the perceptron has a discontinuous behaviour making it impossible to use for gradient

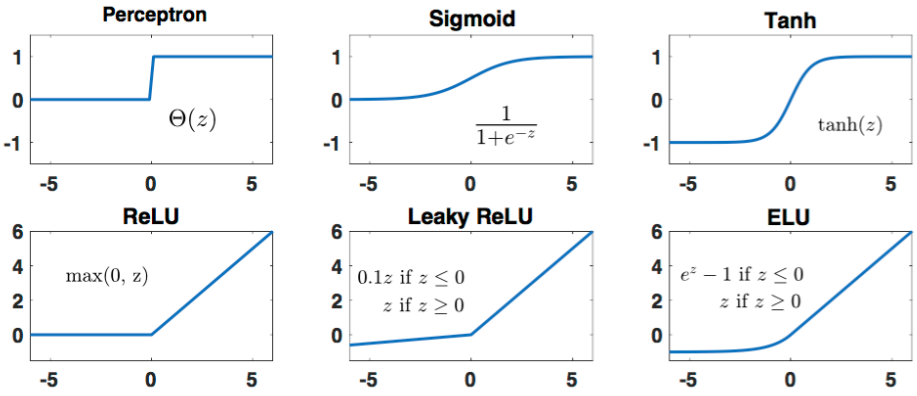

Figure 3.3: The figure shows the different activation functions one can use in the feedforward neural network. Taken from [25]

descent. Secondly, the sigmoid and tanh functions have the drawback that if the weights become large during training, the activation function saturates and the cost function derivatives tend to zero, leading to the **vanishing gradients problem** where the weights, which recieve updates proportional to the gradient (section 3.4), effectively stop changing values. ReLUs, leaky ReLUs and ELUs, on the other hand, have gradients which stay finite even for large inputs.

Following equation 3.27, the hidden units z_j from equation 3.29 are again linearly combined to give **output unit activations**

$$a_k = \sum_{j=1}^{M} w_{kj}^{(2)} z_j + w_{k0}^{(2)} \tag{3.30}$$

where $k = 1, ..., K$, and K is the total number of outputs. This transformation corresponds to the second layer of the network, and again the $w_{k0}^{(2)}$ are bias parameters.

Finally, the output unit activations are transformed using an appropriate activation function to give a set of network outputs y_k. The choice of activation function is determined by the nature of the data and the assumed distribution of target variables. Thus for standard regression problems, the activation function is the identity so that $y_k = a_k$, while for multiple binary classification problems, each output unit activation is transformed using a logistic sigmoid function σ so that $y_k = \sigma(a_k)$.

We can combine these stages to give the overall network function that, for a regression problem with the identity as output activation, takes the form

$$y_k(\boldsymbol{x}, \boldsymbol{w}) = \sum_{j=1}^{M} w_{kj}^{(2)} h\left(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \right) + w_{k0}^{(2)} \tag{3.31}$$

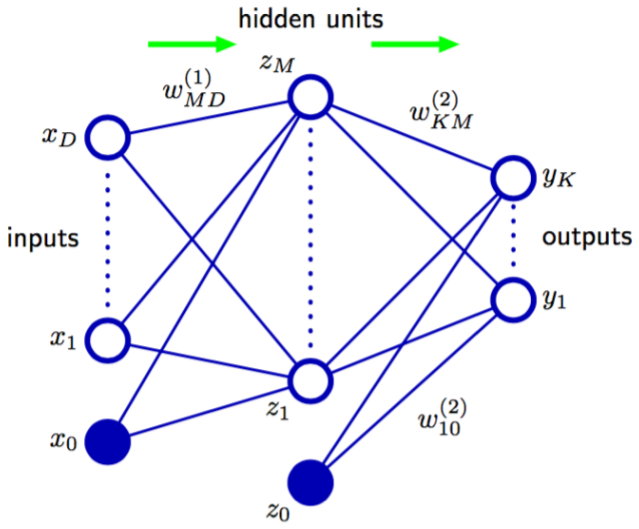

Figure 3.4: The feedforward neural network structure. Here the fixed $x_0 = 1$ and $z_0 = 1$ nodes are additional inputs to be multiplied with the bias terms, introduced for ease of notation. Taken from [26]

where the set of all weights and biases have been grouped together in a vector $\boldsymbol{w}$. Thus the neural network model is simply a nonlinear function from a set of input variables $\boldsymbol{x}$ to a set of output variables $\boldsymbol{y}$ controlled by a vector $\boldsymbol{w}$ of adjustable parameters.

This function can be represented in the form of a network diagram, or more precisely a directed acyclic graph (cyclic graphs may represent **recurrent** neural networks), see figure 3.4. The process of evaluating it can then be interpreted as a forward propagation of information through the network. It should be emphasized that such a diagram does not represent probabilistic graphical models of the kind we will discuss later, because the internal nodes represent **deterministic** variables rather than **stochastic** ones.

We can give a probabilistic interpretation to the neural network similar to the regression model by expressing our uncertainty over the value of a target variable t. We do this by assuming that t has a Gaussian distribution with an $\boldsymbol{x}$-dependent mean given by the output of the network, so that the model can be expressed

$$p(t|\boldsymbol{x}, \boldsymbol{w}) = \mathcal{N}(t|y(\boldsymbol{w}, , \boldsymbol{y}), \sigma^2) \tag{3.32}$$

where σ^2 is the variance of the Gaussian noise. This is a somewhat restrictive assumption, but it is shown in [26] how to extend this approach to allow for more general conditional distributions.

Cost function

Since we are again working with a supervised learning method, the cost function will be constructed to take into account a labeled training dataset, consisting of N i.i.d. observations $X = \{\boldsymbol{x}_1, ..., \boldsymbol{x}_N\}$, along with corresponding target values $t = \{t_1, ..., t_N\}$. The most common cost functions used for feedforward neural networks are similar to those presented in the linear regression section. For example one can construct a cost function based on the sum of squared errors

$$\mathcal{C}_{SSE} = \sum_n^N ||\boldsymbol{y}(\boldsymbol{x}_n, \boldsymbol{w}) - \boldsymbol{t}_n||^2. \tag{3.33}$$

Like before we can also construct the likelihood function

$$p(\boldsymbol{t}|X, \boldsymbol{w}, \sigma^2) = \prod_{n=1}^N p(t_n|\boldsymbol{x}_n, \boldsymbol{w}, \sigma^2) \tag{3.34}$$

and express the cost function as the negative log-likelihood

$$\mathcal{C}_{LL} = -\log p(\boldsymbol{t}|X, \boldsymbol{w}, \sigma^2) = -\sum_{n=1}^N \log p(t_n|\boldsymbol{x}_n, \boldsymbol{w}, \sigma^2) \tag{3.35}$$

In the case of linear regression we could minimize the cost function by solving a linear algebra problem. This is not possible with neural networks. Instead it is necessary to employ approximation schemes such as iterative optimization. Some of the most popular methods in machine learning are based on the gradient descent algorithm, discussed in section 3.4. To use it it is necessary to compute the gradient of the cost function. In the case of feedforward neural networks this is made possible by the backpropagation algorithm. How this algorithm works is outside the scope of this work.

3.3 Unsupervised learning

Unsupervised learning is concerned with finding patterns and structure in unlabeled data. Unsupervised learning includes dimensionality reduction, for example **principal component analysis**, and clustering, for example the **K-means algorithm**. Many supervised learning tasks involve accurately representing the underlying probability distribution from which a dataset is drawn. One class of such methods is variational methods, for example the **Expectation Maximation procedure (EM)**. They are closely connected to methods in physics, for example the Mean-Field Theory of statistical physics. Other examples are **generative** methods, which differ from **discriminative** methods. Linear regression

Figure 3.5: Pictures of cats generated by a generative adversarial network (GAN) and trained with the ADAM optimization procedure (see section 3.4.3). Image found https://ajolicoeur.wordpress.com/cats/. A video visualization of the training procedure showing vague features turn into cats can be seen https://www.youtube.com/watch?v=JRBscukr7ew&feature=youtu.be.

and feedforward neural netowrks from the previous section are examples of discriminative methods. They are named so because they are designed to percieve differences between groups or labels. With generative methods we can do the same as discriminative methods but also perform an increased number of tasks, including generate *new* samples of data. A generative model for images could, if trained on images of cats and dogs, learn to construct new examples of cats and dogs (see figure 3.5). One can sample from the approximated probability distribution using MCMC methods in order to generate new examples. Some generative methods which have gained popularity over the past years include generative adversarial networks (GANs) and variational autoencoders (VAEs).

One class of generative methods are called **energy-based models**. The RBM falls within this category. Another example is **Maximum Entropy (MaxEnt)** models. We will see that here as well one can find close connections to statistical physics. The term energy-based comes from the fact that one constructs an energy function which determines the probability of different configurations of features. The goal of training is then to adjust this energy function to assign high probability to configurations representing frequently occuring samples, and low probability to unlikely scenarios. Since the method of choice in this work is the restricted Boltzmann machine, we will spend this section outlining **probabilistic graphical models** and **Markov Random Fields**, which provide the framework

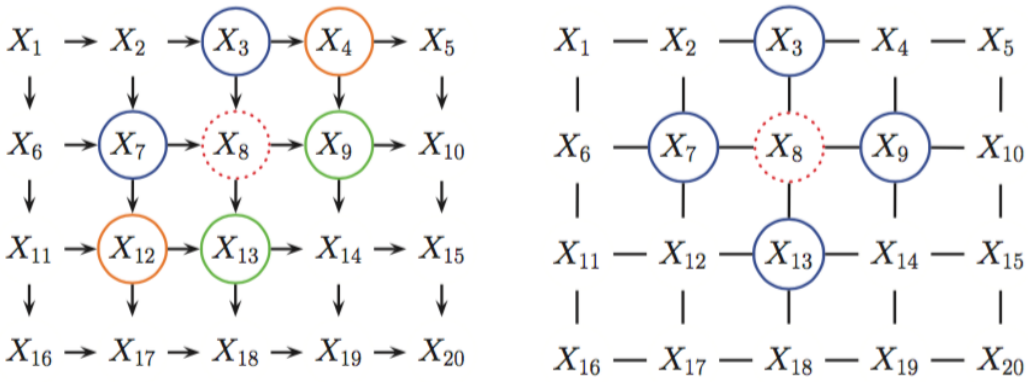

Figure 3.6: (Left) A 2D lattice represented by a directed graphical model. The red node X_s is independent of the black nodes given the colored nodes, which represent its children (green), parents (blue) and co-parents (orange). (Right) The same model represented by an undirected graphical model. The red node X_s is independent of the black nodes given its blue colored neighbors. Figure taken from [27].

for the Boltzmann machine. As the reader might know the Markov Random Fields were devoloped in order to provide a general setting for a popular model in physics, the Ising model.

3.3.1 Probabilistic Graphical Models and Markov Random Fields

A **random field** is a generalization of a multivariate stochastic process (section 2.3.3), such that the underlying parameter need no longer be a simple real-valued "time" variable, but can instead be multidimensional vectors, often interpreted as spatial sites corresponding to each of the random variables. A Markov random field is a random field whose set of random variables satisfy the Markov properties, in this context given by

1. **Pairwise Markov Property** Two non-adjacent variables are conditionally independent given all other variables.

2. **Local Markov Property** A variable is conditionally independent of all other variables given its neighbors.

3. **Global Markov Propery** Any two subsets of variables are conditionally independent given a separating subset.

This structure of conditional probabilities is well represented by an undirected probabilistic graphical model. A **graph** consists of **nodes** connected by **edges**.

In a probabilistic graphical model, each node represents a random variable and the edges express probabilistic relationships between these variables. The graph captures the way in which the joint distribution over all the random variables can be decomposed into a product of conditional probabilities each depending only on a subset of the variables.

There are two major classes of graphical modeles; directed and undirected. Examples of directed models include Bayesian networks. In directed graphs the edges have directions indicated by arrows, while in undirected graphs the edges have no directional significance. While directed models are useful in some problems, it might be unnatural in others. When modeling an image, for example, we might assume the intensity of neighboring pixels are correlated. Figure 3.6 illustrates how this could be modeled in a directed graph (left) and undirected graph (right). We see that the conditional independence structure of the directed model is rather awkward, with the middle node X_s depending on the other colored nodes, instead of just its four neighbors in the undirected case, which is what one would expect.

We define a **clique** as a subset of the nodes in a graph such that there exists an edge between all pairs of nodes in the subset. That is, the nodes in a clique are fully connected. A **maximal clique** is a clique such that it is impossible to include any other nodes from the graph in the set without it ceasing to be a clique. Let us denote a clique by C and the set of variables in that clique by $\boldsymbol{x}_C$. Then the joint distribution can be written as a product of **potential functions** ϕ over the maximal cliques of the graph

$$p(\boldsymbol{x}) = \frac{1}{Z} \prod_C \phi_C(\boldsymbol{x}_C) \tag{3.36}$$

where Z, the partition function, is a normalization constant given by

$$Z = \int \prod_C \phi_C(\boldsymbol{x}_C) \, \mathrm{d}\boldsymbol{x} \tag{3.37}$$

By considering only potential functions which satisfy $\phi_C(\boldsymbol{x}_C) \geq 0$ we ensure that $p(\boldsymbol{x}) \geq 0$. The formal result that the conditional independence properties of an undirected graphical model can be expressed by equation 3.36 is stated in the Hammersley-Clifford Theorem [27].

Because we are restricted to potential functions which are positive it is convenient to express them as exponentials, so that

$$\phi_C(\boldsymbol{x}_C) = e^{-E_C(\boldsymbol{x}_C)} \tag{3.38}$$

where $E(\boldsymbol{x}_C)$ is called an **energy function**, and the exponential representation is the **Boltzmann distribution**. The joint distribution is defined as the product of potentials, and so the total energy is obtained by adding the energies of each of the maximal cliques.

The joint distribution of the random variables is then

$$\begin{aligned} p(\boldsymbol{x}) =& \frac{1}{Z}\prod_C \phi_C(\boldsymbol{x}_C) \\ =& \frac{1}{Z}\prod_C e^{-E_C(\boldsymbol{x}_C)} \\ =& \frac{1}{Z}e^{-\sum_C E_C(\boldsymbol{x}_C)} \\ =& \frac{1}{Z}e^{-E(\boldsymbol{x})}. \end{aligned} \tag{3.39}$$

Cost function

The goal of supervised, discriminative models is straight forward: predict a label given the features. It is harder to define what is meant by a good generative model. When trained on a dataset we want it to generate samples similar to those in the training data, while also generalize well and not reproduce noisy details in the training data. In the end the most common approach is similar to that used for supervised learning: maximize the log-likelihood of the training data set. In generative modeling the log-likelihood characterizes the log-probability of generating the observed data using our model. By choosing the negative log-likelihood as the cost function, the learning procedure tries to find parameters that maximize the probability of the data.

Our derivation here will be true for all energy-based models because we do not assume a specific form of the energy function, only that the generative model is of the Boltzmann form. The energy function is parameterized by parameters $\boldsymbol{\theta}$ which determine what configurations of features have a high or low probability. When training with an unlabeled dataset of i.i.d. samples $\boldsymbol{X}_D = (\boldsymbol{x}_1, ..., \boldsymbol{x}_D)$ the log-likelihood cost function is defined

$$\mathcal{C}_{LL} = -\log \prod_i^D p(\boldsymbol{x_i}|\boldsymbol{\theta}) \tag{3.40}$$

$$= -\sum_i^D \log p(\boldsymbol{x_i}|\boldsymbol{\theta}) \tag{3.41}$$

$$= -\sum_i^D \log \frac{1}{Z}e^{-E(\boldsymbol{x})} \tag{3.42}$$

$$= \sum_i^D E(\boldsymbol{x}) + D\log Z. \tag{3.43}$$

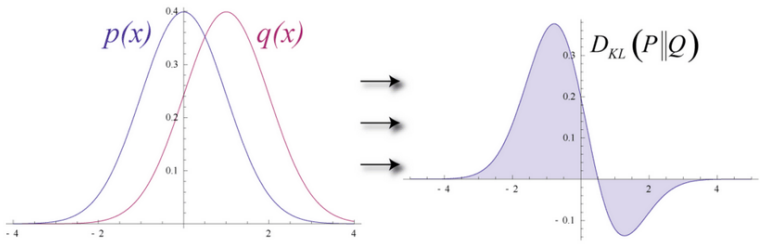

Figure 3.7: Illustration of the Kullback-Leibler divergence which measures the difference between two probability distributions $p(x)$ and $q(x)$. Figure taken from here.

Since it does not make a difference in the optimization procedure to multiply the cost function by a constant, we can choose the cost function to be the above expression divided by the number of data samples D, yielding

$$\mathcal{C}_{LL} = \frac{1}{D}\sum_{i}^{D} E(\boldsymbol{x}) + \log Z \tag{3.44}$$

$$= \langle E(\boldsymbol{x}) \rangle_{data} + \log Z \tag{3.45}$$

where $\langle \cdot \rangle_{data}$ is the expectation value over the probability distribution of the data.

When the goal of the training is to approximate a probability distribution, as it is in generative modeling, another relevant measure is the **Kullback-Leibler divergence**, also known as the relative entropy or Shannon entropy (see figure 3.7). It is a non-symmetric measure of the dissimilarity between two probability density functions p and q. If p is the unkown probability which we approximate with q, we can measure the difference by

$$\mathrm{KL}(p||q) = \int_{-\infty}^{\infty} p(\boldsymbol{x}) \log \frac{p(\boldsymbol{x})}{q(\boldsymbol{x})} \, \mathrm{d}\boldsymbol{x}. \tag{3.46}$$

Thus, the Kullback-Leibler divergence between the distribution of the training data $f(\boldsymbol{x})$ and the model distribution $p(\boldsymbol{x}|\boldsymbol{\theta})$ is

$$\mathrm{KL}(f(\boldsymbol{x})||p(\boldsymbol{x}|\boldsymbol{\theta})) = \int_{-\infty}^{\infty} f(\boldsymbol{x}) \log \frac{f(\boldsymbol{x})}{p(\boldsymbol{x}|\boldsymbol{\theta})} \, \mathrm{d}\boldsymbol{x} \tag{3.47}$$

$$= \int_{-\infty}^{\infty} f(\boldsymbol{x}) \log f(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} - \int_{-\infty}^{\infty} f(\boldsymbol{x}) \log p(\boldsymbol{x}|\boldsymbol{\theta}) \, \mathrm{d}\boldsymbol{x} \tag{3.48}$$

$$= \langle \log f(\boldsymbol{x}) \rangle_{f(\boldsymbol{x})} - \langle \log p(\boldsymbol{x}|\boldsymbol{\theta}) \rangle_{f(\boldsymbol{x})} \tag{3.49}$$

$$= \langle \log f(\boldsymbol{x}) \rangle_{data} + \langle E(\boldsymbol{x}) \rangle_{data} + \log Z \tag{3.50}$$

$$= \langle \log f(\boldsymbol{x}) \rangle_{data} + \mathcal{C}_{LL}. \tag{3.51}$$

The first term is constant with respect to $\boldsymbol{\theta}$ since $f(\boldsymbol{x})$ is independent of $\boldsymbol{\theta}$. Thus the Kullback-Leibler Divergence is minimal when the second term is minimal. The second term is the log-likelihood cost function, hence minimizing the Kullback-Leibler divergence is equivalent to maximizing the log-likelihood.

