Learning Correlations in Quantum Mechanics with Neural Networks

by

Bendik Samseth

Thesis

for the degree of

Master of Science

Faculty of Mathematics and Natural Sciences
University of Oslo

September, 2019

Abstract

Recent applications of machine learning for quantum mechanics have shown encouraging results in efforts to overcome the exponential scaling complexity of the many-body wave function. We continue this exploration by introducing a neural network as an additional Jastrow factor to the wave function ansatz within the framework of variational Monte Carlo, with the hope of learning correlations beyond what traditional methods have been able to do.

We begin with a review of the relevant parts of quantum mechanics with particular emphasis on Monte Carlo methods. We then introduce central elements of machine learning, focusing on neural networks, followed by the necessary glue to bring it all together. Finally we discuss implementation considerations and challenges.

We test our approach first on two interacting electrons in a harmonic oscillator potential. Where the benchmark estimates $\langle E_0 \rangle = 3.000\,64(4)\,\mathrm{a.u.}$, we outperform this by a lowering of more than an order of magnitude, resulting in $\langle E_0 \rangle = 3.000\,021(4)\,\mathrm{a.u.}$. The energy estimates approach those of diffusion Monte Carlo, which can produce exact solutions to the Schrödinger equation.

Secondly we apply the same technique to the strongly correlated system of liquid ^{4}He. Again we obtain significant lowering of the ground state energy, from the benchmark result of $\langle E_0 \rangle = 6.76(2)\,\mathrm{K}$ to $\langle E_0 \rangle = 6.96(2)\,\mathrm{K}$, although the optimization problem proves far more challenging.

These improvements come at the cost of greatly increased computing time. Nevertheless, we argue that the time complexity can be made to scale as $\mathcal{O}(N^2)$ with the number of particles, while still offering improvements.

Acknowledgements

I would like to thank my supervisor, Morten, for the continued positivity and encouragement. I have felt a great deal of freedom to pursue the topics that I wanted to, which made the work so much more enjoyable and interesting.

I also want to thank everyone at computational physics (now computational science). Between studying in Australia and raising a puppy I have not been the most visible member of the group, but nevertheless I've gotten a lot from you. Helpful discussions, game nights and good times at lunch, I was always happy when I actually made it to the office.

A special thank you to my wife, Mina, for always being there for me. From enduring seemingly endless rants about broken code, to the victory dances after finally getting it to work, having you by my side throughout it all means more than anything.

Also to Nala, who, despite being an absolute productivity killer, has provided me with much needed breaks on our many, many walks.

Finally, a big thank you to the rest of my family for your constant support. I would not be where I am today without you all cheering me on.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Quantum mechanics (QM) is the fundamental theory which describes nature at its smallest scales. Its predictions have been verified by every experiment devised to test it, and it is considered by many to be our most successful physical theory ever developed. Still, QM is far from fully understood, and there is an ocean of questions we still want to get out of it. Continued study of QM is arguably one of the most worthwhile scientific undertakings available to us.

Nevertheless, to a layperson QM might just mean "complicated science" and it is happily ignored in favor of more relatable topics. But if we could turn off QM for a day, its effects on our daily life would become blazingly obvious. You no longer need to go to that doctors appointment for an MRI, because that not a thing anymore. Better call and cancel – too bad your phone is now the size of your apartment. At least you can still watch the world fall apart on your new TV - never mind, LEDs don't work anymore either.

The point of this thought experiment is not to say everyone should become a theoretical physicist, but rather to point out the immense technological progress we have made as side effects of our pursuit to unravel nature's inner workings. So while QM may be the law of the microscopic, its study has real life, large scale effects from which we benefit on a daily basis.

The most challenging problem in QM is the study of strongly correlated many-body systems. Put 100 electrons in a box and press play. What happens next turns out to be extremely complicated to accurately predict. As the first step, we turn to the equation that governs this world, the Schrödinger equation:

$$\hat{H} |\Psi\rangle = i\hbar \frac{\partial}{\partial t} |\Psi\rangle \,. \tag{1.1}$$

We will revisit this equation in Chapter 2, but for now it suffices to know that we want to solve it for $|\Psi\rangle$, the so called *wave function*. This mysterious thing describes everything about our box, and it can tell us *anything* we wish to know about the system.

The reason for our troubles is that Eq. (1.1) turns out to be horrendously hard to solve for all but the simplest possible systems we can imagine. Attempting to solve it for e.g. our box of electrons lead us to a many-body wave function plagued with *exponential* scaling complexity. For instance, the wave function might require 2^N components, which for our 100 electrons means we would need a *nonillion* (10^{30}) parts. That's more parts than there are bits in all the computers of the world, with plenty to spare.

In the words of one of the founding fathers of QM:

> *The underlying physical laws necessary for the mathematical theory of a large part of physics and the whole of chemistry are thus completely known, and the difficulty is only that the exact application of these laws leads to equations much too complicated to be soluble.* — P.A.M. Dirac

So how do we get around this fundamental issue of an equation so complicated we literally cannot solve it through analytical means? Depending on the complexity of the system under investigation and our required level of precision, a variety of approximate methods have been developed. Throughout this thesis, we will be concerned with one of the more versatile methods: *Educated guessing*. Based on our theoretical understanding we can make an educated guess as to what $|\Psi\rangle$ should look like. Even if the guess is not completely accurate, we can now simulate the system as if it was described by this wave function and measure its properties. From fundamental principles, we know nature prefers the state of smallest possible energy. That means that if we tweak our guess slightly and obtain a smaller predicted energy, then our new guess is closer to the underlying truth. If we keep updating our guess, keeping those that lead to smaller energies, we will eventually end up with something that hopefully is a much better approximate solution. The problem is now how to make good guesses, and how tweak them in the right direction. One possible approach is known as variational Monte Carlo (VMC), and will be discussed in depth in Chapter 3.

Enter the world of machine learning (ML). Over the last years we have seen huge advancements in a range of real-world problems, including computer vision and natural language processing, each with a scaling complexity similar to the quantum many-body problem. Physicists have taken notice and are now trying to apply techniques from ML in an attempt to make progress in QM. Machine learning has demonstrated how we can train computer models to see patterns and connections far beyond human comprehension, and be able to condense these down to tangible predictions that we can use.

The goal of this thesis is to build upon the existing machinery for variational Monte Carlo (VMC) by extending theoretical guesses for $|\Psi\rangle$ with a neural network component, thus massively increasing the flexibility and expressiveness of our guesses. If successful, this network will be able to learn the physical correlations present in the system and correct for them more accurately than we could ever do ourselves. In addition, this could open up ways for us to investigate systems for which we have limited theoretical understanding by being less reliant on human-made guesses.

Reading This Thesis

This thesis is written with someone just starting their master's in physics as the intended audience. It is written in a way that I would have liked to read it when first embarking on my own degree. Regardless of how many readers fall into this category, it is our hope that anyone comfortable with first year university level mathematics should be able to understand the majority of the results of this thesis, even if not every detail is equally clear. While the topics at hand might be inherently complex, we have made an attempt at guiding the reader through the background in a way that explains all necessary components as they arise, but also limit ourselves as much as possible to just what is needed.

Structure

Part I starts with a introduction to all the underlying theory necessary to understand the results of this work. Chapter 2 starts with introducing the relevant bits of quantum mechanics, followed immediately by a presentation of variational Monte Carlo and Monte Carlo methods in general (Chapters 3 and 4). This forms the backbone of the thesis. Next we shift gears and discuss machine learning in general (Chapter 5), focusing again on the parts which are most relevant to us. We end Part I by talking about the glue that connects all of this together (Chapter 6).

Part II is all about the technical details of how we implement the algorithms from Part I into efficient and correct code. We discuss design choices (Chapters 7 and 8), and culminate with presenting QFLOW, the library we have developed for all our computing needs (Chapter 9). Lastly we make an effort to convince the reader that the code is correct by verifying the results on some selected, idealized cases (Chapter 10).

Part III finally presents the results of the new method we have developed, testing it on both a few-body system (Chapter 11) and a more complicated many-body system (Chapter 12).

Finally, Part IV offers conclusions and future prospects.

Reproducibility

One of the most frustrating parts of writing this thesis has been attempting to reproduce results from published articles. Far too often vital details are left out, both in theory and implementation, leaving poor souls to guess as to how they achieved the results they present.

We have made a conscious effort to be better in this regard. To that end, all of the code, data, figures etc. that you find in this thesis are openly available at one central repository [1]. That even includes the source code for this very document. The QFLOW library is permissively licensed under the MIT license, allowing anyone to use the code how ever they see fit. Furthermore, *every* table and figure has a reference to where you may find the exact source code which generated it, leaving no doubt as to the details of any results.

Notation and Nomenclature

Another point of possible frustration is unclear use of notation and terminology. While we assume a certain general familiarity with mathematics, we strive to make our notation as clear as possible. All symbols should have an accompanying explanation following their first use. Furthermore, while we attempt to use standard notation where possible, any doubt should be removed by consulting the notation reference in Appendix A.

Finally, you will find a list of all the abbreviations in use at the end.

Part I

Theory

Chapter 2

The Quantum Problem

2.1 Problem Statement

Say you want to investigate the properties of some quantum mechanical system. The first step is then to firmly establish how we should describe this system and the laws that govern its behaviour.[†]

If our system of interest consisted of non-quantum entities (e.g. the trajectory of a baseball as it is thrown through the air towards a batter), we would likely turn to our classical laws, such as Newton's second law of motion

$$\sum_i \mathbf{F}_i(t) = \frac{d\mathbf{p}(t)}{dt}, \tag{2.1}$$

where $\mathbf{F}_i$ are the forces acting on the ball, and $\mathbf{p}$ is its momentum at any given point in time, t. Using the law of motion we can use our knowledge about how the environment affects the object to *deterministically* calculate the resulting behaviour. The really nice thing is that, if we also know the mass of the object, we can derive the value of any other measurable physical quantity of interest. As such, we can say that *solving* a classical system consists of the following steps:

1. Define the environment, i.e. the forces acting on the object(s)
2. Use the second law of motion to obtain momentum $\mathbf{p}(t)$ and position $\mathbf{x}(t)$
3. Compute quantity of interest, $Q(\mathbf{p}, \mathbf{x}; t)$

[†]For the entirety of this thesis, we shall assume that the systems we consider do not show any significant relativistic behaviour, so that no such considerations are necessary.

Moving to the quantum world, much of the same procedure remains the same. For the quantum case, we have a different law of motion. In our non-relativistic view, this is the time-dependent Schrödinger equation (TDSE):

$$\hat{H} |\Psi\rangle = i\hbar \frac{\partial}{\partial t} |\Psi\rangle\,, \tag{2.2}$$

where $i = \sqrt{-1}$ is the imaginary unit and $\hbar = h/2\pi$ is the reduced Planck constant. The thing we want to solve for in this case is the so called wave function $|\Psi\rangle$ (explained momentarily), while the description of the system (analogous to the forces in classical mechanics) goes into $\hat{H}$. We refer to the latter as the Hamiltonian operator, and it should be a complete description of the kinetic and potential energies of the particles involved. As an example, we write the equation for a single particle at position x in an energy potential $V(x;t)$ as follows (where we explicitly use the position basis):

$$\left[-\frac{\hbar^2}{2m}\nabla^2 + V(x;t)\right]\Psi(x;t) = i\hbar\frac{\partial\Psi(x;t)}{\partial t}, \tag{2.3}$$

where the first term constitutes the kinetic energy of the particle (with mass m), and the second term is naturally the potential energy. For the systems that we shall consider in this thesis, the Hamiltonians will all take this form, only varying the functional form of V.

Knowing the wave function Ψ of a system is analogous to knowing position and momentum in the classical view in that we can compute any observable quantity from it (more on this in Section 2.5). As such, obtaining the full expression for the correct wave function is of immense use.

The wave function lacks a clear physical intuition for what exactly it *is*, like we have for position and momentum in classical mechanics. Perhaps the most helpful way to view Ψ is through the fact that its squared absolute value, $|\Psi(x;t)|^2$, is the probability of finding a particle at position x at time t. Thinking of the (squared norm of the) wave function as a substitute for the classical position x can therefore be a helpful aid, as long as we keep the probabilistic nature of it in mind.

Summarizing the steps for *solving* a quantum system, analogous to the classical approach, we have the following plan:

1. Define the environment through choosing a form for the Hamiltonian $\hat{H}$
2. Use the TDSE to obtain the wave function $|\Psi\rangle$
3. Use the wave function to compute quantities Q of interest

2.2 Stationary States

The TDSE (Eq. (2.3)) is a partial differential equation, since it contains partial derivatives of the wave function with respect to both position and time. The standard approach to solving this equation is through *separation of variables*. We assume that we can factorize the full wave function as follows:

$$\Psi(x;t) = \psi(x)\phi(t). \tag{2.4}$$

In addition, we assume that $V(x;t) = V(x)$, i.e. that the potential is time-independent.* With these assumptions, we can divide through with Ψ in Eq. (2.3) and obtain the following:†

$$\left[-\frac{\hbar^2}{2m}\nabla^2\psi + V(x)\psi\right]\psi^{-1} = i\hbar\frac{d\phi}{dt}\phi^{-1}. \tag{2.5}$$

We now make the following subtle observation: Since the lhs., a function of x, is equal to the rhs, a function of t, they must both be equal to a constant. If this was not true, we could vary one of x or t and alter only one side of the equation, leaving it invalid. As both sides have units of energy, let's denote this constant energy as E, and proceed to solve each equation by itself.

The time dependent equation becomes:

$$i\hbar\frac{d\phi}{dt} = E\phi(t), \tag{2.6}$$

which is trivial to solve:

$$\phi(t) = Ae^{-iEt/\hbar}, \tag{2.7}$$

for some constant $A = \phi(0)$ determined by boundary conditions.

The time-independent equation, known as the time-independent Schrödinger equation (TISE), is:

$$-\frac{\hbar^2}{2m}\nabla^2\psi + V(x)\psi = E\psi. \tag{2.8}$$

*There are systems for which this assumption does not hold. We will, however, restrict ourself to consider only Hamiltonians for which this description is valid

†It could be tempting to simply strike Ψ from the lhs. of Eq. (2.3) when dividing by Ψ. Nevertheless, we must remember that the Hamiltonian is an operator (specifically seen through the ∇^2 in this case), and so we must divide only after letting this operate on Ψ.

The solutions to this equation are the *stationary states* of the system. If we are able to find these solutions, then we automatically have also the full time dependent solution through Eq. (2.4).

If we return Eq. (2.8) to the more general form,

$$\hat{H} |\psi\rangle = E |\psi\rangle , \tag{2.9}$$

we can recognize the problem as an eigenvalue problem where we seek the eigenvalues (E) and eigenvectors ($|\psi\rangle$) of the operator $\hat{H}$. In light of this, we prefer to explicitly label the equation to account for the possibility that the equation could have multiple (potentially infinite) solutions, and write this as

$$\hat{H} |\psi_n\rangle = E_n |\psi_n\rangle . \tag{2.10}$$

Each of the $|\psi_n\rangle$ represents one possible stationary state, and could for instance be different levels of energy excitations within an atom. For our purposes, we will only care about the so called *ground state*, i.e. the state $|\psi_n\rangle$ corresponding to the lowest possible E_n. By convention, we assume that the energies are ordered such that $E_i \leqslant E_j$ if $i < j$, and denote the ground state as $|\psi_0\rangle$ and the corresponding ground state energy as E_0.

2.3 Many-Body Systems

Up until now, for simplicity, we've only considered the description of single-particle systems. Changing the number of particles is a change in the system description, and as such it entails modifying the Hamiltonian operator accordingly. Everything presented thus far generalizes well to the case of more than one particle, simply by introducing the appropriate sums. The general form of the many-body Hamiltonian we will consider is now:

$$\hat{H} = -\sum_{i=1}^{N} \frac{\hbar^2}{2m_i} \nabla_i^2 + V(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N) \tag{2.11}$$

$$= -\sum_{i=1}^{N} \frac{\hbar^2}{2m_i} \nabla_i^2 + V(\mathbf{X}), \tag{2.12}$$

were $\mathbf{X} := (\mathbf{x}_1\ \mathbf{x}_2\ \dots\ \mathbf{x}_N)^T \in \mathbb{R}^{N\times D}$ is the matrix of D-dimensional row vectors of coordinates for each particle. For further clarity, $\mathbf{x}_i := \sum_{d=1}^{D} x_{i,d}\mathbf{e}_d$ is a D-dimensional vector described by its coordinates $x_{i,d}$ (with unit vectors $\mathbf{e}_d$), and the corresponding Laplacian operator is

$$\nabla_k^2 := \sum_{d=1}^{D} \frac{\partial^2}{\partial x_{k,d}^2}. \tag{2.13}$$

2.4 Requirements of Wave Functions

We have stated earlier that by solving the Schrödinger equation and obtaining the wave function, we can compute any desirable quantity of interest. In order for the wave function to fulfill this rather impressive encoding of everything about the system, it has to satisfy certain criteria. We now devote some special consideration to make these requirements explicit.

In order to represent a physically observable system, a wave function Ψ must:

1. Be a solution to the Schrödinger equation
2. Be normalizable (in order to represent a probability)
3. Be a continuous function of space
4. Have a continuous first order spacial derivative
5. Obey suitable symmetry requirements

While the first requirement is obvious, points 2-4 boil down to Ψ taking a functional form that is well behaved, satisfying required boundary conditions and being possible to view as a probability density function (PDF). The last point is perhaps less clear, and we devote some further attention to this point in particular.

2.4.1 Symmetry of Wave Functions

Nature has many examples of systems made up of particles of the same species. That is, the particles all have the same mass, spin, electromagnetic charge etc. such that there is no way to distinguish one from the other by measuring their properties. An example could be the electrons of an atom, all of which have the exact same physical properties.

In classical mechanics, we can still distinguish identical particles by other means. Imagine for instance a set of perfectly identical planets in orbit. Even though they have all of the same physical properties, we can still enumerate them and keep track of which is which. This is due to the fact that their position in time and space is deterministically defined by their current state, which allows us to track them.

In quantum mechanics, however, we no longer have this deterministic view. In this world, even if we know where all the individual electrons are at a specific point in time, we cannot say with certainty where they will be at a later time. We blame this on the uncertainty principle, and the result is that systems of identical particles become systems of *indistinguishable* particles in quantum mechanics.

Consider now a system of two indistinguishable particles, labeled x_1 and x_2, where x_i contains all the quantum numbers required to describe particle i (e.g. position coordinates and the z component of spin). The system is then described by a wave function

$$\Psi(x_1, x_2). \tag{2.14}$$

Because the particles are indistinguishable, this labeling of 1 and 2 is arbitrary, and so we should be able to relabel them:

$$\Psi(x_2, x_1). \tag{2.15}$$

These two expressions, which represent exchanging the two particles, *must* describe the same physical system. That is, the probabilities of both states must be equal:

$$|\Psi(x_1, x_2)|^2 = |\Psi(x_2, x_1)|^2 \tag{2.16}$$
$$\iff \Psi(x_1, x_2) = e^{i\alpha}\Psi(x_2, x_1), \tag{2.17}$$

i.e. they can only differ in their complex phase, which doesn't affect any measurable quantity. Repeating the exchange once more yields the original wave function,

$$\Psi(x_1, x_2) = e^{2i\alpha}\Psi(x_1, x_2) \tag{2.18}$$
$$\iff e^{i\alpha} = \pm 1. \tag{2.19}$$

This result states that any wave function, upon the exchange of indistinguishable particles, must be either symmetric (same sign) or anti-symmetric (opposite sign) to that of the original. This is generalizable to any number of particles, and is known as the *Pauli exclusion principle*. The following theorem summarizes the result [2]:

THEOREM 1 (Spin-Statistic Theorem). *The wave function of a system of identical integer spin particles has the same value when the positions of any two particles are swapped. Particles with wave functions symmetric under exchange are called bosons.*

The wave function of a system of identical half-integer spin particles changes sign when two particles are swapped. Particles with wave functions antisymmetric under exchange are called fermions.

2.5 Observables - From Wave Function to Measurement

We have repeatedly claimed that armed with the correct wave function we can compute any measurable quantity of interest. Finally, we consider how exactly we can go about doing so.

Assume we want to compute an observable O. The first step is to determine the corresponding *operator* $\hat{O}$. This is in general done by taking the classical description of the observable and performing a canonical transformation.* Most notably, we have for the following transformations for position and momentum:

$$\mathbf{x} \to \hat{\mathbf{x}}, \tag{2.20}$$
$$\mathbf{p} \to -i\hbar\boldsymbol{\nabla}. \tag{2.21}$$

For example, as is often the case, let's say we want to compute the total energy of the system. For N particles that would classically be:

$$H = \sum_{i=1}^{N} \frac{p_i^2}{2m_i} + V(\mathbf{X}), \tag{2.22}$$

where $\mathbf{p}_i$ denotes the momentum of particle i, all of which are placed in some spacial potential V. It is easily verified that if we perform the above mentioned substitutions we will recover Eq. (2.12) and recognize it as the Hamiltonian operator, $\hat{H}$.

Finally, having both the wave function and the appropriate operator $\hat{O}$ we can proceed. Observables no longer have definite values in general as in classical mechanics. Instead, we associate an expectation value with respect to the PDF described by Ψ:†

$$\langle O \rangle = \langle \hat{O} \rangle = \frac{\langle \Psi | \hat{O} | \Psi \rangle}{\langle \Psi | \Psi \rangle} \tag{2.23}$$
$$= \frac{\int d\mathbf{X}\, \Psi^*(\mathbf{X}) \hat{O}(\mathbf{X}) \Psi(\mathbf{X})}{\int d\mathbf{X}\, |\Psi(\mathbf{X})|^2}, \tag{2.24}$$

where $\int d\mathbf{X}\,(\cdot)$ indicates an integral over all possible configurations of the system (e.g. all possible position and spin values for each particle). Often

*There are also quantities that do not have a classical analog (e.g. spin) for which we can still find operator forms.

†Note that quantities can still have definite values in certain states. This is then evident by the expectation values having zero associated variance.

we have required the wave function to be normalized in such a way that the denominator is equal to unity, and it can then be omitted.

For many-body systems it should be apparent that this integral quickly becomes intractable to compute analytically. In practice we employ a numerical strategy to evaluate these integrals, where the technique we use depends on the dimensionality of the integral and the required level of accuracy. In our case, due to the large number of degrees of freedom in the systems we shall investigate, we will use Monte Carlo integration (MCI). This will be discussed in more detail in Chapter 4.

2.6 Example Systems

So far we have not presented any particular systems. In this thesis we focus our attention on two particular systems for illustrative purposes. We chose these systems for their simplicity and/or the amount of preexisting results available in the literature. We do this in order to benchmark our results against known exact solutions, or when these do not exist, against verified approximate results available in the literature.

2.6.1 Quantum Dots

We consider a system of electrically charged particles (e.g. electrons) confined in a pure isotropic harmonic oscillator potential, with an idealized total Hamiltonian given by:

$$\begin{aligned}\hat{H} &= \sum_{i=1}^{N}\left(-\frac{1}{2}\nabla_i^2 + V_{\text{ext}}(\mathbf{r}_i)\right) + \sum_{i<j} V_{\text{int}}(\mathbf{r}_i, \mathbf{r}_j) \\ &= \sum_{i=1}^{N}\left(-\frac{1}{2}\nabla_i^2 + \frac{1}{2}\omega^2 r_i^2\right) + \sum_{i<j}\frac{1}{r_{ij}},\end{aligned} \tag{2.25}$$

where we use natural units ($\hbar = c = m_e = 1$) with energies in atomic units (a.u.), N denotes the number of particles in the system, and ω is the oscillator frequency of the trap. Further, $\mathbf{r}_i$ denotes the position vector of particle i, with $r_i := \|\mathbf{r}\|$ and $r_{ij} := \left\|\mathbf{r}_i - \mathbf{r}_j\right\|$ defined for notational brevity.

This system describes particles trapped in a parabolic potential well that pulls them towards the bottom at all times, while simultaneously feeling the repulsive Coulomb forces from the other particles. This hinders all particles from settling together at the bottom. Even for this somewhat idealized system, the interplay between these two opposing forces gives rise to a surprisingly complex problem, which will prove remarkably hard to solve analytically even for two particles, and utterly impossible for higher N.

With the natural units in place, the only involved quantity without a proper unit is length, i.e. what unit does the $\mathbf{r}_i$ have. A convenient choice is to consider the mean square vibrational amplitude, $\langle r^2 \rangle$, for a single particle at $T = 0\,\mathrm{K}$ placed in the oscillator trap. Computing the expectation value we get $\langle r^2 \rangle = \hbar/2m\omega$, and we define the unit of length as the characteristic length of the trap, $a_{ho} = \left(2\langle r^2\rangle\right)^{1/2} = (\hbar/m\omega)^{1/2}$ [3].

In our case, we limit ourselves to $N = 2$ interacting electrons in two dimensions in a trap with a frequency such that $\hbar\omega = 1$.* We do this because for this case we have exact, analytical solutions for the ground state energy. With the interaction term included, the ground state energy is $E_0 = 3$ a.u. [4]. This limitation is purely one of convenience, as having exact benchmarks makes for better verification of results. Furthermore, limiting the size of the problem makes the required computation time manageable, which is good when experimenting with different techniques.

Simple Non-Interacting Case

If we omit the interacting terms in Eq. (2.25) we have the standard harmonic oscillator Hamiltonian:

$$\hat{H}_0 = \sum_{i=1}^{N} \left(-\frac{1}{2}\nabla_i^2 + \frac{1}{2}\omega^2 r_i^2 \right). \tag{2.26}$$

This Hamiltonian lends itself to analytical solutions, and the stationary single particle states are (in 2D) [5]:

$$\phi_{n_x,n_y}(x,y) = A H_{n_x}(\sqrt{\omega}x) H_{n_y}(\sqrt{\omega}y) e^{-\frac{\omega}{2}(x^2+y^2)}, \tag{2.27}$$

for quantum numbers $n_x, n_y = 0, 1, \dots$, and the Hermite polynomials H_n (not to be confused with the Hamiltonians, and never to be mentioned again). The ground state, $n_x = n_y = 0$ is simply

$$\phi_{00}(x,y) = \sqrt{\frac{\omega}{\pi}} e^{-\frac{\omega}{2}(x^2+y^2)}. \tag{2.28}$$

Using this wavefunction we can calculate the ground state energy for one particle,

$$\epsilon_{00} = \frac{\langle \phi_{00} | \hat{H}_0 | \phi_{00} \rangle}{\langle \phi_{00} | \phi_{00} \rangle} = \omega = 1 \text{ a.u.} \tag{2.29}$$

*Note that, due to the natural units, this implies that $\omega = 1$, which further means that $a_{ho} = 1$. It should be apparent why we use these definitions, as it simplifies both units and expressions.

The ground state wavefunction for the (unperturbed) two-electron case is simply the product of the one-electron wave functions,

$$\begin{aligned}\Phi(\mathbf{r}_1, \mathbf{r}_2) &= \phi_{00}(\mathbf{r}_1)\phi_{00}(\mathbf{r}_2) \\ &= \frac{\omega}{\pi} e^{-\frac{\omega}{2}(r_1^2+r_2^2)}.\end{aligned} \tag{2.30}$$

We can once again evaluate the ground state energy analytically, which yields

$$E_0 = \frac{\langle\Phi|\hat{H}_0|\Phi\rangle}{\langle\Phi|\Phi\rangle} = 2\omega = 2 \text{ a.u.} \tag{2.31}$$

This result is not surprising, as adding one more particle, without any interactions, should simply double the energy. Another way to look at it is that the simple harmonic oscillator solution gives $\omega/2$ per degree of freedom, so adding another two yields and extra ω.

When the two particles are electrons, we may say something about their total spin. As electrons are fermions, their total wavefunction must be anti-symmetric upon interchanging the labels 1 and 2. Equation (2.30) is obviously symmetric, and so the spin-wavefunction must necessarily be anti-symmetric. For the combination of two spin-1/2 particles, there is only one candidate, namely the spin-0 singlet:

$$\chi_0 = \frac{1}{\sqrt{2}}(|\uparrow\downarrow\rangle - |\downarrow\uparrow\rangle). \tag{2.32}$$

A similar argument can be made for particles with different spins.

Considerations from the Virial Theorem

The virial theorem gives a general relation for the time-averaged kinetic energy $\langle K\rangle$ and the corresponding potential energy $\langle V_{\text{pot}}\rangle$ of a stable system of N particles. In general the theorem states:

$$\langle K\rangle = -\frac{1}{2}\sum_{k=1}^{N} \langle \mathbf{F}_k \cdot \mathbf{r}_k\rangle, \tag{2.33}$$

where $\mathbf{F}_k$ denotes the combined forces acting on particle k, located at position $\mathbf{r}_k$. For a radial potential on the form $V(r) = ar^n$, such that the potential between any two particles in the system depends on some power of the inter-particle distance, the theorem takes the following form:

$$\langle K\rangle = \frac{n}{2}\langle V_{TOT}\rangle \tag{2.34}$$

where V_{TOT} denotes the sum of the potential energy $V(r)$ over all pairs of particles.

Although the harmonic oscillator potential does not depend on the *inter-particle* distance, but rather on the positions of each particle, it works out to the same relation in our case. Computing the full relation for our Hamiltonian for two electrons in two dimensions, it even works out so that we can use the same relation on the harmonic oscillator potential and the Coulomb potential separately, and add the result. This means that the virial theorem predicts the following [6]:

$$\langle K \rangle = \langle V_{\text{ext}} \rangle - \frac{1}{2} \langle V_{\text{int}} \rangle . \tag{2.35}$$

Note that this implies that we should consider the *total* kinetic energy, and the *total* external and internal potential energies, as opposed to per particle.

2.6.2 Liquid ^{4}He

Consider now an infinite collection of helium atoms (^{4}He) packed with a given density, ρ. As infinities are hard to work with, we model this by considering a cubic simulation box with side lengths L and periodic boundary conditions. The infinite collection is then composed of stacking copies of such simulation boxes together. Fig. 2.1 shows an illustration of the idea.

The Hamiltonian for this system is

$$\hat{H} = -\sum_{i=1}^{N} \frac{\hbar^2}{2m} \nabla_i^2 + \sum_{i<j} V(r_{ij}) \tag{2.36}$$

i.e. the kinetic energy of all atoms, plus an interaction potential dependent on the distance between all pairs of atoms. The mass m is the mass of one ^{4}He atom. The form of V is not known analytically, but is experimentally probed to great accuracy. Theorists have since fitted specific functional forms to the experimental data, and we will do our calculations using one of these potentials. The most commonly used is the simple Lennard-Jones (LJ) potential [7]:

$$V(r) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \tag{2.37}$$

with $\epsilon/\kappa = 10.22\,\text{K}^*$ and $\sigma = 2.556\,\text{Å}$. This models the competing forces of the atoms' mutual repulsion and attraction. The positive term describes the short

$^*\kappa$ is the Boltzmann constant.

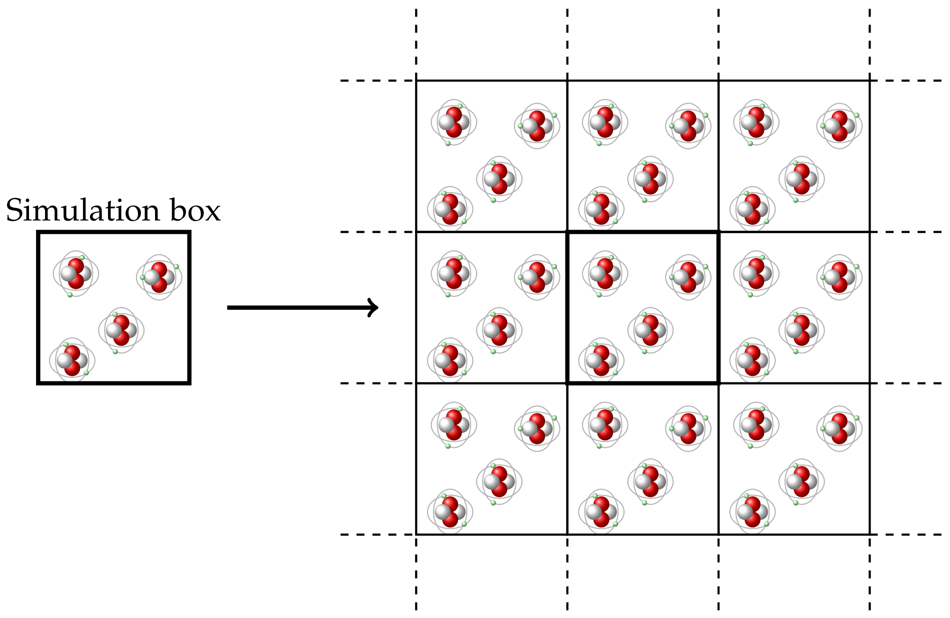


Figure 2.1: Illustration of ^{4}He organized into a grid of identical simulation boxes. The actual boxes are three-dimensional.
Source [1, `writing/illustrations/PBC-illustration.tex`]

range Pauli repulsion due to overlapping electron orbitals, and the negative term describes the long range attraction due to phenomena such as van der Waals forces.

We can also use the slightly more accurate (and complicated) potential named HFDHE2 [8]:

$$V(r) = \epsilon \left\{ \begin{aligned} &A \exp\left(-\alpha \frac{r}{r_m}\right) \\ &- F(r)\left[C_6\left(\frac{r_m}{r}\right)^6 + C_8\left(\frac{r_m}{r}\right)^8 + C_{10}\left(\frac{r_m}{r}\right)^{10}\right] \end{aligned} \right\} \tag{2.38}$$

with

$$F(r) = \begin{cases} \exp\left(-[D\frac{r_m}{r} - 1]^2\right) & \text{for } \frac{r}{r_m} \leqslant D \\ 1 & \text{otherwise} \end{cases} \tag{2.39}$$

with the following parameters:

$$\begin{aligned} A &= 0.544\,850\,4 \cdot 10^6 & \epsilon/\kappa &= 10.8\,\text{K} \\ \alpha &= 13.353\,384 & C_6 &= 1.377\,324\,12 \\ D &= 1.241\,314 & C_8 &= 0.425\,378\,5 \\ r_m &= 2.9673\,\text{Å} & C_{10} &= 0.178\,100 \end{aligned} \tag{2.40}$$

Both potentials grow rapidly for small r, and tend to 0 for large r. The interesting sections of both potentials are shown in Fig. 2.2. The potentials are very similar, with the main difference being the depth of the well and how sharply the potential dies off.

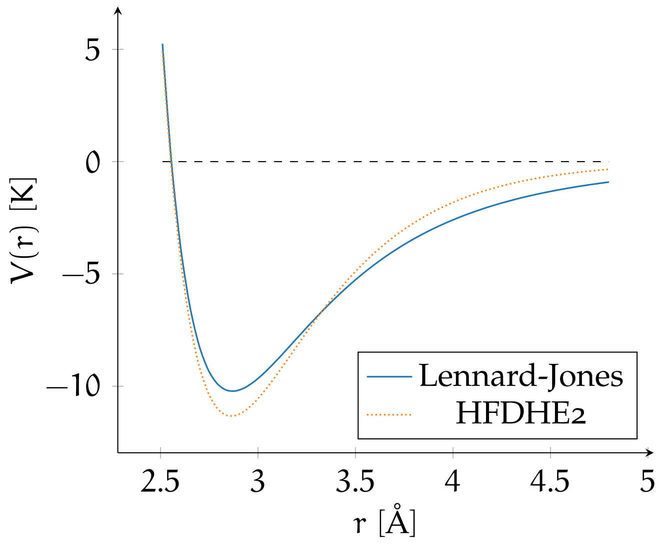


Figure 2.2: Lennard-Jones and HFDHE2 potentials used to model the potential between pairs of ^{4}He atoms. Both potentials grow rapidly towards infinity when $r \to 0$ and approach zero when $r \to \infty$.
Source [1, `writing/illustrations/helium-potentials.tex`]

A Note About Units

In the literature it is common to express the energies in Kelvin per particle, and lengths in angstrom [7, 8, 9]. In order to convert energies to temperatures we divide by the Boltzmann constant, κ, because it has the unit of Joules per Kelvin. The Hamiltonian becomes:

$$\hat{H} = -\sum_{i=1}^{N} \frac{\hbar^2}{2m\kappa} \nabla_i^2 + \sum_{i<j} \frac{1}{\kappa} V(r_{ij}). \tag{2.41}$$

If we use SI units for the constants involved we get:

$$\frac{\hbar^2}{2m\kappa} = 6.059\,651\,974 \cdot 10^{-20}\,\text{m}^2\text{K} = 6.059\,651\,974\,\text{Å}^2\text{K}, \tag{2.42}$$

which turns out to be a reasonably sized number when we use angstrom as units for lengths.* We will use these values in the implementation, and

*Note that ∇^2 has units of length^{-2}, so the units work out to Kelvin.

simply refer to energies in Kelvin when we study this system. However, if the reader ever wants to convert the units, for comparison with other works perhaps, simply multiplying with the value of κ in the unit system of choice should yield the corresponding energy.

Minimum Image

Because we assume a periodic structure we must take this into account when calculating distances. Consider two particles, A and B, located at opposite corners of the simulation box. What is the distance between them? The intuitive answer is $\|\mathbf{r}_A - \mathbf{r}_B\| = \sqrt{3}\,L$, i.e. the length of the diagonal of the cubic box. Nevertheless, the answer we should use is zero. The reason is that there is a periodic copy of the box stacked such that A and the periodic B copy are located at the same corner. This way of calculating distances is called minimum image, and says that we should use the shortest possible distance. In general, if two particles have a distance of $\Delta x = L/2 + \delta_x$ ($\delta_x \geqslant 0$, each spatial coordinate handled individually), the minimum image distance is $\Delta x_{min} = \Delta x - L = L/2 - \delta_x$.

In our implementation, whenever we need a distance between two particles, we use the following prescription (example in Python):

```python
import numpy as np

# Example coordinates.
L = 5.0
p1, p2 = np.array([0, 0, 0]), np.array([2, 3, 4])

diff = p2 - p1  # [2, 3, 4]
diff_minimum = diff - L * np.round(diff / L)  # [2, -2, -1]
```

The last variable, `diff_minimum` represents the minimum image distance vector and this is the one used for any further calculations.

Correcting for Periodicity in Potentials

The potential $V(r_{ij})$ depend on the inter-particle distances. However, there is an infinite amount of particle pairs if we consider the system as a whole. We use periodic boundary conditions, and shall only consider pairs where both particles are in the simulation box (but still respecting the minimum image convention). This limitation excludes any interactions that act on length scales larger than $L/2$, and this can have a significant impact on the total system.

In an attempt to limit this effect, we modify the potentials slightly. First we explicitly truncate the potential to be zero for large distances, and shift it slightly so that the function remains continuous. That is, considering $V(r)$ from Eq. (2.37), we change it as follows:

$$V_{trunc}(r) = \begin{cases} V(r) - V(L/2) & \text{for} \quad r \leqslant L/2 \\ 0 & \text{otherwise} \end{cases}. \tag{2.43}$$

Note that in order for this truncation to be sensible, we must use a sufficiently large box so that $V(L/2)$ is sufficiently close to zero. What exactly *sufficiently* means is left rather vague, but we mention it as a potential source of error.

The truncation obviously leads to slightly less precise results, but this can partially be corrected for with so-called *tail corrections*. The approach is to model the potential contribution of all particles further away than $L/2$ in a mean-field manner. The result is simply adding a constant term, and acts only to shift the total potential in a given direction. As the purpose of this thesis is not to obtain the most realistic results possible, and rather a *relative* comparison of methods, we will not spend more time on specific ways of implementing such corrections.

Chapter 3

Variational Monte Carlo

Variational Monte Carlo (VMC) is a method for obtaining the ground state wave function of a quantum mechanical system, and it constitutes the framework for all the calculations we perform in this thesis. As this method is of great significance to us, this chapter is devoted to all relevant definitions, derivations and technical details required for a successful study of the systems of interest.

3.1 The Variational Principle

The fundamental principle that enables VMC is the observation that the wave function that describes the ground state of a system, is the wave function which produces the lowest expectation value of the energy. Read this statement a couple of times and it almost seems like a circular definition, because we define the ground state as the state with the lowest energy. Still, this simple observation is what allows us a way to go forward.

Let's formalize the above statement.

Theorem 2 (Variational Principle). *Suppose a time-independent Hamiltonian, $\hat{H}$, an associated set of eigenvalues, $E_0 \leqslant E_1 \leqslant \ldots$ and a set of orthonormal eigenvectors, $\{|\phi_n\rangle\}$ spanning the associated Hilbert space, such that the time-independent Schrödinger equation is satisfied,*

$$\hat{H}\,|\phi_n\rangle = E_n\,|\phi_n\rangle\,. \tag{3.1}$$

For any arbitrary $|\psi\rangle$ in the Hilbert space, the expectation value of $\hat{H}$ for this wave function must satisfy

$$\langle\hat{H}\rangle := \langle\psi|\hat{H}|\psi\rangle \geqslant E_0, \tag{3.2}$$

where E_0 is the exact ground state energy, with equality if and only if $|\psi\rangle = |\phi\rangle_0$.

Proof. Because $\{|\phi_n\rangle\}$ constitutes a complete orthonormal basis of the Hilbert space, we can expand $|\psi\rangle$ as

$$|\psi\rangle = \sum_n c_n |\phi_n\rangle \quad \text{with} \quad \sum_n |c_n|^2 = 1. \tag{3.3}$$

We then have

$$\begin{aligned}
\langle\psi|\hat{H}|\psi\rangle &= \left(\sum_m c_m^* \langle\phi_m|\right)\hat{H}\left(\sum_n c_n |\phi_n\rangle\right) && (3.4)\\
&= \sum_n \sum_m c_m^* c_n \langle\phi_m|\hat{H}|\phi_n\rangle && (3.5)\\
&= \sum_n \sum_m c_m^* c_n E_n \langle\phi_m|\phi_n\rangle && (3.6)\\
&= \sum_n \sum_m c_m^* c_n E_n \delta_{mn} && (3.7)\\
&= \sum_n |c_n|^2 E_n && (3.8)\\
&\geqslant \sum_n |c_n|^2 E_0 = E_0. && (3.9)
\end{aligned}$$

It is clear that the equality only occurs when $c_n = \delta_{0n}$, which implies $|\psi\rangle = |\phi_0\rangle$. □

If we require the test wave function $|\psi\rangle$ to be orthogonal to $|\phi_0\rangle$, then the above derivation results in a similar condition for determining the first excited state. By applying the same reasoning iteratively, the same can be said about any higher energy state. So in principle, we can use the variational principle to first find the ground state, then find an orthogonal state that gives the first excited energy, then find a state orthogonal to both that gives the second excited energy etc. We will not pursue any excited states in this thesis, but it is useful to know that we could in principle extend our results if need be.

3.2 The Variational Monte Carlo Algorithm

Rooted in the variational principle, the general approach to finding a good estimate of the ground state goes as follows:

1. Define the Hamiltonian of interest, $\hat{H}$
2. Propose a trial wave function $|\boldsymbol{\alpha}\rangle$ dependent on some free parameters $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \dots, \alpha_m)$

3. Return the best possible wave function $|\boldsymbol{\alpha}\rangle$ such that $\langle\boldsymbol{\alpha}|\hat{H}|\boldsymbol{\alpha}\rangle$ is minimized w.r.t the parameters $\boldsymbol{\alpha}$.

The first step is trivial, as we shall assume that the Hamiltonians we want to examine are given in advance.*

The second and third points are where we will spend our efforts, and so we devote extra attention to these two points in particular.

3.3 The Trial Wave Function

Critical to the outcome of a VMC calculation is the quality of the proposed trail wave function. In order to have stable and sensible results we should also take care that the wave function obeys the requirements stated in Section 2.4. Beyond these basic assertions, it is imperative that the functional form of the proposed wave function is capable of expressing the true underlying ground state, or at least a good approximation of it. As an example, take the simple case of the idealized one-dimensional harmonic oscillator presented in Section 2.6.1, with the ground state expressed as

$$\phi_0(x) \propto \exp\left(-\frac{1}{2}x^2\right). \tag{3.10}$$

If we proposed a trial wave function on a form

$$\psi(x;\alpha) \propto \exp\left(-\alpha x^2\right), \tag{3.11}$$

we could expect VMC to produce the exact ground state after optimizing α. The above is a good trial wave function because its functional form contains the exact state.

Suppose now we proposed something different, maybe because of a lack of theoretical understanding. For the sake of example, say we chose this visually similar function:†

$$\psi(x;\alpha) \propto \frac{1}{1+\alpha x^2}. \tag{3.12}$$

No matter how we tune the α parameter here, we will never achieve the lowest ground state energy, simply because the ground state is not possible to express within this functional form. We can now only hope to get approximations, acting as upper bounds on the ground state energy.

*Clearly, deriving/defining Hamiltonians that accurately describe a given system is not a trivial task in general, and might require great levels of theoretical work and/or experimental data analysis. But from our perspective, this work has already been done.

†Visually similar in the sense that plotting them both yield similar bell shaped curves, not in terms of their algebra.

3.3.1 Slater-Jastrow: Standard Approach

In order to come up with good trail wave functions we have a standard approach which often serves us well. Typically we are considering a system of many interacting particles in some external potential (e.g. the quantum dots system). In such cases the typical approach is to build a trial wave function as a product of a single-particle part, S, and a correlation part, J. The two are referred to as the *Slater* factor and the *Jastrow* factor, respectively:

$$\psi = S \cdot J. \tag{3.13}$$

Slater Factor

The idea behind the decomposition above is that we put all the physical insight we have into these two components individually. For the single particle part we include what we can deduce about the system if we idealize it in some way. Typically that involves looking at the states when there is only one particle, or considering some simplified interaction between all of them (e.g. Hartree-Fock approximation). The basis functions we obtain from such an analysis are then placed in a determinant/permanent structure so as to fix the symmetry requirements of fermions/bosons, respectively. This factor is then the Slater factor [10, 11].

For clarity, we present a couple of different Slater forms. We consider the two-dimensional quantum dots system from Section 2.6.1 for N particles. We denote the position of particle i by $\mathbf{x}_i = (x_i, y_i)$. Let $\mathbf{\Phi}$ be a matrix defined by $\Phi_{i,j} := \phi_i(\mathbf{x}_j)$, where the ϕ_i is part of the set of orbitals $\{\phi_{0,0}, \phi_{1,0}, \phi_{0,1}, \phi_{1,1}, \dots\}$, as defined in Eq. (2.27). Lastly, let $\mathbf{\Phi}^{(M)}$ denote the resulting matrix with M rows, one for each of the M lowest energy orbitals. Finally, if the particles are fermions we define the Slater factor as:[*]

$$S(\mathbf{X})_{\text{fermions}} := \text{determinant}\left(\mathbf{\Phi}^{(M)}\right). \tag{3.14}$$

Due to the properties of the determinant, this has the anti-symmetry required for fermions.

In the case of bosons, we could just change the above to

$$S(\mathbf{X})_{\text{bosons}} := \text{permanent}\left(\mathbf{\Phi}^{(M)}\right). \tag{3.15}$$

[*]Spin considerations not mentioned here, as they will not be important for the particular systems we will investigate. In general though, the structure of the Slater factor needs to account for spin in a well defined way.

However, there are far less computationally expensive ways to get a *symmetric* result, as required for bosons. A simple product of single particle ground states is sufficient in this case.

$$S(\mathbf{X})_{\text{bosons}} := \prod_{i}^{N} \phi_{0,0}(x_i). \tag{3.16}$$

The general structure is constant across different systems, changing only the basis functions we use to express the matrix elements.

In some cases, however, we will struggle to find basis functions that describe a meaningful portion of the system's behaviour in this way. In such challenging cases, the Slater factor could be reduced to only being responsible for supplying the correct symmetry. In the case of bosons, that would mean the complete removal of the Slater factor from the trial wave function. We will see this when looking further into the liquid helium system.

Jastrow Factor

Anything not covered by this analysis is meant to be accounted for in the correlation term, J. This is typically a function on the following generalized form, commonly called a Jastrow factor [12]:

$$J(\mathbf{X}) = \exp\left(\sum_{i<j} U(r_{ij})\right), \tag{3.17}$$

where $U(r_{ij})$ is some function dependent on the inter-particle distance only. Note that by convention the Slater factor has the correct symmetry, so the correlation factor should always be symmetrical in order to maintain the same total symmetry.

Any variational parameters $\boldsymbol{\alpha}$ are typically introduced as part of the Jastrow factor. The Slater factor represents everything we are sure should be included from a theoretical approach, while the Jastrow term should ideally account for all our ignorance. It is therefore typically expressed with a few free parameters.

Again, for the sake of clarity, we state an example of a Jastrow factor. One of the most commonly used is the Pade-Jastrow factor [13]:

$$J(\mathbf{X}) = \prod_{i<j} \exp\left(\frac{\alpha r_{ij}}{1 + \beta r_{ij}}\right). \tag{3.18}$$

Here β is the only variational parameter, and α is fixed depending on the dimensionality and spin of the particles. More complicated versions exist, containing higher order polynomials of r_{ij} in the exponential. This particular form will prove to be a very good choice for the quantum dots system.

3.4 Optimization

At this point, we have a Hamiltonian $\hat{H}$ defining our system of interest, and we have proposed a trial wave function $|\boldsymbol{\alpha}\rangle$ based on the theoretical intuition we have. The wave function has by design some free parameters $\boldsymbol{\alpha}$ which we now want to pin down to their optimal values.

From the variational principle (Theorem 2), we know that the best set of parameters are those that minimize the expected energy, $\langle\boldsymbol{\alpha}|\hat{H}|\boldsymbol{\alpha}\rangle$. As such, a natural place to start would be to determine an efficient way to evaluate the energy.