To further understand generative models it is useful to study the gradient of the cost function which is needed in order to minimize it using methods presented in section 3.4. As in statistical physics the partition function is the generating function of expectation values, in particular there are mathematical relationships between expectation values and the log-partition function. In this case we have [25]

$$\left\langle \frac{\partial E(\boldsymbol{x};\theta_i)}{\partial \theta_i} \right\rangle_{model} = \int p(\boldsymbol{x}|\boldsymbol{\theta}) \frac{\partial E(\boldsymbol{x};\theta_i)}{\partial \theta_i} \, \mathrm{d}\boldsymbol{x} = -\frac{\partial \log Z(\theta_i)}{\partial \theta_i}. \tag{3.52}$$

Here $\langle \cdot \rangle_{model}$ is the expectation value over the model probability distribution $p(\boldsymbol{x}|\boldsymbol{\theta})$. Using this relationship we can express the gradient of the cost function as

$$\frac{\partial \mathcal{C}_{LL}}{\partial \theta_i} = \left\langle \frac{\partial E(\boldsymbol{x};\theta_i)}{\partial \theta_i} \right\rangle_{data} + \frac{\partial \log Z(\theta_i)}{\partial \theta_i} \tag{3.53}$$

$$= \left\langle \frac{\partial E(\boldsymbol{x};\theta_i)}{\partial \theta_i} \right\rangle_{data} - \left\langle \frac{\partial E(\boldsymbol{x};\theta_i)}{\partial \theta_i} \right\rangle_{model} \tag{3.54}$$

$$\tag{3.55}$$

This expression shows that the gradient of the log-likelihood cost function is a **difference of moments**, with one calculated from the data and one calculated from the model. The data-dependent term is called the **positive phase** and the model-dependent term is called the **negative phase** of the gradient. We see now that minimizing the cost function results in lowering the energy of configurations $\boldsymbol{x}$ near points in the training data and increasing the energy of configurations not observed in the training data. That means we increase the model's probability of configurations similar to those in the training data.

The gradient of the cost function also demonstrates why gradients of unsupervised, generative models must be computed differently from for those of for example FNNs. While the data-dependent expectation value is easily calculated based on the samples $\boldsymbol{x}_i$ in the training data, we must sample from the model in order to generate samples from which to caclulate the model-dependent term. We sample from the model by using MCMC-based methods. We can not sample from the model directly because the partition function Z is generally intractable.

As in supervised machine learning problems, the goal is also here to perform well on *unseen* data, that is to have good generalization from the training data. The distribution $f(x)$ we approximate is not the *true* distribution we wish to

estimate, it is limited to the training data. Hence, in unsupervised training as well it is important to prevent overfitting to the training data. Thus it is common to add regularizers to the cost function in the same manner as described for the supervised methods.

3.4 Gradient Descent Optimization

Gradient descent is the most common method in machine learning for minimizing the cost function during the training stage. The goal is to iteratively update the parameters of the model in directions where the gradient of the cost function is large and negative. This procedure ensures that the parameters move towards a local minimum. In the simplest gradient descent scheme we start with initial parameter values $\boldsymbol{\theta}_0$ and update according to

$$\mathbf{v}_t = \eta_t \nabla_\theta \mathcal{C}(\boldsymbol{\theta}_t) \tag{3.56}$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t \tag{3.57}$$

where η_t is a parameter which controls the step size and is called the **learning rate**.

To see the benefits and limitations to the gradient descent method we can compare it to Newton's method, which is a *second-order method*. In Newton's method the update $\boldsymbol{v}$ of the parameters is chosen in order to minimize the second-order Taylor expansion of the cost function:

$$\mathcal{C}(\boldsymbol{\theta} + \mathbf{v}) \approx \mathcal{C}(\boldsymbol{\theta}) + \nabla_\theta \mathcal{C}(\boldsymbol{\theta})\mathbf{v} + \frac{1}{2}\mathbf{v}^T H(\theta)\mathbf{v} \tag{3.58}$$

where $H(\theta)$ is the Hessian matrix of second derivatives. Differentiating this expression with respect to $\boldsymbol{v}$ and using that we should have $\nabla_\theta E(\boldsymbol{\theta} + \mathbf{v}_{opt}) = 0$ we find

$$0 = \nabla_\theta E(\boldsymbol{\theta}) + H(\theta)\mathbf{v}_{opt} \tag{3.59}$$

rewriting yields the update rules for Newton's method

$$\mathbf{v}_t = H^{-1}(\theta_t)\nabla_\theta E(\theta_t) \tag{3.60}$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t \tag{3.61}$$

We cannot be sure that the Hessian is well conditioned so it is common to replace its inverse with a reregularized pseudo-inverse such as $[H(\theta_t) + \epsilon I]^{-1}$, where ϵ is a small parameter. We see that Netwon's method has the same form as gradient descent, with the inverse Hessian acting as the learning rate. The benefit of this is that, while gradient descent has the same learning rate for all the parameters, in this case it is automatically adapted to each parameter individually, taking

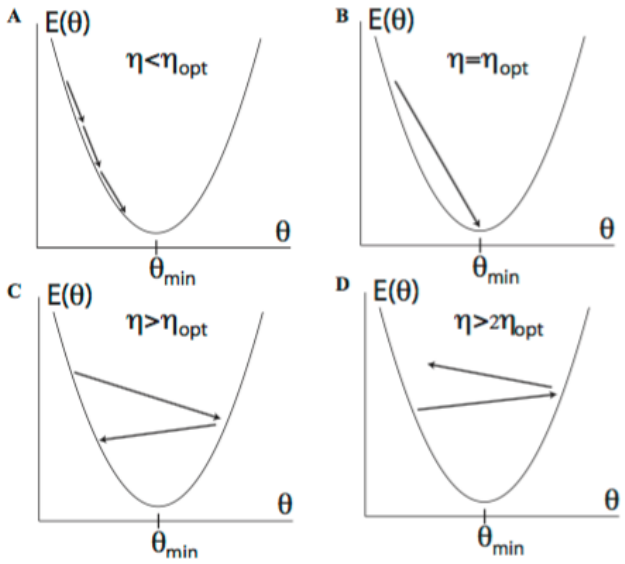

Figure 3.8: Effect of different sizes of the learning rate η on the gradient descent minimization algorithm. Taken from https://deeplearning4j.org/restrictedboltzmannmachine.

larger step in flat directions and smaller steps in steep directions. The reason is that the Hessian encodes the curvature of the surface, in the sense that its singular values are *inversly proportional to the squares of the local curvatures of the surface*. However there are reasons why gradient descent is often preferred to second-order methods. It is an extremely intensive computational operation to calculate the Hessian. And even if one uses first-order methods to approximate the Hessian, known as quasi-Newton methods, one must still store and invert a matrix with n^2 entries, where n is the number of parameters. [25]

The main consideration when choosing the gradient descent learning rate is a trade-off between accuracy and speed. While a small learning rate is guaranteed to converge to a local minimum, it may be very slow. A learning rate that is too big, on the other hand, can make the algorithm unstable, either oscillating around the minimum or even moving away from it, see fiugre 3.8. In the following sections we will look at some of the various improvements one can make to the simple gradient descent scheme.

3.4.1 Momentum

One of the simplest modifications to the simple gradient descent algorithm is to introduce a term called **momentum**, which functions as a memory of the

direction the algorithm is moving in parameter space. The update rule becomes

$$\mathbf{v}_t = \gamma \mathbf{v}_{t-1} + \eta_t \nabla_\theta E(\boldsymbol{\theta}_t) \tag{3.62}$$
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t, \tag{3.63}$$

where $0 \leq \gamma \leq 1$ is the momentum hyperparameter, and $\mathbf{v}_t$ becomes a moving average over the most recent gradients with $(1-\gamma)^{-1}$ setting the characteristic time scale for the memory. Introducing momentum is useful because it increases the step size in directions with persistent but small gradients, and prevents oscillations in directions with high curvature.

3.4.2 RMS-prop

As mentioned, one ideally wants to adapt the learning rate to the curvature of the landscape without having to calculate the computationally expensive inverse of the Hessian matrix of second derivatives. Over the last years a number of algorithms have been developed which achieve this by keeping track of the second moment of the gradient in addition to the gradient itself. Some of these methods include AdaGrad, AdaDelta, RMS-prop, and ADAM. We will discuss RMS-prop and the ADAM optimizer. Multiplication and division by vectors is to be taken as element-wise operations.

The RMS-prop algorithm keeps track of the second moment of the gradient with the term $\mathbf{s}_t = \mathbb{E}[\mathbf{g}_t^2]$. The update rule is then given by

$$\mathbf{g}_t = \nabla_\theta E(\boldsymbol{\theta}) \tag{3.64}$$
$$\mathbf{s}_t = \beta \mathbf{s}_{t-1} + (1-\beta)\mathbf{g}_t^2 \tag{3.65}$$
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \frac{\mathbf{g}_t}{\sqrt{\mathbf{s}_t + \epsilon}}. \tag{3.66}$$

Here β sets the averaging time of the second moment and typically has a value around $\beta = 0.9$. η_t is the learning rate, typically 10^{-3}, and $\epsilon \sim 10^{-8}$ is a regularization constant in order to prevent divergencies. As desired this update rule ensures that the learning rate is reduced in directions where the magnitude of the gradient is consistently large. We can therefore use a larger learning rate which speeds up the process.

3.4.3 ADAM

The ADAM optimizer keeps a moving average of both the first and second moment of the gradient, that is $\mathbf{m}_t = \mathbb{E}[\mathbf{g}_t]$ and $\mathbf{s}_t = \mathbb{E}[\mathbf{g}_t^2]$ respectively.

$$\mathbf{g}_t = \nabla_\theta E(\boldsymbol{\theta}) \tag{3.67}$$

$$\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\mathbf{g}_t \tag{3.68}$$

$$\mathbf{s}_t = \beta_2 \mathbf{s}_{t-1} + (1-\beta_2)\mathbf{g}_t^2 \tag{3.69}$$

$$\hat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{1-\beta_1^t} \tag{3.70}$$

$$\hat{\mathbf{s}}_t = \frac{\mathbf{s}_t}{1-\beta_2^t} \tag{3.71}$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{s}}_t} + \epsilon} \tag{3.72}$$

Here the variables denoted by hats represent a bias correction introduced because we estimate the moments of the gradient using a moving average. η and ϵ are the same as for RMS-prop and β_1 and β_2 control the memory lifetime of the first and second moment, typically taken as 0.9 and 0.99 respectively.

Like in RMS-prop the effective step size of a parameter depends on the magnitude of its gradient squared. We can investigate this further by looking at how a single parameter θ_t is updated in terms of the variance $\sigma_\mathbf{t}^\mathbf{2} = \hat{\mathbf{s}}_\mathbf{t} - (\hat{\mathbf{m}}_\mathbf{t})^\mathbf{2}$. The update can be written

$$\Delta\theta_{t+1} = -\eta_t \frac{\hat{m}_t}{\sqrt{\sigma_t^2 + m_t^2} + \epsilon}. \tag{3.73}$$

If we consider the limits of this expression we see that

- If $\sigma_t^2 \ll \hat{m}_t^2$ then $\Delta\theta_{t+1} \to -\eta_t$. If the variance is small while the gradient is big, meaning we are moving persistently in steep directions, the maximum step size is limited to η_t.
- If $\sigma^2 \gg \hat{m}_t^2$ then $\Delta\theta_{t+1} \to -\eta_t \hat{m}_t/\sigma_t$. This means that if the variance is big, so that there are great fluctuations in the gradient, the learning rate becomes proportional to the mean of the gradients in units of the standard deviation. This is good because the standard deviation is a natural and adaptive scale for determining whether the gradient is large or small.

In conclusion, the ADAM optimizer is able to configure the learning rate to not become too big in steep directions, avoiding oscillations and divergencies, as well as being adaptive when the gradient is experiencing large fluctuations.

While RMS-prop and ADAM allow for using a bigger learning rate which speeds up computations some have observed that these methods might not generalize as well. Generalization refers to whether the trained model preforms well on unseen test data after having been trained on the training data and is an important goal in machine learning.

Chapter 4

The Restricted Boltzmann Machine

The restricted Boltzmann machine (RBM) is an unsupervised, generative neural network which became known when Hinton *et al* presented an efficient algorithm for training it [28] and soon after used it as a building block in deep belief networks, which were introduced in a series of seminal papers that coined the term deep learning and reawakened interest in artificial neural networks [21] [22] [23].

This chapter gives an introduction to the restricted Boltzmann machine. First, the general Boltzmann machine, then the RBM and the original variant with binary visible and hidden units, the binary-binary RBM. Finally we will look at the variant used in this work, the Gaussian-binary RBM, as well as the RBM training procedure.

4.1 The Boltzmann Machine

The Boltzmann machine [29] is a type of Markov Random Field (section 3.3.1) where each pair of nodes is connected by an edge (see figure 4.1). This means it is a fully connected network, or a **complete graph**. Furthermore the nodes are defined as either **visible** nodes, denoted by $\boldsymbol{x}$, or **hidden** nodes, denoted $\boldsymbol{h}$. The visible nodes are the input and output. The hidden nodes are latent variables of which the purpose is to encode complex interactions between the visible nodes.

The joint probability density function of the variables is, similar to the Markov Random Field, given by

$$p_{BM}(\boldsymbol{x}, \boldsymbol{h}) = \frac{1}{Z_{BM}} e^{-\frac{1}{T} E_{BM}(\boldsymbol{x}, \boldsymbol{h})}, \tag{4.1}$$

with the partition function

$$Z_{BM} = \int \int e^{-\frac{1}{T} E_{BM}(\tilde{\boldsymbol{x}}, \tilde{\boldsymbol{h}})} \, \mathrm{d}\tilde{\boldsymbol{x}} \, \mathrm{d}\tilde{\boldsymbol{h}}. \tag{4.2}$$

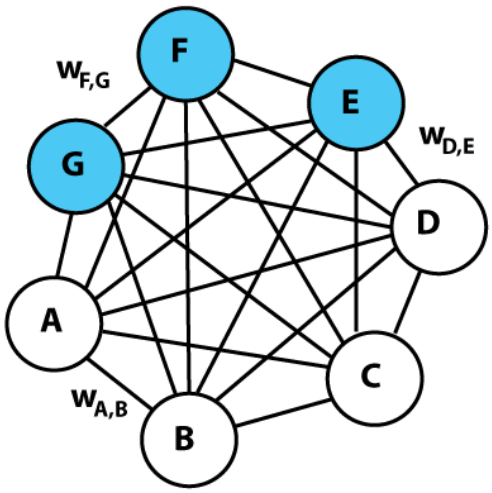

Figure 4.1: The the Boltzmann machine represented by an undirected, graphical model. Blue nodes are hidden units and white nodes are visible units. The Boltzmann machine is a complete graph, meaning each pair of nodes is connected by an edge. The nodes are stochastic, which makes it a probabilistic graphical model. Figure from https://en.wikipedia.org/wiki/Boltzmann_machine.

T is a physics-inspired parameter named temperature and will be assumed to be 1 unless otherwise stated. The energy function of the Boltzmann machine determines the interactions between the nodes and is defined

$$E_{BM}(\boldsymbol{x}, \boldsymbol{h}) = -\sum_{i,k}^{M,K} a_i^k \alpha_i^k(x_i) - \sum_{j,l}^{N,L} b_j^l \beta_j^l(h_j) - \sum_{i,j,k,l}^{M,N,K,L} \alpha_i^k(x_i) w_{ij}^{kl} \beta_j^l(h_j) - \sum_{i,m=i+1,k}^{M,M,K} \alpha_i^k(x_i) v_{im}^k \alpha_m^k(x_m) - \sum_{j,n=j+1,l}^{N,N,L} \beta_j^l(h_j) u_{jn}^l \beta_n^l(h_n). \quad (4.3)$$

Here $\alpha_i^k(x_i)$ and $\beta_j^l(h_j)$ are one-dimensional transfer functions or mappings from the given input value to the desired feature value. They can be arbitrary functions of the input variables and are independent of the parameterization (parameters referring to weight and biases), meaning they are not affected by training of the model. The indices k and l indicate that there can be multiple transfer functions per variable. Furthermore, a_i^k and b_j^l are the visible and hidden bias. w_{ij}^{kl} are weights of the **inter-layer** connection terms which connect visible and hidden units. v_{im}^k and u_{jn}^l are weights of the **intra-layer** connection terms which connect the visible units to each other and the hidden units to each other, respectively.

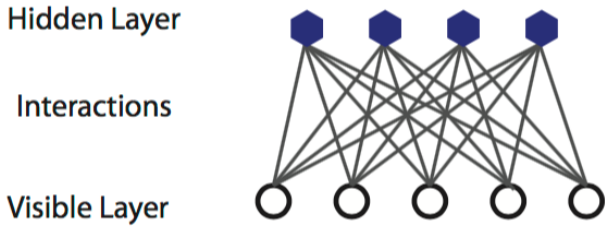

Figure 4.2: The bipartite graph structure of the restricted Boltzmann machine. In contrast to the general Boltzmann machine there are no connections within layers. Figure from [25].

4.1.1 Restricted Boltzmann Machines

The RBM was originally introduced in [30] with the name **harmonium**, since the energy function was referred to as the *harmony* at the time. It was introduced as the restricted Boltzmann machine by Hinton in [28]. The RBM is a simpler version of the Boltzmann machine where there are no connections within layers. This means the units within the visible layer are conditionally independent of each other and the same for the units in the hidden layer. This makes the RBM a **bipartite** graph, see figure 4.2.

The conditional independency within layers leads to a factorization property of RBMs when marginalizing out the visible or hidden units. The integral over all possible states of the visible layer factorizes into a product of one dimensional integrals over all possible values for the corresponding unit. Therefore, conditional probabilities can be computed efficiently. Having the conditional probabilities means Gibbs sampling, a simpler version of Metropolis-Hastings sampling, can be employed.

Energy Function

We remove the intra-layer connections by setting v_{im} and u_{jn} to zero. The expression for the energy of the RBM is then

$$E_{RBM}(\boldsymbol{x}, \boldsymbol{h}) = -\sum_{i,k}^{M,K} a_i^k \alpha_i^k(x_i) - \sum_{j,l}^{N,L} b_j^l \beta_j^l(h_j) - \sum_{i,j,k,l}^{M,N,K,L} \alpha_i^k(x_i) w_{ij}^{kl} \beta_j^l(h_j). \quad (4.4)$$

4.1.2 Binary-Binary Restricted Boltzmann Machines

The original RBM had binary visible and hidden nodes. They were showned to be universal approximators of discrete distributions in [31]. It was also shown that adding hidden units yields strictly improved modelling power. The common

Table 4.1: This table shows how the terms in the restricted Boltzmann machine (RBM) energy function (equation 4.4) should be implemented in order to yield the binary-binary restricted boltzmann machine, that is an RBM where both visible and hidden units take binary values.