3.4.1 Local Energy

Denote the trial wave function as $\psi_{\boldsymbol{\alpha}}(\mathbf{X})$, where $\boldsymbol{\alpha}$ are the variational parameters and $\mathbf{X} = (x_1\ x_2\ \dots\ x_n)^T$ is the matrix of all the degrees of freedom for each particle (e.g. position coordinates). The expectation value for the energy of the system is:

$$\left\langle\hat{H}\right\rangle = \frac{\langle\boldsymbol{\alpha}|\hat{H}|\boldsymbol{\alpha}\rangle}{\langle\boldsymbol{\alpha}|\boldsymbol{\alpha}\rangle} \tag{3.19}$$

$$= \frac{\int d\mathbf{X}\,\psi_{\boldsymbol{\alpha}}^*\hat{H}\psi_{\boldsymbol{\alpha}}}{\int d\mathbf{X}\,|\psi_{\boldsymbol{\alpha}}|^2}. \tag{3.20}$$

We now restructure this integral a bit. Remembering the interpretation of the wave function as a probability density function (PDF), we can write the *probability density* for a given system configuration $\mathbf{X}$ as:

$$P_{\psi_{\boldsymbol{\alpha}}}(\mathbf{X}) := \frac{|\psi_{\boldsymbol{\alpha}}|^2}{\int d\mathbf{X}\,|\psi_{\boldsymbol{\alpha}}|^2}, \tag{3.21}$$

where we account for the possibility of $\psi_{\boldsymbol{\alpha}}$ not being normalized to unity. Further, we define a quantity we will call the *local energy*,

$$E_L := \frac{1}{\psi_{\boldsymbol{\alpha}}}\hat{H}\psi_{\boldsymbol{\alpha}}. \tag{3.22}$$

The reason for these definitions is that now we can write the energy expectation as follows:

$$\left\langle\hat{H}\right\rangle = \int d\mathbf{X}\,P_{\psi_{\boldsymbol{\alpha}}}(\mathbf{X})E_L(\mathbf{X}). \tag{3.23}$$

Why is this better? Because this is simply the weighted average of the local energy, something which has a straightforward discrete formulation:

$$\langle \hat{H} \rangle = \lim_{N\to\infty} \frac{1}{N} \sum_{i=1}^{N} E_L(\mathbf{X}_i) \quad \text{where} \quad \mathbf{X}_i \overset{d}{=} P_{\psi_\alpha}. \tag{3.24}$$

So, if we can sample $\mathbf{X}$ from the distribution described by our trial wave function, we simply need to evaluate the local energy for a sufficiently large number of configurations to obtain a good estimate of the expectation value of the Hamiltonian. How exactly to do the sampling is a rather involved topic, and will be handled in Chapter 4.

3.4.2 Updating Parameters

Now that we can evaluate the energy, the function to minimize, we could in principle start optimizing our parameters. If the parameters are few and discrete we could simply evaluate the energy for all of them and pick the one which gives the best result. However, this approach will quickly become intractable for even a modest number of parameters, and more intelligent methods should be pursued.

Perhaps the most common optimization strategy relies on computing the gradient of the energy w.r.t. $\boldsymbol{\alpha}$, and then change $\boldsymbol{\alpha}$ in the opposite direction of this. After all, a gradient by definition points in the direction of steepest (local) ascent, so going the other way will be the direction of steepest descent. How exactly to make these changes is a topic of its own, and discussed in depth in Chapter 5, but the simplest implementation is as follows:

$$\boldsymbol{\alpha}^{(i+1)} = \boldsymbol{\alpha}^{(i)} - \eta \boldsymbol{\nabla}_{\boldsymbol{\alpha}} \langle \hat{H} \rangle, \tag{3.25}$$

for a suitably chosen hyperparameter $\eta \in \mathbb{R}$, typically $\eta \ll 1$. This says that the updated parameters $\boldsymbol{\alpha}^{(i+1)}$ are equal to the previous ones, $\boldsymbol{\alpha}^{(i)}$, minus a small step in the direction of the gradient of $\langle \hat{H} \rangle$ w.r.t. each parameter. After a sufficient number of iterations, we hope that the parameters converge to their optimal values.

This raises another question, namely how to compute $\boldsymbol{\nabla}_{\boldsymbol{\alpha}} \langle \hat{H} \rangle$. Using the basic properties of derivatives we get (temporarily omitting the subscript on

ψ_α):

$$\nabla_\alpha \left\langle \hat{H} \right\rangle = \nabla_\alpha \left(\frac{\int dX\, \psi^* \hat{H} \psi}{\int dX\, |\psi|^2} \right) \tag{3.26}$$

$$= \frac{\int dX\, |\psi|^2 \left[\int dX\, \nabla_\alpha(\psi^*) \hat{H} \psi + \psi^* \hat{H} \nabla_\alpha(\psi) \right]}{\left(\int dX\, |\psi|^2 \right)^2} \tag{3.27}$$

$$- \frac{\left(\int dX\, \psi^* \hat{H} \psi \right) \left(\int dX\, \nabla_\alpha(|\psi|^2) \right)}{\left(\int dX\, |\psi|^2 \right)^2} \tag{3.28}$$

$$= \left\langle \frac{\nabla_\alpha(\psi^*) \hat{H} \psi + \psi^* \hat{H} \nabla_\alpha(\psi)}{|\psi|^2} \right\rangle - \left\langle \hat{H} \right\rangle \left\langle \frac{\nabla_\alpha |\psi|^2}{|\psi|^2} \right\rangle, \tag{3.29}$$

where we have used the fact that $\hat{H}$ does not depend on $\boldsymbol{\alpha}$ in any way.

To further simplify this, we use the trick $\nabla(f)/f = \nabla(\ln f)$, as well as the hermiticity of $\hat{H}$:

$$\nabla_\alpha \left\langle \hat{H} \right\rangle = \left\langle \frac{\nabla_\alpha(\psi^*) \hat{H} \psi + \nabla_\alpha(\psi) (\hat{H} \psi)^*}{|\psi|^2} \right\rangle - \left\langle \hat{H} \right\rangle \left\langle \frac{\nabla_\alpha |\psi|^2}{|\psi|^2} \right\rangle \tag{3.30}$$

$$= \langle \nabla_\alpha(\ln \psi^*) E_L + \nabla_\alpha(\ln \psi) E_L^* \rangle - 2 \left\langle \hat{H} \right\rangle \langle \nabla_\alpha \ln |\psi| \rangle . \tag{3.31}$$

At this point, we are going to explicitly assume that the wave function is real. We are not going to propose any complex trial wave functions in this thesis, so this assumption will be valid for our purposes. The final expression then becomes (using $\left\langle \hat{H} \right\rangle = \langle E_L \rangle$):

$$\nabla_\alpha \left\langle \hat{H} \right\rangle = 2[\langle \nabla_\alpha(\ln \psi) E_L \rangle - \langle E_L \rangle \langle \nabla_\alpha \ln |\psi| \rangle]. \tag{3.32}$$

Computationally we already know how to evaluate $\langle E_L \rangle$, and we employ a completely analogous approach for the two other expectation values. With this final piece of the puzzle we have everything we need in order to perform a VMC calculation for the ground state of a system.

Chapter 4

Monte Carlo Methods

In Chapter 3 we found that we needed a way to sample system configurations $\mathbf{X}$ from the probability density function (PDF) described by the wave function. We needed this as a means to evaluate integrals. Specifically, we needed to evaluate the expectation value of the Hamiltonian w.r.t. a given wave function ψ_α:

$$\langle \hat{H} \rangle = \frac{\langle \psi_\alpha | \hat{H} | \psi_\alpha \rangle}{\langle \psi_\alpha | \psi_\alpha \rangle} = \int d\mathbf{X}\, P_{\psi_\alpha}(\mathbf{X}) E_L(\mathbf{X}) \tag{4.1}$$

$$\approx \frac{1}{N} \sum_{i=1}^{N} E_L(\mathbf{X}_i) \quad \text{where} \quad \mathbf{X}_i \overset{d}{=} P_{\psi_\alpha}. \tag{4.2}$$

where N is set sufficiently large to satisfy the required accuracy. Due to the central nature of this method, we will start by a proper presentation of this Monte Carlo (MC) technique as it is used here, followed the details of how to obtain the samples we need.

4.1 Monte Carlo Integration

Monte Carlo integration (MCI) is a general technique for numeric evaluation of any arbitrary integral. In general it concerns evaluating any multidimensional definite integral, written as:

$$I = \int \cdots \int_\sigma d\mathbf{x}\, f(\mathbf{x}), \tag{4.3}$$

for some integrand $f(\mathbf{x})$ and where σ denotes a subset of $\mathbb{R}^m$ with a m-dimensional volume given by:

$$V = \int \cdots \int_\sigma d\mathbf{x} \tag{4.4}$$

In its simplest form, Monte Carlo integration (MCI) works by sampling N *uniformly* distributed points $x_1, \ldots, x_N$ from σ, and uses a Riemann sum formulation of the integral I:*

$$I = \lim_{N\to\infty} M_N := \lim_{N\to\infty} \sum_{i=1}^{N} f(x_i)\frac{V}{N} = V\,\langle f(x)\rangle_\sigma, \tag{4.5}$$

where $\langle\cdot\rangle_\sigma$ denotes the expectation value over all points in σ when all points are equally likely. The estimate becomes increasingly accurate for increasing N, and is exact in the limit where $N \to \infty$.

4.1.1 Estimating the Error

For any finite N the result of using MCI will not be exact. Because of this, it is immensely useful to be able to assign a statistical certainty to the result. For instance, say we estimate the ground state energy of a system using two different wave functions. We perform the integrals and get (to the first five digits) $\langle H\rangle_A = 1.0315\,\mathrm{eV}$ and $\langle H\rangle_B = 1.0943\,\mathrm{eV}$. Recalling that the lowest energy is the most physically accurate, can we say that A is better than B? No, because we lack information on the precision of the integrals. Assume that in reality, the numbers were $\langle H\rangle_A = 1.03(8)\,\mathrm{eV}$ and $\langle H\rangle_B = 1.09(8)\,\mathrm{eV}$, where the numbers in parenthesis indicate the uncertainty in the last digit. Now we know that the two numbers are not different in a statistically significant way, and any difference could be entirely random. We would need to use more sample points to lower the uncertainty to be able to say anything more about them.

We obtain a statistical estimate of the error we make in approximating the integral by considering the variability of the individual terms of Eq. (4.5). Let's start by estimating the variance of the integrand:

$$\mathrm{Var}[f(x)] := \sigma_N^2 = \frac{1}{N-1}\sum_{i=1}^{N}\left(f(x_i) - \frac{1}{N}\sum_{j=1}^{N} f(x_j)\right)^2. \tag{4.6}$$

It then follows, by the properties of variance,

$$\mathrm{Var}[M_N] = \mathrm{Var}\left[\frac{V}{N}\sum_{i=1}^{N} f(x_i)\right] = \frac{V^2}{N^2}\sum_{i=1}^{N}\mathrm{Var}[f(x_i)] = \frac{V^2\sigma_N^2}{N} \tag{4.7}$$

*Notice that x_i now denotes one random sample, and has nothing to do with the i'th particle. For the time being we don't talk about particles, and the subscripts indicate different samples.

$$\implies \mathrm{Std}[M_N] = \sqrt{\mathrm{Var}[M_N]} = \frac{V\sigma_N}{\sqrt{N}}. \tag{4.8}$$

Equation (4.8) is the standard deviation of the sample mean, otherwise known as the standard error of the mean (SEM). It is common use the SEM as the estimate of uncertainty on expectation values. Another common choice is to give confidence intervals based on the SEM. Assuming the variance is normally distributed,[*] we can say with e.g. 95 % certainty that the true expectation value, I, is in the interval

$$\left[\mathrm{CI}_{-}^{95}, \mathrm{CI}_{+}^{95}\right] := [M_N - 1.96\,\mathrm{Std}[M_N], M_N + 1.96\,\mathrm{Std}[M_N]]. \tag{4.9}$$

Equation (4.8) tells us that the expected statistical error we make when using MCI goes like $\mathcal{O}(1/\sqrt{N})$, and depends linearly on the size of the volume and standard deviation of the integrand itself. This illustrates both the advantage, and disadvantage of MCI compared to other, deterministic integration methods. Its advantage is its simple dependency on the volume, and its independence from the particular number of dimensions in the integral. Other methods tend to depend exponentially on the dimensionality, and as such MCI is often the best choice for multidimensional integrals. Its disadvantage is the relatively slow convergence rate, which is asymptotically much worse than other approaches [14].

Correction for Autocorrelation

Actually, Eq. (4.7) includes a mistake. We applied a property of variance on a sum of random variables. The equation is correct if and only if the samples $\mathbf{x}_i$ are perfectly independent. Unfortunately for us, the samples will not always be so, depending on the algorithm used to generate them. We can sample independently from a uniform distribution and some other notable distributions (Section 4.2), but not in general. This will be especially true for the algorithms we use to sample from arbitrary wave functions (Section 4.2.3).

The version of Eq. (4.7) that is true in general accounts for autocorrelated samples:[†]

$$\mathrm{Var}[M_N] = \mathrm{Var}\left[\frac{V}{N}\sum_{i=1}^{N} f(\mathbf{x}_i)\right] = \frac{V^2}{N^2}\sum_{i=1}^{N}\sum_{j=1}^{N} \mathrm{Cov}[f(\mathbf{x}_i), f(\mathbf{x}_j)]. \tag{4.10}$$

[*]Which for our wave functions is often times at least approximately true.

[†]Covariance is a measure of the joint variability between two random variables. Autocorrelation is used to refer to the covariance between a random variable and itself at pairs of time points.

This is always greater than or equal to Eq. (4.7), and equal only if the off-diagonal covariance terms (where $i \neq j$) are all zero. Not accounting for the autocorrelation in the samples will therefore lead us to underestimate the error, which is arguably much worse than the opposite.

There is, unfortunately a heavy cost to computing the full covariance. Typically, to keep the expectation values accurate, we will use very large values for N, often on the order of tens or hundreds of millions. The double loop in Eq. (4.10) will then quickly become unfeasible to compute, with trillions or quadrillions of iterations. In practice we need to estimate it. Yes, we are going to estimate the estimate of the error we make in an estimate.

The strategy we shall use for all later error analysis is called *blocking*, and specifically we use an automated form presented in an article by Jonsson [15]. Say we have a sequence of sample values $f(x_i)$:

$$D_0 = \begin{bmatrix} f(x_1) & f(x_2) & \dots & f(x_N) \end{bmatrix}, \tag{4.11}$$

and let's assume that there is some non-zero autocovariance between samples that are sufficiently close in the sequence. The idea of blocking is to group "blocks" of samples together, replacing them by their mean value. For instance, after one blocking transformation of the above sequence we get:

$$D_1 = \begin{bmatrix} \left(\frac{f(x_1)+f(x_2)}{2}\right) & \left(\frac{f(x_3)+f(x_4)}{2}\right) & \dots & \left(\frac{f(x_{n-1})+f(x_n)}{2}\right) \end{bmatrix}. \tag{4.12}$$

If $N = 2^d$ with $d > 1$ we can repeat the process to obtain d different sequences D_i. These transformations conserve the value of Eq. (4.10) while decreasing the covariance between samples [15]. That implies that measuring the variance with Eq. (4.7) on each D_i will yield greater and greater errors, until there is no more covariance and the error estimate converges.

Figure 4.1 shows this process in practice. The example data is 2^{27} samples of the mean radial displacement of a single one-dimensional particle placed in a harmonic oscillator. Due to how the data is generated, we expect there to be a significant portion of autocorrelation present. The figure shows the standard error from Eq. (4.7), calculated on the data sets obtained by performing repeated blocking transformations. We can see it rises steadily during the first 6-7 transformations, after which the block sizes are big enough to remove all the covariance and the estimate converges. As the data set becomes too small the results start to become unpredictable. The optimal estimate is indicated by the red circle, as determined by the automated procedure developed by Jonsson [15]. In this particular case this occurred after 12 blocking transformations.

If not explicitly stated otherwise, all error estimates in this thesis will include this correction.

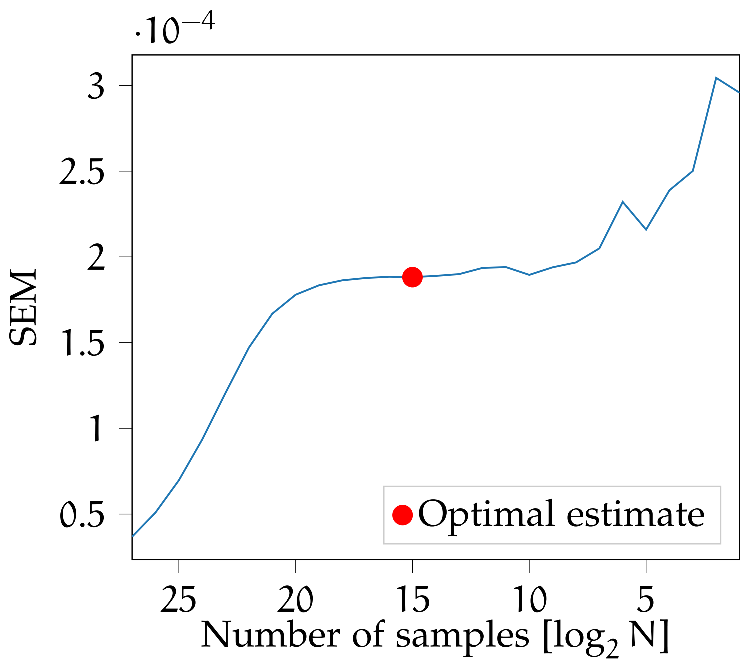


Figure 4.1: The estimated standard error of a random process excibiting symptoms of autocorrelation, shown as a function of the effective size of the data set as blocking transformations are applied. As the block sizes increase the covariance goes away, while the variance increases. After convergence, the optimal estimate is illustrated by the red dot, as determined by an automated blocking procedure [15].
Source [1, `writing/scripts/blocking-example-diagram.py`]

4.1.2 Importance Sampling

In some suitable cases we can improve quite dramatically on the simple, straightforward integration approach given in Eq. (4.5) with a technique known as importance sampling (IS). To illustrate this, say we would like to evaluate the following integral:

$$I = \int_{-\infty}^{\infty} dx\, f(x) = \int_{-\infty}^{\infty} dx\, \frac{\exp\{-x^2/2\}}{\sqrt{2\pi(1+x^2)}} = 0.789\,64. \tag{4.13}$$

The integrand is plotted in Fig. 4.2, and the observant reader might recognise this as the product of a normal distribution and a Student-t distribution. The correct value for the integral is also given, so we have a reference for the results.

For comparison, let's start with the straightforward approach from before. Because the integral goes to infinity the "volume" V would not be well defined, and we are forced to truncate the region manually. From looking at

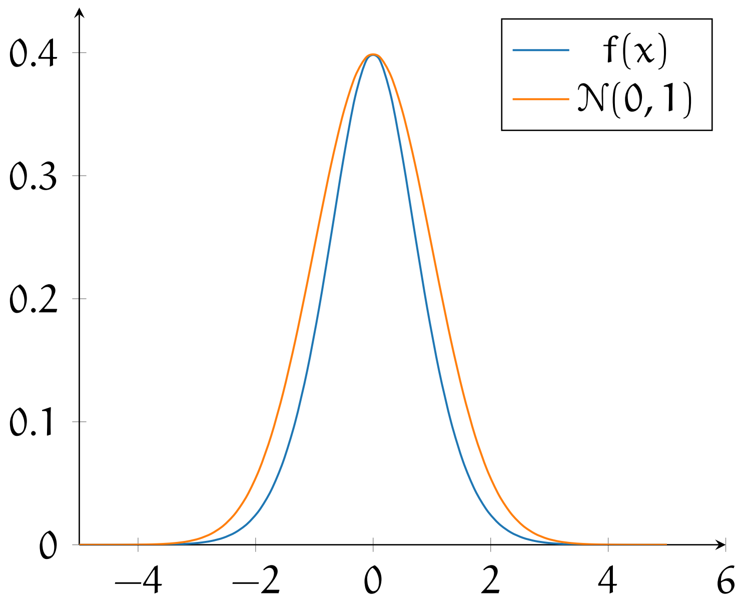


Figure 4.2: Plot of the function in Eq. (4.13), enveloped by a standard normal distribution.
Source [1, `writing/MonteCarlo.tex`]

the graph in Fig. 4.2 we may say that $x \in [-5,5]$ should account for the vast majority of the total integral. That means we use the following estimate:

$$I \approx \frac{10}{N}\sum_{i=1}^{N} f(x_i) \quad \text{where} \quad x_i \overset{d}{=} \text{Uniform}(-5,5). \tag{4.14}$$

The main issue with this approach is that 1) we need to truncate the integral manually and 2) no matter where we place the box boundaries we will tend to sample a lot of x_i's in areas where f gives very small contributions to the integral. Ideally we would like our sample points to be distributed as closely as possible to f, in order to capture as much information as we can. This is the idea of IS. Instead of using the uniform distribution for sampling, we use some probability distribution which more closely resembles the integrand, call it $g(x)$. Formally we then restate the integral as follows:

$$I = \int_{-\infty}^{\infty} dx\, f(x) = \int_{-\infty}^{\infty} dx\, \frac{f(x)}{g(x)} g(x). \tag{4.15}$$

This can be interpreted as the expectation of $z(x) = f(x)/g(x)$ for x's drawn from g, and so the corresponding estimation is then:

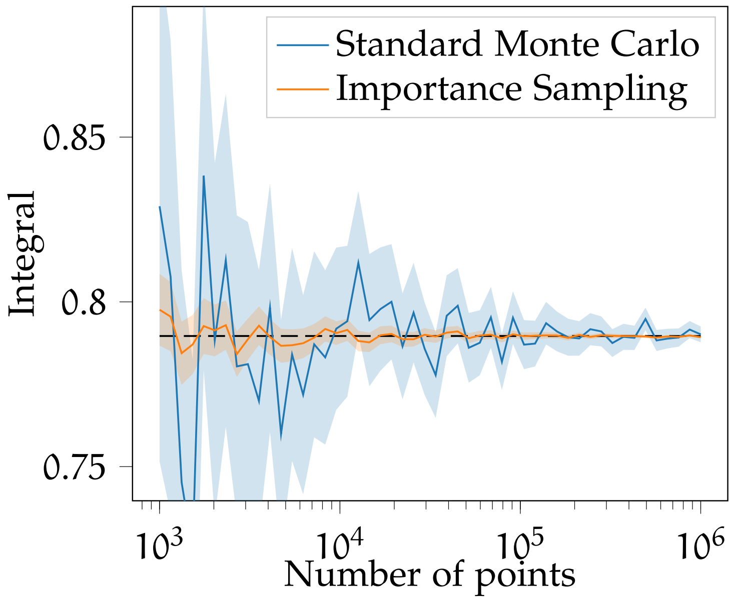


Figure 4.3: Convergence of the integral in Eq. (4.13), using regular MCI and IS with $g = \mathcal{N}(0,1)$. The points indicate the approximated value of the integral for each particular experiment, and the shaded areas indicate the corresponding 95 % confidence intervals for the expected value. We see the latter displaying both greater accuracy and tighter confidence intervals.
Source [1, `writing/scripts/monte_carlo_int_example.py`]

$$I \approx \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{g(x_i)} \quad \text{where} \quad x_i \overset{d}{=} g. \tag{4.16}$$

Setting $g = \text{Uniform}(-5, 5)$ recovers Eq. (4.14), so this is simply the natural generalization of the standard approach for an arbitrary PDF.

Going back to the example, we should now choose a distribution function that closely resembles the integrand, while still being simple to sample from. In this (contrived) example, a natural choice is to use the standard normal distribution, $g = \mathcal{N}(0,1)$. In Fig. 4.2 we can see how $g(x)$ is enclosing $f(x)$ much more tightly than any rectangular box might hope to.

Figure 4.3 shows the convergence of the MC approximations towards the correct value for an increasing number of sampled points. The drawn line is the mean value at each run, while the shaded areas show the corresponding 95 % confidence intervals. Because of the more suited sampling distribution, we obtain results which are more accurate, and perhaps most importantly, tighter confidence bounds. In general, using better sampling distributions will tend to give us lower variance results.

4.2 Sampling from Arbitrary Probability Density Functions

So far we have taken for granted the ability to sample numbers from various PDFs. We dedicate this section to discuss sampling from the bottom up, culminating in a detailed presentation of the most central algorithm in this thesis, the Metropolis-Hastings algorithm.

4.2.1 The Uniform Distribution

The basic building block upon which we shall build all other random sampling techniques is the ability to sample random numbers from the uniform distribution, $\mathrm{Uniform}(a, b)$. This is the simplest distribution possible, where all numbers in the range $[a, b]$ have the same probability density. The definition of its PDF is simply:

$$p_U(x \mid a, b) := \begin{cases} \frac{1}{b-a} & \text{for} \quad x \in [a, b] \\ 0 & \text{otherwise} \end{cases}. \tag{4.17}$$

Most commonly we operate only with the *standard uniform distribution*, $\mathrm{Uniform}(0, 1)$. Any other range can be simply related via the trivially verifiable identity

$$p_U(x \mid a, b) = p_U\left(\frac{x-a}{b-a} \,\middle|\, 0, 1\right). \tag{4.18}$$

But how exactly do we obtain random realizations from this PDF? First we need to depart from the typical notion of randomness, and realize that we are unable to write an algorithm which produces a truly random result. While we could in principle rely on nature to provide sources of complete, true randomness (such as the exact behavior of a quantum mechanical observable), this is impractical when we need fast, on-demand samples. Instead we settle for *pseudo-random* samples. A pseudo-random sequence $x_1, x_2, \dots x_n$ implies that one cannot[*] predict x_{n+1} without insight into how the sequence is generated. In other words, the (ideal) pseudo-random sequence is indistinguishable from a truly random sequence for anyone observing the numbers x_i alone.

There are a multitude of algorithms that can generate a sequence of pseudo-random *integers*, all with varying properties.[†] The simplest family of such

[*]In any practical way, barring exhaustive trial and error of every conceivable underlying algorithm.

[†]Such properties could include range of possible numbers, period, level of bias and how cryptographically secure they are.

algorithms is a linear congruential generator (LCG) [16]. It defines a pseudo-random sequence by the simple recurrence relation

$$x_{n+1} = (a x_n + c) \mod m, \tag{4.19}$$

where a, c and m are constants which determine the behaviour of the algorithm, and x_0 needs to be specified manually. For instance, Press et al. [14] use $a = 1664525$, $c = 1013904223$ and $m = 2^{32}$.

This produces uniformly distributed integers in the range $[0, m)$. Numbers from the standard uniform distribution can then simply be obtained by dividing the integers by m. The complete algorithm is shown in Algorithm 4.1.

Algorithm 4.1 Sampling from Uniform(l, u)

```
Require: $a, c, m$ and $x_0$
1: function UNIF(lower bound $l$, upper bound $u$)
2:     $x \leftarrow x_0$
3:     repeat
4:         $x \leftarrow (ax + c) \mod m$
5:         yield $(x/m) \times (u - l) + l$
6:     until done
7: end function
```

4.2.2 Inverse Transform Sampling

Armed with uniformly distributed random numbers, we now turn towards generating numbers from other PDFs. Let $U \overset{d}{=} \text{Uniform}(0, 1)$ be a uniformly distributed stochastic variable, and let X be a stochastic variable associated with a PDF, p, and a corresponding cumulative distribution function (CDF), F, i.e.

$$F(x) = \Pr(X \leqslant x) = \int_{-\infty}^{x} dt\, p(t). \tag{4.20}$$

We would like to define a transformation $T : [0, 1] \mapsto \mathbb{R}$ that can map from uniform numbers to numbers that follow the given CDF, i.e. define T such that $T(U) \overset{d}{=} X$. We have:

$$F(x) = \Pr(X \leqslant x) = \Pr(T(U) \leqslant x) = \Pr\Big(U \leqslant T^{-1}(x)\Big) = T^{-1}(x), \tag{4.21}$$

because $T^{-1}(x) \in [0, 1]$ by definition, and assuming that $T^{-1}(x)$ is strictly monotone. It follows then that $F^{-1}(U) \overset{d}{=} X$, i.e. if the CDF is strictly monotone (as CDFs should be) then we can get realizations of X by applying F^{-1} to realizations of U.

The algorithm to sample from any PDF with a tractable inverse CDF is then extremely simple, and for completeness it is listed in Algorithm 4.2.

Algorithm 4.2 Inverse transform sampling

```
Require: Cumulative distribution function $F$
Ensure: Random $x$ with CDF equal to $F$
1: repeat
2:     $u \leftarrow \text{unif}(0, 1)$
3:     $x \leftarrow F^{-1}(u)$
4:     yield $x$
5: until done
```

4.2.3 The Metropolis-Hastings Algorithm

Algorithm 4.2 works great for a number of standard PDFs. For many cases, however, we do not have a tractable form for the inverse CDF, which renders the algorithm useless. So what about the wave functions we are interested in? The PDF in question is, as defined in Eq. (3.21),

$$P_\psi(\mathbf{X}) := \frac{|\psi|^2}{\int d\mathbf{X}\, |\psi|^2}. \tag{4.22}$$

Sadly, computing F^{-1} for this PDF, if even possible, would be a very costly operation. In fact, the normalization integral in the denominator is enough to make practical sampling from P_ψ impossible. Even worse, considering that ψ is updated continuously throughout a variational Monte Carlo (VMC) calculation, we could not even cache the result of the integral after computing it once.

Luckily, we have a solution: The Metropolis-Hastings Algorithm. This algorithm is the workhorse behind thousands of applications, including VMC, and is considered to be among the most influential algorithms of the 20th century. While the algorithm itself is quite simple, the argument for *why* it works is a little more involved. The following sections will be dedicated to the mathematical underpinnings. In the mean time, the full algorithm is shown in Algorithm 4.3.

The most important thing about Algorithm 4.3 is the fact that we only need to know the ratios of probabilities, $P(\mathbf{x}')/P(\mathbf{x})$. This means that we

Algorithm 4.3 Metropolis-Hastings sampling

```
Require: Probability density $P(x)$, proposal distribution $q(x' \mid x)$
Ensure: Random $x$ drawn from $P$
 1: Initialize $x$, randomly or otherwise
 2: repeat
 3:     $x' \leftarrow q(x' \mid x)$
 4:     $u \leftarrow \text{unif}(0, 1)$
 5:     $A \leftarrow P(x')q(x \mid x')/P(x)q(x' \mid x)$
 6:     if $u \leqslant A$ then
 7:         $x \leftarrow x'$
 8:     end if
 9:     yield $x$
10: until done
```

don't need the probabilities to be normalized to unity*, meaning we don't have to compute the costly integral in Eq. (3.21). For our purposes this is essential, and Algorithm 4.3 is the reason why VMC is possible.

Formal Derivation

The Metropolis-Hastings algorithm builds on what is called a *Markov Chain*. A Markov Chain is a type of stochastic model that describes a sequence of possible states. We imagine some space of possible states and a set of transition probabilities, $T(x' \mid x)$, describing the likelihood of moving between them. The defining property of a Markov Chain, as opposed to other stochastic processes is that the transition probabilities $T(x' \mid x)$ from a state x to a state x' depends only on the current state and not on any previous states. In other words, it matters only where you are, not where you have been.

For well behaved Markov Chains, as the number of steps in the chain increases, the distribution of the states will asymptotically approach a stationary distribution, $\pi(x)$. The Metropolis-Hastings algorithm works by carefully constructing $T(x' \mid x)$ such that $\pi(x) = P(x)$, where $P(x)$ is the desired PDF.

In order to find the correct $T(x' \mid x)$, we require that the stationary distribution $\pi(x)$ exists and that it is unique:

Existence: A sufficient condition for the existence of $\pi(x)$ is that of *detailed balance*. This says that the probability of being in state x and transitioning to x' is equal to the probability of being in state x' and transitioning

*We do need the probabilities to be *normalizable* though. In our case, where $P \propto |\Psi|^2$, we have already assumed this requirement in Section 2.4.

to x:

$$\pi(x)T(x' \mid x) = \pi(x')T(x \mid x'). \tag{4.23}$$

Uniqueness: The distribution $\pi(x)$ is unique if the Markov chain is *ergodic*. This is the case when we don't return to the same state after a fixed number of transitions, and that all states can be reached in a finite number of transitions.

Deriving the correct $T(x' \mid x)$ starts by factoring the transition probabilities into a proposal distribution, $q(x' \mid x)$, and an acceptance ratio $A(x', x)$. The idea is to use the former to propose the next state, and use the latter to accept or reject the proposal. Inserting this into Eq. (4.23) and rearranging we get:

$$\frac{A(x', x)}{A(x, x')} = \frac{P(x')}{P(x)} \frac{q(x \mid x')}{q(x' \mid x)}. \tag{4.24}$$

We can now freely choose an acceptance ratio that satisfies the above. The common choice is:

$$A(x', x) = \min\left(1, \frac{P(x')}{P(x)} \frac{q(x \mid x')}{q(x' \mid x)}\right). \tag{4.25}$$

At this point, all that remains is to specify a proposal distribution $q(x' \mid x)$. Depending on the choice of q we get a sampler with different attributes. We have implemented two version, presented in the following sections.

Metropolis Algorithm

The simplest choice, referred to as simply the Metropolis algorithm, is to choose a distribution such that $q(x' \mid x) = q(x \mid x')$. That way the acceptance ratio simplifies significantly, fully canceling out q. In our implementation we have used a uniform proposal distribution:

$$q(x' \mid x) = p_U\left(x' \,\middle|\, x - \frac{\Delta x}{2}, x + \frac{\Delta x}{2}\right), \tag{4.26}$$

where Δx is a *step size* that regulates how large each perturbation should be. This leads to the simplest form of Algorithm 4.3, given for completeness in Algorithm 4.4. The algorithm given here is adapted particularly for our purposes, where we want to sample a matrix, $\mathbf{X} \in \mathbb{R}^{N \times D}$, for N particles in D dimensions. In particular, our implementation yields a new sample after moving only one of the particles. This is a common case for VMC applications, because this allows for a code optimization that is crucial when using Slater determinants in wave functions. We don't use such wave functions in this thesis, but future work might benefit from this choice.

Algorithm 4.4 Metropolis sampling

```
Require: Probability density $P(\mathbf{X}) = |\psi(\mathbf{X})|^2$, step size $\Delta x$
Ensure: Random $\mathbf{X}$ drawn from $P$
 1: Initialize $\mathbf{X}$, randomly or otherwise
 2: repeat
 3:     for $i \leftarrow 1 : N$ do
 4:         $\mathbf{X}' \leftarrow \mathbf{X}$
 5:         for $d \leftarrow 1 : D$ do
 6:             $\delta \leftarrow \text{unif}(-0.5, 0.5)$
 7:             $X'_{i,d} \leftarrow X_{i,d} + \Delta x \cdot \delta$
 8:         end for
 9:         $u \leftarrow \text{unif}(0, 1)$
10:         $A \leftarrow P(\mathbf{X}')/P(\mathbf{X})$
11:         if $u \leqslant A$ then
12:             $\mathbf{X} \leftarrow \mathbf{X}'$
13:         end if
14:         yield $\mathbf{X}$
15:     end for
16: until done
```

Importance Sampling

Algorithm 4.4 suffers from the same issues that ordinary MCI does, as opposed to the enhancement of IS. The proposal distribution used in the Metropolis algorithm, while simple to implement, can certainly be improved upon. The idea is to guide the random walk in space towards areas of greater probability. We will now give a brief physical motivation for the strategy, glossing over most technical details.

Particles will tend towards regions of space where $P(\mathbf{X})$ is large. We may say that this is the result of a *quantum drift force*. Without further motivation, we define the drift force acting on particle k as follows:

$$\mathbf{F}_k(\mathbf{X}) = \frac{2\boldsymbol{\nabla}_k\Psi(\mathbf{X})}{\Psi(\mathbf{X})}, \tag{4.27}$$

which points in the direction of greatest increase in Ψ with respect to particle k's position. We also expect the system to exhibit some degree of random motion, as it is a quantum system. This combination of drift and random motion is described by the Langevin equation [17],

$$\frac{\partial \boldsymbol{x}_k}{\partial t} = D\mathbf{F}_k + \boldsymbol{\eta}, \tag{4.28}$$

where D is called the drift coefficient, and $\boldsymbol{\eta}$ is a vector of uniformly distributed random values (with zero mean) and accounts for the random motion. We fix $D = 1/2$. Applying Euler's method to the Langevin equation we can get:

$$\mathbf{x}_k' = \mathbf{x}_k + \frac{1}{2}\mathbf{F}_k\,\Delta t + \boldsymbol{\xi}\sqrt{\Delta t}, \tag{4.29}$$

where Δt is a time step similar to Δx for plain Metropolis, and $\boldsymbol{\xi}$ is a vector of values drawn from the standard normal distribution. This is our new recipe for generating proposals $\mathbf{x}'$. The final piece is to find the corresponding PDF, $q(\mathbf{x}' \mid \mathbf{x})$.

To this end, we consider the Fokker-Planck equation, which for one particle in one dimension is written as

$$\frac{\partial \Psi}{\partial t} = D\frac{\partial}{\partial x}\left(\frac{\partial}{\partial x} - F\right)\Psi, \tag{4.30}$$

where D is as before and F is the one-dimensional analog of $\mathbf{F}_k$. This describes the time-evolution of a probability distribution under the influence of a drift force and random impulses. We've let Ψ play the role of the probability distribution. Eq. (4.30) yields a solution given by the following Green's function (for one particle):

$$G\left(\mathbf{x}_k', \mathbf{x}_k, \Delta t\right) \propto \exp\left[-\frac{\left\|\mathbf{x}_k' - \mathbf{x}_k - D\Delta t\,\mathbf{F}_k\right\|^2}{4D\Delta t}\right]. \tag{4.31}$$

This is interpreted as the probability of transitioning to position $\mathbf{x}_k'$ from $\mathbf{x}_k$ within a time interval Δt, so that finally we have the proposal distribution:

$$q\left(\mathbf{x}' \mid \mathbf{x}\right) = G\left(\mathbf{x}', \mathbf{x}, \Delta t\right). \tag{4.32}$$

Note that because we only need the ratio of q values, there is no problem that Eq. (4.31) only gives the proportionality. The final algorithm is shown in Algorithm 4.5. See Section 10.6 for a demonstration of how Algorithm 4.5 compares to Algorithm 4.4.

Algorithm 4.5 Metropolis-Hastings importance sampling

```
Require: Probability density $P(\mathbf{X}) = |\psi(\mathbf{X})|^2$, step size $\Delta t$
Ensure: Random $\mathbf{X}$ drawn from $P$
 1: Initialize $\mathbf{X}$, randomly or otherwise
 2: repeat
 3:     for $i \leftarrow 1 : N$ do
 4:         $\mathbf{X}' \leftarrow \mathbf{X}$
 5:         for $d \leftarrow 1 : D$ do
 6:             $\xi \leftarrow \mathcal{N}(0, 1)$
 7:             $X'_{i,d} \leftarrow X_{i,d} + \sqrt{\Delta t} \cdot \xi + \frac{1}{2}\Delta t\,(\mathbf{F}_i(\psi))_d$
 8:         end for
 9:         $u \leftarrow \mathrm{unif}(0, 1)$
10:         $A \leftarrow P(\mathbf{X}')/P(\mathbf{X})$
11:         $A \leftarrow A \cdot q(\mathbf{x}_i \mid \mathbf{x}'_i)/q(\mathbf{x}'_i \mid \mathbf{x}_i)$
12:         if $u \leqslant A$ then
13:             $\mathbf{X} \leftarrow \mathbf{X}'$
14:         end if
15:         yield $\mathbf{X}$
16:     end for
17: until done
```

Chapter 5

Machine Learning

Machine learning (ML) is a field of study concerned with building models[†] that can perform a task without the need for explicit instructions on how to do so. The key word is *learning*; we want to enable the model to discover the best way to solve the task at hand within the constraints that we have imposed on it.

A simple example of an ML algorithm is linear regression, where we aim at finding a linear function that best fits some observed data, with respect to some metric of what constitutes a *good fit*. Other examples include dimensionality reduction, image classification, algorithmic trading and playing chess. All of these examples *could* be implemented in terms of a large set of if-this-then-that rules, covering the explicit response to every conceivable input. The problem with that is that we often have little to no idea of how exactly to determine what to do for each case, not to mention the potential infinity of rules we would need to cover all cases.

The idea of ML is to instead describe a parameterized model that maps inputs to outputs, and a metric to quantitatively describe how good or bad the outputs are. Then we ask the model to find the set of parameters that will maximize the goodness (or minimize badness) of the metric. In short, ML is saying *what* you want done, rather than saying *how* you want it done.

This chapter is dedicated to presenting the relevant bits of ML that will be relevant to the work in this thesis. As the field is vast, only a very minimal selection of topics will be covered, and the reader is encouraged to dive deeper into topics where interest is sparked. We start with an overview of the general process, exemplified by linear regression for simplicity. From there we shall introduce neural networks, along with a more in depth discussion of optimization strategies.

[†]The term *model* is used in a very general sense, and refers to anything that takes information as input and produces a corresponding output.

5.1 General Procedure

For our purposes, we will define the steps involved in developing a machine learning model as follows:

1. Define a model, $\hat{f}_{\hat{\alpha}}(x)$ dependent on some parameters $\hat{\boldsymbol{\alpha}}$
2. Define a cost function $C(\hat{f}_{\hat{\alpha}})$ - a metric of how far away from the ideal model $\hat{f}_{\hat{\alpha}}$ is
3. Minimize $C(\hat{f}_{\hat{\alpha}})$ with respect to $\boldsymbol{\alpha}$

It is common to divide ML into two main sub-domains: supervised and unsupervised learning. Supervised learning is characterized by us having access to a data set where both inputs *and* desired outputs are specified. This is possibly the most common case, or at least the case most people *want* to work with. Examples are many, and include most regression and classification tasks.

Unsupervised learning is (naturally) the other case, where we do not have information about any output responses. In these cases we don't focus on learning the output (as we have no idea of what we want that to be), but rather on what we can learn about the inputs themselves. This can include dimensionality reduction, clustering analysis and more.

Often we need to introduce another sub-domain for the cases that fall in between; Reinforcement Learning. This is the domain of problems where we do not have explicit knowledge of what outputs should be, but where we can still infer something about whether or not an output was good. An example of this is the problem of playing chess. While we can't label all board states with a corresponding "correct move", we can say that moves that lead to checkmate are probably good. Reinforcement learning tries to adapt a model so as to maximize/minimize the reward/punishment that follows as a consequence of its actions.

Variational Monte Carlo (VMC) is a type of reinforcement learning. This is because we do not have information about exactly what the value of the wave function should be for every configuration, but we still have a way to evaluate the goodness of the wave function by calculating the energy it predicts. We'll go more in depth into how all this connects to VMC in Section 5.4.

5.2 Supervised Learning

Even though our particular interest in machine learning is for its use in VMC, a lot of the ideas and techniques we need come from the supervised learning domain. Because of that, we will quickly introduce all the relevant components in the following sections. For simplicity and clarity, we will do this through the simplest example we have; linear regression.

In general for a supervised learning task, we have the following ingredients:

- A data set, $\mathcal{D} = \{x_i, y_i\}_{i=1}^{n}$, of n inputs x_i and corresponding expected outputs y_i
- A proposed model $\hat{y} = \hat{f}_{\hat{\alpha}}(x)$ that we want to fit to the data
- A cost function $\mathcal{C}(\hat{f}_{\hat{\alpha}}; \mathcal{D})$ to minimize with respect to the model parameters $\hat{\alpha}$. Note that for supervised learning, the cost function is dependent on the data set, allowing us to define metrics on how well the model reproduces the data.
- An optimization strategy for how we should approach the minimization

5.2.1 Example: Linear Regression

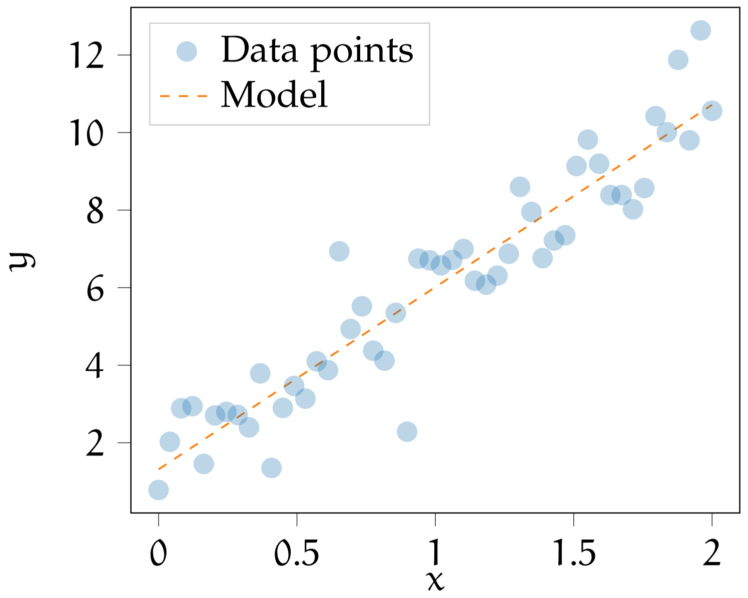


Figure 5.1: Example of linear regression applied to a simple data set of a single input variable. The regression tries to filter away the noise and find the most likely actual trend line.
Source [1, `writing/scripts/linear_regression_example.py`]

We assume that the true underlying process, $f_{\alpha}(x)$, that generated $\mathcal{D}$ is linear in the inputs x. For simplicity, we will also assume that the outputs are scalar, $\mathbf{y} := y$. That is, we assume the data was generated in the following way:

$$y_i = f_{\boldsymbol{\alpha}}(\mathbf{x}_i) + \epsilon = \mathbf{x}^T \boldsymbol{\alpha} + \epsilon, \tag{5.1}$$

where $\boldsymbol{\alpha}$ is a column vector of coefficients, and $\epsilon \sim \mathcal{N}(0, \sigma^2)$ is normally distributed noise with zero mean and variance σ^2, introduced via measurement inaccuracy etc.*

Model

Based on the above assumption, we propose the following model:

$$\hat{f}_{\hat{\boldsymbol{\alpha}}}(\mathbf{x}) = \mathbf{x}^T \hat{\boldsymbol{\alpha}}. \tag{5.2}$$

Ideally we want to learn $\hat{\boldsymbol{\alpha}}$ from the data such that $\hat{\boldsymbol{\alpha}} \simeq \boldsymbol{\alpha}$.

Cost Function

In order to quantify which $\hat{\boldsymbol{\alpha}}$ is optimal, we need a metric to compare by. There are several conceivable choices here, many of which might lead us to the correct $\boldsymbol{\alpha}$. Most common, at least in this case, is to use the so-called mean squared error (MSE):

$$\mathcal{C}_{\mathrm{MSE}}(\hat{f}_{\hat{\boldsymbol{\alpha}}}; \mathcal{D}) = \frac{1}{n} \sum_{i=1}^{n} \left| \hat{f}_{\hat{\boldsymbol{\alpha}}}(\mathbf{x}_i) - y_i \right|^2. \tag{5.3}$$

The "learning" task is now expressed simply as the following optimization problem:

$$\hat{\boldsymbol{\alpha}} = \underset{\boldsymbol{\alpha}'}{\operatorname{argmin}}\ \mathcal{C}_{\mathrm{MSE}}(\hat{f}_{\boldsymbol{\alpha}'}; \mathcal{D}). \tag{5.4}$$

Optimization

With data, model and cost function defined, the last step is to solve Eq. (5.4). While this can in general be a hard task (more on this in Section 5.3), we can

*Note that this definition implies that $f_{\boldsymbol{\alpha}}$ has zero intercept, i.e. $f_{\boldsymbol{\alpha}}(\mathbf{0}) = 0$. We can easily work around that, however, by adding a constant element to every input vector, i.e. let $\mathbf{x}' := (1 \;\; \mathbf{x})$ be the new inputs. The second option is to simply center the data beforehand, by subtracting the mean $\bar{y} = 1/n \sum_{i=1}^{n} y_i$ from every y_i, and then optionally revert back whenever needed.

actually do this particular exercise analytically with some rather simple linear algebra. Skipping the derivation, we get the following solution:

$$\hat{\boldsymbol{\alpha}} = \left(\mathbf{X}^{T}\mathbf{X}\right)^{-1}\mathbf{X}\mathbf{y}, \tag{5.5}$$

where $\mathbf{X} = (\mathbf{x}_1\ \mathbf{x}_2\ \dots\ \mathbf{x}_n)^T$ is the matrix with inputs laid out in rows, and $\mathbf{y}$ is the column vector of all outputs.

Regularization

Often it turns out to be useful to add an extra term to the cost function that depends on the size of the parameters $\hat{\boldsymbol{\alpha}}$. This typically takes the following form:

$$\mathcal{C}(\hat{f}_{\hat{\boldsymbol{\alpha}}}; \mathcal{D}) = \mathcal{C}_{\mathrm{MSE}}(\hat{f}_{\hat{\boldsymbol{\alpha}}}; \mathcal{D}) + \gamma \|\hat{\boldsymbol{\alpha}}\|_p^p, \tag{5.6}$$

where $\|\cdot\|_p$ is the L^p norm, and γ is a parameter we set to control the degree of regularization (typically $\gamma \ll 1$). Most used are $p = 1$ (called the LASSO loss) and $p = 2$ (called the Ridge loss).

The motivation for why we might want to do this is as follows: Imagine we have a data set with $\mathbf{x}_i = (x_i, 2x_i)$, that is to say we have two variables per data point, but the two variables are directly correlated. Lastly, assume that $y_i = 3x_i$ is the underlying function. Without any regularization, all of the following models are equally good:

$$\hat{\boldsymbol{\alpha}}^{(1)} = (3, 0),\ \hat{\boldsymbol{\alpha}}^{(2)} = (1, 1),\ \hat{\boldsymbol{\alpha}}^{(3)} = (-997, 500). \tag{5.7}$$

The problem is that without regularization, this particular optimization problem ill-formed, and does not have a unique solution. While all of the above give perfect reconstruction of the data, solutions like $\hat{\boldsymbol{\alpha}}^{(3)}$ yield strange interpretations. According to this model, the output is strongly negatively correlated with x_1, in complete contradiction with the underlying truth. Adding regularization guides the model to prefer $\hat{\boldsymbol{\alpha}}$'s with smaller coefficients.