Transfer functions	Biases	Weights
$\alpha_i^1(x_i) = x_i$	$a_i^1 = a_i$	$w_{ij}^{11} = w_{ij}$
$\beta_j^1(h_j) = h_j$	$b_j^1 = b_j$	

choice of binary values are 0 and 1. However, in some physics applications, -1 and 1 might be a more natural choice. We will here use 0 and 1.

Energy Function

Table 4.1 shows how the RBM energy function should be implemented for the binary-binary RBM, using only one transfer function, the identity, for each layer. This results in the energy

$$E_{BB}(\boldsymbol{x}, \mathbf{h}) = -\sum_{i}^{M} x_i a_i - \sum_{j}^{N} b_j h_j - \sum_{i,j}^{M,N} x_i w_{ij} h_j. \tag{4.5}$$

Joint Probability Density Function

With the energy given in equation 4.5, the joint probability density function of the units in the binary-binary RBM becomes

$$p_{BB}(\boldsymbol{x}, \boldsymbol{h}) = \frac{1}{Z_{BB}} e^{\sum_i^M a_i x_i + \sum_j^N b_j h_j + \sum_{ij}^{M,N} x_i w_{ij} h_j} \tag{4.6}$$

$$= \frac{1}{Z_{BB}} e^{\boldsymbol{x}^T \boldsymbol{a} + \boldsymbol{b}^T \boldsymbol{h} + \boldsymbol{x}^T \boldsymbol{W} \boldsymbol{h}} \tag{4.7}$$

with the partition function

$$Z_{BB} = \sum_{\boldsymbol{x}, \boldsymbol{h}} e^{\boldsymbol{x}^T \boldsymbol{a} + \boldsymbol{b}^T \boldsymbol{h} + \boldsymbol{x}^T \boldsymbol{W} \boldsymbol{h}}. \tag{4.8}$$

Marginal Probability Density Functions

In order to find the probability of any configuration of the visible units we derive the marginal probability density function.

$$
\begin{aligned}
p_{BB}(\boldsymbol{x}) =& \sum_{\boldsymbol{h}} p_{BB}(\boldsymbol{x}, \boldsymbol{h}) \qquad (4.9)\\
=& \frac{1}{Z_{BB}} \sum_{\boldsymbol{h}} e^{\boldsymbol{x}^T \boldsymbol{a} + \boldsymbol{b}^T \boldsymbol{h} + \boldsymbol{x}^T \boldsymbol{W} \boldsymbol{h}} \\
=& \frac{1}{Z_{BB}} e^{\boldsymbol{x}^T \boldsymbol{a}} \sum_{\boldsymbol{h}} e^{\sum_j^N (b_j + \boldsymbol{x}^T \boldsymbol{w}_{*j}) h_j} \\
=& \frac{1}{Z_{BB}} e^{\boldsymbol{x}^T \boldsymbol{a}} \sum_{\boldsymbol{h}} \prod_j^N e^{(b_j + \boldsymbol{x}^T \boldsymbol{w}_{*j}) h_j} \\
=& \frac{1}{Z_{BB}} e^{\boldsymbol{x}^T \boldsymbol{a}} \Bigg(\sum_{h_1} e^{(b_1 + \boldsymbol{x}^T \boldsymbol{w}_{*1}) h_1} \times \sum_{h_2} e^{(b_2 + \boldsymbol{x}^T \boldsymbol{w}_{*2}) h_2} \times \\
& \ldots \times \sum_{h_2} e^{(b_N + \boldsymbol{x}^T \boldsymbol{w}_{*N}) h_N} \Bigg) \\
=& \frac{1}{Z_{BB}} e^{\boldsymbol{x}^T \boldsymbol{a}} \prod_j^N \sum_{h_j} e^{(b_j + \boldsymbol{x}^T \boldsymbol{w}_{*j}) h_j} \\
=& \frac{1}{Z_{BB}} e^{\boldsymbol{x}^T \boldsymbol{a}} \prod_j^N (1 + e^{b_j + \boldsymbol{x}^T \boldsymbol{w}_{*j}}). \qquad (4.10)
\end{aligned}
$$

A similar derivation yields the marginal probability of the hidden units

$$
p_{BB}(\boldsymbol{h}) = \frac{1}{Z_{BB}} e^{\boldsymbol{b}^T \boldsymbol{h}} \prod_i^M (1 + e^{a_i + \boldsymbol{w}_{i*}^T \boldsymbol{h}}). \qquad (4.11)
$$

Conditional Probability Density Functions

We derive the probability of the hidden units given the visible units using Bayes' rule:

$$\begin{aligned} p_{BB}(\boldsymbol{h}|\boldsymbol{x}) =& \frac{p_{BB}(\boldsymbol{x},\boldsymbol{h})}{p_{BB}(\boldsymbol{x})} \\ =& \frac{\frac{1}{Z_{BB}} e^{\boldsymbol{x}^T\boldsymbol{a}+\boldsymbol{b}^T\boldsymbol{h}+\boldsymbol{x}^T\boldsymbol{W}\boldsymbol{h}}}{\frac{1}{Z_{BB}} e^{\boldsymbol{x}^T\boldsymbol{a}} \prod_j^N (1+e^{b_j+\boldsymbol{x}^T\boldsymbol{w}_{*j}})} \\ =& \frac{e^{\boldsymbol{x}^T\boldsymbol{a}} e^{\sum_j^N (b_j+\boldsymbol{x}^T\boldsymbol{w}_{*j})h_j}}{e^{\boldsymbol{x}^T\boldsymbol{a}} \prod_j^N (1+e^{b_j+\boldsymbol{x}^T\boldsymbol{w}_{*j}})} \\ =& \prod_j^N \frac{e^{(b_j+\boldsymbol{x}^T\boldsymbol{w}_{*j})h_j}}{1+e^{b_j+\boldsymbol{x}^T\boldsymbol{w}_{*j}}} \\ =& \prod_j^N p_{BB}(h_j|\boldsymbol{x}). \end{aligned} \tag{4.12}$$

From this we find the probability of a hidden unit being "on" or "off":

$$p_{BB}(h_j=1|\boldsymbol{x}) = \frac{e^{(b_j+\boldsymbol{x}^T\boldsymbol{w}_{*j})h_j}}{1+e^{b_j+\boldsymbol{x}^T\boldsymbol{w}_{*j}}} \tag{4.13}$$

$$= \frac{e^{(b_j+\boldsymbol{x}^T\boldsymbol{w}_{*j})}}{1+e^{b_j+\boldsymbol{x}^T\boldsymbol{w}_{*j}}} \tag{4.14}$$

$$= \frac{1}{1+e^{-(b_j+\boldsymbol{x}^T\boldsymbol{w}_{*j})}}, \tag{4.15}$$

and

$$p_{BB}(h_j=0|\boldsymbol{x}) = \frac{1}{1+e^{b_j+\boldsymbol{x}^T\boldsymbol{w}_{*j}}}. \tag{4.16}$$

We see that the outcome is the sigmoid function, the same as is frequently used as non-linear activation functions in feedforward neural networks (figure 3.3).

Similarly we have that the conditional probability of the visible units given the hidden are

$$p_{BB}(\boldsymbol{x}|\boldsymbol{h}) = \prod_i^M \frac{e^{(a_i+\boldsymbol{w}_{i*}^T\boldsymbol{h})x_i}}{1+e^{a_i+\boldsymbol{w}_{i*}^T\boldsymbol{h}}} \tag{4.17}$$

$$= \prod_i^M p_{BB}(x_i|\boldsymbol{h}). \tag{4.18}$$

Table 4.2: This table shows how the terms in the restricted Boltzmann machine (RBM) energy function (equation 4.4) should be implemented in order to yield the Gaussian-binary restricted boltzmann machine, that is an RBM where the visible units take continuous values and the hidden units take binary values.

Transfer functions	Biases	Weights
$\alpha_i^1(x_i) = -x_i^2$	$a_i^1 = \frac{1}{2\sigma_i^2}$	$w_{ij}^{11} = 0$
$\alpha_i^2(x_i) = x_i$	$a_i^2 = \frac{a_i}{\sigma_i^2}$	$w_{ij}^{21} = \frac{w_{ij}}{\sigma_i^2}$
$\alpha_i^3(x_i) = 1$	$a_i^3 = -\frac{a_i^2}{2\sigma_i^2}$	$w_{ij}^{31} = 0$
$\beta_j^1(h_j) = h_j$	$b_j^1 = b_j$	

Thus

$$p_{BB}(x_i = 1|\boldsymbol{h}) = \frac{1}{1 + e^{-(a_i + \boldsymbol{w}_{i*}^T \boldsymbol{h})}} \tag{4.19}$$

$$p_{BB}(x_i = 0|\boldsymbol{h}) = \frac{1}{1 + e^{a_i + \boldsymbol{w}_{i*}^T \boldsymbol{h}}}. \tag{4.20}$$

4.1.3 Gaussian-Binary Restricted Boltzmann Machines

In a number of problems the values we want to model are not binary, but continuous. There are several ways one might accomodate this. One way is the Gaussian-binary RBM [32]. In this model the hidden units are still binary, while the visible units are assumed to take real values $x_i \in [-\infty, \infty]$ and be normally distributed with variance σ_i^2.

Energy Function

We find the energy function of the Gaussian-binary RBM by implementing the RBM energy as shown in table 4.2. As seen there are now three transfer functions for the visible units ($K = 3$) and one for the hidden units ($L = 1$). Inserting into the expression for $E_{RBM}(\boldsymbol{x}, \boldsymbol{h})$ in equation 4.4 results in the energy

$$\begin{aligned} E_{GB}(\boldsymbol{x}, \boldsymbol{h}) &= \sum_i^M \frac{(x_i - a_i)^2}{2\sigma_i^2} - \sum_j^N b_j h_j - \sum_{ij}^{M,N} \frac{x_i w_{ij} h_j}{\sigma_i^2} \\ &= \left\| \frac{\boldsymbol{x} - \boldsymbol{a}}{2\boldsymbol{\sigma}} \right\|^2 - \boldsymbol{b}^T \boldsymbol{h} - (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T \boldsymbol{W} \boldsymbol{h}. \end{aligned} \tag{4.21}$$

Joint Probability Density Function

Given the energy in equation 4.21, joint probability density function of the Gaussian-binary RBM is

$$\begin{aligned} p_{GB}(\boldsymbol{x}, \boldsymbol{h}) =& \frac{1}{Z_{GB}} e^{-\left\| \frac{\boldsymbol{x}-\boldsymbol{a}}{2\boldsymbol{\sigma}} \right\|^2 + \boldsymbol{b}^T \boldsymbol{h} + (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T \boldsymbol{W} \boldsymbol{h}} \\ =& \frac{1}{Z_{GB}} e^{-\sum_i^M \frac{(x_i - a_j)^2}{2\sigma_i^2} + \sum_j^N b_j h_j + \sum_{ij}^{M,N} \frac{x_i w_{ij} h_j}{\sigma_i^2}} \\ =& \frac{1}{Z_{GB}} \prod_{ij}^{M,N} e^{-\frac{(x_i - a_j)^2}{2\sigma_i^2} + b_j h_j + \frac{x_i w_{ij} h_j}{\sigma_i^2}}, \end{aligned} \tag{4.22}$$

with the partition function given by

$$Z_{GB} = \int \sum_{\tilde{\boldsymbol{h}}}^{\tilde{\boldsymbol{H}}} e^{-\left\| \frac{\tilde{\boldsymbol{x}}-\boldsymbol{a}}{2\boldsymbol{\sigma}} \right\|^2 + \boldsymbol{b}^T \tilde{\boldsymbol{h}} + (\frac{\tilde{\boldsymbol{x}}}{\boldsymbol{\sigma}^2})^T \boldsymbol{W} \tilde{\boldsymbol{h}}} \, \mathrm{d}\tilde{\boldsymbol{x}}. \tag{4.23}$$

Marginal Probability Density Functions

We proceed to find the marginal probability densitites of the Gaussian-binary RBM. We first marginalize over the binary hidden units to find $p_{GB}(\boldsymbol{x})$

$$\begin{aligned} p_{GB}(\boldsymbol{x}) =& \sum_{\tilde{\boldsymbol{h}}}^{\tilde{\boldsymbol{H}}} p_{GB}(\boldsymbol{x}, \tilde{\boldsymbol{h}}) \\ =& \frac{1}{Z_{GB}} \sum_{\tilde{\boldsymbol{h}}}^{\tilde{\boldsymbol{H}}} e^{-\left\| \frac{\boldsymbol{x}-\boldsymbol{a}}{2\boldsymbol{\sigma}} \right\|^2 + \boldsymbol{b}^T \tilde{\boldsymbol{h}} + (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T \boldsymbol{W} \tilde{\boldsymbol{h}}} \\ =& \frac{1}{Z_{GB}} e^{-\left\| \frac{\boldsymbol{x}-\boldsymbol{a}}{2\boldsymbol{\sigma}} \right\|^2} \prod_j^N (1 + e^{b_j + (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T \boldsymbol{w}_{*j}}). \end{aligned} \tag{4.24}$$

We next marginalize over the visible units. This is the first time we marginalize over continuous values. We rewrite the exponential factor dependent on $\boldsymbol{x}$ as a Gaussian function before we integrate in the last step.

$$
\begin{aligned}
p_{GB}(\boldsymbol{h}) &= \int p_{GB}(\tilde{\boldsymbol{x}}, \boldsymbol{h})\, \mathrm{d}\tilde{\boldsymbol{x}} \\
&= \frac{1}{Z_{GB}} \int e^{-\left\| \frac{\tilde{\boldsymbol{x}}-\boldsymbol{a}}{2\boldsymbol{\sigma}} \right\|^2 + \boldsymbol{b}^T\boldsymbol{h} + (\frac{\tilde{\boldsymbol{x}}}{\boldsymbol{\sigma}^2})^T \boldsymbol{W}\boldsymbol{h}}\, \mathrm{d}\tilde{\boldsymbol{x}} \\
&= \frac{1}{Z_{GB}} e^{\boldsymbol{b}^T\boldsymbol{h}} \int \prod_i^M e^{-\frac{(\tilde{x}_i - a_i)^2}{2\sigma_i^2} + \frac{\tilde{x}_i \boldsymbol{w}_{i*}^T \boldsymbol{h}}{\sigma_i^2}}\, \mathrm{d}\tilde{\boldsymbol{x}} \\
&= \frac{1}{Z_{GB}} e^{\boldsymbol{b}^T\boldsymbol{h}} \Bigg(\int e^{-\frac{(\tilde{x}_1 - a_1)^2}{2\sigma_1^2} + \frac{\tilde{x}_1 \boldsymbol{w}_{1*}^T \boldsymbol{h}}{\sigma_1^2}}\, \mathrm{d}\tilde{x}_1 \\
&\quad \times \int e^{-\frac{(\tilde{x}_2 - a_2)^2}{2\sigma_2^2} + \frac{\tilde{x}_2 \boldsymbol{w}_{2*}^T \boldsymbol{h}}{\sigma_2^2}}\, \mathrm{d}\tilde{x}_2 \\
&\quad \times ... \\
&\quad \times \int e^{-\frac{(\tilde{x}_M - a_M)^2}{2\sigma_M^2} + \frac{\tilde{x}_M \boldsymbol{w}_{M*}^T \boldsymbol{h}}{\sigma_M^2}}\, \mathrm{d}\tilde{x}_M \Bigg) \\
&= \frac{1}{Z_{GB}} e^{\boldsymbol{b}^T\boldsymbol{h}} \prod_i^M \int e^{-\frac{(\tilde{x}_i - a_i)^2 - 2\tilde{x}_i \boldsymbol{w}_{i*}^T \boldsymbol{h}}{2\sigma_i^2}}\, \mathrm{d}\tilde{x}_i \\
&= \frac{1}{Z_{GB}} e^{\boldsymbol{b}^T\boldsymbol{h}} \prod_i^M \int e^{-\frac{\tilde{x}_i^2 - 2\tilde{x}_i(a_i + \tilde{x}_i \boldsymbol{w}_{i*}^T \boldsymbol{h}) + a_i^2}{2\sigma_i^2}}\, \mathrm{d}\tilde{x}_i \\
&= \frac{1}{Z_{GB}} e^{\boldsymbol{b}^T\boldsymbol{h}} \prod_i^M \int e^{-\frac{\tilde{x}_i^2 - 2\tilde{x}_i(a_i + \boldsymbol{w}_{i*}^T \boldsymbol{h}) + (a_i + \boldsymbol{w}_{i*}^T \boldsymbol{h})^2 - (a_i + \boldsymbol{w}_{i*}^T \boldsymbol{h})^2 + a_i^2}{2\sigma_i^2}}\, \mathrm{d}\tilde{x}_i \\
&= \frac{1}{Z_{GB}} e^{\boldsymbol{b}^T\boldsymbol{h}} \prod_i^M \int e^{-\frac{(\tilde{x}_i - (a_i + \boldsymbol{w}_{i*}^T \boldsymbol{h}))^2 - a_i^2 - 2a_i \boldsymbol{w}_{i*}^T \boldsymbol{h} - (\boldsymbol{w}_{i*}^T \boldsymbol{h})^2 + a_i^2}{2\sigma_i^2}}\, \mathrm{d}\tilde{x}_i \\
&= \frac{1}{Z_{GB}} e^{\boldsymbol{b}^T\boldsymbol{h}} \prod_i^M e^{\frac{2a_i \boldsymbol{w}_{i*}^T \boldsymbol{h} + (\boldsymbol{w}_{i*}^T \boldsymbol{h})^2}{2\sigma_i^2}} \int e^{-\frac{(\tilde{x}_i - a_i - \boldsymbol{w}_{i*}^T \boldsymbol{h})^2}{2\sigma_i^2}}\, \mathrm{d}\tilde{x}_i \\
&= \frac{1}{Z_{GB}} e^{\boldsymbol{b}^T\boldsymbol{h}} \prod_i^M \sqrt{2\pi\sigma_i^2} e^{\frac{2a_i \boldsymbol{w}_{i*}^T \boldsymbol{h} + (\boldsymbol{w}_{i*}^T \boldsymbol{h})^2}{2\sigma_i^2}}.
\end{aligned} \tag{4.25}
$$

Conditional Probability Density Functions

We finish by deriving the conditional probabilities.

$$\begin{aligned} p_{GB}(\boldsymbol{h}|\boldsymbol{x}) =& \frac{p_{GB}(\boldsymbol{x},\boldsymbol{h})}{p_{GB}(\boldsymbol{x})} \\ =& \frac{\frac{1}{Z_{GB}} e^{-\left\| \frac{\boldsymbol{x}-\boldsymbol{a}}{2\boldsymbol{\sigma}} \right\|^2 + \boldsymbol{b}^T\boldsymbol{h} + (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T \boldsymbol{W}\boldsymbol{h}}}{\frac{1}{Z_{GB}} e^{-\left\| \frac{\boldsymbol{x}-\boldsymbol{a}}{2\boldsymbol{\sigma}} \right\|^2} \prod_j^N (1 + e^{b_j + (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T \boldsymbol{w}_{*j}})} \\ =& \prod_j^N \frac{e^{(b_j + (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T \boldsymbol{w}_{*j}) h_j}}{1 + e^{b_j + (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T \boldsymbol{w}_{*j}}} \\ =& \prod_j^N p_{GB}(h_j|\boldsymbol{x}) \end{aligned} \tag{4.26}$$