While this particular example is contrived, the point should be clear: In some cases, depending on both the data and what model we choose, the model might be too flexible for the task at hand. This is generally referred to as the problem of *overfitting*, and can lead to strange behavior. The opposite problem, called *underfitting*, refers to the case where the model is too constrained, e.g. trying to fit a first order polynomial to quadratic data.

5.3 Gradient-based Optimization

Most supervised learning tasks rely on a gradient-based optimization strategy, with the umbrella name gradient decent (GD). These methods can be summarized as follows: We want to find x^* such that

$$x^* = \underset{x}{\operatorname{argmin}}\, g(x), \tag{5.8}$$

for some function g that should be minimized.* We solve this by starting at some initial guess $x^{(0)}$ and improve it by iterating the following recurrence relation:

$$\begin{aligned} x^{(n+1)} &= x^{(n)} + \Delta x^{(n)} \\ \Delta x^{(n)} &= -\eta \boldsymbol{\nabla} g(x^{(n)}), \end{aligned} \tag{5.9}$$

where η is a suitably chosen number, typically $\eta \ll 1$. In the limit $n \to \infty$ (if η is small enough), this will converge to a minimum for g.

When $g = \mathcal{C}(\hat{f}_{\hat{\alpha}}; \mathcal{D})$ we have some choice in exactly how we should compute $\boldsymbol{\nabla}\mathcal{C}$. We could compute the cost for the entire data set, or just a single data point. These two options are referred to as GD and stochastic gradient decent (SGD), respectively. The latter is most used because it is less computationally expensive for large data sets, as well as less likely to get stuck in local minima. A hybrid of the two is also possible, called mini-batch GD.

Lastly we have various ways of extending the basic algorithm in Eq. (5.9), with more sophisticated ways of determining $\Delta x^{(n)}$, some of which we will mention here.

5.3.1 Fixed Learning Rate

This is the version presented in Eq. (5.9), in where η is chosen in advance and remains fixed throughout all iterations:

$$\Delta x^{(n)} = -\eta \boldsymbol{\nabla} g(x^n). \tag{5.10}$$

- Pros:
  - Easy to implement
  - Intuitive and easy to adapt based on results
- Cons:

*For maximization, simply minimize $-f(x)$.

- If η is to large it can lead to divergence or inaccurate results
- If η is to small we have a higher chance of getting stuck in small, local minima.
- If η is small we will need many iterations before convergence
- All elements of $\mathbf{x}$ get the same η, even though the gradient elements can have significantly different magnitude and/or stability.

5.3.2 Momentum

Two common problems arise when using the update rule in Eq. (5.10), namely that we tend to get stuck in local minima and that unstable gradients can throw us off the right track. The idea of momentum, first introduced by Rumelhart, Hinton, and Williams [18], aims to combat this by letting the update be a linear combination of the previous step and the current one:

$$\Delta\mathbf{x}^{(n)} = \alpha\Delta\mathbf{x}^{(n-1)} - \eta\nabla g(\mathbf{x}^{(n)}). \tag{5.11}$$

This introduces another hyperparameter α, which again should be set to an appropriate value. Letting $\alpha = 0$ recovers Eq. (5.10), and increasing values increases the update terms' inertia.

5.3.3 Individual Learning Rates per Element

The magnitude and/or stability of the elements of ∇g can be significantly different from each other. An optimal learning rate η might therefore not be optimal for any single element, but rather the best compromise available. A simple fix is to use a separate learning rate for each dimension:

$$\Delta\mathbf{x}^{(n)} = -\boldsymbol{\eta} \circ \nabla g(\mathbf{x}^{(n)}), \tag{5.12}$$

where $\circ$ denotes the Hadamard product (element-wise multiplication). This allows us more flexibility and will improve results, while the obvious downside is the increased need for hyperparameter tuning.

5.3.4 Averaging

Another idea useful to overcome unstable gradients or poorly converging updates is that of *averaging*, invented by Polyak and Juditsky [19]. We keep the simple update rule from Eq. (5.10), but the final $\mathbf{x}^*$ we use is set to the average of all the intermediate steps:

$$\boldsymbol{x}^* = \frac{1}{n}\sum_{i=0}^{n-1} \boldsymbol{x}^{(i)}. \tag{5.13}$$

We may also find it useful to use a weighted average, so as to give more emphasis to later estimates. An example is exponential decay weighting, used in the next section.

5.3.5 ADAM: Adaptive Moment Estimation

Among the state-of-the-art algorithms available, employing momentum, averaging and several other ideas, is the ADAM algorithm, invented by Kingma and Ba [20]. It is slightly more involved, and the reader is encouraged to read the aforementioned paper for an excellent presentation. The short story has the algorithm defined as follows:*

$$\begin{aligned}
\boldsymbol{m}^{(n+1)} &= \beta_1 \boldsymbol{m}^{(n)} + (1-\beta_1)\boldsymbol{\nabla} g(\boldsymbol{x}^{(n)}) && (5.14)\\
\boldsymbol{v}^{(n+1)} &= \beta_2 \boldsymbol{v}^{(n)} + (1-\beta_2)\boldsymbol{\nabla} g(\boldsymbol{x}^{(n)}) \circ \boldsymbol{\nabla} g(\boldsymbol{x}^{(n)}) && (5.15)\\
\hat{\boldsymbol{m}} &= \frac{\boldsymbol{m}^{(n+1)}}{1-\beta_1^{n+1}} && (5.16)\\
\hat{\boldsymbol{v}} &= \frac{\boldsymbol{v}^{(n+1)}}{1-\beta_2^{n+1}} && (5.17)\\
\Delta\boldsymbol{x}^{(n)} &= -\eta\, \hat{\boldsymbol{m}} \oslash \sqrt{\hat{\boldsymbol{v}}+\epsilon}, && (5.18)
\end{aligned}$$

where η is as before, β_1 and β_2 are decay rates for moment estimates, and ϵ is a small number used to avoid division by zero issues ($\oslash$ denotes Hadamard division, element-wise division). Kingma and Ba [20] provide default values for all the parameters, and in many cases these work excellently right out of the box.

$$\begin{aligned}
\eta &= 0.001 & \beta_1 &= 0.9\\
\epsilon &= 10^{-8} & \beta_2 &= 0.999
\end{aligned} \tag{5.19}$$

Especially important for us is how ADAM adapts *individual* learning rates per parameter. After all, given tens of thousands of parameters, what are the chances that all of them have the same optimal learning rate? ADAM allows us to account for different degrees of variability in the parameters, without the need to manually tune them all.

*All operations on vectors are element-wise. Exponentiation is expressed by superscripts *not* surrounded in parenthesis.

We will make extensive use of ADAM in our VMC calculations, as it proved superior to vanilla SGD in all cases. Section 10.5.2 includes a comparison on a couple of selected optimization problems.

5.4 Reinforcement Learning for VMC

As mentioned, VMC is a type of reinforcement learning problem. While we don't have access to a data set of configurations and their corresponding wave function amplitude, we can say something about the goodness of any particular wave function by measuring the expected ground state energy.

Even though reinforcement learning typically differs quite a lot from supervised learning in both model type, optimization strategy etc., they are not so by definition. When considering VMC in particular, we can treat it exactly the same way we would on a supervised problem, with one major difference: The cost function is *not* dependent on a data set. We use the same type of models, and the same strategies to minimize the cost function.

With this key change follows some differences. In particular, in some sense it is impossible to overfit a model. Overfitting happens when a model performs significantly better on the training data compared to data it has not seen before. We detect overfitting in supervised learning by withholding some of the training data and then evaluate the performance on this test data. This measures the models ability to generalize information, as opposed to memorizing exact values from the training data.

For reinforcement learning, however, what would the equivalent be? There are no data to withhold. If a VMC calculation suddenly results in an amazingly accurate result, then we simply have an amazingly accurate wave function. With this in mind, why not use an incredibly complex model? To some extent, this is the exact argument of this thesis, and we will use some highly complex models later on. There is, however, two limiting issues with this approach. First is the computational cost, which limits what we can feasibly do. Second, and more importantly, learning becomes harder with more complex models. Just because the model is theoretically capable of representing the wave function to great accuracy, it does not mean that we will ever be able to learn the correct parameters. With increasing complexity we increase the number of local minima in the parameter space. In here lies the challenge, attempting to balance the *exploration* of all possible parameter configurations with *exploitation* of promising minima.

5.5 Artificial Neural Networks

Artificial neural networks (ANNs, or just NNs) are a broad category of computational models, a representation of some general function $f : \mathbb{R}^m \to \mathbb{R}^n$.* Their name and origin stems from an attempt to model how neurons in animal brains react to inputs, and how they adapt to learn new skills. As animals are quite effective at learning based on the inputs from their surroundings, it seemed promising to emulate this process when attempting to teach a computer new tricks. ANNs have since been the target of extensive research, and has branched out in numerous variations that have little to no analog in animal brains.

While the original idea to use ANNs first appeared around the middle of the 20th century, they where initially limited, in large part due to the lacking computational capacity of that time, and saw little use for a long time. As computing power increased exponentially, most state-of-the-art machine learning models today are some variation on a neural network. This includes tasks such as natural language processing [21], image classification and segmentation [22], and playing games such as Go [23], Chess [24] and Starcraft II [25].

The following sections are devoted to presenting the type of artificial neural networks that we will employ in this work, including mathematical underpinnings, architecture choices and training strategies.

5.5.1 Nodes and Connections

First things first; What is a neural network? A neural network† is a way to express a certain type of complex computation. We model a computation as a sequence of *layers*, with each layer consisting of *nodes*, and some connections between the nodes in each layer. By convention, we call the first layer the *input* layer, the last layer the *output* layer, and layers in between are generally referred to as *hidden* layers. Figure 5.2 shows a graphical depiction of this general structure, exemplified for a certain number of layers/nodes.

Each node can hold a scalar value, and this value is propagated forward thorough the connections. Each connection has an associated weight (scalar) which is multiplied to the node's value before it is passed along. Nodes with multiple incoming connections take the sum as its value. Finally we might also include bias nodes, which are nodes with a fixed value. These serve the

*While there is nothing inherent about ANNs that limit us to only real values, we will not use complex valued networks in this work and therefore skip to mental burden of thinking about the complex numbers. Furthermore, complex values can simply be modeled by two real valued outputs if need be.

†Specifically feed forward neural networks. There are some types of exotic networks that does not fit well into this description, but they are beyond the scope of this discussion.

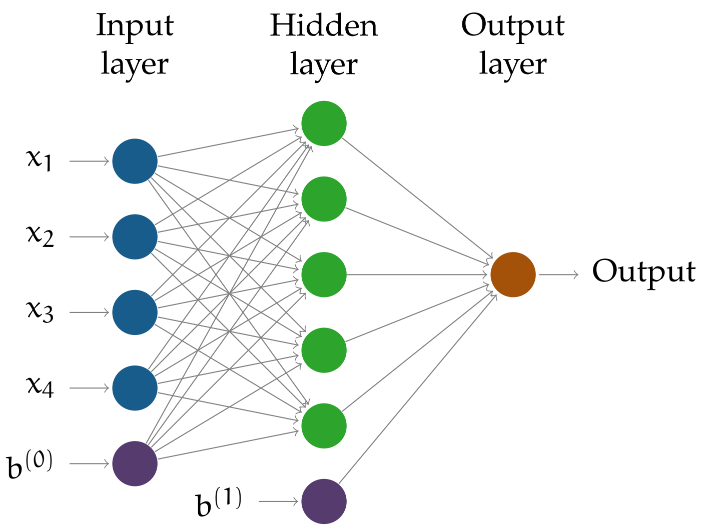


Figure 5.2: Illustration of a general feed-forward neural network architecture, here shown with four inputs nodes, one hidden layer of 5 nodes and a single output node.
Source [1, `writing/illustrations/neural-network.tex`]

same role as the intercept in a linear model, and enable us to – well, bias the output in some direction. If we initialize the input layer with some values, we can think of the network as a computation flow graph – visually representing the way information is manipulated as it moves towards a final output.

Definition

Let a_i^l be the value of the i'th node in the l'th layer, with $a_i^{(0)} = x_i$ being the inputs to the network. Each layer l in the network consists of a weight matrix $\boldsymbol{W}^{(l)}$, such that W_{ij}^l is the connection between $a_j^{(l-1)}$ and $a_i^{(l)}$. We also have a set of biases $b_i^{(l)}$ for each $a_i^{(l)}$, and finally a set of activation functions $\sigma^{(l)} : \mathbb{R} \to \mathbb{R}$ (to be explained).* All this comes together to define the network output $f(\boldsymbol{x})$ as follows:†‡

*In Fig. 5.2 the bias is illustrated as a single node with multiple connections, while in the mathematical notation we express the bias as a vector of values. These two are equivalent, and the figure is drawn as is for clarity. In all equations and code, bias is represented by as vector.

†Repeated indices within a term imply a summation. Note however that this is not implied for superscripts. That is, $A_{ij}^{(l)} B_{jk}^{(l)} := \sum_j A_{ij}^{(l)} B_{jk}^{(l)}$

‡While the example diagram only has a single output, there is nothing stopping us from having multidimensional outputs.

$$f(\mathbf{x}) = \mathbf{a}^{(L)} \tag{5.20}$$
$$a_k^{(l)} = \sigma^{(l)}(z_k^{(l)}) \tag{5.21}$$
$$z_k^{(l)} = W_{ki}^{(l)} a_i^{(l-1)} + b_k^{(l)} \tag{5.22}$$

This recursively defined result is calculated by starting at $l = 0$ with $\mathbf{a}^{(0)} := \mathbf{x}$ and calculating $\mathbf{a}^{(l+1)}$ iteratively. The process of doing this is sometimes referred to as a *forward propagation*.

5.5.2 Activation Functions

Say we set all $\sigma^{(l)}$ to the identity transformation (i.e. no transformation). Then the above definition would be reduced to the following:

$$f(\mathbf{x}) = \mathbf{z}^{(L)} \quad \text{with} \quad z_k^{(l)} = W_{ki}^{(l)} z_i^{(l-1)} + b_k^{(l)} \tag{5.23}$$

If you have had your morning coffee you might already see the issue here. This is simply a series of affine transformations, and as such could be replaced with one single, compound transformation: $\mathbf{z}^L = \mathbf{W}\mathbf{x} + \mathbf{b}$. Turns out, without an activation function, no matter how many layers we place in between the input and output, we have just defined a convoluted way of performing a simple affine transformation.

That is why we pump the value of each node through a *non-linear* function, $\sigma^{(l)} : \mathbb{R} \to \mathbb{R}$. The functions $\sigma^{(l)}$ can really be any function we choose, and what to pick largely depends on what computation we are trying to model. Some often used options, with varying input/output domains, are listed in Table 5.1, and their plots are shown in Fig. 5.3.

Which Activation Functions to Choose

While there is a great deal to possibly discuss about the pros and cons of various activation functions, a deep dive into this is beyond the scope of this thesis. In practice, choosing an activation function comes down to trial and error. Much of the neural network theory we have today is really more a set of best practices and experiences documented by others. Seldom do we have a rigorous backing for the choices we make, and most often a simple "this worked best" is as good as we get.*

*This might be slight exaggeration in some cases. But consider the state-of-the-art, massive networks of today - understanding every aspect of their behavior seems a daunting, if at all possible task. Sometimes the reasonable thing is simply to make some guesses and see what happens.

Table 5.1: Selection of activation functions. See Fig. 5.3 for the corresponding plots. $a \in \mathbb{R}$ is a (hyper)parameter tuning the behavior of some of the activation functions.

Activation Function	Definition
Sigmoid	$\sigma(x) = \frac{1}{1+\exp(-x)}$
Hyperbolic Tangent	$\sigma(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
Exponential	$\sigma(x) = \exp(x)$
Rectified **L**inear **U**nit (ReLU)	$\sigma(x) = \max(0, x)$
Leaky ReLU (LReLU)	$\sigma(x) = \max(ax, x) \quad \text{where} \quad 0 < a < 1$
Exponential **L**inear **U**nit (ELU)	$\sigma(x) = \begin{cases} x & \text{where} \quad x \geqslant 0 \\ a(\exp(x) - 1) & \text{otherwise} \end{cases}$

That said there are some things we *can* consider, especially for the output layer. If we want to interpret the output as a probability, something like the Sigmoid function could be useful because it outputs numbers in $[0, 1]$. Similarly, if we want to do regression, perhaps the output layer should have the identity function (i.e. no activation) so that the full spectrum $(-\infty, \infty)$ is possible. Finally, in our case, wave functions tend to involve exponentials and so we shall make heavy use of the exponential function as the output transformation on our networks.

5.5.3 Computational Flexibility

Part of the reason why neural networks are such a great tool is their flexibility. By simply changing the design of the network, or changing the type of activation function in a layer, we can adapt and change the model as we need. Even two identical networks can be tuned to model completely different data. Neural networks merely define a loose framework for how layers and nodes come together, and free us up to experiment.

The expressiveness of neural networks turns out to be just about as powerful as anything could be. That is because *any* (reasonable) function can be arbitrarily well approximated by a neural network with a single hidden layer [26]. With just a single hidden layer it turns out we might need exponentially many nodes, but if allowed to be deeper (more layers), we can have reasonable number of nodes as well and still achieve arbitrarily good results [27]. This result is known as the Universal Approximation Theorem [26, 27]. A formal description for this is not reproduced here, as the main point is to stress the expressive power of neural networks.

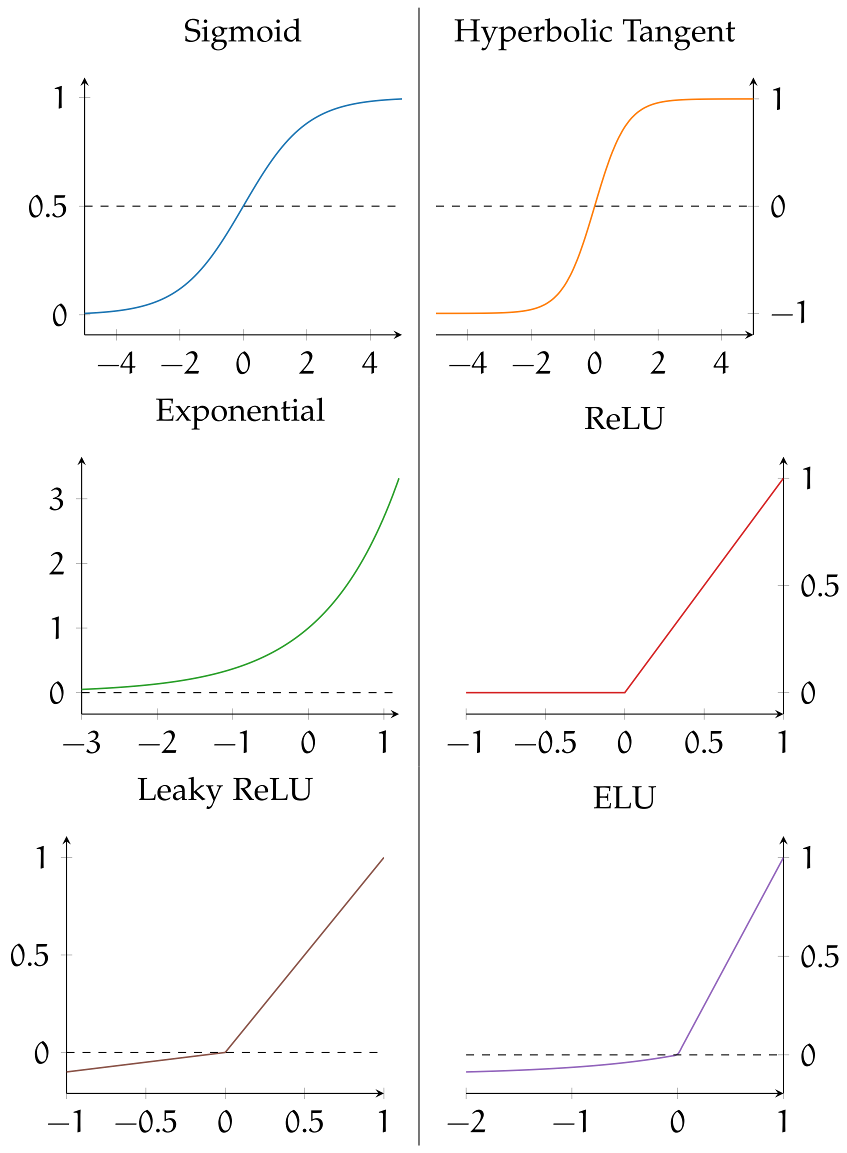


Figure 5.3: Selection of activation functions, with their plots. See Table 5.1 for their definitions. Note that the plots for LReLU and EReLU involve a parameter which has been arbitrarily set to 0.1 in the plots, and that the actual behavior of the function might be different for other choices.
Source [1, `writing/illustrations/activation.tex`]

5.5.4 Backpropagation

Now suitably enthusiastic about the power of neural networks, we must put our feet back on the ground for a bit and do some working out. After all, defining a complex model does not help us much if we cannot efficiently train it. With all these layers and connections, a legitimate concerns is how we can determine how to optimize the weights and biases involved.

We will be employing the gradient decent strategy from Section 5.3, which requires us to compute the gradient of a cost function, $\boldsymbol{\nabla}_{\boldsymbol{\alpha}}\mathcal{C}(f(\boldsymbol{x}))$, w.r.t. all model parameters $\boldsymbol{\alpha}$. In our case, the parameters are all the weights, $\boldsymbol{W}^{(l)}$, and biases $\mathbf{b}^{(l)}$.

While the following derivation might appear to have a daunting amount of indices and partial derivatives, it is really just a chain rule bonanza, applying the same trick over and over until the problem gives up and yields a neat solution.

Starting with the derivatives for the biases of the final layer:*

$$\frac{\partial \mathcal{C}}{\partial b_k^{(L)}} = \frac{\partial \mathcal{C}}{\partial z_i^{(L)}}\frac{\partial z_i^{(L)}}{\partial b_k^{(L)}} = \frac{\partial \mathcal{C}}{\partial z_i^{(L)}}\frac{\partial}{\partial b_k^{(L)}}\left(W_{ij}^{(L)}a_j^{(L-1)} + b_i^{(L)}\right) = \frac{\partial \mathcal{C}}{\partial z_k^{(L)}} := \delta_k^{(L)}, \quad (5.24)$$

where

$$\delta_k^{(L)} := \frac{\partial \mathcal{C}}{\partial z_k^{(L)}} = \frac{\partial \mathcal{C}}{\partial a_i^{(L)}}\frac{\partial a_i^{(L)}}{\partial z_k^{(L)}} = \frac{\partial \mathcal{C}}{\partial a_i^{(L)}}\frac{\partial \sigma^{(L)}(z_i^{(L)})}{\partial z_k^{(L)}} := \frac{\partial \mathcal{C}}{\partial a_k^{(L)}}\dot{\sigma}^{(L)}(z_k^{(L)}). \quad (5.25)$$

We can easily calculate this last expression as the product of the derivative of the cost function w.r.t. the network output (which should be doable for any sensible cost function), and the derivative of the output layer activation function evaluated at $z_k^{(L)}$ (which again is easy for any reasonable choice of $\sigma^{(L)}$).

For the weights of the last layer we get the following:

$$\frac{\partial \mathcal{C}}{\partial W_{kl}^{(L)}} = \frac{\partial \mathcal{C}}{\partial z_i^{(L)}}\frac{\partial z_i^{(L)}}{\partial W_{kl}^{(L)}} = \delta_i^{(L)}\frac{\partial}{\partial W_{kl}^{(L)}}\left(W_{im}a_m^{(L-1)} + b_i^{(L)}\right) = \delta_k^{(L)}a_l^{(L-1)}. \quad (5.26)$$

This way we can express all the derivatives we need through $\boldsymbol{a}^{(i)}$ and $\boldsymbol{\delta}^{(i)}$. We already know how to compute $\boldsymbol{a}^{(i)}$ for all layers, so we just need $\boldsymbol{\delta}^{(i)}$ for $i < L$. We get this, of course, using the chain rule some more:†

*For clarity: Repeated indices within a factor (still) imply a sum over said indices, but this is *not* implied for superscripts in parenthesis.

†While this derivation "just" entails the chain rule and some bookkeeping, exactly how to apply the chain rule is not that obvious at all times. It is "easy" once presented, but not necessarily trivial to reproduce from scratch.

$$\delta_k^{(l)} := \frac{\partial \mathcal{C}}{\partial z_k^{(l)}} \tag{5.27}$$

$$= \frac{\partial \mathcal{C}}{\partial z_i^{(l+1)}} \frac{\partial z_i^{(l+1)}}{\partial z_k^{(l)}} \tag{5.28}$$

$$= \delta_i^{(l+1)} \frac{\partial z_i^{(l+1)}}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_k^{(l)}} \tag{5.29}$$

$$= \delta_i^{(l+1)} \frac{\partial}{\partial a_j^{(l)}} \left(W_{im}^{(l+1)} a_m^{(l)} + b_i^{(l+1)} \right) \frac{\partial \sigma^{(l)}(z_j^{(l)})}{\partial z_k^{(l)}} \tag{5.30}$$

$$= \delta_i^{(l+1)} W_{ik}^{(l+1)} \dot{\sigma}^{(l)}(z_k^{(l)}). \tag{5.31}$$

These results can be summarized more succinctly in matrix form. We have:

$$\boldsymbol{\delta}^{(l)} = \begin{cases} \nabla_{\boldsymbol{a}^L} \mathcal{C} \circ \dot{\sigma}^{(L)}(\boldsymbol{z}^{(L)}) & \text{for} \quad l = L \\ \left[\left(\boldsymbol{W}^{(l+1)} \right)^T \boldsymbol{\delta}^{(l+1)} \right] \circ \dot{\sigma}^{(l)}(\boldsymbol{z}^{(l)}) & \text{for} \quad l < L \end{cases} \tag{5.32}$$

and the final gradients:

$$\nabla_{\boldsymbol{b}^{(l)}} \mathcal{C} = \boldsymbol{\delta}^{(l)} \quad \text{and} \quad \nabla_{\boldsymbol{W}^{(l)}} \mathcal{C} = \boldsymbol{\delta}^{(l)} \boldsymbol{a}^{(l-1)}. \tag{5.33}$$

The process of calculating these gradients involve starting at the output layer, working our way backwards, and is therefore referred to as *backpropagation*, with Eqs. (5.32) and (5.33) referred to as the *backpropagation algorithm*.

With this final piece of the puzzle, we know everything we need from ML for the purposes of this thesis. The last remaining issue is how to introduce these models into the framework of VMC, to which we dedicate the next chapter.

Chapter 6

Merging Variational Monte Carlo and Machine Learning

The purpose of variational Monte Carlo (VMC) is to find a wave function which minimizes the expected energy of the system. Stated like this, VMC clearly fits under the machine learning umbrella as a general function optimization problem. Still, it is not a standard supervised learning problem such as linear regression, because we don't have a data set of expected outputs for every input. Additionally there are some extra challenges introduced from the quantum problems themselves, such as symmetry requirements and cusp conditions. I believe these extra challenges are part of the reason why VMC optimizations have traditionally been done independently from the rest of the machine learning (ML) research advancements seen in the last few decades.

In the last few years, more and more research seems to go into applying various techniques from the more general field of ML into the natural sciences, including quantum mechanics and VMC. A notable example is Carleo and Troyer [28], who demonstrated that an restricted Boltzmann machine (RBM), a particular machine learning model, was capable of representing the wave function for some notable systems to great effect.

Despite of these recent advancements, we argue that there still exists a great deal of hesitancy towards using general models from machine learning for VMC. This chapter aims to enable the use of *any* arbitrary machine learning model, and in so doing open the door for many new applications and advancements in the field of VMC.

6.1 Discriminative vs. Generative Models

The RBM used by Carleo and Troyer [28] was chosen (partly) because it is a *generative* model. A distinction is often made between two general types of

ML models: *discriminative* and *generative*.[*] To understand their differences, imagine we have a data set $\mathcal{D} = \{x_i, y_i\}_{i=1}^{n}$ of n inputs x_i and outputs y_i. In general we want to uncover the relationship between x and y, and we can imagine two different ways of doing this. We could learn:

$p(y|x)$
The conditional probability density function (PDF) of y given x.

$p(x, y)$
The joint PDF of inputs and outputs.

The most familiar option is the first one. Linear regression is an example of such a case where we have an input and want to get the corresponding output. Additionally, instead of considering the PDF $p(y|x)$ we most often just define the output to be the output with the highest probability, $y = \operatorname{argmax}_y p(Y = y|X = x)$.

Less familiar for most is the second case. Here we model both inputs and outputs at the same time. Using Bayes rule we can get both $p(y|x)$ and $p(x|y)$ from the joint distribution. This enables us to not only make predictions on outputs, but also to generate inputs. This means if we train a generative model on images of cats and dogs, it could learn to not only classify the images correctly, but also to generate new images.

Benefits of a Generative Model

So what does this have to do with VMC? Our ultimate goal in VMC is to model a probability amplitude for system configurations $\mathbf{X} \to \psi(\mathbf{X})$. We want this, in part, so that we can sample configurations from it. So wouldn't it be nice if we could have the model generate the configurations directly, instead of having to go through the whole machinery of the Metropolis-Hastings algorithm and its associated downsides?

When Carleo and Troyer [28] used an RBM for their wave function, this opened the door for new ways of sampling (specifically Gibbs sampling), playing on the generative nature of the RBM. Such possibilities can improve both computational performance and accuracy of estimates obtained. It also provides a nice conceptual link between the model and the generation of configurations.

Limitations of Generative Models

While we can in principle obtain the same conditional probabilities of discriminative models from the joint distribution, it turns out that learning the

[*]See for instance Ng and Jordan [29] for a more in-depth discussion of the topic.

joint distribution can be a harder task [29]. This means that while the result of a generative model has the potential to be more useful, the increased simplicity of discriminative models can lead to more accurate results. This is the classic dilemma of whether to do one thing well, or two things decently. Ideally we would do both, but sometimes the trade off in accuracy is significant.

Second, and arguably more importantly, limiting ourselves to generative models is – well, limiting. A vast pool of potential models are discriminative, including the neural networks from Section 5.5. Given that we have the Metropolis-Hastings algorithm which works excellently in most situations, it seems unnecessary to completely disregard so many options. Part of the intent of this entire thesis is to illustrate how discriminative models can be used effectively for VMC purposes.

6.2 Arbitrary Models as Trial Wave Functions

Suppose we propose some arbitrary wave function $\psi_\alpha(\mathbf{X})$, parameterized by an arbitrary number of parameters $\boldsymbol{\alpha}$. Subject to the requirements on wave functions from Section 2.4, there are a few things we need from this function in order to successfully run a VMC optimization. The complete list of required operations is as follows:[*]

Evaluation:

Given a configuration $\mathbf{X}$, produce a scalar value $\psi_\alpha(\mathbf{X})$

Gradient w.r.t. parameters:

In order to compute the cost function gradient, we need the following quantity:

$$\frac{1}{\psi_\alpha(\mathbf{X})}\nabla_\alpha\psi_\alpha(\mathbf{X}) = \nabla_\alpha \ln\psi_\alpha(\mathbf{X}). \tag{6.1}$$

Gradient w.r.t. inputs: (optional)

For use with importance sampling (IS), we need to compute the drift force associated with the wave function (for particle k):

$$\frac{2}{\psi_\alpha(\mathbf{X})}\nabla_k\psi_\alpha(\mathbf{X}) = 2\nabla_k \ln\psi_\alpha(\mathbf{X}). \tag{6.2}$$

Note that this is only strictly needed when IS is used, and can be omitted.

[*]These operations are rooted in the assumption that we are working with coordinates as the system configuration specification, and where the kinetic energy is part of the energy. We acknowledge that there are a whole class of Hamiltonians for which this is not suitable. In these cases, the required operations will likely need some changes. Nevertheless, the overarching procedure and techniques discussed should be agnostic to such changes.

Laplacian w.r.t. inputs:

For use with any Hamiltonian which includes kinetic energy, we need to compute the Laplacian of the wave function with respect to the configuration $\mathbf{X}$.

$$\sum_i \frac{1}{\psi_\alpha(\mathbf{X})}\nabla_i^2\psi_\alpha(\mathbf{X}). \tag{6.3}$$

Any function that supports these four operations can be used as a trial wave function. We should of course take care to choose functions that satisfy the standard requirements from Section 2.4, but also to choose a model which is likely to be a good candidate.

6.2.1 Artificial Neural Networks as Trial Wave Functions

artificial neural networks (ANNs) give us an incredibly flexible way to define computationally expressive models, and to easily adapt them by changing their architecture. For a long time, (human) theorists have hand crafted Jastrow factors, resulting in (good) trial wave functions rooted in the physics of the system. The problem is that hand crafting only goes so far, and we reach the limits of what we can construct and reason about analytically. Enter now a way to experiment and play with complex models, in a flexible manner. Using ANNs could enable us to prototype wave functions more rapidly, as well as possibly increase the accuracy of the resulting simulations.

Convinced this is a great idea, we have some more working out to do. While evaluating the output of the network is easy enough (see last chapter), we need to derive the general expression for the other quantities we need.

In the following, let $\psi(\mathbf{x})$ be an artificial neural network which has the quantum numbers of every particle as its inputs. $\mathbf{x}$ is the concatenation of all $\{\mathbf{x}_i\}_{i=1}^N$ for N particles. For our case, that means each particle coordinate is one input to the network, and the output of the network is the value of wave function.

Gradient w.r.t. Parameters

Back in Section 5.5.4 we derived the formulas for the gradient of a general cost function, with respect to the weights and biases of any network. In the case of VMC, we need the quantity given in Eq. (6.1). We can get this trivially through the backpropagation algorithm by defining the cost function to be the output of the network:

$$\mathcal{C}(\psi(\mathbf{x})) = \psi(\mathbf{x}). \tag{6.4}$$

Computing the "cost function" gradient using backpropagation, we simply divide by $\psi(\mathbf{x})$ to get the quantity in Eq. (6.1):

$$\frac{1}{\psi(\mathbf{X})}\boldsymbol{\nabla}_\alpha\psi(\mathbf{X}) = \frac{1}{\psi(\mathbf{x})}[\boldsymbol{\nabla}_\alpha\mathcal{C}(f(\mathbf{x}))]. \tag{6.5}$$

First Order Partial Derivative w.r.t. Inputs

Equation (6.2) requires us to compute the first order derivatives of the networks output w.r.t. its individual inputs. With reference to the notation used in Section 5.5, we get:

$$\begin{aligned}
\frac{\partial a_k^{(l)}}{\partial x_j} &= \frac{\partial \sigma^{(l)}\left(z_k^{(l)}\right)}{\partial z_i^{(l)}}\frac{\partial z_i^{(l)}}{\partial x_j} && (6.6)\\
&= \dot\sigma^{(l)}\left(z_k^{(l)}\right)\frac{\partial z_k^{(l)}}{\partial x_j} && (6.7)\\
&= \dot\sigma^{(l)}\left(z_k^{(l)}\right)\frac{\partial}{\partial x_j}\left(W_{ki}^{(l)}a_i^{(l-1)} + b_k^{(l)}\right) && (6.8)\\
&= \dot\sigma^{(l)}\left(z_k^{(l)}\right)\left(W_{ki}^{(l)}\frac{\partial a_i^{(l-1)}}{\partial x_j}\right), && (6.9)\\
\frac{\partial a_k^{(0)}}{\partial x_j} &= \frac{\partial x_k}{\partial x_j} = \delta_{kj}. && (6.10)
\end{aligned}$$

Once again we divide by ψ (and multiply by 2) to get the values needed in Eq. (6.2).

Second Order Partial Derivatives w.r.t. Inputs

For Eq. (6.3) we also need the second order derivatives. Continuing from Eq. (6.9) we get:

$$\frac{\partial^2 a_k^{(l)}}{\partial x_j^2} = \dot{\sigma}^{(l)}\left(z_k^{(l)}\right) W_{ki}^{(l)} \frac{\partial^2 a_i^{(l-1)}}{\partial x_j^2} + \frac{\partial \dot{\sigma}^{(l)}\left(z_k^{(l)}\right)}{\partial x_j} W_{ki}^{(l)} \frac{\partial a_i^{(l-1)}}{\partial x_j} \tag{6.11}$$

$$= \dot{\sigma}^{(l)}\left(z_k^{(l)}\right) W_{ki}^{(l)} \frac{\partial^2 a_i^{(l-1)}}{\partial x_j^2} + \ddot{\sigma}^{(l)}\left(z_k^{(l)}\right) \frac{\partial z_k^{(l)}}{\partial x_j} W_{ki}^{(l)} \frac{\partial a_i^{(l-1)}}{\partial x_j} \tag{6.12}$$

$$= \dot{\sigma}^{(l)}\left(z_k^{(l)}\right) W_{ki}^{(l)} \frac{\partial^2 a_i^{(l-1)}}{\partial x_j^2} + \ddot{\sigma}^{(l)}\left(z_k^{(l)}\right) \left(W_{ki}^{(l)} \frac{\partial a_i^{(l-1)}}{\partial x_j}\right)^2, \tag{6.13}$$

$$\frac{\partial^2 a_k^{(0)}}{\partial x_j^2} = \frac{\partial^2 x_k}{\partial x_j^2} = 0. \tag{6.14}$$

Dividing by ψ yields the desired result.

6.2.2 Imposing Symmetry Requirements

We know from Section 2.4 that we have certain requirements which need to be met in order for the wave function to be sensible. Assuming we pick a model that is suitably smooth and continuous, the most challenging requirement is that of particle exchange symmetry. As discussed in Section 3.3.1, we typically want to design our wave function candidates to be symmetric, and add a Slater determinant for fermions if need be. The problem is that arbitrary functions are not likely to be symmetric to the interchange of its inputs. Neural networks are intrinsically not symmetric.

We present some different strategies of imposing such a symmetry:

Learn the Symmetry

The simplest strategy is perhaps the most appealing, as we simply let the model realize on its own that it is best to be symmetric. After all we know the ideal wave function is, so an ideally trained model *should* be able to learn this.

In practice this might be too much to ask, and the model might settle on a non-symmetric local minimum. Additionally, one might argue that the symmetry requirement *must* be satisfied at all times during training for the results to be physical. This will depend on the particular system we investigate.

Pool the Output

We can draw inspiration from the world of convolutional neural networks (CNNs), and obtain the symmetry by way of a pooling operation. Let $F(\mathbf{X})$ be the final model output and let $f(\mathbf{X})$ be the originally proposed model.

The simplest version is to let F be the sum of f applied to every permutation of $\mathbf{X}$. We could use the product instead of the sum, or any other commutative binary operator. While this would fix the symmetry, it would have a complexity of $\mathcal{O}(N!)$ for N particles.

A computationally better idea is to let f be a function of only two particles, $f(\mathbf{x}_i, \mathbf{x}_j)$, and then let F be the sum of all possible permutations of two particles out of the N available:

$$F(\mathbf{X}) = \sum_{i=1}^{N} \sum_{j \neq i}^{N} f(\mathbf{x}_i, \mathbf{x}_j) \quad \text{(Sum Pooled)} \quad . \tag{6.15}$$

This has complexity $\mathcal{O}(N^2)$, but limits us to learn two-body interactions. Three-body interactions could be included with another sum, but would increase the complexity accordingly. For problems which require higher order correlations this approach might not be suitable.

Sort the Inputs

Perhaps the most computationally efficient way to impose the symmetry is to simply transform the inputs in a symmetric manner before passing them on to the model. An intuitive choice is to sort the inputs in ascending order of "size". Importantly we should sort by particle, and not sort all coordinates independently. One way to do this is to sort the particles by their absolute distance from the origin. This approach has the benefit of a much improved complexity of $\mathcal{O}(N \log N)$. Additionally, we might imagine that ordering the particles by some extrinsic metric might help the model extract useful information more easily.*

*Consider a Hamiltonian that depends strongly on the closest/furthest distance of any particle to the origin. Any model will quickly be able to pick up on the fact that the first/last input is highly correlated with the energy. If these particles could be at any input, understanding this connection could be more challenging.

Part II

Implementation

Chapter 7

Parallelization

Variational Monte Carlo (VMC) calculations take a lot of computing power. It is easy to set up a calculation that can take lifetimes to complete. Some systems are simply to large for us to ever be able to work with, but at least we want to go as far as possible. To do this we need to consider all available resources and attempt to utilize them as efficiently as possible. This chapter is dedicated to the various ways of speeding up calculations by means of parallelization.

Firstly we should make a few important notes. Parallelization is an effective tool *only* if we have more than one processing unit available. On a single-core machine you may have the experience of multiple things happening at once, but this is simply due to the single core switching back and fourth faster than we can observe. This is *concurrency*, or multitasking as it is more often called when applied to humans. As we know, you are never twice as productive when you attempt to do two things at once - often the opposite is closer to the truth. The same is true for computers. So when we now discuss ways of speeding up calculations by means of *parallelization*, we are at all times limited by the number of processing units available.

Lastly, we are going to make a distinction between two types of parallelization. For any task that is not trivially parallel (i.e. two completely independent tasks), we expect to need some sort of *synchronization*, or communication between the workers. We have two options for how to achieve this:

Shared memory: Workers communicate through a shared section of memory, by reading/writing to it

Distributed memory: Workers have independent memory sections, and communicate explicitly through sending/receiving messages (chunks of data)

7.1 Distributed Memory

The main bottleneck in VMC computations is the evaluation of expectation values, such as Eq. (3.24) and Eq. (3.32). They entail computing a large number of individual terms and taking the average. The important thing about this computation is that addition is commutative, and so the order of the terms (obviously) does not matter to the result. As such we could run p parallel processes, each of which computes n/p terms (for n total terms) before aggregating the partial sums together and taking the mean.

This is a so called embarrassingly parallel problem, in the sense that it is trivial to divide the workload up into smaller tasks. It has as close to linear speedup potential[*] as one can hope for, and is an ideal candidate for parallelization through distributed memory multi-processing. We want to run p independent instances of the algorithm, and pause momentarily to communicate the results. In the simple example above, we just send the final partial sum acquired by each process. The cost of sending the message is negligible compared to the main loop.

The main benefit of distributed memory is that it is easy to scale up to any arbitrary number of machines, as long as they are connected in some way. These machines could in principle be placed all around the world and still work well for our purposes (some extra latency would be introduced by the distance).

7.1.1 Implementation API[†]: Message Passing Interface

For the actual communication between distributed memory processes, we have the standardized message passing interface (MPI). This is typically implemented as a set of C/C++ library functions with which we define the communication we want, along with a special compiler (`mpicc`/`mpic++`) and a utility to start multiple instances of a program (`mpiexec`[‡]).

7.1.2 Example: Expectation Value of the Local Energy

The following example is taken from the source code of our code library QFLOW (Chapter 9), and illustrates a simply parallelized task: Computing the expected local energy. While some details are specific to QFLOW's implementation, the general concept should be clear.

[*]Linear speedup implies that if a task takes time T to finish with one process, then it takes T/n time with n processes.

[†]Application programming interface (API).

[‡]Quite common is the use of `mpirun`, however, this is not part of the MPI standard. The more stable and portable command is `mpiexec`.

Listing 1 Example excerpt from the source code of QFLOW, showing an MPI parallelized computation of the expected local energy.
Source [1, qflow/hamiltonians/hamiltonian.cpp]

```
double Hamiltonian::local_energy(Sampler& sampler,
    Wavefunction& psi, long samples) const
{
     const int  n_procs = mpiutil::proc_count();
     const int  rank    = mpiutil::get_rank();
     const long samples_per_proc      // n / p + remainder.
       = samples/n_procs + (rank<samples % n_procs ? 1 : 0);

     double E_L = 0;

     for (long i = 0; i < samples_per_proc; ++i)
         E_L += local_energy(sampler.next_configuration(),
             psi);

     double global_E_L;
     MPI_Allreduce(&E_L, &global_E_L, 1,
                   MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

     return global_E_L / samples;
}
```

To illustrate the speedup we obtain from using MPI parallelization, we have run Listing 1 on different number of cores. In particular, we computed the local energy of a simple harmonic oscillator system of one particle in three dimensions. Figure 7.1 shows the number of Monte Carlo (MC) iterations per second as a function of the number of cores used. As advertised, we see a speedup that follows closely the ideal linear line.

Still, we should emphasize that in order to obtain these results, we had to increase the total workload linearly with increasing number of central processing units (CPUs). If we don't, the task finishes so quickly that communication becomes a significant part of the total time. So in order to fully utilize the power of extra CPUs, we should make sure to structure the program such that the bulk of it is spent in the independent parts of the program. In practice that means to use a sufficiently large number of MC samples.

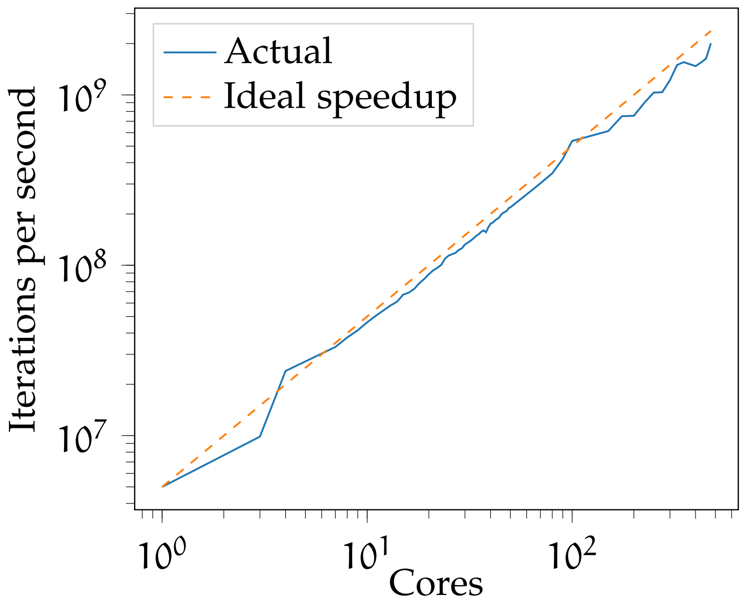


Figure 7.1: Iterations per second when running Listing 1 with varying number of processors. These results where obtained by running a test on the Abel computing cluster. Exact results will vary greatly depending on the availability of the cores, size of the workload and several other factors. Source [1, `writing/scripts/mpi-speed-test.py`]

7.2 Shared Memory

As mentioned, the other possible way to allow for communication between workers is through shared memory. This typically means running multiple threads within the same process memory space, so that each thread has access to the same memory. This is a faster type of communication compared with message passing, but comes at the cost of having to consider data races. A data race occurs when multiple workers try to access (with at least one write operation) the same location in memory at the same time. The result of such behavior is undefined and must be avoided.

7.2.1 Implementation API: OpenMP

Similarly to how distributed memory parallelization has the MPI standard, we have the OpenMP standard for describing shared memory parallelization. The use of this type of parallelization is typically made entirely through compiler instructions embedded in the code, with some library functions also available. The number of threads to use is decided at runtime, and can even change during the course of the program. Importantly there is only one process running the code, which lives for the duration of the program, and threads get created and destroyed throughout the course of the computation.

7.2.2 Example: Monte Carlo Estimation of π

The following example shows a task equivalent in nature to the one used in the last section, now parallelized using OpenMP.

Running Listing 2 with a varying number of threads results in Fig. 7.2 which shows how the speedup is about linear until we run out of processing units (16 cores on the machine used), after which it flattens out.

This showcases how shared memory has excellent potential for speedup, but also its major drawback: Scalability. It is considerably harder to scale a shared memory architecture to an arbitrary number of processors because the "shared" part forces the processing units to be in relative close proximity to each other.* For this reason, we have focused our parallelization efforts to MPI, because we anticipate the need for a large number of parallel jobs. To some extent, a happy marriage of the two is sometimes possible, but in general we expect that dedicating as many cores to MPI processes as possible should be the most beneficial strategy.

7.3 GPU Acceleration

The final form of hardware acceleration we will mention is one that is particularly central for machine learning (ML) in particular. While the two prior techniques boil down to having multiple CPUs work together, certain tasks

*You could in principle emulate shared memory for separate machines, but the emulation it self would have to use a message passing interface, which would defeat the purpose.

Listing 2 Example of parallel estimation of π using OpenMP. The example is complete and can be compiled as is.

```
#include <iostream>
#include <omp.h>
int main ()
{
    constexpr long steps = 1000000000;
    constexpr double step = 1.0 / (double) steps;

    int i, j;
    double x;
    double pi = 0;
#pragma omp parallel for reduction(+:pi) private(x)
    for (i=0; i < steps; i++)
    {
        x = (i+0.5)*step;
        pi += 4.0 / (1.0+x*x);
    }
    pi *= step;

    std::cout << "Pi = " << pi << std::endl;
}
```

are particularly well suited to acceleration by using a graphics processing unit (GPU). These are pieces of hardware typically designed to enable fast rendering of graphics for computers, hence the name. They have a highly parallel structure optimized for algorithms that need to process large blocks of data in parallel. A typical example that relates to ML is matrix-matrix multiplication. Say we compute $\mathbf{C} = \mathbf{AB}$, or more explicitly:

$$C_{kl} = A_{ki} B_{il}. \tag{7.1}$$

The order in which we do any of the above $\mathcal{O}(n^3)$ operations really doesn't matter, and so one can see how this can be an extremely parallel process. A high-end GPU will have a number of cores in the four digit range, facilitating speedups of several orders of magnitude compared to a single CPU.

The recent (last decade) advancements in GPU technology has been an integral part of the enormous amount of success in recent ML applications. Their use has facilitated building models far beyond the capacity of an ordinary CPU.