The conditional probability of a binary hidden unit h_j being on or off again take the form of sigmoid functions

$$\begin{aligned} p_{GB}(h_j = 1|\boldsymbol{x}) =& \frac{e^{b_j + (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T \boldsymbol{w}_{*j}}}{1 + e^{b_j + (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T \boldsymbol{w}_{*j}}} \\ =& \frac{1}{1 + e^{-b_j - (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T \boldsymbol{w}_{*j}}} \end{aligned} \tag{4.27}$$

$$p_{GB}(h_j = 0|\boldsymbol{x}) = \frac{1}{1 + e^{b_j + (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T \boldsymbol{w}_{*j}}}. \tag{4.28}$$

The conditional probability of the continuous $\boldsymbol{x}$ now has another form, how-

ever.

$$
\begin{aligned}
p_{GB}(\boldsymbol{x}|\boldsymbol{h}) =& \frac{p_{GB}(\boldsymbol{x},\boldsymbol{h})}{p_{GB}(\boldsymbol{h})} \\
=& \frac{\frac{1}{Z_{GB}} e^{-\left\| \frac{\boldsymbol{x}-\boldsymbol{a}}{2\boldsymbol{\sigma}} \right\|^2 + \boldsymbol{b}^T\boldsymbol{h} + (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T \boldsymbol{W}\boldsymbol{h}}}{\frac{1}{Z_{GB}} e^{\boldsymbol{b}^T\boldsymbol{h}} \prod_i^M \sqrt{2\pi\sigma_i^2} e^{\frac{2a_i \boldsymbol{w}_{i*}^T\boldsymbol{h} + (\boldsymbol{w}_{i*}^T\boldsymbol{h})^2}{2\sigma_i^2}}} \\
=& \prod_i^M \frac{1}{\sqrt{2\pi\sigma_i^2}} \frac{e^{-\frac{(x_i-a_i)^2}{2\sigma_i^2} + \frac{x_i \boldsymbol{w}_{i*}^T\boldsymbol{h}}{2\sigma_i^2}}}{e^{\frac{2a_i \boldsymbol{w}_{i*}^T\boldsymbol{h} + (\boldsymbol{w}_{i*}^T\boldsymbol{h})^2}{2\sigma_i^2}}} \\
=& \prod_i^M \frac{1}{\sqrt{2\pi\sigma_i^2}} \frac{e^{-\frac{x_i^2 - 2a_i x_i + a_i^2 - 2x_i \boldsymbol{w}_{i*}^T\boldsymbol{h}}{2\sigma_i^2}}}{e^{\frac{2a_i \boldsymbol{w}_{i*}^T\boldsymbol{h} + (\boldsymbol{w}_{i*}^T\boldsymbol{h})^2}{2\sigma_i^2}}} \\
=& \prod_i^M \frac{1}{\sqrt{2\pi\sigma_i^2}} e^{-\frac{x_i^2 - 2a_i x_i + a_i^2 - 2x_i \boldsymbol{w}_{i*}^T\boldsymbol{h} + 2a_i \boldsymbol{w}_{i*}^T\boldsymbol{h} + (\boldsymbol{w}_{i*}^T\boldsymbol{h})^2}{2\sigma_i^2}} \\
=& \prod_i^M \frac{1}{\sqrt{2\pi\sigma_i^2}} e^{-\frac{(x_i - b_i - \boldsymbol{w}_{i*}^T\boldsymbol{h})^2}{2\sigma_i^2}} \\
=& \prod_i^M \mathcal{N}(x_i | b_i + \boldsymbol{w}_{i*}^T\boldsymbol{h}, \sigma_i^2) \qquad (4.29) \\
\Rightarrow p_{GB}(x_i|\boldsymbol{h}) =& \mathcal{N}(x_i | b_i + \boldsymbol{w}_{i*}^T\boldsymbol{h}, \sigma_i^2). \qquad (4.30)
\end{aligned}
$$

The form of these conditional probabilities explain the name "Gaussian" and the form of the Gaussian-binary energy function. We see that the conditional probability of x_i given $\boldsymbol{h}$ is a normal distribution with mean $b_i + \boldsymbol{w}_{i*}^T\boldsymbol{h}$ and variance σ_i^2.

4.2 Training the RBM

The cost function of the RBM is most commonly chosen to be the negative log-likelihood and will have a similar form as discussed in 3.3.1. It is usually minimized using gradient descent algorithms like those discussed in 3.4. The one thing we will explain here however is the Gibbs sampling used to compute the gradient of the log likelihood cost function. In order to compute the gradient we need to sample configurations $\boldsymbol{x}$ from the model. This is done using MCMC methods. Since we usually know the conditional probabilities we can use a special case of Metropolis-Hastings called Gibbs sampling.

4.2.1 Gibbs Sampling

The Gibbs sampling method use the same framework as the Metropolis-Hastings algorithm (section 2.3.4) employing Markov Chains and Monte Carlo sampling. The only difference lies in how the update step is implemented. Recall that the Metropolis Hastings acceptance step is

$$A(\boldsymbol{x}^f, \boldsymbol{x}^b) = \min\Big(1, \frac{P(\boldsymbol{x}^f)Q(\boldsymbol{x}^b|\boldsymbol{x}^f)}{P(\boldsymbol{x}^b)Q(\boldsymbol{x}^f|\boldsymbol{x}^b)}\Big) \tag{4.31}$$

where the before and final states are denoted by b and f respectively and these are made superscripts in order to easily index the vector components with subscripts.

Gibbs sampling offer a smart way of choosing the proposal distribution depending on the desired distribution. Given the desired distribution $p(\boldsymbol{x}) = p(x_1, ...x_D)$ we need to formulate the proposal distribution as the conditional probability of a variable x_i given all the other variables $\boldsymbol{x}_{\setminus i} = \{x_1, ..., x_D\} \setminus \{x_i\}$. The proposal distribution is then given by $p(x_i|\boldsymbol{x}_{\setminus i})$ and we can use it to express the desired distribution by $p(\boldsymbol{x}) = p(x_i|\boldsymbol{x}_{\setminus i})p(\boldsymbol{x}_{\setminus i})$. We then insert this into the Metropolis-Hastings acceptance probability.

$$\begin{aligned}
A(\boldsymbol{x}^f, \boldsymbol{x}^b) =& \min\Big(1, \frac{P(\boldsymbol{x}^f)Q(\boldsymbol{x}^b|\boldsymbol{x}^f)}{P(\boldsymbol{x}^b)Q(\boldsymbol{x}^f|\boldsymbol{x}^b)}\Big) && (4.32)\\
=& \min\Big(1, \frac{P(\boldsymbol{x}^f)P(x_i^b|\boldsymbol{x}_{\setminus i}^f)}{P(\boldsymbol{x}^b)P(x_i^f|\boldsymbol{x}_{\setminus i}^b)}\Big) && (4.33)\\
=& \min\Big(1, \frac{P(x_i^f|\boldsymbol{x}_{\setminus i}^f)P(\boldsymbol{x}_{\setminus i}^f)P(x_i^b|\boldsymbol{x}_{\setminus i}^f)}{P(x_i^b|\boldsymbol{x}_{\setminus i}^b)P(\boldsymbol{x}_{\setminus i}^b)P(x_i^f|\boldsymbol{x}_{\setminus i}^b)}\Big) && (4.34)\\
=& \min\Big(1, \frac{P(x_i^f|\boldsymbol{x}_{\setminus i}^b)P(\boldsymbol{x}_{\setminus i}^b)P(x_i^b|\boldsymbol{x}_{\setminus i}^b)}{P(x_i^b|\boldsymbol{x}_{\setminus i}^b)P(\boldsymbol{x}_{\setminus i}^b)P(x_i^f|\boldsymbol{x}_{\setminus i}^b)}\Big) && (4.35)\\
=& 1, && (4.36)
\end{aligned}$$

where in 4.36 we used that only $\boldsymbol{x}_i^b$ is updated to $\boldsymbol{x}_i^f$ and so $\boldsymbol{x}_{\setminus i}^f = \boldsymbol{x}_{\setminus i}^b$. It turns out that the ratio becomes one, which means all samples are accepted in Gibbs sampling.

In the RBM the visible units are conditionally independent of each other and it is the same for the hidden units. The proposal distribution is therefore

$$\begin{aligned}
Q(x_i|\boldsymbol{x}_{\setminus i}^b, \boldsymbol{h}) =& p_{RBM}(x_i|\boldsymbol{h}) && (4.37)\\
Q(h_j|\boldsymbol{x}, \boldsymbol{h}_{\setminus j}^b) =& p_{RBM}(h_j|\boldsymbol{x}). && (4.38)
\end{aligned}$$

4.3 Neural Quantum States

4.3.1 The wavefunction

The wavefunction should be a probability amplitude depending on $\boldsymbol{x}$. The RBM model is given by the joint distribution of $\boldsymbol{x}$ and $\boldsymbol{h}$

$$F_{rbm}(\mathbf{x}, \mathbf{h}) = \frac{1}{Z} e^{-\frac{1}{T}E(\mathbf{x},\mathbf{h})} \tag{4.39}$$

To find the marginal distribution of $\boldsymbol{x}$ we set:

$$F_{rbm}(\mathbf{x}) = \sum_{\mathbf{h}} F_{rbm}(\mathbf{x}, \mathbf{h}) \tag{4.40}$$

$$= \frac{1}{Z} \sum_{\mathbf{h}} e^{-E(\mathbf{x},\mathbf{h})} \tag{4.41}$$

Now this is what we use to represent the wave function, calling it a neural-network quantum state (NQS)

$$\Psi(\mathbf{X}) = F_{rbm}(\mathbf{x}) \tag{4.42}$$

$$= \frac{1}{Z} \sum_{\boldsymbol{h}} e^{-E(\mathbf{x},\mathbf{h})} \tag{4.43}$$

$$= \frac{1}{Z} \sum_{\{h_j\}} e^{-\sum_i^M \frac{(x_i-a_i)^2}{2\sigma^2} + \sum_j^N b_j h_j + \sum_{i,j}^{M,N} \frac{x_i w_{ij} h_j}{\sigma^2}} \tag{4.44}$$

$$= \frac{1}{Z} e^{-\sum_i^M \frac{(x_i-a_i)^2}{2\sigma^2}} \prod_j^N (1 + e^{b_j + \sum_i^M \frac{x_i w_{ij}}{\sigma^2}}) \tag{4.45}$$

$$\tag{4.46}$$

Positive definite wave function

The above wavefunction is the most general one because it allows for complex valued wavefunctions. However it fundamentally changes the probabilistic foundation of the RBM, because what is usually a probability in the RBM framework is now a an amplitude. This means that a lot of the theoretical framework usually used to interpret the model, i.e. graphical models, conditional probabilities, and Markov random fields, breaks down. It becomes harder to interpret what the hidden nodes represent, however an example from the literature of similar use of latent variables to be marginalized over are those of ghost wavefunctions, introduced in the 1980s. They are however constructed with connections between units different from the RBM structure.

If we assume the wavefunction to be postive definite, however, we can use the RBM to represent the squared wavefunction, and thereby a probability. This also makes it possible to sample from the model using Gibbs sampling, because we can obtain the conditional probabilities.

$$
\begin{aligned}
|\Psi(\mathbf{X})|^2 &= F_{rbm}(\mathbf{X}) && (4.47)\\
\Rightarrow \Psi(\mathbf{X}) &= \sqrt{F_{rbm}(\mathbf{X})} && (4.48)\\
&= \frac{1}{\sqrt{Z}}\sqrt{\sum_{\{h_j\}} e^{-E(\mathbf{X},\mathbf{h})}} && (4.49)\\
&= \frac{1}{\sqrt{Z}}\sqrt{\sum_{\{h_j\}} e^{-\sum_i^M \frac{(X_i-a_i)^2}{2\sigma^2}+\sum_j^N b_j h_j+\sum_{i,j}^{M,N}\frac{X_i w_{ij} h_j}{\sigma^2}}} && (4.50)\\
&= \frac{1}{\sqrt{Z}} e^{-\sum_i^M \frac{(X_i-a_i)^2}{4\sigma^2}}\sqrt{\sum_{\{h_j\}}\prod_j^N e^{b_j h_j+\sum_i^M \frac{X_i w_{ij} h_j}{\sigma^2}}} && (4.51)\\
&= \frac{1}{\sqrt{Z}} e^{-\sum_i^M \frac{(X_i-a_i)^2}{4\sigma^2}}\sqrt{\prod_j^N\sum_{h_j} e^{b_j h_j+\sum_i^M \frac{X_i w_{ij} h_j}{\sigma^2}}} && (4.52)\\
&= \frac{1}{\sqrt{Z}} e^{-\sum_i^M \frac{(X_i-a_i)^2}{4\sigma^2}}\prod_j^N\sqrt{e^0+e^{b_j+\sum_i^M \frac{X_i w_{ij}}{\sigma^2}}} && (4.53)\\
&= \frac{1}{\sqrt{Z}} e^{-\sum_i^M \frac{(X_i-a_i)^2}{4\sigma^2}}\prod_j^N\sqrt{1+e^{b_j+\sum_i^M \frac{X_i w_{ij}}{\sigma^2}}} && (4.54)\\
& && (4.55)
\end{aligned}
$$

4.3.2 Cost function

This is where we deviate from what is common in machine learning. Rather than defining a cost function based on some dataset, our cost function is the energy of the quantum mechanical system. From the variational principle we know that minizing this energy should lead to the ground state wavefunction. As stated previously the local energy is given by

$$E_L = \frac{1}{\Psi}\hat{\mathbf{H}}\Psi \tag{4.56}$$

and the gradient is

$$G_i = \frac{\partial\langle E_L\rangle}{\partial\alpha_i} = 2(\langle E_L\frac{1}{\Psi}\frac{\partial\Psi}{\partial\alpha_i}\rangle - \langle E_L\rangle\langle\frac{1}{\Psi}\frac{\partial\Psi}{\partial\alpha_i}\rangle) \tag{4.57}$$

where $\alpha_i = a_1, ..., a_M, b_1, ..., b_N, w_{11}, ..., w_{MN}$.

We use that $\frac{1}{\Psi}\frac{\partial \Psi}{\partial \alpha_i} = \frac{\partial \ln \Psi}{\partial \alpha_i}$ and find

$$\ln \Psi(\mathbf{X}) = -\ln Z - \sum_m^M \frac{(X_m - a_m)^2}{2\sigma^2} + \sum_n^N \ln(1 + e^{b_n + \sum_i^M \frac{X_i w_{in}}{\sigma^2}}) \tag{4.58}$$

Giving

$$\frac{\partial}{\partial a_m} \ln \Psi = \frac{1}{\sigma^2}(X_m - a_m) \tag{4.59}$$

$$\frac{\partial}{\partial b_n} \ln \Psi = \frac{1}{e^{-b_n - \frac{1}{\sigma^2}\sum_i^M X_i w_{in}} + 1} \tag{4.60}$$

$$\frac{\partial}{\partial w_{mn}} \ln \Psi = \frac{X_m}{\sigma^2(e^{-b_n - \frac{1}{\sigma^2}\sum_i^M X_i w_{in}} + 1)} \tag{4.61}$$

If $\Psi = \sqrt{F_{rbm}}$

$$\ln \Psi(\mathbf{X}) = -\frac{1}{2}\ln Z - \sum_m^M \frac{(X_m - a_m)^2}{4\sigma^2} + \frac{1}{2}\sum_n^N \ln(1 + e^{b_n + \sum_i^M \frac{X_i w_{in}}{\sigma^2}}) \tag{4.62}$$

Giving

$$\frac{\partial}{\partial a_m} \ln \Psi = \frac{1}{2\sigma^2}(X_m - a_m) \tag{4.63}$$

$$\frac{\partial}{\partial b_n} \ln \Psi = \frac{1}{2(e^{-b_n - \frac{1}{\sigma^2}\sum_i^M X_i w_{in}} + 1)} \tag{4.64}$$

$$\frac{\partial}{\partial w_{mn}} \ln \Psi = \frac{X_m}{2\sigma^2(e^{-b_n - \frac{1}{\sigma^2}\sum_i^M X_i w_{in}} + 1)} \tag{4.65}$$

Essentially we just multiply by a factor half.

Computing E_L

We repeat the Hamiltonian of the quantum dot system, which is given by

$$\hat{\mathbf{H}} = \sum_p^P (-\frac{1}{2}\nabla_p^2 + \frac{1}{2}\omega^2 r_p^2) + \sum_{p<q} \frac{1}{r_{pq}} \tag{4.66}$$

where the first summation term represents the standard harmonic oscillator part and the latter the repulsive interaction between two electrons. Natural units

($\hbar = c = e = m_e = 1$) are used, and P is the number of particles. This gives us the following expression for the local energy (D being the number of dimensions)

$$E_L = \frac{1}{\Psi}\mathbf{H}\Psi \tag{4.67}$$

$$= \frac{1}{\Psi}(\sum_p^P(-\frac{1}{2}\nabla_p^2 + \frac{1}{2}\omega^2 r_p^2) + \sum_{p<q}\frac{1}{r_{pq}})\Psi \tag{4.68}$$

$$= -\frac{1}{2}\frac{1}{\Psi}\sum_p^P \nabla_p^2\Psi + \frac{1}{2}\omega^2\sum_p^P r_p^2 + \sum_{p<q}\frac{1}{r_{pq}} \tag{4.69}$$

$$= -\frac{1}{2}\frac{1}{\Psi}\sum_p^P\sum_d^D \frac{\partial^2\Psi}{\partial x_{pd}^2} + \frac{1}{2}\omega^2\sum_p^P r_p^2 + \sum_{p<q}\frac{1}{r_{pq}} \tag{4.70}$$

$$= \frac{1}{2}\sum_p^P\sum_d^D(-(\frac{\partial}{\partial x_{pd}}\ln\Psi)^2 - \frac{\partial^2}{\partial x_{pd}^2}\ln\Psi + \omega^2 x_{pd}^2) + \sum_{p<q}\frac{1}{r_{pq}} \tag{4.71}$$

$$\tag{4.72}$$

Now if each visible node in the Boltzmann machine represents one coordinate of one particle, this can be written as

$$E_L = \frac{1}{2}\sum_m^M(-(\frac{\partial}{\partial v_m}\ln\Psi)^2 - \frac{\partial^2}{\partial v_m^2}\ln\Psi + \omega^2 v_m^2) + \sum_{p<q}\frac{1}{r_{pq}} \tag{4.73}$$

Where we have that

$$\frac{\partial}{\partial x_m}\ln\Psi = -\frac{1}{\sigma^2}(x_m - a_m) + \frac{1}{\sigma^2}\sum_n^N \frac{w_{mn}}{e^{-b_n - \frac{1}{\sigma^2}\sum_i^M x_i w_{in}} + 1} \tag{4.74}$$

$$\frac{\partial^2}{\partial x_m^2}\ln\Psi = -\frac{1}{\sigma^2} + \frac{1}{\sigma^4}\sum_n^N \omega_{mn}^2\frac{e^{b_n + \frac{1}{\sigma^2}\sum_i^M x_i w_{in}}}{(e^{b_n + \frac{1}{\sigma^2}\sum_i^M x_i w_{in}} + 1)^2} \tag{4.75}$$

We now have all the expressions neeeded to calculate the gradient of the expected local energy with respect to the RBM parameters $\frac{\partial\langle E_L\rangle}{\partial\alpha_i}$.

If $\Psi = \sqrt{F_{rbm}}$

$$\frac{\partial}{\partial x_m}\ln\Psi = -\frac{1}{2\sigma^2}(x_m - a_m) + \frac{1}{2\sigma^2}\sum_n^N \frac{w_{mn}}{e^{-b_n - \frac{1}{\sigma^2}\sum_i^M x_i w_{in}} + 1} \tag{4.76}$$

$$\frac{\partial^2}{\partial x_m^2}\ln\Psi = -\frac{1}{2\sigma^2} + \frac{1}{2\sigma^4}\sum_n^N \omega_{mn}^2\frac{e^{b_n + \frac{1}{\sigma^2}\sum_i^M x_i w_{in}}}{(e^{b_n + \frac{1}{\sigma^2}\sum_i^M x_i w_{in}} + 1)^2} \tag{4.77}$$

Again the only difference being that we multiply by a factor half.

Part II

Implementation and Results

Chapter 5

Implementation

5.1 Structure

The code developed as a part of this work can be visited at https://github.com/vildemf/Master. In developing the structure of the code other libraries with relevant methods have been considered for inspiration. Both QMC packages such as QWALK and QMCPACK and RBM libraries such as Paysage and PyDeep, and finally the NetKet library which provides neural quantum states for spin lattice problems.

With inspiration from QMC methods we have a `Hamiltonian` class and a `NeuralQuantumState` class (figure 5.1). The former stores information about the Hamiltonian of the system, for example the harmonic oscillator frequency ω and provides methods for computing kinetic and potential energy. The latter stores information about the NQS wavefunction, such as the weights and biases, and provide methods for a number of tasks like computing the wavefunction and computing its Laplacian and the gradient with respect to the network parameters.

With inspiration from RBM libraries (and machine learning libraries in general) we have a class which represents the *model* called `QuantumModel`, a class which allows for *sampling* from the model called `MonteCarloMethod` and a class which *trains* the model called `Trainer` (figure 5.1). The `QuantumModel` class controls the `Hamiltonian` and `NeuralQuantumState` classes. If a user creates an instance of `MonteCarloMethod` the `QuantumModel` is also provided with a `Sampler` object. While the `Sampler` object is used by the `MonteCarloMethod` class and can be made quite general, an efficient implementation is model-dependent to the extent that it was deemed most useful to let it be controlled by the model class. Finally, there is the `GradientDescent` class of which an instance is controlled by the `Trainer` class.

While a central goal for the code has been efficiency, another has been extendability. Since there are several parts of the code which solve almost distinct problems implementing distinc methods, a natural choice of programming paradigm

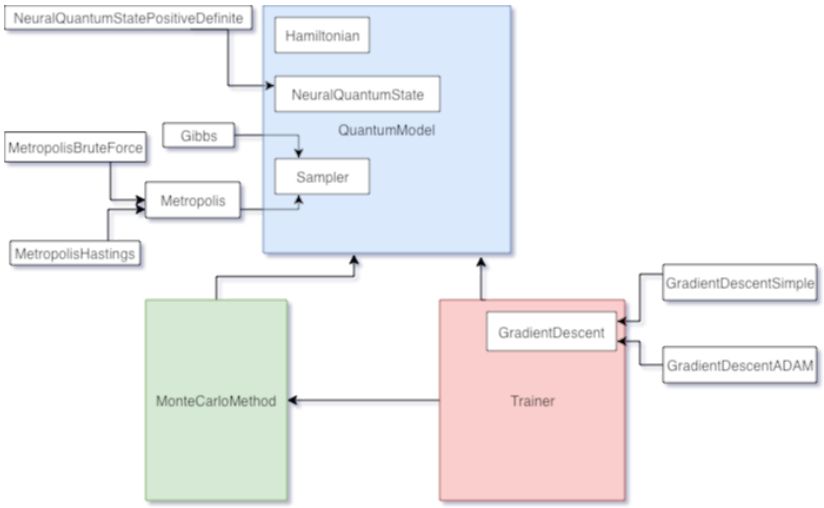

Figure 5.1: Class structure. Colored classes are the ones the user interacts with. Colored classes are wrapped around the white ones (instances) placed inside them. Arrows between colored classes indicate which colored classes interact with each other. The direction of the arrow signifies that one class "acts on" the other. Arrows between white classes signifies inheritence. The direction of the arrow signifies the base class.

is **object oriented programming (OOP)**. Some of the benefits of OOP is that it allows programmers to focus on parts of the code without taking into account details of how the other parts are implemented. This means extensions or changes can be made within a part of the code without the rest of the program being affected. Some of the means to reach this end inclue **encapsulation**, which means that by making details of a class private we know that other parts of the code do not depend on them, and **modularity**, which means organizing the code in a modular structure where it is natural apply encapsulation. The class structure described in the previous paragraph is built on these principles.

Another important concept in object oriented programming is **polymorphism**. The greek word translates to "many forms" and is the idea of the same interface having different implementations. This can be extremely useful. A simple example of polymorphism which has been employed is function **overloading**. This happens when a class implements the same function twice but with different input parameters. When the function is called, it is the implementation which corresponds to the given input parameters that automatically overloads the other. This is for example used in the function `quantumForce()` in the `NeuralQuantumState` class where a slightly different implementation is given depending on whether one wants to compute the quantum force for the current state of the system or for a trial state. Function overloading is an example of compile time polymorphism because it is known at compile time which

implementation will be executed.

Another example of polymorphism lends itself by considering the implementation of the sampling algorithm. As we have seen, there are several versions of this method to choose between. However, most parts of the code do not need to be aware of this. For example, the `QuantumModel` class only needs to know that it should call a `sample()` method, not how this method is implemented. Polymorphism allows for en elegant way of making use of this relationship in the implementation. We have done so by making `Sampler` a base class with derived classes `Metropolis` and `Gibbs`. All of them have the method `sample()`. While the derived classes inherit the base class method and thus have two implementations of the same method, one implementation will **override** the other and be the one actually executed. By default it is the derived implementation that overrides the base. Making several implementations of the same function interface is signlaled by the `virtual` keyword. When the base class represents a general concept of which only more concrete object variants are actually used, we can make it an **abstract base class (ABC)** [33]. This is done by making the base class' virtual method a **pure** virtual method by setting it to zero. ABCs are only meant to be used as a base class for subsequent derivations. It is therefore illegal to create objects of ABCs. One can, however, declare **pointers** to them. Pointers allow for efficient use of the polymorphic implementation. If we make the `QuantumModel` class store the sampler object in a `Sampler` pointer, it is possible for this pointer to point to an object of either `Metropolis` or `Gibbs`. The appropriate implementation to execute is invoked at run-time depending on what derived type the `Sampler` pointer is pointing to. As opposed to function overloading, then, function overriding is referred to as run-time polymorphism. Using virtual functions adds some time and space overhead to the class size and function invocation time. Using them is thus a trade-off between flexibility and efficiency.

One should also consider when to use **dynamic memory allocation**. While often deemed dangerous because it is the programmer's responsibility to ensure that memory is freed at the right time, it is useful in cases where one does not know the size or numbers of objects needed, the precise type or one wants several objects to share data. When using dynamically allocated memory in this project we have used **smart pointers**, which automatically deletes objects when no pointers are referring to them, making the use more "safe".

5.2 How to Use

The user has to specify three class instances in order to run the code: the model, the sampling method and the trainer. In this section we will go through how they work and what options the user has.

The Model

First, we show an example of how to initialize the model.

```
random_device rd;

double harmonicoscillatorOmega = 1.0;
bool coulombinteraction        = true;
string nqsType                 = "positivedefinite";
string nqsInitialization       = "randomgaussian";
int nParticles                 = 2;
int nDimensions                = 2;
int nHidden                    = 2;
int seed1                      = rd();

shared_ptr<QuantumModel> model = make_shared<QuantumModel>
            (harmonicoscillatorOmega,
              coulombinteraction,
              nqsType,
              nqsInitialization,
              nParticles,
              nDimensions,
              nHidden,
              seed1);
```

First one specifies the ω of the harmonic oscillator potential, which can take any range of values of physical interest. The boolean coulombinteraction argument determines whether the system is interacting or non-interacting. The argument nqsType specifies the type of neural quantum state model to be used. Specifically one can choose between the general model, indicated by the string "general", or the model which assumes the wavefunction to be positive definite, indicated by the string "positivedefinite" as shown above.

The parameter nqsInitialization determines how the weights and biases of the model should be initialized and consists of three options. The choice "randomuniform" initializes the parameters with uniform random numbers between -1 and 1, while "randomgaussian" initializes the parameters from a Gaussian distribution with zero mean and standard deviation of 0.1. Finally, the user can instead give a filename from which to read parameters. This is useful if one wishes to use a model that has already been trained in a different run. The format of such a file should be as shown in figure 5.2. If one chooses to write the parameter of the current model to file, using the memberfunction writeParametersToFile(filename) it will be written on this format ("filename" here and in the following paragraphs refers to any string with the user's preferred filename).

The next three input arguments, nParticles, nDimensions, and nHidden, are quite intuitive and are the number of particles, dimensions and hidden units respectively. The final argument is a seed which is given to a pseudo-random Mersienne Twister generator. This is used for random initialization of weights and biases as discussed above as well as for random initialization of particle positions. One can either use a non-deterministic random device as shown above to generate the

seed, or one can give it a fixed number. The latter option is useful for debugging purposes or when investigating the effects of other choices in the code, creating the need to keep everyhting else fixed.

The Sampling Method

As has perhaps become clear, in order to be able to do anything with a generative model of this kind, we need a sampling method. This is provided by the MonteCarloMethod class. An example of how to intialize one for using the Metropolis method is shown below. We assume that a model has already been initialized in the manner outlined in the previous section.

```
int numberOfSamples = 1e6;
string samplertype  = "importancesampling";
double step         = 0.4;
int seed2           = rd();

MonteCarloMethod method(numberOfSamples, model, seed2, samplertype,
    step);
```

Here, numberOfSamples determines how many samples will be generated. Keep in mind that the 10% first samples will not be used in the calculation of expectation values due to the equilibration time before the Markov chain reaches the desired stationary distribution, as discussed in section 2.3.4 (a rather brute force way of accounting for this effect, which could be made more sophisticated).

The argument samplertype determines what kind of sampler method will be used. For the Metropolis method, the user can choose between "bruteforce" and "importancesampling". To use the Gibbs sampler the argument is "gibbs". In case of Metropolis brute force sampling the step argument refers to the parameter Δx while in the case of importance sampling it refers to δt. The Gibbs sampler needs no tuning by the user, rendering the step argument superfluous such that the method class should be initialized by

```
MonteCarloMethod method(numberOfSamples, model, seed2, samplertype);
```

There is one restriction on which model can be used with what sampler: the Gibbs sampler can only sample from a positive definite model, since only this model allows for the necessary conditional probabilities. Otherwise, all cominbations are allowed.

There are two possibilities for the user to store quantities during sampling. The first is to store the sample energies, which is needed in order to compute standard errors with the blocking method after the run. To do this the user should use the member function setWriteEnergiesForBlocking(filename). The second possibility is to count particle positions in order to compute one body densities. To do so the user should call the memberfunction setOneBodyDensities(filename).

When the method is set up, it is run by calling

```
method.runMonteCarlo();
```

The Trainer

Finally, we will look at how to initialize the trainer class. If the model has been initialized randomly and not read from a file with pre-trained parameters, the user will want to train it using this class before doing any sampling from it. The trainer class does however use the sampling method in order to compute energies and gradients at each minimization iteration. Hence, both a model and a method is assumed to have been already defined according to the description given in previous sections in the following. An example of initialization is shown below.

```
int numberOfIterations = 500;
double learningrate    = 0.02;
double gamma           = 0.05
string minimizertype   = "simple";

Trainer trainer(numberOfIterations, learningrate, minimizertype, gamma);
```

The first two arguments determine the number of iterations, or learning epochs, and the learning rate of the gradient descent algorithm. The third argument determines the γ parameter of the momentum method. minimizertype determines which type of gradient descent should be used, and the user can choose between "simple" for the basic version or "adam" for the ADAM optimizer. Momentum is only applied to the simple gradient descent method and can be left out if the ADAM optimizer is used, that is the initialization becomes

```
Trainer trainer(numberOfIterations, learningrate, minimizertype);
```

It can also be left out if the user just wants simple gradient descent without momentum.

If the user wants to store quantities during training, the memberfunction setWriteIterativeExpectations(filename) should be called. It stores the expected energy and the norm of the gradient of the energy with respect to weights and biases at each iteration.

When the trainer class has been set up, the user can train a model by calling the method

```
trainer.train(method, *model);
```

An example of the output of such a run is given in figure 5.3. Note that the standard error reported here is the naïve error that does not take into account

Table 5.1: Validation of the non-interacting case.

		2D		3D	
# particles	ω	E_{NQS} [a.u.]	E_0 [a.u.]	E_{NQS} [a.u.]	E_0 [a.u.]
2	0.1	0.2000000(3)	0.2	0.3000000(1)	0.3
	0.5	1.000000(1)	1.0	1.500000(2)	1.5
	1.0	2.000000(7)	2.0	3.000000(3)	3.0
10	0.1	1.0000000(4)	1.0	1.5000000(4)	1.5
	0.5	5.000000(3)	5.0	7.500000(4)	7.5
	1.0	10.00000(1)	10.0	15.000000(9)	15.0
20	0.1	2.0000000(6)	2.0	3.0000000(4)	3.0
	0.5	10.000000(5)	10.0	15.000000(4)	15.0
	1.0	20.000000(7)	20.0	30.00000(1)	30.0

autocorrelation, since proper errors must be found with the blocking method after the run is completed.

5.3 Validation

The code has been validated mainly by comparing results with analytically known values. This is documented in the following two sections. Furthermore, section 6.1 on method selection also provides a thorough inspection of the results of different variants of each method and parameter, yielding further validation of the implementation.

The runs in the following sections have been done using the positive definite model and the Gibbs sampler with 10^{-5} samples during training and 10^{-6} samples in the final sampling from the trained model. Training has been done using the ADAM optimizer with learning rate $\eta = 0.02$ and 125 iterations in the non-interacting case and 500 iterations in the interacting case.

5.3.1 Non-Interacting Case

Recall that for non-interacting bosons in a harmonic oscillator potential, the energy can be calculated analytically using the formula

$$E_0 = \frac{1}{2}\omega \cdot D \cdot P, \tag{5.1}$$

```
a0 a1 a2 ...
b0 b1 b2 ...
w00 w01 w02 ...
w10 w11 w12 ...
...
```

Figure 5.2: Format of model parameter input and output file storing weights and biases. The first row is the visible bias, the seond row the hidden bias, and remaining rows are the rows of the weight matrix.

```
----------------------------------------
Local energy:           2.01341
Standard error:         0.000169661
Acceptance ratio:       0.91247
Gradient norm:          0.215756
Training epoch 0
----------------------------------------
Local energy:           2.00606
Standard error:         0.000109725
Acceptance ratio:       0.911805
Gradient norm:          0.134874
Training epoch 1
----------------------------------------
Local energy:           2.0027
Standard error:         7.25173e-05
Acceptance ratio:       0.912383
Gradient norm:          0.086243
Training epoch 2
----------------------------------------
```

Figure 5.3: Example of output from the training procedure.

Table 5.2: Validation of the interacting case for a system of two particles.

# dimensions	ω	E_{NQS} [a.u.]	E_0 [a.u.]
2	1.0	3.065(2)	3.0
3	0.1	0.5155(1)	0.5
	0.5	2.0260(3)	2.0

where ω is the harmonic oscillator frequency, D the number of dimensions and P the number of particles. We use this to provide a number of analytical results for different frequencies and number of particles and dimensions in order to benchmark the code. See the results in table 5.1.

In the table, we observe that the code provides excellent results for the energy with a standard deviation at the order of $\sim 10^{-5} - 10^{-7}$. This corresponds to a variance of the order $10^{-10} - 10^{-14}$.

5.3.2 Interacting Case

For the interacting case it is harder to provide analytical results. However, since the case of two particles interacting can be interpreted as two fermions, we can use the analytical results found by Taut in [3] and [4]. That is, we have analytical results for two dimensions with $\omega = 1.0$ and three dimensions with $\omega = 0.1$ and $\omega = 0.5$. See the comparison to the results produced by the code in table 5.2.

We observe that the results are far from as good as in the non-interacting case. First of all the model, and/or the minimization, does not succeed as well, since the absolute error is a couple of orders of magnitude bigger than the standard error from the sampling method. Secondly, the standard error is also worse than in the previous section, which also corresponds to a bigger variance, at the order of $\sim 10^{-6} - 10^{-8}$. Again, this is a sign that the model has not been as successful, since we know that a good model should produce a small variance. We do however believe that this has to do with the model, and not with the implementation. This will be further discussed in section 6.2, but one main reason to believe so is that the model does not provide an electron-electron cusp like the Jastrow factor, for example, usually does. Hence, given these limitations of the model, we believe the above results are still close enough to conclude that the implementation is satisfactory.

Chapter 6

Results

6.1 Method Selection

Our study of the restricted Boltzmann machine for the quantum dot system is dependent on several methods, of which there are several variants to choose between and several parameters to fine-tune. We will therefore start this chapter by documenting the effects of different choices. This allows us to determine the optimal combinations for further use while also ensuring that each method works as expected.

6.1.1 Sampling Method

We first study the different sampling methods, where the implemented methods are Gibbs sampling, Metropolis brute force and Metropolis with importance sampling. The metric of interest is the computational time taken to produce the highest number of independent samples and the smallest standard error. The number of independent samples and the autocorrelation time is computed with the automated blocking code provided by [15].

Before we compare the sampling methods we must first determine the optimal parameters of each method. While Gibbs sampling does not require the user to choose parameters, both versions of the Metropolis method do. In Metropolis brute force we have to choose the step size Δx while for importance sampling we have to choose the time interval δt. The goal of both these choices is to minimize the autocorrelation time τ so that the sampling is as efficient as possible - that is, as many as possible of the generated samples are independent samples which improve the Monte Carlo approximation. The fact that there is a minimum of τ with resepect to both these quantities can be understood intuitively. Both quantities contribute to the proposed new sample configuration. The bigger these quantities are, the bigger the change from the previous sample configuration will be, and the less the two configurations are correlated. However, due to the

Metropolis acceptance step, if the change is very big, the likelihood of the change beeing accepted decreases. If the change is not accepted, the new sample is the same as the previous one, and the two configurations are very correlated. This means we expect an optimal middle ground that mininize τ.