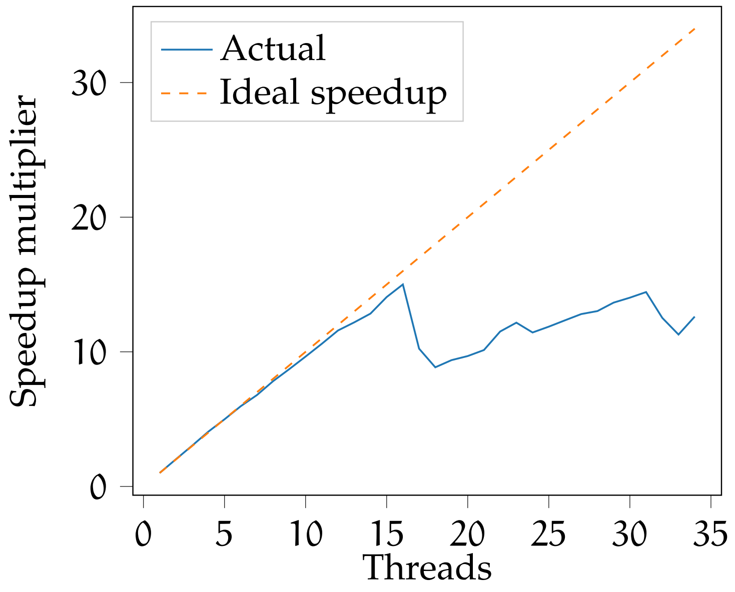


Figure 7.2: Achieved speedup when running Listing 2 with varying number of threads. These results where obtained by running a test on the Abel computing cluster, on a node with 16 cores. We see a clear speedup until we run out of cores, at which point using more threads hurts performance. Speedup is here defined as the ratio of iterations per second to iterations per second using one thread.
Source [1, `writing/scripts/openmp_test.py`]

7.3.1 Implementation API: CUDA

One popular way to program for a GPU is through CUDA, a programming interface developed by NVIDIA. Similarly to MPI programming, its use is (for C/C++) through a specialized compiler and some library functions. The typical flow is to implement the parallel computation (say matrix-matrix multiplication as above) in a dedicated CUDA function, and then call to this function from regular code running on the CPU.

Many ML frameworks such as TensorFlow and PyTorch have built in support for GPU, and offloading work to them is "trivial". Unfortunately, we where unable to use such frameworks in this work (mainly due to the reasons described in Chapter 8). Further, we have not actually implemented GPU support in the current codes, despite this having large potential benefits. This is simply a matter of time and realizing that it would require a considerable amount of effort to get right. With the introduction of neural networks as wave functions, we see no obvious reason why using GPUs would not cause significant speedups also in our application of ML. The main challenge will likely be to join GPUs into the existing MPI codes. We leave this as a potential point of future work.

Chapter 8

Automatic Differentiation

One of the central parts of the success of neural networks in machine learning is the backpropagation algorithm of Eqs. (5.32) and (5.33). The algorithm is able to describe how parameters should be updated in arbitrary networks by propagating the error estimates backwards through the network. While we could derive these equations relatively simply in the case of a pure feed-forward neural network such as the ones described here, there is a whole world of variants for which we would need to derive different equations, such as recurrent networks, residual networks, convolutional networks etc. Nevertheless, they all still rely on the basic premise of applying the chain rule.

One of the perhaps most appealing attributes of machine learning (ML) frameworks such as TensorFlow [30] is how we can design arbitrary networks *without* having to manually work out the associated derivatives. The way this is done is through automatic differentiation (AD), an algorithmic application of the chain rule that can compute derivatives of any function specified by a computer program.

This is possible by exploiting the fact that any computer program must be built up of elementary operations $(+, -, *, /)$ and perhaps some elementary functions $(\exp, \sin$ etc.). We know how to handle derivatives for all of these, and importantly we know how to combine them in arbitrary ways through the chain rule. This allows us to construct a computation for any function which will compute its derivative.

AD promises a general purpose implementation that can calculate derivatives exactly (within numerical precision) in a manner that uses at most a constant factor more operations than the original function. We can also generalize to higher order derivatives by repeated application of the algorithm. That is, viewing the derivative computation as a function in its own right we can apply the same trick to construct a computation for the second derivative.

8.1 Example

Let the function we wish to differentiate be the following:*

$$\begin{aligned} f(x_1, x_2) &= x_1 x_2 + \sin(x_1) && (8.1)\\ &= w_1 w_2 + \sin(w_1) && (8.2)\\ &= w_3 + w_4 && (8.3)\\ &= w_5 = y, && (8.4) \end{aligned}$$

where we have introduced the temporary variables w_i for later notational brevity. Assume that we are interested in $\partial y/\partial x_1$ and $\partial y/\partial x_2$. There are two main approaches in use: *forward* and *reverse* mode.

8.1.1 Forward Mode

We first decide on one independent variable to differentiate with respect to, say x_1. We then compute $\dot{w}_i = \partial w_i/\partial x_1$ in order of increasing i (forward through the computation).

When we step through the computation of $f(x_1, x_2)$, we can simultaneously compute the derivative we want. By normal rules of differentiation:

Value	Derivative (w.r.t. x_1)
$w_1 = x_1$	$\dot{w}_1 = 1$
$w_2 = x_2$	$\dot{w}_2 = 0$
$w_3 = w_1 w_2$	$\dot{w}_2 = \dot{w}_1 w_2 + w_1 \dot{w}_2$
$w_4 = \sin(w_1)$	$\dot{w}_4 = \cos(w_1)\dot{w}_1$
$w_5 = w_3 + w_4$	$\dot{w}_5 = \dot{w}_3 + \dot{w}_4$

Substituting in the values from top to bottom we see that we indeed get $\frac{\partial y}{\partial x_1} = \cos(x_1) + x_2$ as one would expect. We can now repeat the process once more for x_2. Evidently, we only need to change the initial values $\dot{w}_1$ and $\dot{w}_2$, while the rest remains the same. Because of this, these "trivial" first derivatives are sometimes referred to as *seed* values.

8.1.2 Reverse Mode

Instead of first fixing the independent variable, we instead fix the dependent variable which should be differentiated (only one in this case, y), and then

*This example is in large part similar to one found on the AD Wikipedia page, simply because I quite like this explanation.

compute its derivative w.r.t. the different sub expressions w_i. We define $\bar{w}_i = \partial y / \partial w_i$, and construct the following table of operations (backwards through the computation):

$$
\begin{aligned}
\bar{w}_5 &= \frac{\partial y}{\partial w_5} = 1 \\
\bar{w}_4 &= \frac{\partial y}{\partial w_4} = \bar{w}_5 \frac{\partial w_5}{\partial w_4} = \bar{w}_5 \\
\bar{w}_3 &= \frac{\partial y}{\partial w_3} = \bar{w}_5 \frac{\partial w_5}{\partial w_3} = \bar{w}_5 \\
\bar{w}_2 &= \frac{\partial y}{\partial w_2} = \bar{w}_3 \frac{\partial w_3}{\partial w_2} = \bar{w}_3 w_1 \\
\bar{w}_1 &= \frac{\partial y}{\partial w_1} = \bar{w}_4 \frac{\partial w_4}{\partial w_1} + \bar{w}_3 \frac{\partial w_3}{\partial w_1} = \bar{w}_4 \cos(w_1) + \bar{w}_3 w_2.
\end{aligned}
$$

Notice how reverse mode required only one pass though the algorithm in order to find the derivatives of both independent variables, compared to two passes for forward mode.

Keen readers might also recognize backpropagation as a special case of reverse mode AD.

8.2 Forward Mode vs. Reverse Mode

While the two described approaches both obtain the same result, one is generally preferable to the other depending on the structure of the computation. Let the computation be a function $f : \mathbb{R}^m \to \mathbb{R}^n$. Forward mode requires m passes (one per input), while reverse mode requires n passes (one per output). So in general, if $m \gg n$ we prefer reverse mode and when $m \ll n$ we prefer forward.

Because of this, reverse mode automatic differentiation is by far the most prevailing choice in ML frameworks. This is because models almost always have more inputs than outputs.

8.3 Automatic Differentiation for VMC

As variational Monte Carlo (VMC) also needs quite a lot of derivatives it is natural to wonder if we can have AD tools work to our advantage also in this area.

8.3.1 First Order Derivatives

To see how this could work, we should revisit the list of required operations for wave functions in Section 6.2. In the list we see the need for first order derivatives w.r.t. both the inputs to the wave function and its parameters. Both of these are straightforward to obtain in an efficient manner using reverse mode AD. To illustrate why, we can consider the *Jacobian* matrix, $\mathbf{J}$, which for a function $f : \mathbb{R}^m \to \mathbb{R}^n$ looks like:

$$\mathbf{J} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_m} \\ \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_m} \end{pmatrix}. \tag{8.5}$$

A forward mode sweep calculates a column of $\mathbf{J}$, while a reverse mode sweep calculates a row [31]. In our case, f is a wave function which has a scalar output, and $\mathbf{J}$ is really a row vector. Reverse mode is clearly superior in this case, requiring only a single pass.

8.3.2 Higher Orders

Returning to the list of operations, we still have the ominous Laplace operator, ∇^2. To see why this will be harder, we can consider the *Hessian* matrix, $\mathbf{H}$:

$$\mathbf{H} = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_m} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_M \partial x_1} & \frac{\partial^2 f}{\partial x_M \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_m^2} \end{pmatrix}. \tag{8.6}$$

We can compute these second order derivatives by applying AD twice. First we do a reverse sweep and obtain the Jacobian (the row vector). The act of computing the Jacobian is it self a function, $f : \mathbb{R}^m \to \mathbb{R}^m$, and as such we can apply AD to it an obtain the elements of $\mathbf{H}$.

Computing the required Laplace operator equates to computing the trace (sum of diagonal) of the Hessian matrix. Remember that a forward/reverse sweep calculates a column/row of the matrix. That means that in order to get the diagonal elements, we actually need m sweeps, regardless of what direction we go. Sadly, the complexity of this operation is so large that we have found it unfeasible to use for even a moderate number of inputs.

8.4 Why Automatic Differentiation Failed

Because AD is so central to most ML frameworks, like TensorFlow and PyTorch, a great deal of effort was put in to attempt to make a viable VMC implementation within these. That would give out-of-the-box functionality for neural networks, optimization strategies, GPU acceleration etc. Sadly, the inefficiency of the calculation of the Laplacian operator put a stop to such a marriage. The time spent by AD to compute ∇^2 was orders of magnitude worse than the manually coded analytically expressions, and scaled significantly worse with increasing number of particles. This severe deficiency of AD resulted in the unfortunate need to depart from the existing ML frameworks and implement everything manually (i.e. analytically expressions for derivatives). This decision was not made lightly, and only after a painstaking effort to somehow attempt to defy the above mathematical argument. Unfortunately, mathematics was right.

We sincerely hope that AD can somehow be efficiently incorporated in future works. Research into how AD can be efficiently applied to higher order derivatives is ongoing, see for instance Wang and Pothen [32]. If nothing else, we could potentially use AD for Hamiltonians which do not include the Laplacian operator.

Chapter 9

QFLOW: Software Package for VMC with Neural Networks

Everything discussed so far has culminated into the development of a self contained package available to the public under the permissive MIT license. All original results presented in the next part are produced with this package, and the source code to reproduce them are also presented in all cases.

This chapter is dedicated to the presentation of the package, explanations of design decisions and comparison with other tools. A more complete technical documentation is published online [1], and it is recommended to consult this resource for practical usage.

QFLOW is implemented as an object oriented Python package with bindings into a monolithic C++ backend library. This allows for easy setup of the computation and post processing of data can be done entirely from Python. Still, all the computationally intensive code runs entirely in C++, so there is no performance penalty in starting things from Python. This design allows us to take the best of both languages, and it makes for a much improved user experience.

9.1 Installation

Refer to [1] for up to date instructions. Here we only mention the required components for using QFLOW:

- Python 3.7 or greater
- A C++17 compliant compiler
- OpenMPI 3.1.3 or greater (or equivalent MPI implementations)

Every other dependency should be automatically resolved during installation.

9.2 Usage

The following example aims to showcase the basic usage pattern of QFLOW.

Say we want run a variational Monte Carlo (VMC) optimization of two electrons in two dimensions confined in a harmonic oscillator trap, i.e. the quantum dot system described in Section 2.6.1. The following complete code uses a Pade-Jastrow wave function and optimizes the variational parameters before evaluating the final ground state energy integral. All in a few readable lines of Python:

```python
import numpy as np
import matplotlib.pyplot as plt

from qflow.wavefunctions import SimpleGaussian, JastrowPade,
    WavefunctionProduct
from qflow.hamiltonians import CoulombHarmonicOscillator
from qflow.samplers import ImportanceSampler
from qflow.optimizers import SgdOptimizer
from qflow.training import train, EnergyCallback, ParameterCallback
from qflow.statistics import compute_statistics_for_series
from qflow.mpi import mpiprint, master_rank

P, D = 2, 2  # Particles, dimensions

# Define Hamiltonian:
H = CoulombHarmonicOscillator(omega_ho=1)

# Define trial wave function:
gaussian = SimpleGaussian(alpha=0.8)
jastrow = JastrowPade(alpha=1, beta=1)
psi = WavefunctionProduct(gaussian, jastrow)

# Set up sampling strategy:
sampler = ImportanceSampler(np.empty((P, D)), psi, step_size=0.1)

# Train wave function:
training_energies = EnergyCallback(samples=100000)
training_params = ParameterCallback()
train(
    psi,
    H,
    sampler,
    iters=150,      # Optimization steps.
    samples=1000,   # MC cycles per optimization step.
    gamma=0,        # Regularization parameter (disabled here).
    optimizer=SgdOptimizer(0.1),
    call_backs=(training_energies, training_params),
)

# With a trained model, time to evaluate!
energy = H.local_energy_array(sampler, psi, 2 ** 21)
stats = compute_statistics_for_series(energy, method="blocking")
mpiprint(stats, pretty=True)
```

```
OUTPUT:
    {'CI':   (3.0005518930024406, 3.000882266251301),
     'max':   3.0203576916357706,
     'mean':  3.0007170796268707,
     'min':   2.9820746671373106,
     'sem':   8.425552793020659e-05,
     'std':   0.005391695501040196,
     'var':   2.9070380375937085e-05}
```

The reported output is the full energy estimate produced after having optimized the variational parameters (the `mean` field), along with various related statistics about the energy estimate. Figure 9.1 shows how the energy estimate decreases as we tune the parameters, before eventually converging.

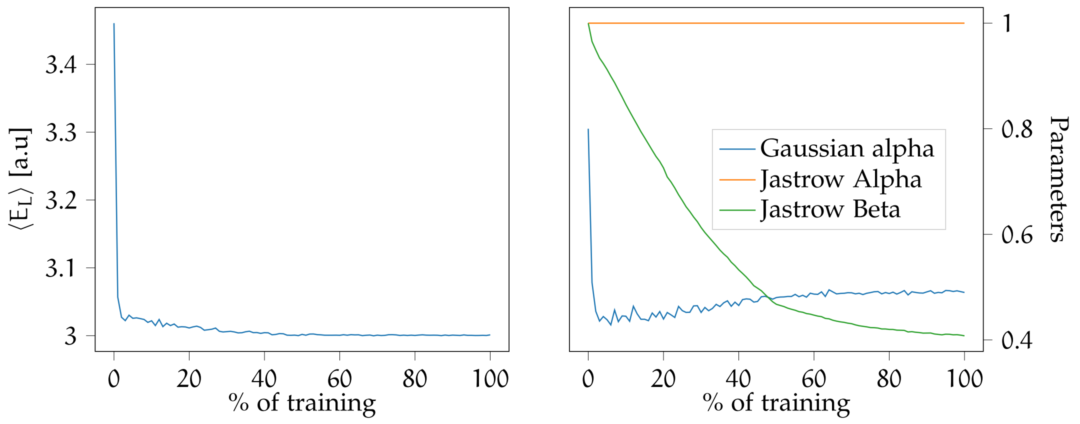


Figure 9.1: Optimization of variational parameters as a function of update iterations. The left plot shows the expected local energy and the right plot shows the corresponding variational parameters at each point during the training.
Source [1, `writing/scripts/quickstart.py`]

The script can be run under MPI (`mpiexec -n X python script.py`) with any number of processes, which results in almost completely linear speedup.

9.3 Structure

The package has been structured around four basic building blocks of a VMC calculation, each of which is represented by a base class:

- Hamiltonians
- Wave functions
- Samplers

- Optimizers

Due to this object oriented design approach, each part needs only consider how to interact with the base classes of the other three, as opposed to duplicating code per combination. As well as aiding development, this makes it easy the prototype various combinations of wave functions, sampling strategies etc., as well as trivial to exchange the Hamiltonian in question.

9.3.1 Hamiltonians

The `qflow.hamiltonians.Hamiltonian` class in QFLOW is responsible for defining the energy (kinetic, external potentials and interaction potentials) of the system. This class is queried for local energy evaluations/gradients as well other system related quantities of interest. Each particular Hamiltonian is implemented as a subclass of `qflow.hamiltonians.Hamiltonian`, and at a minimum it needs to define the external and internal potentials (can be set equal to zero).

9.3.2 Wave functions

In a similar way, the `qflow.wavefunctions.Wavefunction` class in QFLOW defines the basic operations required for arbitrary wave functions, and each particular trial wave function is implemented as a subclass to this.

Neural Networks

The neural networks are used to define wave functions, and as such their relevant definitions are structured under the `qflow.wavefunction` module. The class `qflow.wavefunctions.Dnn` is a subclass of `qflow.wavefunctions.Wavefunction` and allows us to define deep neural networks (DNNs). Setting up the network itself is done by iteratively adding layer objects (`qflow.wavefunctions.nn.layers`).

9.3.3 Samplers

Monte Carlo (MC) sampling is done through one of the implemented methods in the `qflow.samplers` module. Again the code is structured around a base class, `qflow.samplers.Sampler`, which defines the basic interface to the sampler, including querying for samples (system configurations) and obtaining the acceptance rate. Specific algorithms are subclasses, such as `qflow.samplers.MetropolisSampler` and `qflow.samplers.ImportanceSampler`.

9.3.4 Optimizers

The last major piece is the `qflow.optimizers` module. The base class here is `qflow.optimizers.SgdOptimzier` which is the vanilla implementation of stochastic gradient decent (SGD) from Eq. (5.10). Similarly we have `qflow.-optimizers.AdamOptimizer` as a subclass of the former, which unsurprisingly implements Eq. (5.19).

9.4 Inheritance vs. Templates

C++ has a powerful templating feature which allows us to generate code for particular types on demand. We could have used this instead of inheritance in order to compose Hamiltonians, wave functions etc., without the extra overhead and bloat often associated with inheritance. While we acknowledge that a pure template implementation would potentially give a speed increase, the inheritance method was chosen for the following reasons:

- The speed increase would likely be small in comparison to the main bottlenecks.
- Compilation time would increase significantly, as unique template instantiations would be required for each combination
- Selecting a different set of Hamiltonians, wave functions, samplers and optimizers would require another compilation run
- A Python interface would not be possible. In order to generate the Python bindings we need a pre-built C++ library.

9.5 Testing

Vital to any large software project is thorough testing. To this end QFLOW is heavily tested, including unit tests of most functions in use and larger integration tests to test for correct interplay between components. The next chapter presents some of these larger tests, while we here list the ways in which the code is tested on a continued basis.

The tests are written in a mix of C++, using the Google Test framework, and Python using `pytest`. The language mix is partly due to legacy code from before the Python bindings were included, but also because large parts of the C++ code are not exposed to Python.

In addition, were applicable we have made use of Hypothesis to do property based testing from Python. This makes test far superior to hand-crafted examples, and the package actively tries to come up with edge cases that will break the code. The amount of subtle (as well as obvious) bugs that this has found is astounding, and much of our confidence in the code is owed to these property based unit tests.

Because a lot of the code in QFLOW involves implementing mathematical expressions for derivatives, we needed a good way to test this. If we want to go beyond testing just a few hand-made examples, we need a way to programmatically produce the expected derivatives. The problem then is quite typical when writing tests: How do you generate the expected answers without simply retyping the same code twice? This is a place where automatic differentiation (AD) actually became useful. The derivatives of all wave functions are compared with the values obtained through AD by using the Python package Autograd. We were even able to use AD to verify the Laplace operators, because the time inefficiency of AD is less of an issue for unit tests.

On top of all these tests, we have also included so called doctests in the documentation of QFLOW. This is documentation through examples, where the output of the examples are given in the documentation. We can then have a tool parse the documentation and verify that we indeed get the results that are stated. This is a useful way to both improve documentation and to make sure that usage does not break in unexpected ways.

Finally, we use Travis CI for continuous integration. That means all tests are checked on every commit made to the version control system, by building the project on a remote server and running all tests. For convenience, Travis is also set up to update the documentation at the project website, calculate test coverage and scan for potential code issues.

Chapter 10

Verification and Benchmarking

This chapter is dedicated to presenting some of the tests that we have done to verify that the software we have developed is indeed functioning as advertised. These types of tests are more large scale (integration tests) compared to low level unit tests which is a part of the source code for QFLOW. By checking that we can reproduce known benchmarks and observe behavior that is consistent with our expectations we hope to increase the amount of trust assigned to the implementation.

10.1 Setup

We focus our tests on the idealized harmonic oscillator system in $D = 3$ dimensions with N non-interacting particles, i.e. the Hamiltonian given by Eq. (2.26):

$$\hat{H}_0 = \sum_{i=1}^{N} \left(-\frac{1}{2}\nabla_i^2 + \frac{1}{2} r_i^2 \right), \tag{10.1}$$

where we have $\hbar = m = \omega = 1$, and $r_i^2 = x_i^2 + y_i^2 + z_i^2$. This has the ground state given by Eq. (2.30), generalized to N particles in three dimension and omitting normalization constants:

$$\Phi(\mathbf{X}) = \exp\left[-\frac{1}{2} \sum_{i=1}^{N} r_i^2 \right], \tag{10.2}$$

where as before we have defined $\mathbf{X}$ as

$$\mathbf{X} := \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_N \end{pmatrix} := \begin{pmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ \vdots & \vdots & \vdots \\ x_N & y_N & z_N \end{pmatrix} \tag{10.3}$$

For the trail wave function we shall use two different ones. First, the simple Gaussian form of the ground state itself:

$$\psi_G(\mathbf{X}) = \exp\left(-\alpha \sum_{i=1}^{N} r_i^2\right), \tag{10.4}$$

with α the only variational parameter. Learning the ideal parameters should be trivial in this case, and we should expect perfect results.

Second, we use an ansatz resulting from a Gaussian-binary restricted Boltzmann machine (RBM) [33], presented in Eq. (11.3):

$$\psi_{RBM}(\mathbf{X}) = \exp\left[-\sum_{i}^{M} \frac{(X_i - a_i)^2}{2\sigma^2}\right] \prod_{j}^{H} \left(1 + \exp\left[b_j + \sum_{i}^{M} \frac{X_i W_{ij}}{\sigma^2}\right]\right), \tag{10.5}$$

where $M = \#$ of particles $\times$ # of dimensions is the number of degrees of freedom and H is the number of hidden nodes (set to 4 through this section). Note also that X_i in the above refers to the i'th degree of freedom, counting through $\mathbf{X}$ in row major order. The variational parameters are $\mathbf{a}, \mathbf{b}$ and $\boldsymbol{W}$, and we hold $\sigma^2 = 1$ constant in this case.

We use this wave function simply to make learning the true ground state slightly more challenging than proposing a simple Gaussian straight away. Note that setting $\mathbf{a}, \mathbf{b}$ and $\boldsymbol{W}$ all to zero yields the correct ground state in this particular case.

10.2 Energy Estimates and Statistics

We start by verifying that we can reproduce the expected values for $\langle E_L \rangle$, $\langle r \rangle$ and $\langle r^2 \rangle$ for the ideal harmonic oscillator. We still use $D = 3$ dimensions and set $N = 100$ particles.

Table 10.1 shows the energy obtained using ψ_G with two different values of α_G. For the optimal choice $\alpha_G = 0.5$ we get the exact analytic ground state energy, $\langle E_L \rangle / N = D/2$. The reported variance is entirely due to the limited precision of 64 bit floating point numbers.

For the non-optimal $\alpha_G = 0.51$ we get a larger energy, as we would expect. Furthermore, we get a reported standard error of the mean (SEM) of $\sim 2 \cdot 10^{-5}$ a.u.. This includes a correction from the automated blocking mechanism by Jonsson [15]. The results were obtained using 2^{23} Monte Carlo (MC) samples, so by the standard prescription for SEM we should have gotten $8 \cdot 10^{-4}\,\text{a.u.} \big/ \sqrt{2^{23}} \approx 3 \cdot 10^{-7}$ a.u.. This large discrepancy can be explained by the presence of a large amount of autocorrelation in the energy estimates. Inherent to Monte Carlo integration (MCI) is some degree of autocorrelation, but the reason for this large amount is because of the large number of particles and how sampling is implemented. Each MC step we move only a single particle. For large N this means it takes a lot of samples to substantially change the positions of all particles. For smaller N the correction from blocking is smaller, albeit still present. This exemplifies the importance of a proper calculation of statistical errors. Every SEM presented in this thesis will include this correction.