Throughout this section we are looking at a system of two particles in two dimensions with two hidden units and a harmonic oscillator frequency of $\omega = 1$. We are generating 10^6 Monte Carlo samples (only 10^5 during training iterations) and are training with the ADAM optimizer with a learning rate $\eta = 0.02$.

Metropolis Brute Force Step Size

We begin the search for the optimal step size Δx by looking at different orders of magnitude. The result is seen in table 6.1. We have included both the non-interacting and interacting case in order to detect whether interaction may affect the optimal parameter choice.

The first observation is that, as expected, the naïve estimate of the standard error σ continuously underestimates the standard deviation compared to the more accurate blocking estimate which takes correlation into account. In fact, in the non-interacting case, there is a difference of several orders of magnitude, demonstrating the importance of applying the blocking method in order to have a realistic idea of the variance.

Secondly, we observe the minimum we expected in the autocorrelation time. Both in the non-interacting and interacting case this occurs at $\Delta x \sim 10^0$. We notice how a smaller correlation time leads to a larger number of independent samples, as desired. Observing the numbers one realizes the severe effects of the autocorrelation time. A run that generated $\sim 10^6$ samples has in some cases only produced $\sim 10^2$ *independent* samples, which, as seen in section 2.3.2 greatly reduces the quality of the Monte Carlo approximation. There is a direct correspondence between τ and the number of independent samples: if the total number of samples is 2^{20} and τ is found to be 2^8, then the number of independent samples is $2^{20-8} = 2^{12}$. The reason for using powers of two is that the automated blocking implementation used here assumes it. When a certain block size is found to be insufficient, the next attempted size is the previous size multplied by two. This means that the estimates we obtain for τ here are rather coarse-grained, however they are found to be sufficient for our purpose, and it is a huge benefit that the implementation provides an automated process.

Finally the acceptance ratio has been included. Since it can be calculated directly rather than by applying the blocking method after computations it is the quickest and most practical quantity to keep track of during program runs. From this table it seems that the most optimal value is in the range of 20-30%. However, we will study all quantities in more detail.

Having found the optimal order of magnitude for Δx, we next test different values within this range. See the result in figure 6.1. We observe that the value

Table 6.1: Comparison of autocorrelation time τ and effective number of independent samples for different step sizes in the brute force Metropolis method run with $\sim 10^6$ cycles. The system istwo particles in two dimensions.

Δx	E [a.u.]	naïve σ	blocking σ	τ	# i.s.	accept
Non-interacting						
$5 \cdot 10^{-2}$	2.00011	$1.06 \cdot 10^{-6}$	$8 \cdot 10^{-5}$	8192	128	0.989
$5 \cdot 10^{-1}$	2.00001	$9.86 \cdot 10^{-7}$	$1 \cdot 10^{-5}$	512	2048	0.892
$5 \cdot 10^{0}$	2.00000	$9.70 \cdot 10^{-7}$	$4 \cdot 10^{-6}$	256	4096	0.229
$5 \cdot 10^{1}$	2.00003	$9.74 \cdot 10^{-7}$	$3 \cdot 10^{-5}$	4096	256	0.003
Interacting						
$5 \cdot 10^{-2}$	3.09795	$2.4 \cdot 10^{-3}$	$4 \cdot 10^{-2}$	4096	256	0.990
$5 \cdot 10^{-1}$	3.08866	$2.2 \cdot 10^{-3}$	$5 \cdot 10^{-3}$	1024	1024	0.904
$5 \cdot 10^{0}$	3.06929	$1.2 \cdot 10^{-3}$	$3 \cdot 10^{-3}$	128	8192	0.273
$5 \cdot 10^{1}$	3.10412	$2.2 \cdot 10^{-3}$	$5 \cdot 10^{-2}$	1024	1024	0.003

of Δx which minimize τ is in the range $2.5 - 4.0$. We see that the shortest autocorrelation time coincides with the smallest standard error. This corresponds to the fact that a higher number of independent samples should reduce the variance of the Monte Carlo approximate. We notice that the interacting case produces less correlated samples, however the optimal step size is roughly the same for both the interacting and non-interacting case.

As expected, the acceptance ratio decreases as the step size increases and proposed changes become more drastic. We observe that the optimal step size seem to correspond to an acceptance ratio of around 50%. This corresponds to what is typically used as a guiding value in VMC methods using the brute force Metropolis algorithm. [34].

Metropolis Importance Sampling Time Interval

We approach the choice of the parameter δt in a similar manner and begin by studying different orders of magnitude. See the results in table 6.2. Again we find that there is an order of magnitude which minimize τ both in the interacting and non-interacting case, that is $\delta t \sim 10^{-1}$.

As before we make a more detailed study of the found optimal range of δt. See the result in figure 6.2. Although the optimal choice within this range seems less pointed than in the brute force case, the smallest values of τ are found for

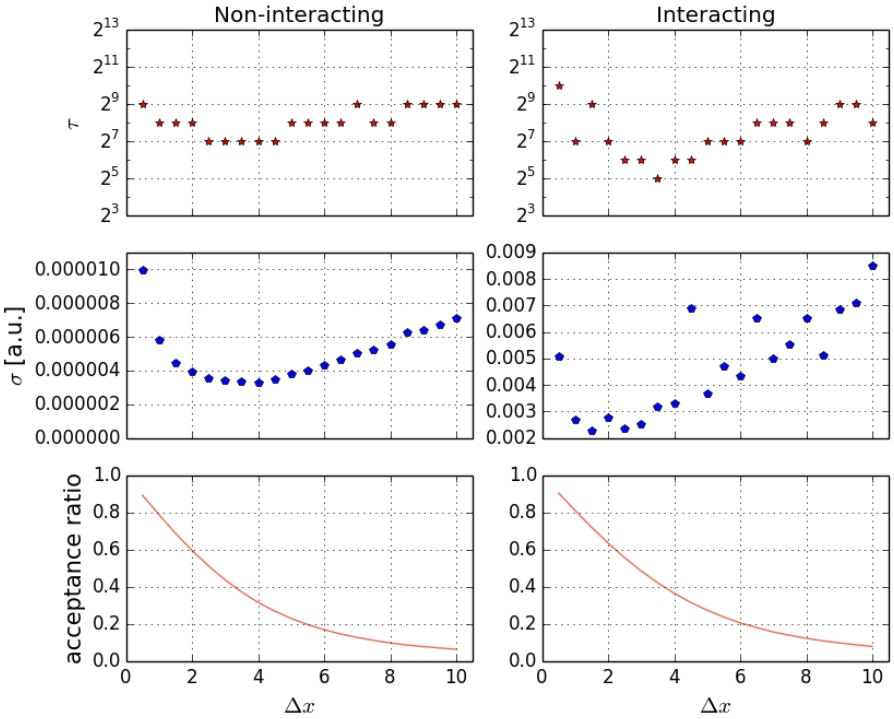

Figure 6.1: Detailed inspection of the effect of the Metropolis brute force step size.

Table 6.2: Comparison of autocorrelation time τ and effective number of independent samples for different step sizes in the importance sampled Metropolis method run with $\sim 10^6$ cycles. The system is two particles in two dimensions.

δt	E [a.u.]	naïve σ	blocking σ	τ	# i.s.	accept
Non-interacting						
$5 \cdot 10^{-3}$	2.00003	$1.0 \cdot 10^{-6}$	$2 \cdot 10^{-5}$	2048	512	0.9999
$5 \cdot 10^{-2}$	2.00001	$9.9 \cdot 10^{-7}$	$6 \cdot 10^{-6}$	256	4096	0.9960
$5 \cdot 10^{-1}$	2.00000	$9.8 \cdot 10^{-7}$	$2 \cdot 10^{-6}$	64	16384	0.8756
$5 \cdot 10^{0}$	1.99992	$8.3 \cdot 10^{-7}$	$2 \cdot 10^{-5}$	2048	512	0.0311
Interacting						
$5 \cdot 10^{-3}$	3.07128	$1.4 \cdot 10^{-3}$	$8.4 \cdot 10^{-3}$	1024	1024	0.9999
$5 \cdot 10^{-2}$	3.08210	$2.0 \cdot 10^{-3}$	$3.3 \cdot 10^{-3}$	128	8192	0.9968
$5 \cdot 10^{-1}$	3.07796	$1.7 \cdot 10^{-3}$	$1.8 \cdot 10^{-3}$	32	32768	0.8999
$5 \cdot 10^{0}$	3.08855	$1.2 \cdot 10^{-3}$	$7.2 \cdot 10^{-3}$	1024	1024	0.0521

δt around 0.3-0.6.

Comparing Gibbs, Metropolis Brute Force and Metropolis Importance Sampling

Having found the best parameters for the Metropolis methods, we can now compare them to each other and to the Gibbs method. We start with a visual comparison shown in figure 6.3. The figure shows the accuracy of each method during the training procedure in the non-interacting and interacting case, with the shaded area showing the standard error of each method. We observe that all three methods produce very similar energies. This is good, because the sampling methods should not alter the results of the model or of the optimization procedure. We do however notice a difference in the standard error produced by each method. Both in the non-interacting and interacting case the standard error of the Metropolis brute force method is wider than the other two. We would expect this because we assume the Metropolis brute force method is producing fewer independent samples. In the non-interacting case we see that the Gibbs method has an even smaller error than the Metropolis importance sampling method. One reason for this is probably that the Gibbs method updates the position of all particles for each new sample, whereas the Metropolis method has been implemented to only update the position of one particle at the time. In the interacting case it

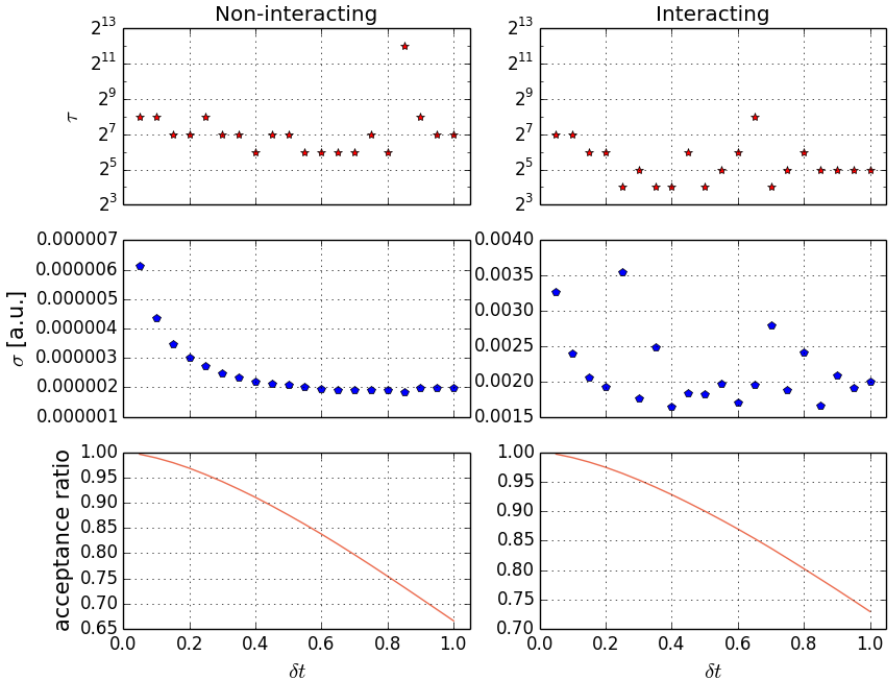

Figure 6.2: Detailed importance sampling step size.

seems that this effect is over-shadowed by the effect of interaction reducing the autocorrelation time, which we saw in the two previous sections.

The final comparison between the three sampling methods is provided in table 6.3. As observed in figure 6.3, the standard error of the Metropolis brute force method is an order of magnitude larger than the other two. Furthermore, the standard error of the Gibbs method is less than half the size of the Metropolis importance sampling method. As expected we also see that a small standard error correlates with a small autocorrelation and large number of independent samples. In this table we have also included the computational time, since the goal is to find the method that produces the largest number of independent samples (and thereby the smallest standard error) in the shortest amount of time. The observed computational times are as expected. The Gibbs method is the slowest, probably because, as mentioned, it updates of the positions of all particles rather than just one in each step. The second slowest is Metropolis importance sampling, which is expected because it has more elaborate computations of the proposed step and acceptance probability, including quantities such as the quantum force and Greens function ratio. Finally, the quickest method is the Metropolis brute force method, which has the simplest procedure for updating new configurations. The final column in the table shows the number of independent samples produced pr. second and we conclude that the Gibbs sampling method provides the most

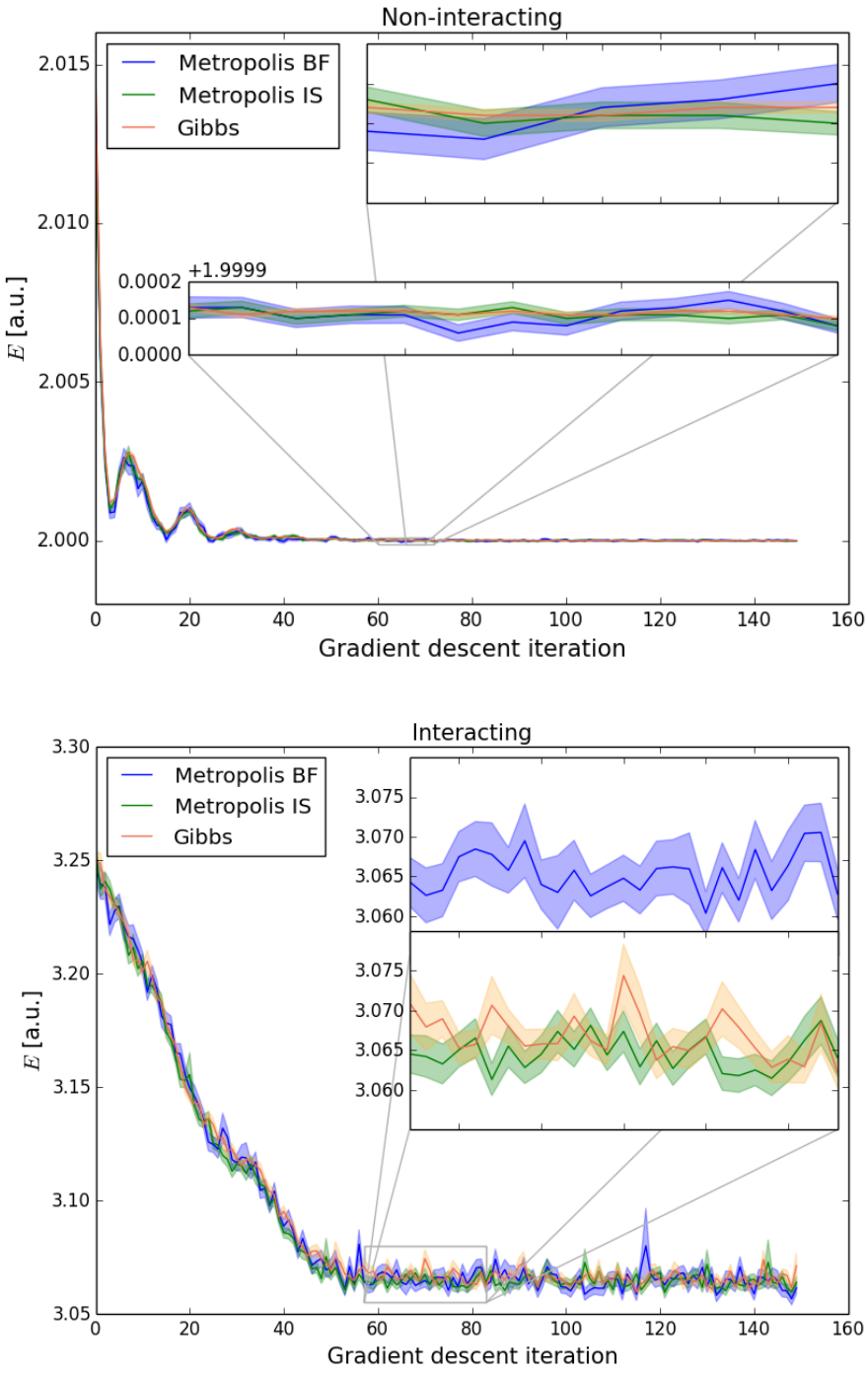

Figure 6.3: Comparison of the standard error (shaded erea) of the Gibbs method, Metropolis brute force and Metropolis importance sampling methods.

Table 6.3: Comparison of autocorrelation time τ and effective number of independent samples for different step sizes for Gibbs sampling, Metropolis brute force and Metropolis importance sampling, run with $\sim 10^6$ cycles. The system is a non-interacting case of two particles in two dimensions.

Method	Energy [a.u.]	blocking σ	τ	# i. s.	time [s]	$\frac{\#\text{ i. s.}}{\text{time}}$ $[\frac{1}{s}]$
Gibbs	2.000000	$3 \cdot 10^{-6}$	32	32768	5.7	5749
M. BF	2.000000	$1 \cdot 10^{-5}$	128	8192	2.9	2825
M. IS	2.000000	$8 \cdot 10^{-6}$	64	16384	4.1	3996

efficient solution.

6.1.2 Optimization Method

Next, we study the optimization methods. The implementations considered are gradient descent, gradient descent with momentum and the ADAM optimizer.

Gradient Descent Learning Rate

First we consider simple gradient descent with three different orders of the learning rate η, see figure 6.4. Based on the absolute error ΔE and the magnitude of the gradient $\nabla_\alpha E$ with respect to the weights and biases we see that the minimization is most efficiently carried out with a learning rate of the order $\sim 10^{-1}$.

Gradient Descent Momentum

Next we consider the gradient descent algorithm where a momentum term has been added to the updating rule. In this case the parameter γ controls the characteristic time scale of the memory of the direction the algorithm is moving. Figure 6.5 shows the minimization procedure for three different γ. The learning rate is set to the best value found from figure 6.4, $\eta = 0.5$. We see that the the learning procedure is already efficient due to this choice of η. However, we also see that the momentum term makes a difference. In particular the term with γ of the order $\sim 10^{-1}$ has an absolute error and gradient which decrease faster than the others.

The ADAM Optimizer Learning Rate

Finally we study the ADAM optimization procedure for three different choice of the learning rate η, see figure 6.6. While we see that the biggest learning rate is

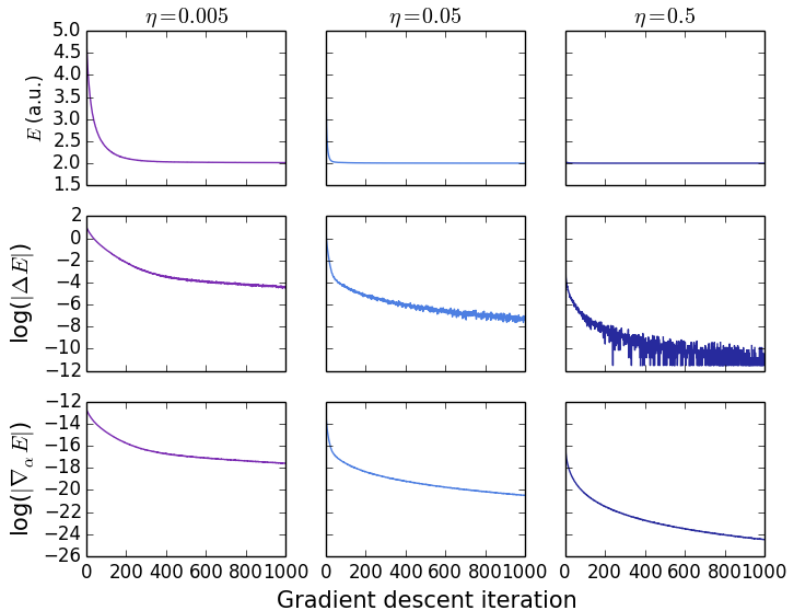

Figure 6.4: Three different values of the learning rate η of a simple gradient descent minimization procedure. The system is two non-interacting particles in two dimensions.

the fastest to achieve a minimum during the first iterations, it flattens out and seemingly starts to oscillate. The next biggest learning rate $\eta = 0.5$ on the other hand continues with a steady decrease. Hence a learning rate of the order $\sim 10^{-2}$ seem to be the best choice here.

Comparing Gradient Descent, Gradient Descent with Momentum and the ADAM Optimizer

Comparing the best results of these three methods, we see that all achieve an absolute error of the order $\sim 10^{-10}$ within a thousand training epochs. Simple gradient descent is the slowest, reaching this accuracy at around 800 iterations. The momentum variant and the ADAM optimizer are quicker, reaching this accuracy after 6-700 epochs. We will in the following be using the ADAM optimizer.

6.1.3 Number of Hidden Units

We study the effect of the number of hidden nodes in figure 6.7. In this case we included interaction which makes the energy values slightly less stable. While none of the different numbers of hidden units stand out in particular the number $N = 4$ seem to provide slightly more stable results and quicker convergence. Since this is a system of two particles in two dimensions we have 4 visible units. This

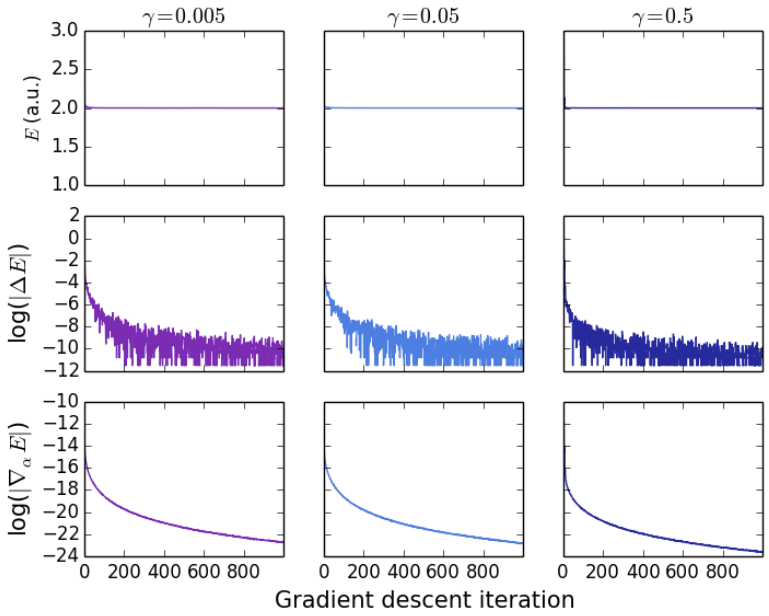

Figure 6.5: Three different values of the momentum hyperparameter γ of a simple gradient descent minimization procedure. γ controls the characteristic time scale of the memory of the direction the algorithm is moving. The system is two non-interacting particles in two dimensions.

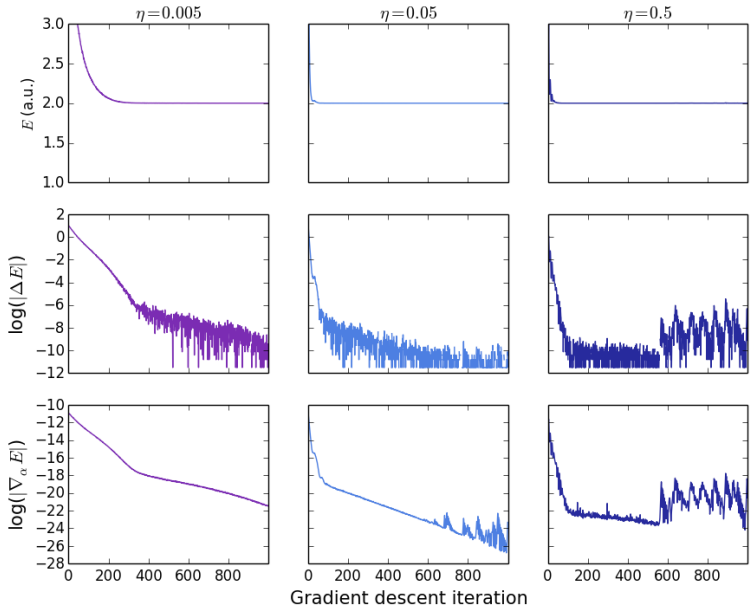

Figure 6.6: Results from the ADAM optimizer for different sizes of the learning rate η.

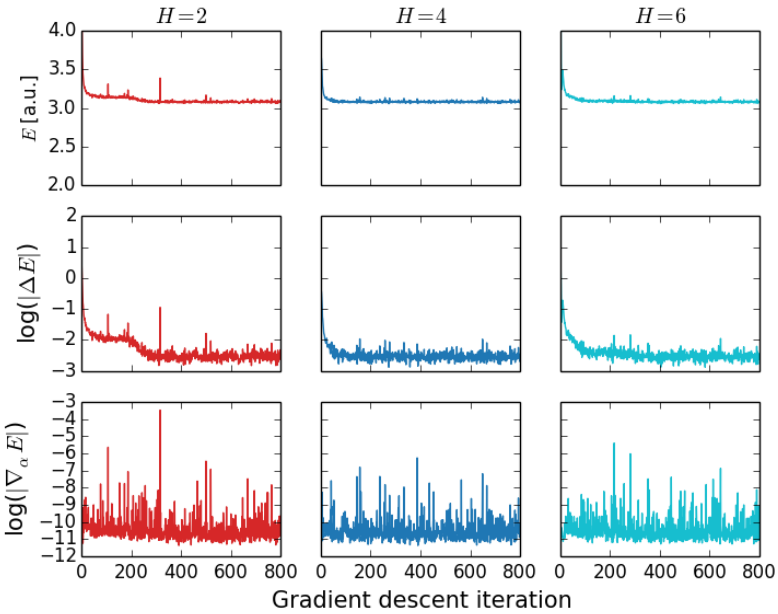

Figure 6.7: Studying the effect of different numbers of hidden nodes, using the ADAM optimizer with $\eta = 0.05$. The system is two interacting particles in two dimensions of which the analytically correct result of the energy is 3 a.u.

Table 6.4: Ground state energies in 2D. E_{HF}, E_{VMC} and E_{VMC-J} are the Hartree Fock energy, and the VMC energy without and with a Jastrow factor respectively, from [35]. E_{DMC} are DMC energies, calculated in [**?**]. E_{DMC} may be assumed to represent exact results.

ω	E_{NQS}	E_{HF}	E_{VMC}	E_{VMC-J}	E_{DMC}	ϵ_{NQS}	ϵ_{VMC-J}
0.01	0.07855(2)	-	-	0.0797255(1)	0.073839(2)	6%	8.00%
0.1	0.4707(3)	-	-	0.45156772(1)	0.44079(1)	7%	3.00%
0.28	1.0590(6)	-	-	1.0264107(2)	1.02164(1)	4%	1.00%
0.5	1.710(1)	-	-	1.6633891(2)	1.65977(1)	3%	0.20%
1.0	3.062(2)	3.253	3.172	3.0010648(3)	3.00000(1)	2%	0.04%

result then could indicate that a number of hidden units equal to that of visible units is the most useful choice. While the representational powers of the model should only improve with the number of hidden nodes, increasing the number increases the number of parameters to train, which can cause the optimization procedure requiring more time and a smaller learning rate in order to remain stable. Additionally, an increase in hidden nodes increase the computational time of quantities like the energy and gradient. Therefore using as many hidden nodes as possible is not a preferred choice.

6.2 Results

6.2.1 Benchmarking

We begin the result section by benchmarking our method against other standard methods in order to compare performance. Since the non-interacting case has already been documented to perform well in the validation section (see table 5.1) we will here focus on the interacting case. See the results for two particles in two dimensions in table 6.4. Here, the most interesting method to benchmark against is the VMC method, since it is similar to our method in every aspect except for the trial wavefunction and its number of variational parameters. It is also the method that we hope to be able to enhance if the neural quantum state model were to be successful. In short, the only difference between the neural quantum state results and the VMC method presented here is that the trial wavefunction of the former consists of a restricted Boltzmann machine with eight variational parameters (two hidden units have been used), while the trial wavefunction of the latter consists of a slater determinant of single-particle wavefunctions multplied with a Jastrow factor to account for correlations, with a total of two variational

parameters. Furthermore, we have included diffusion Monte Carlo (DMC) results as these can be considered exact results for our purposes [35, 36]. In order to compare the performance of the NQS to the standard VMC method we have also included the relative errors of these two methods with respect to the DMC results.

We observe that with one exception, the VMC results are consistently better than for the NQS. Since the NQS models well the non-interacting case, this is likely due to the fact that it does not fulfill the electron-electron cusp conditions. This assumption is supported by studying the results in more detail. We see that the difference in relative error between the NQS and VMC method increases with the harmonic oscillator frequency. In fact, for a very small frequency of $\omega = 0.01$, the NQS performs *better* than the VMC method, with a relative error of 6% compared to the VMC relative error of 8%. For the largest frequencies, on the other hand, the VMC method's relative error is better by an order of magnitude.

A high frequency decreases the space in which the electrons are confined, hence for a high frequency the electrons are more likely to cross each other's path, causing diverging energies when one does not have the electron-electron cusp to cancel this divergence. For a small frequency on the other hand, this becomes less of an issue as the electrons are not as confined and will spread out more, so that path-crossing and diverging energies will occur less often. The tendency to spread out for a lower frequency is demonstrated in the one-body densities in section 6.2.2.

Given this behavior, the fact that the relative error of the NQS improves compared to, and is even able to surpass, that of the standard VMC method for smaller harmonic oscillator frequencies agrees with our physical understanding. Finally, we have also included results from the Hartree Fock method and from a VMC calculation without a Jastrow factor as examples where, as for the NQS, the electron-electron energy divergence has not been explicitly accounted for. We see that the NQS is able achieve an accuracy that is 4-6% better than both these methods.

In table 6.5 we present results for the three dimensional case. Both the NQS and the VMC method perform generally better in this case, with the VMC method achieving better accuracy than the NQS by one order of magnitude in three of four cases. While the NQS continues to underperform compared to standard VMC the results are still quite encouraging with a relatively good agreement.

6.2.2 One-Body Densities

In this section we present the radial one-body densities of two particles in two dimensions with and without interaction for different harmonic oscillator frequen-

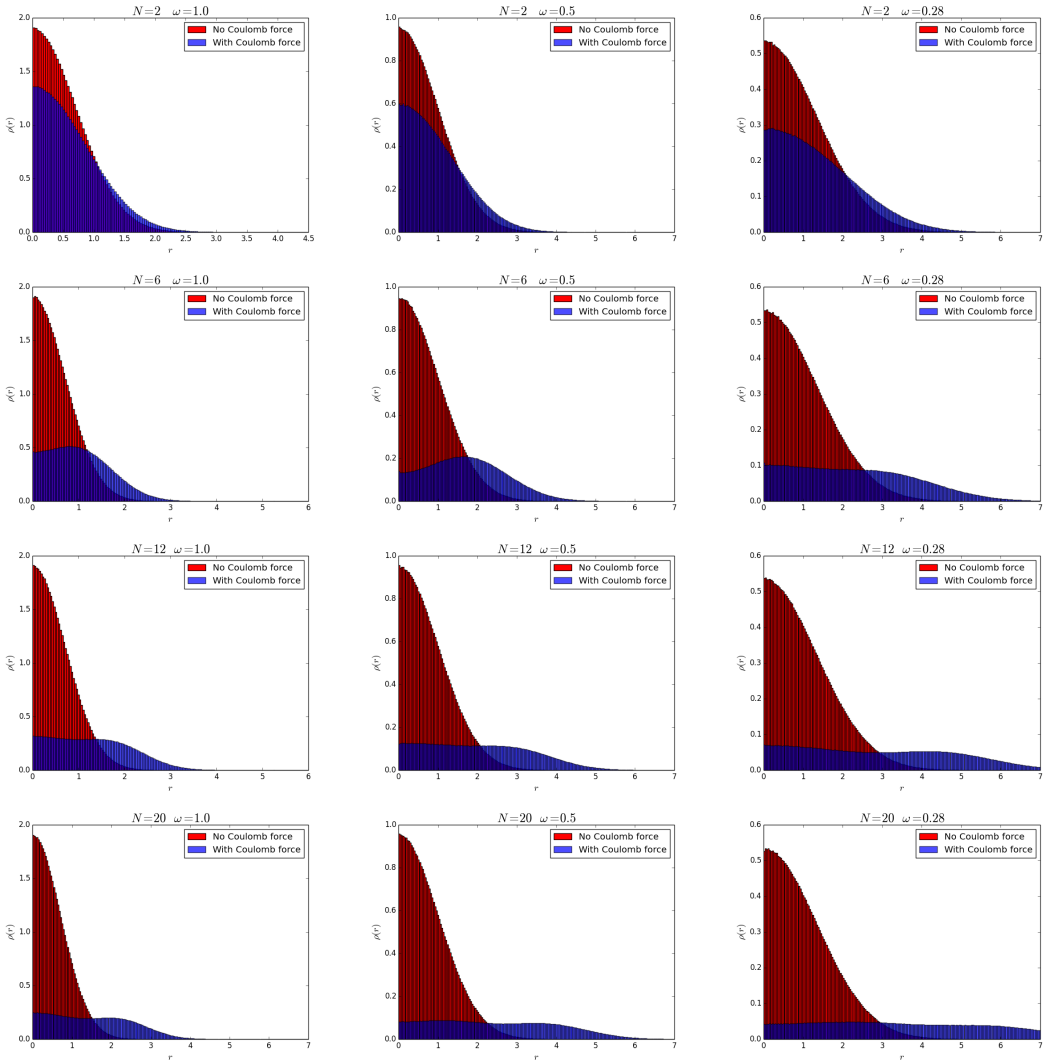

Figure 6.8: One body densities of the 2D NQS for bosons.

Table 6.5: Ground state energies in 3D. E_{HF}, E_{VMC} and E_{VMC-J} are the Hartree Fock energy, and the VMC energy without and with a Jastrow factor respectively, from [35]. E_{DMC} are DMC energies, calculated in [**?**]. E_{DMC} may be assumed to represent exact results.

ω	E_{NQS}	E_{VMC-J}	E_{DMC}	E_0	ϵ_{NQS}	ϵ_{VMC-J}
0.1	0.5156(1)	0.5033689(2)	0.499997(3)	0.5	3.1%	0.7%
0.28	1.2243(2)	1.2037916(2)	1.201725(2)	-	1.9%	0.2%
0.5	2.0262(3)	2.0018226(2)	2.000000(2)	2.0	1.3%	0.1%
1.0	3.7600(5)	3.7355844(3)	3.730123(3)	-	0.8%	0.2%

cies. See figure 6.8. While the two-particle case can be interpreted as fermions, the cases with a higher number of particles here represent bosons. One effect of this is that while the non-interacting case of fermions would see the density spread out as the number of particles were increased due to the Pauli principle (see for example the one-body densities in [35]), we observe here that the densities for non-interacting bosons are more or less unchanged as the number of particles increases, with the shape of a single peak stayng preserved. The only effect that makes the peak spread out is decreasing the frequency. Decreasing the frequency relaxes the confinement of the particles and allows them to move in a bigger space.

For the interacting case, however, we observe that the densities are spread out both for a decreasing frequency and for an increased number of particles. As the number of particles increase the repellent Coulomb force becomes more prominent, causing the particles to take up more of the available space. In several cases we see the shape of two seperate peaks emerging. We conclude that the NQS wavefunction is able to model the effect of both the harmonic oscillator frequency and the Coulomb interaction on the particle positions.

For comparison we have included results for two and six particles with $\omega = 1.0$ in figure 6.9. These have been computed with a slater-Jastrow trial function *combined* with a NQS wavefunction in collaboration with the author in [17]. The two-particle case should exhibit the exact same behavior as the two-particle case in figure 6.8, as these both correspond to electrons. They seem to agree well with a peak close to the origin and a tail wich flattens out around $r = 2.5$. For the case of six particles we would expect them to behave a little differently, as figure 6.8 represents six bosons while figure 6.9 represents fermions acting according to the Pauli exclusion principle. The difference we observe is not huge, however we notice that the bosons has a slightly sharper peak closer to the origin, with the tail of the distribution flattening out around $r = 3$, while the fermions peak

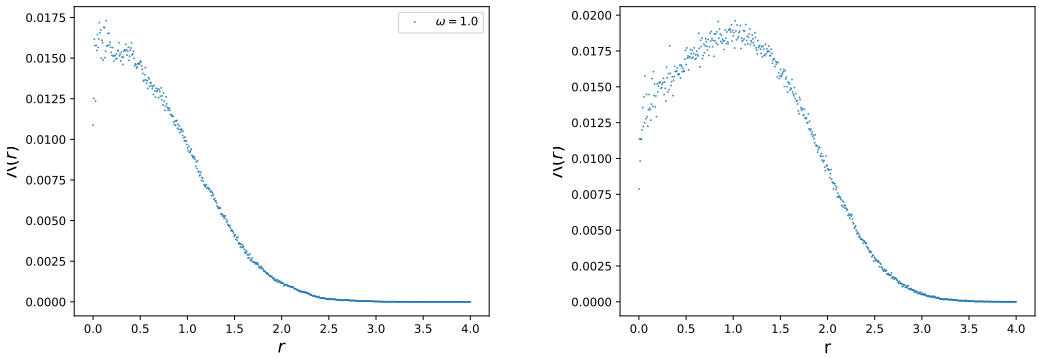

Figure 6.9: One body densities of fermions in 2D with an NQS wavefunction combined with a Slater-Jastrow wavefunction.

slightly after $r = 1.0$ and flattens out around $r = 3.5$.

6.2.3 Comparing to a Jastrow Wavefunction

In this section we attempt to study more thoroughly exactly how interactions are modeled by the NQS wavefunction as compared to a Jastrow wavefunction. To this end, we have in figure 6.10 compared a trained NQS two-electron wavefunction to an optimized two-electron Jastrow wavefunction. We observe how the probability of the position of one electron is affected by the position of the other, by keeping one of them fixed.

Recall that a simple two-electron un-normalized trial wavefunction with a Jastrow function is expressed

$$\Psi(\boldsymbol{r}_1, \boldsymbol{r}_2) = e^{\alpha(r_1^2+r_2^2)} e^{\frac{r_{12}}{2(1+\beta r_{12})}} \tag{6.1}$$

where α and β are the variational parameters. We have here used the optimized values found in [35], that is $\alpha = 0.987$ and $\beta = 0.398$.

The first row of figure 6.10 visualizes the NQS probability of electron 1 when electron 2 is fixed in two different positions. We see that the probability of electron 1 takes the shape of a peak which is translated away from wherever electron 2 is situated. The probability of electron 1 corresponding to the first position of electron 2 is colored red, while the probability of electron 1 corresponding to the second position of electron 2 is colored blue. We clearly see the red and blue peak appearing at slightly different locations, as well as the circular contour lines beneath them having sligthly different centers. The second row visualizes the same scenario. Again we notice that while without the second electron the first electron would peak at the origin, with the second electron placed to the left of the origin the first electron's probability peak is shifted slightly to the right,

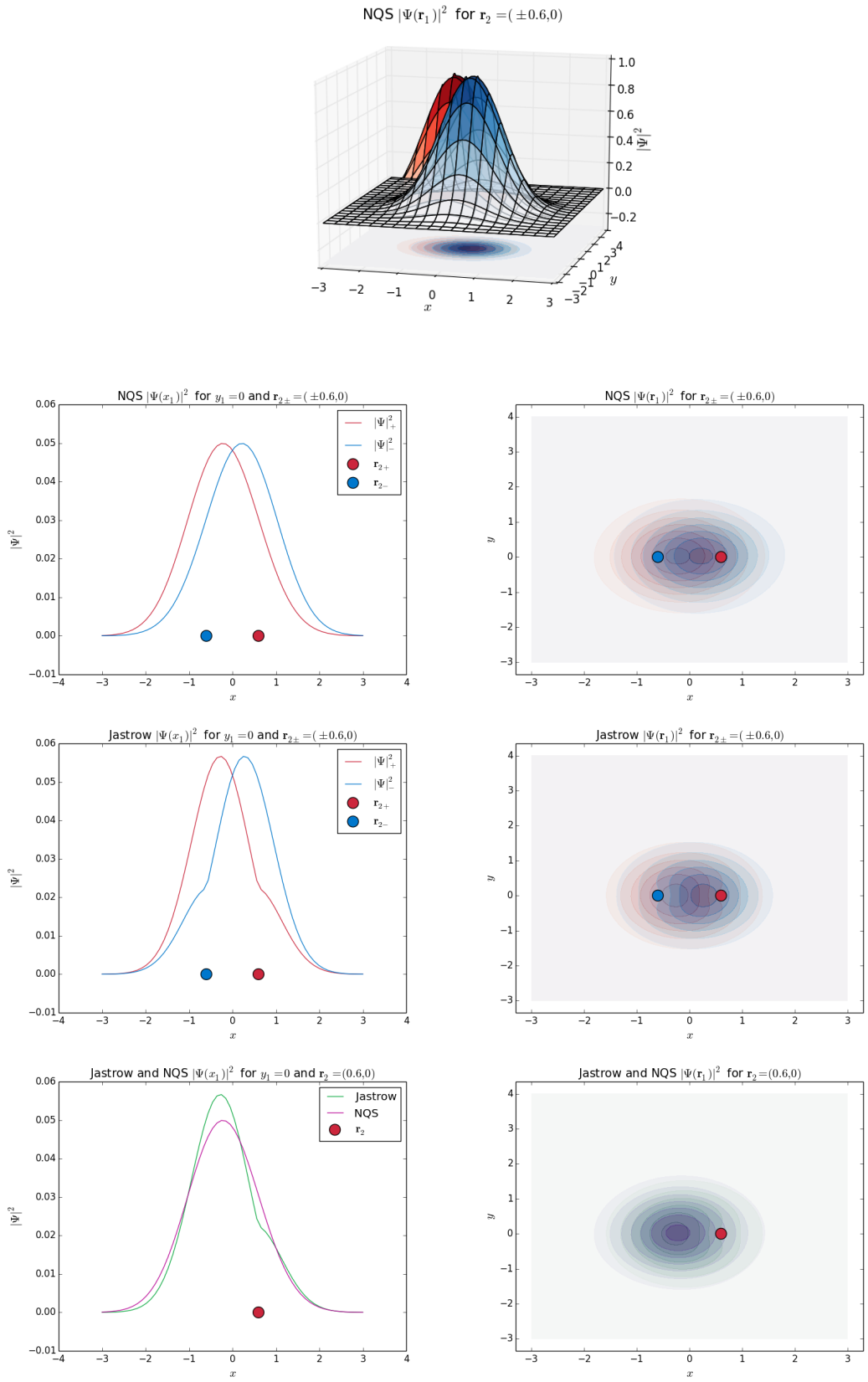

Figure 6.10: Row 1,2: $|\Psi(\boldsymbol{r}_1)|^2$ for an NQS wavefunction, given two different configurations of $\boldsymbol{r}_2$. Row 3: The same for an unperturbed harmonic oscillator wavefunction times a Jastrow correlation factor. Row 4: The NQS and Jastrow wavefunctions for one configuration of $\boldsymbol{r}_2$.

Table 6.6: The table shows how the neural quantum state weights the interactions between the position coordinates of two particles interacting in two dimensions.

Interaction term	Taylor expansion weight
$x_1 y_1$	-0.00076989
$x_1 x_2$	-0.27617352
$x_1 y_2$	0.00042894
$y_1 x_2$	0.00109446
$y_1 y_2$	-0.27980328
$x_2 y_2$	-0.00075311

while with the second electron placed to the right of the origin the first electron's probability peak is shifted slightly to the left (red distribution of particle 1 corresponds to the red position of particle 2, and same with the blue colors).

The third row visualizes the same scenarios with the probability instead given by the Jastrow wave function in equation 6.1. We notice the difference immediately from looking at the graph. While the Jastrow wavefunction makes the same shift of electron 1 depending on electron 2, it additionally has a *dent* in the probability near the second electron, thus providing a more efficient model of the Coulomb effect. We see this effect in the contour lines as well, which are no longer perfectly circular, but squeezed near the point where the other electron is placed, as if trying to avoid it.

Finally, the last row shows both the NQS and Jastrow wavefunctions together. In this row we only show the distribution of electron 1 with respect to *one* position of electron 2 in order to compare the two models. The clearest difference is the *dent* in the Jastrow function, as observed in the previos paragraph. If one looks closely one can observe this difference in the contour lines as well. Another effect is that it seems that the NQS wavefunction is more spread out than the Jastrow function, resulting in a smaller probability at the peak of the distribution and a higher probability at the tails. It could be that the NQS compensates for the lack of a Jastrow dent by instead spreading out and decreasing the overall probability of being near the origin as opposed to at the tails.

6.2.4 Analysis of Interactions Weighted by the Neural Quantum State Wavefunction

The exponent of the NQS wavefunction can be Taylor expanded to a weighted sum of all orders of interactions between the position coordinates. [?] This shows

how the hidden variables and the corresponding weights combine to model higher order interactions. We have extracted the expressions corresponding to second order interactions and calculated their values according to a trained wavefunction. Table 6.6 shows how 2. order interaction terms are weighted in an NQS wavefunction for two interacting particles. Since the Coulomb potential depends on the term $r_{12} = |\boldsymbol{r}_1 - \boldsymbol{r}_2| = \sqrt{x_1^2 - 2x_1x_2 + x_2^2 + y_1^2 - 2y_1y_2 + y_2^2}$ we might expect the terms x_1x_2 and y_1y_2 to be important. We see that their weights are indeed two to three orders of magnitude larger than the others.

Part III

Summary and Outlook

Chapter 7

Conclusion

In this work we have developed from scratch formalism and software for using unsupervised machine learning methods to study interacting many-particle systems. The motivation for employing machine learning methods for the quantum many-body problem was found in the difficulty of describing the non-trivial correlations encoded in the exponential complexity of the many-body wave function. While there are a range of increasingly successfull approximation methods to tackle this challenge, it is still surprisingly hard to accurately describe even seemingly simple systems of few particles. At the core of this challenge is successfully performing dimensionality reduction and feature extraction. That is, to reduce the exponential complexity of the wavefunction down to its most essential features. Over recent years, neural networks have emerged as strikingly able to solve these classes of problems. Therefore, we have found it to be of fundamental and practical interest that we investigate their use for many-particle systems.

One might say that the principal goal of physicists is to construct models of natural phenomena based on the intuition gained from careful empirical observations. Thus, it would not be surprising if the gut-reaction of many phycisists when considering employing machine learning methods would be deep scepticism. After all, as has been discussed, the machine learning approach can be stereotyped as sacrifising inference and understanding at the alter of improved prediction metrics - that is, to employ models that are too complex to allow interpretation or insight as long as the produced predictions are successfull. However, when the physical models become so complicated that constructing them from physical intuition and adjust all parameters can be a several months long process, as can be the case with for example trial wavefunctions modeling correlations in complex nuclear problems, it seems that examining hugely successfull machine learning methods is worth a try, and it is on account of this that we have conducted our study.

This work started with inspiration from Carleo *et al.* [1], who used the binary-binary restricted Boltzmann machine to model the wavefunction of spin lattice systems. Having studied this work we formulated our own goal - to investigate

the ability of a related neural network, the Gaussian-binary restricted Boltzmann machine, to model continous space models. We first developed proof-of-concept code which demonstrated encouraging results. Thus, we subsequently developed that code into a user-friendly, easily extendable object oriented software in C++ with several options with regards to sampling methods, optimization and variants of the model as part of our investigation. The sampling methods studied include the Gibbs method and the Metropolis method with and without importance sampling, and the optimization algorithms include gradient descent with and without momentum in addition to the ADAM optimizer. Finally, we have studied the theory of probabilistic graphical models underlying the restricted Boltzmann machine, to what extent it is applicable in the case of neural quantum states, and to whether there are wavefunctions in the litterature such as ghost wavefunctions using latent variables in a similar manner, in order to provide the best possible starting point for understanding the new wavefunction formalism.

Specifically the continuous-space systems we have studied are bosons and fermions in harmonic oscillator potentials interacting through the Coulomb force. The observables of interest have been the energy, variance and one-body densities. For the non-interacting case we achieved excellent results for the energy to an arbitrary degree of accuracy and a small variance at the order of $\sim 10^{-11} - 10^{-14}$. For the interacting case of two fermions we achieve an accuracy of 10^{-1} and a variance of 10^{-6}, whereas a standard VMC calculation reaches an accuracy of 10^{-2}. This is best explained by the standard Gaussian-binary restricted Boltzmann machine not having the electron-electron cusp usually provided by a Jastrow factor which is needed to cancel diverging energies. However, since the restricted Boltzmann machine offers great flexibility in how one chooses to implement i.e. the transfer functions in the energy function, whereas we have so far only used one of the standard forms, there is a great chance that the model could be altered to better model this behavior. For the same reason we believe it would be interesting to also use the model for other types of interactions.

7.1 Our Contributions

The main contributions made in this work are

- Propose a new formalism for VMC trial wave functions based on neural networks

- Implement a software for exploring this model with different sampling methods and minimization techniques. It is an object oriented code which is easily extendable in several directions - be it adding more sampling methods, optimization methods, new Hamiltonians to study other physical systems, or exploring new variations of the neural quantum state wavefunction.

- We have studied the implemented methods and their parameters and documented the combinations we find optimal.
- We have studied systems of trapped bosons and fermions and benchmarked our results against standard VMC methods and diffusion Monte Carlo methods.

7.2 Future Prospects

There are several directions in which we believe it would be interesting to develop this work.

- Develop the implementation to also include slater determinants so that fermions can be studied in greater numbers (this work is already on-going in collaboration with the author of [17]).
- The field of optimization is vast, and probably we have merely scratched the surface in this work. However, the model is strongly dependent on the minimization being successful, and so would benefit from more work in this direction. It could also be interesting to implement second-order minimization methods and compare to gradient descent. While it is true that these methods are computatioanlly expensive, we are so far working with a rather small number of parameters compared to some other neural networks where second-order methods are out of the question.
- As mentioned, we have implemented the Gaussian-binary restricted Boltzmann machine, which is one of the standard ways to use the restricted Boltzmann machine to model continuous values. However, there is a lot of freedom in what we choose the RBM energy function to look like. This means we can adapt the model to the physical system and interactions at hand more than has been done in this work. It would be interesting to see what possibilities are opened up by doing so. Similarly, it would be interesting to try other types of visible and hidden units than have been done here. For example one could try rectified linear units as a way of implementing continuous rather than hidden latent variables.
- Finally, the restricted Boltzmann machine consists of only one hidden layer. However it is the building block from which one creates the so-called **deep belief networks**. It would be interesting to see if adding several hidden layers would improve the model.

Bibliography

[1] G. Carleo and M. Troyer, "Solving the Quantum Many-Body Problem with Artificial Neural Networks.," *Science*, vol. 355, pp. 602–606, feb 2017.

[2] B. L. Hammond, W. A. Lester Jr, and P. J. Reynolds, *Monte Carlo Methods in Ab Initio Quantum Chemistry*. World Scientific Publishing Co. Pte. Ltd., 1994.

[3] M. Taut, "Two Electrons in an External Oscillator Potential: Particular Analytic Solutions of a Coulomb Correlation Problem," *Physical Review A*, vol. 48, no. 5, p. 3561, 1993.

[4] M. Taut, "Two Electrons in a Homogeneous Magnetic Field: Particular Analytical Solutions," *Journal of Physics A: Mathematical and General*, vol. 27, no. 3, pp. 1045–1055, 1994.

[5] M. Hjort-Jensen, M. P. Lombardo, and U. Van Kolck, *An Advanced Course in Computational Nuclear Physics: Bridging the Scales from Quarks to Neutron Stars*. Springer International Publishing, 2017.

[6] S. Kvaal, *Lecture Notes for FYS-KJM 4480 Quantum Mechanics for Many-Particle Systems*. University of Oslo, 2017.

[7] M. Hjorth-Jensen, *Computational Physics Lecture Notes*. University of Oslo, 2015.

[8] T. Helgaker, P. Jørgensen, and J. Olsen, *Molecular Electronic-Structure Theory*. John Wiley & Sons Ltd, 2000.

[9] M. H. Kalos and P. A. Whitlock, *Monte Carlo Methods*. WILEY-VCH Verlag GmbH & Co., second, re ed., 2008.

[10] J. Toulouse, R. Assaraf, and C. J. Umrigar, "Introduction to the Variational and Diffusion Monte Carlo Methods," *Advances in Quantum Chemistry*, vol. 73, pp. 285–314, jan 2016.

[11] N. G. Van Kampen, *Stochastic Processes in Physics and Chemistry*. North Holland, 3rd ed., 2007.

[12] C. J. Umrigar, "Variational Monte Carlo Basics and Applications to Atoms and Molecules," in *Quantum Monte Carlo Methods in Physics and Chemistry* (C. J. Umrigar and M. P. Nightingale, eds.), NATO Adv Study Inst; Natl Sci Fdn; Ctr Europeen Calcul Atom Molec; Cornell Univ, 1999.

[13] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, "Equation of State Calculations by Fast Computing Machines," *The Journal of Chemical Physics*, vol. 21, no. 6, p. 1087, 1953.

[14] W. K. Hastings, "Monte Carlo Sampling Methods Using Markov Chains and Their Applications," *Biometrika*, vol. 57, no. 1, pp. 97–109, 1970.

[15] M. Jonsson, *Standard error estimation by an automated blocking method.* Msc thesis, University of Oslo, 2018.

[16] M. Ledum, *A Computational Environment for Multiscale Modeling.* Msc thesis, University of Oslo, 2017.

[17] A. Mariadason, *Quantum many-Body Simulations of Double Dot System.* Msc thesis, University of Oslo, 2018.

[18] J. L. Dubois and H. R. Glyde, "Bose-Einstein Condensation in Trapped Bosons: A Variational Monte Carlo Analysis," *Physical Review A*, vol. 63, 2001.

[19] J. K. Nilsen, J. Mur-Petit, M. Guilleumas, M. Hjorth-Jensen, and A. Polls, "Vortices in Atomic Bose-Einstein Condensates in the Large-Gas-Parameter Region," *Physical Review A*, vol. 75, no. 5, 2005.

[20] P. Langley, "The Changing Science of Machine Learning," *Machine Learning*, vol. 82, no. 3, pp. 275–279, 2011.

[21] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A Fast Learning Algorithm for Deep Belief Nets," *Neural Computation*, vol. 18, no. 7, 2006.

[22] G. E. Hinton and R. R. Salakhutdinov, "Reducing the Dimensionality of Data with Neural Networks," *Science*, vol. 313, no. 5786, pp. 504–507, 2006.

[23] G. E. Hinton, "Learning Multiple Layers of Representation," *Trends in Cognitive Sciences*, vol. 11, no. 10, pp. 428–434, 2007.

[24] D. C. Ciresan, U. Meier, L. M. Gambardella, and U. Schmidhuber, "Deep, Big, Simple Neural Nets Excel on Handwritten Digit Recognition," *Neural Computation*, vol. 22, no. 12, 2010.

[25] P. Mehta, C.-H. Wang, A. G. R. Day, and C. Richardson, "A High-Bias, Low-Variance Introduction to Machine Learning for Physicists." 2018.

[26] C. M. Bishop, "Pattern Recognition and Machine Learning," in *Large-Scale Kernel Machines*, Springer-Verlag New York, 1 ed., 2006.

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

[28] G. E. Hinton, "Training Products of Experts by Minimizing Contrastive Divergence," *Neural Computation*, vol. 14, no. 8, pp. 1771–1800, 2002.

[29] D. H. Ackley, G. E. Hinton, and T. J. Sejnowski, "A Learning Algorithm for Boltzmann Machines," *Cognitive Science*, vol. 9, no. 1, pp. 147–169, 1985.

[30] P. Smolensky, "Information Processing in Dynamical Systems: Foundations of Harmony Theory," in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations* (D. E. Rumelhart and J. L. McClelland, eds.), ch. 6, pp. 194–281, MIT Press, 1986.

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

[32] M. Welling, M. Rosen-Zvi, and G. Hinton, "Exponential Family Harmoniums with an Application to Information Retrieval," in *Advances in Neural Information Processing Systems 17* (L. K. Saul, Y. Weiss, and L. Bottou, eds.), pp. 1481–1488, MIT Press, 2005.

[33] S. B. Lippman, J. Lajoie, and B. E. Moo, *C++ Primer*. Objectwrite Inc, 5th ed., 2013.

[34] B. Rubenstein, "Introduction to the Variational Monte Carlo Method in Quantum Chemistry and Physics," in *Variational Methods in Molecular Modeling* (J. Wu, ed.), ch. 10, pp. 258–313, Springer Science+Business Media Singapore 2017, 2017.

[35] H. B. Mørk, *Quantum Monte Carlo Studies of Many-Electron Systems*. Msc thesis, University of Oslo, 2016.

[36] M. Pedersen Lohne, G. Hagen, M. Hjorth-Jensen, S. Kvaal, and F. Pederiva, "Ab Initio Computation of the Energies of Circular Quantum Dots," *Physical Review B*, vol. 84, p. 115302, 2011.