Table 10.1: Estimated ground state energy using ψ_G with two different values for α_G. The energies, standard deviations and variances are given per particle, and were produced using importance sampling (IS) and 2^{23} samples. Statistical errors are corrected for autocorrelation using blocking. Energies in atomic units [a.u.].
Source [1, `writing/scripts/verify-energy-stats.py`]

	$\langle E_L \rangle / N$	Std	Var
$\alpha_G = 0.50$	1.500 00(0)	$1.1 \cdot 10^{-16}$	$1.2 \cdot 10^{-32}$
$\alpha_G = 0.51$	1.500 34(2)	$8.0 \cdot 10^{-4}$	$6.4 \cdot 10^{-7}$

Moving from energy to distance metrics, Table 10.2 shows the mean radial displacement and its squared sibling for all the particles in the system. The results were obtained using $\alpha_G = 0.5$ and 2^{23} MC samples. For reference, the table also states the exact analytic results. Unlike the energy, which for the ideal α_G is independent of position, these results are not perfectly accurate. However, the results are correct to five significant digits, and both $\langle r \rangle$ and $\langle r^2 \rangle$ are within a few SEMs from the analytic value. We take this as further confirmation of both the integration implementation and the validity of the statistical estimates.

Table 10.2: Estimates for the mean radial displacement, $\langle r \rangle$, and the mean squared radial displacement, $\langle r^2 \rangle$. For reference the exact analytic results are listed as well ($\langle r \rangle = 2/\sqrt{\pi\omega}$ and $\langle r^2 \rangle = D/2$), which can be easily verified by computing the corresponding integrals directly. Lengths in dimensionless units of a_{ho}.
Source [1, writing/scripts/verify-energy-stats.py]

	Exact	$\langle \cdot \rangle$	Std	Var
$\langle r \rangle$	1.128 379	1.128 44(2)	$4.8 \cdot 10^{-2}$	$2.3 \cdot 10^{-3}$
$\langle r^2 \rangle$	1.500 000	1.499 96(4)	$1.2 \cdot 10^{-1}$	$1.5 \cdot 10^{-2}$

10.3 One-body Density

Because the wave functions we typically encounter tend to be multidimensional, visualizing them can be quite challenging. One way of reducing the dimensionality is to integrate out the positions for all but one particle. The result is a probability density function (PDF) called the one-body density:

$$\rho(\mathbf{x}_1) = \int \cdots \int d\mathbf{x}_2\, d\mathbf{x}_2 \ldots d\mathbf{x}_N\, |\Psi(\mathbf{X})|^2, \tag{10.6}$$

where we have arbitrarily chosen to keep particle index 1. This gives the PDF for where one might expect to find particle 1, averaged over all possible configurations of the other particles.

As a validating example, we consider the simple Gaussian wave function ψ_G. We get:

$$\rho(\mathbf{x}_1) = \int \cdots \int d\mathbf{x}_2\, d\mathbf{x}_2 \ldots d\mathbf{x}_N\, |\psi_G(\mathbf{X})|^2 \tag{10.7}$$

$$= e^{-2\alpha r_1^2} \int \cdots \int d\mathbf{x}_2\, d\mathbf{x}_2 \ldots d\mathbf{x}_N\ e^{-2\alpha \sum_{i=2}^N r_i^2} \tag{10.8}$$

$$= Ce^{-2\alpha r_1^2}, \tag{10.9}$$

where C is a normalization constant. Perhaps unsurprisingly, the particle will tend to be located close to the center of the potential well, with exponentially decreasing probability for increasing radii.

We can perform the integral in Eq. (10.6) for any wave function using MCI. We simply make a histogram of the particles position as we sample a large amount of configuration.[*] Figure 10.1 shows the resulting plot of $\rho(r_1)$ in a harmonic oscillator with $N = 100$ three-dimensional particles, using 2^{23} samples. After normalizing both the result and Eq. (10.9) the two curves are indistinguishable, showing that our implementation is indeed correct.

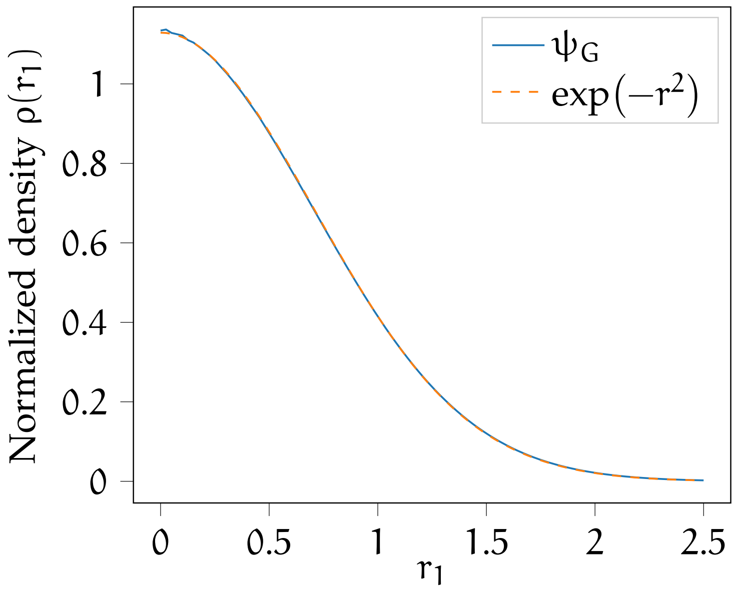


Figure 10.1: One-body density of a particle in a harmonic oscillator potential, as described by ψ_G with $\alpha = 0.5$. The curve is indistinguishable from the analytic result. The small discrepancy around $r_1 = 0$ is an artifact of the vanishing volume of the inner most bins, and the discrepancy becomes increasingly negligible with more samples. This result used 2^{23} MC samples. Source [1, `writing/scripts/verify-onebody-density.py`]

10.4 Two-body Density

Similarly to the one-body density, we can integrate out all degrees of freedom except for two particles. This will give us a two-dimensional PDF showing how the two particles are likely to be located relative to each other.

Mathematically we define it in a similar way,

$$\rho(\mathbf{x}_1, \mathbf{x}_2) = \int \cdots \int d\mathbf{x}_3 \, d\mathbf{x}_4 \dots d\mathbf{x}_N \, |\Psi(\mathbf{X})|^2, \tag{10.10}$$

[*]A technical caveat is that when we discretize the radius r_1 into bins for the histogram we must account for the different volumes (or areas or lengths in two and one dimensions) of the bins. Greater r will correspond to greater volumes, and because of this they will receive a correspondingly greater proportion of the samples. Dividing the bin counts by their respective volumes fixes this.

which for N non-interacting particles governed by ψ_G can be solved analytically:

$$\rho(\mathbf{x}_1, \mathbf{x}_2) = Ce^{-2\alpha(r_1^2 + r_2^2)}. \tag{10.11}$$

Again we can perform the integral numerically using MCI. Figure 10.2 shows a contour plot of the density along with the exact contour lines from Eq. (10.11) indicated by the dashed lines. Visually distinguishing the two is a little harder now, as we have turned to colors to visualize the three-dimensional plot. Still, the two sets of contour lines are very much in agreement, indicating that the implementation is correct.

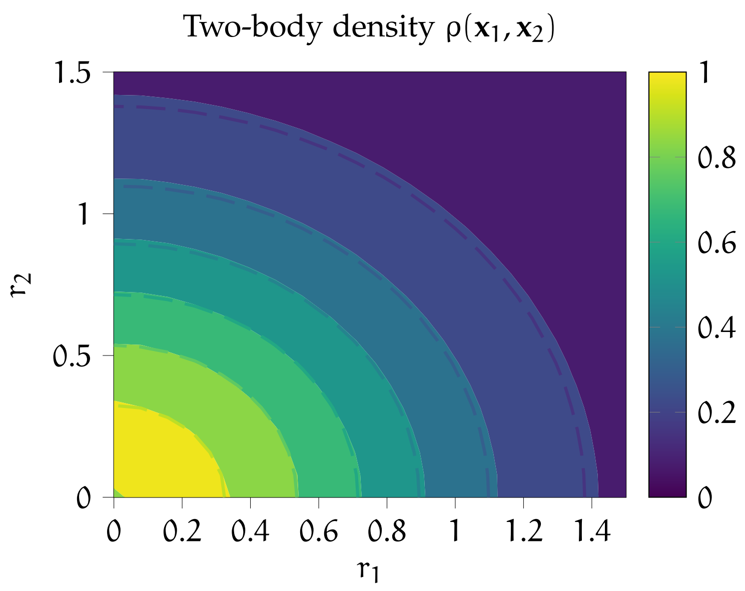


Figure 10.2: Contour plot of the two-body density of $N = 10$ particles in a harmonic oscillator potential, as described by ψ_G with $\alpha = 0.5$. The dotted lines are the contours given by Eq. (10.11). Again the numerical result follows closely that of the exact result. This result used 2^{25} MC samples. Source [1, `writing/scripts/verify-twobody-density.py`]

10.5 Optimization

10.5.1 Integration Test

The simplest complete test is to initialize ψ_G with a non-optimal parameter, e.g. $\alpha = 0.3$, and attempt to learn the optimal value. Optimizing this is trivially accomplished, and Fig. 10.3 shows a training progression using $N =$

10 particles. The hyperparameters have here been artificially tuned to avoid immediate convergence to $\alpha = 0.5$ so as to better illustrate the process.

If we allow the training to progress a little further (or use more optimal hyperparameters), it eventually finds $\alpha = 0.5$ to within machine precision and we get $\langle E_L \rangle / N = D/2$ with exactly zero variance. While this test is not the most challenging, it is nevertheless a useful check.

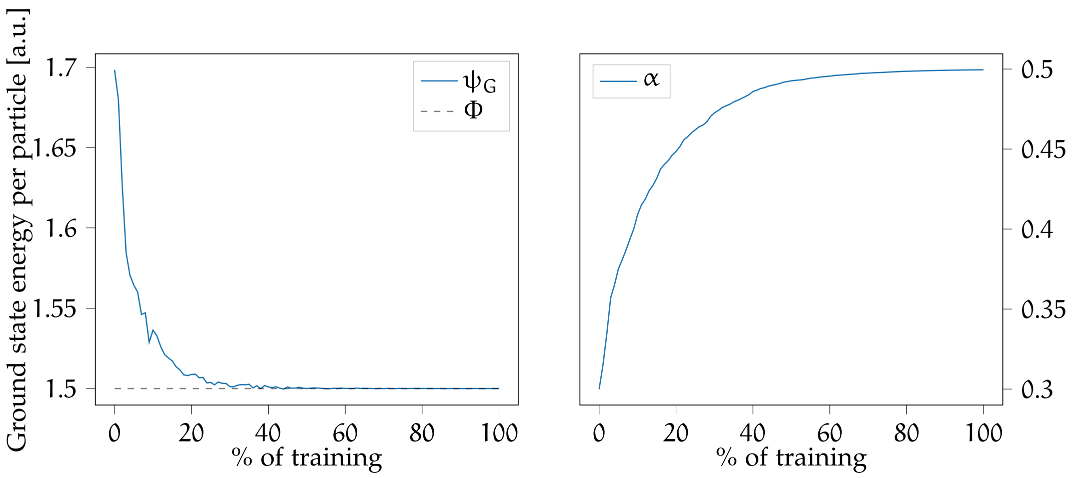


Figure 10.3: Example progression of optimizing the variational parameter α, using ψ_G as the trial wave function. Hyperparameters have been tuned so that we can see what happens, as opposed to immediate convergence to the perfect result.
Source [1, `writing/scripts/verify-simple-gaussian.py`]

10.5.2 Learning Rate Dependency

Successful training is highly dependent on using the correct hyperparameters. Among the most important are the ones controlling the optimization scheme, such as the learning rate in stochastic gradient decent (SGD). The following plots aim at illustrating this dependency, while also serving as a check that the implemented optimization schemes work as expected.

Importantly, these results are not meant to infer that some schemes or learning rates are superior to others. Which scheme works best for a given learning problem will depend on a number of factors, such as the magnitude of the gradients, number of parameters, variance in gradient estimates etc.

Simple Problem - ψ_G

Figure 10.4 shows the absolute error of ψ_G during training with several different schemes. For standard SGD we see that a learning rate around $\eta = 0.1$

(with η defined as in Eq. (5.9)) performs best among these results, and SGD with $\eta = 0.01$ is the slowest to converge. Naturally, values of $0.01 < \eta < 0.1$ perform somewhere between the two.

From these results alone it might seem like larger learning rates always perform better. To an extent this is true, as it allows for more rapid learning. However, setting η too high can lead to divergence and unpredictable behavior. In less extreme cases, it can also keep us from converging properly onto the correct parameters by oscillating around the ideal values.

We have also included some runs using ADAM. Here we have more parameters to play with, but only a few are shown here. In this trivial learning example it is hard to beat properly tuned SGD, but we see how ADAM is able to follow closely. In this particular case, we saw large improvements by reducing β_1, which effectively reduces the momentum applied. An important fact is also that ADAM is designed to be used with many parameters, with individual learning rates per parameter. This enhancement does not show itself in this single-parameter example.

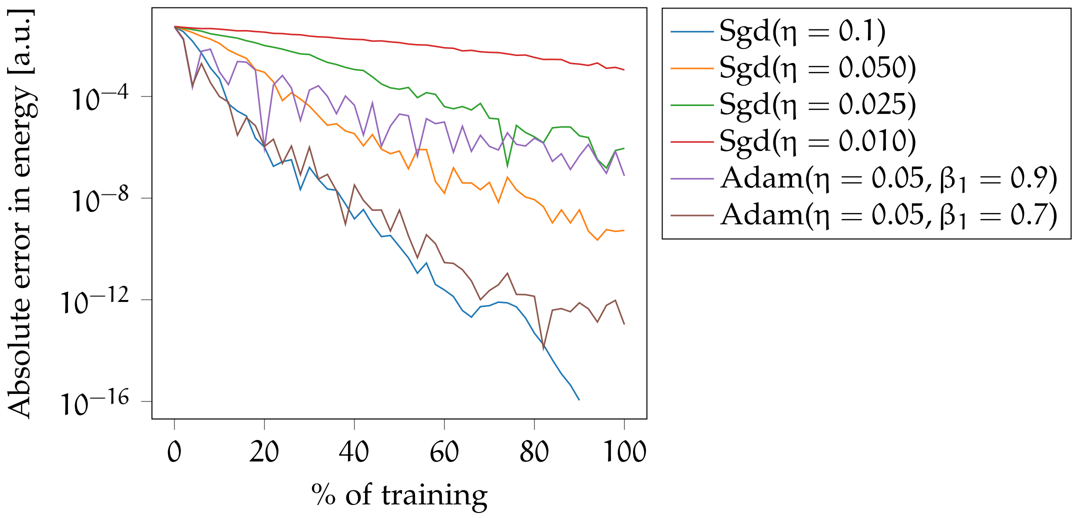


Figure 10.4: Example training progression using ψ_G as trial wave function with different optimization schemes. With sufficient time, all algorithms tend towards zero error.
Source [1, `writing/scripts/verify-learning-rate-gaussian.py`]

More Complex Problem - ψ_{RBM}

We run the same test as above, now with ψ_{RBM} as the trial wave function instead. We do this to illustrate a common pitfall of gradient-based optimization – local minima. Fig. 10.5 shows three runs plateauing around an error $\sim 10^{-6}$a.u.. Interestingly, the worst result is obtained with the middle most value of η. This shows the random nature of SGD, in that it is quite unpredictable when and where we might get stuck due to a local minimum. Repeating the same experiment with different random seeds does not consistently reproduce this particular result.

Similarly, we see that one of the ADAM runs did in fact stumble on to a different, better local minimum. While this is also subject to randomness, we find that ADAM tends to be at least as good as SGD whenever we have more than one parameter to learn. This is to be expected, as ADAM can account for different scales and variability in the components of the parameter gradient.

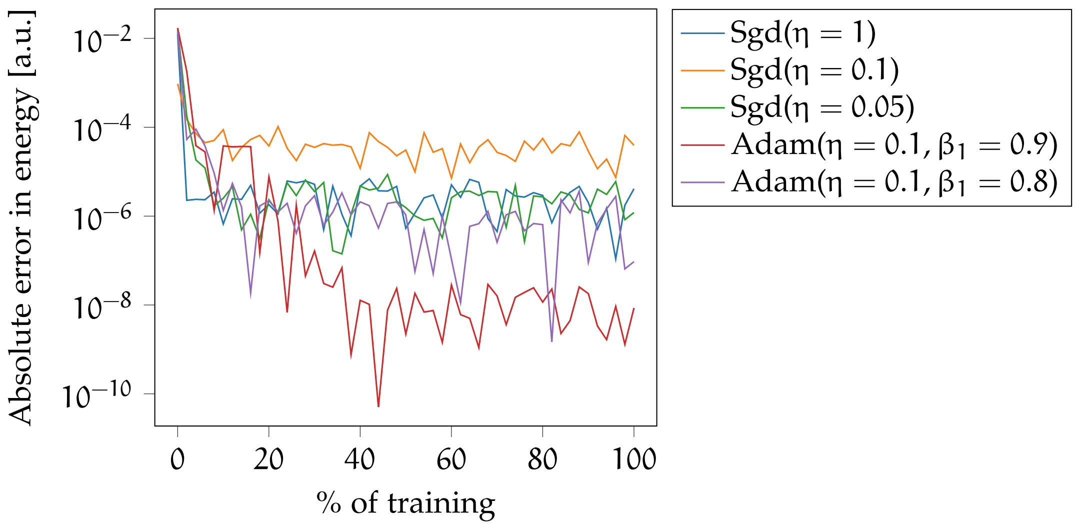


Figure 10.5: Example training progression using ψ_{RBM} as trial wave function with different optimization schemes. We see evidence of learning getting stuck in local minima due to the overly complex wave function anstaz. Source [1, `writing/scripts/verify-learning-rate-rbm.py`]

10.6 Sampling

We will now investigate the behavior of the implemented sampling strategies.

10.6.1 Step Dependency

Similarly to the learning rate in optimization schemes, the MC samplers are highly dependent on an appropriate step parameter (see Algorithms 4.4 and 4.5).* We want to use a step size that balances two opposing attributes:

*We intentionally stick to calling these "step" parameters, without specifically mentioning which of the two (Δx or Δt) we mean. When necessary we will make it clear which type of step parameter is meant.

- Particles should be sufficiently mobile.
  - Unchanging configurations lead to biased energy estimates with high autocorrelation.
- New configurations should be accepted as much as possible.
  - Rejections imply wasted computation time as well as increased autocorrelation.

Figure 10.6 shows how the acceptance rate (AR) changes with different step sizes. For both Metropolis and IS, the AR tends to 100 % for low step sizes, and to 0 % for large values. Both algorithms show a similar pattern in the middle region, with a steeper decline for IS. A good trade-off between the above considerations is achieved when the AR is somewhere in the range 50 % to 99 %, with the exact best value dependent on the particular problem at hand.

The dotted lines show the SEM obtained using the corresponding sampler and step size, when calculating the local energy with ψ_G and $\alpha = 0.51$.[*] The errors were calculated using 2^{21} MC samples and corrected for autocorrelation using blocking [15].

We see Metropolis tending to very small errors in the range shown, with a large spike around $\delta \approx 0.01$. We believe step sizes around this critical point allows enough movement for the system to randomly get into unlikely states, but still so small that getting out of these states takes a long time, leading to a significant portion of samples from unimportant states. The low error for both high and low step sizes can be explained by both extremes resulting in similar behavior; no effective change. When the system remains unchanged, either from rejected samples or effectively equivalent ones (for high and low step sizes, respectively), the resulting local energies must necessarily be very similar as well. Neither case is desirable considering accurate integration results. Recall that the SEM is a statistical measure of the *precision* of the local energy estimate, and not a measure of its *accuracy*.[†]

IS shows different behavior with respect to the SEM. For the entire range of good step size choices, IS results in smaller statistical errors. This is to be expected, and IS will therefore be preferred whenever it is available to us.

Still, we would like to explain the behavior for non-optimal step sizes. The error shows a spike as the step size decreases, in a similar way as seen for Metropolis. Although not shown fully, the error also explodes once the

[*]The value $\alpha = 0.51$ was used to avoid the zero variance of the ideal value $\alpha = 0.5$, but still behaving similarly to the real system.

[†]The number 3.14159 is more precise than the number 6, but if the true value is 6.28, then the latter is more accurate.

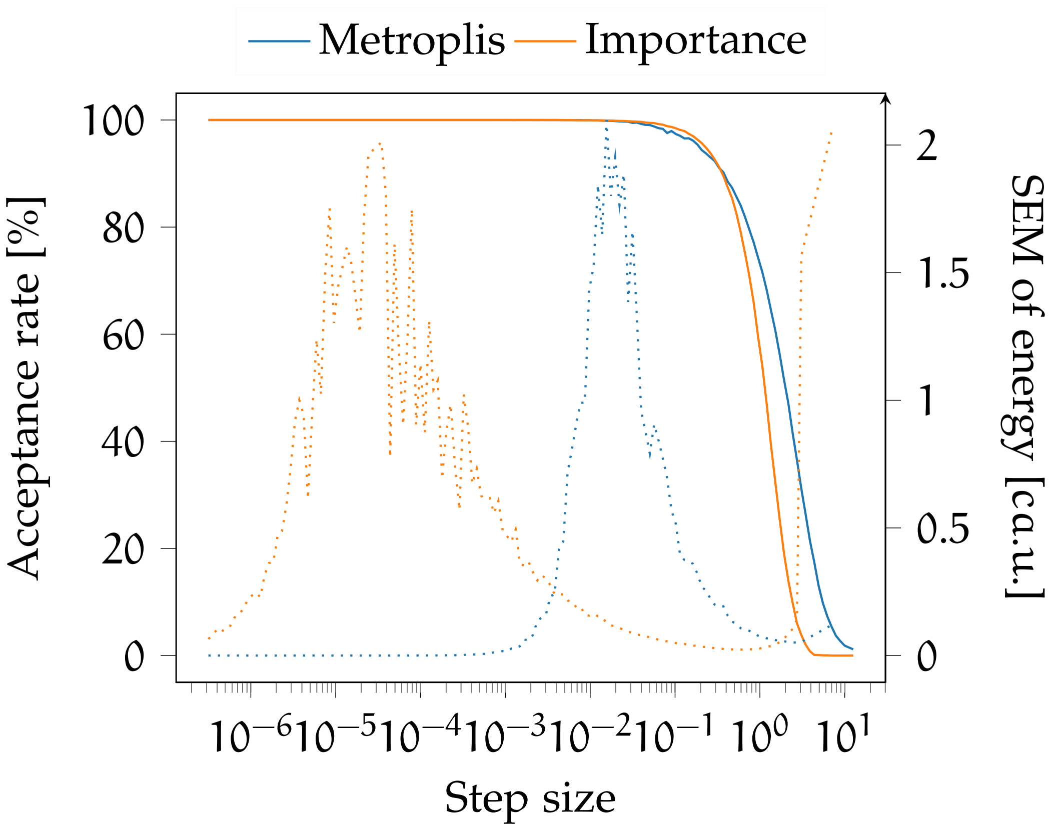


Figure 10.6: *Solid lines:* Acceptance rate of Metropolis and IS as a function of step size. Note that the interpretation of the step size is different for the two algorithms. *Dashed lines:* SEM of energy estimates obtained using the corresponding sampling algorithm and step size, using ψ_G with $\alpha = 0.51$. Source [1, `writing/scripts/verify-sampling-step.py`]

acceptance rate goes below a few percent. We believe this behavior is a result of how the update rule for IS is dependent on both the step size and its square root.

For small step sizes, the square root term will dominate, and the algorithm effectively decays into standard Metropolis. The location of the spike is shifted towards smaller step sizes (approximately the square of the Metropolis spike location), and the spike is wider because the square root function grows slower than linearly.

Finally, the unstable behavior for large step sizes can be explained the other way. The little remaining movement is dominated by the drift force, and the system will quickly get stuck in a local maximum of the wave function, and then be unable to get out again. These are maxima with large probability amplitudes, resulting in large values for the local energy and correspondingly large errors.

Part III

Results

Chapter 11

Quantum Dots

We now present results for all methods discussed, applied on the example system of quantum dots (QD) from Section 2.6.1. We present first some traditional VMC benchmarks using a typical Slater-Jastrow wave function form, followed by the introduction of neural network-based wave functions.

11.1 Benchmark

As we shall restrict this analysis to only two interacting particles for the QD, $\mathbf{X} = (\mathbf{x}_1\ \mathbf{x}_2)$, our benchmark wave function is simple. We build it up using the product of single particle ground states, multiplied by a Pade-Jastrow correlation term:[†]

$$\begin{aligned}\psi_{PJ}(\mathbf{X}) &= \Phi(\mathbf{X})\, J_P(\mathbf{X}) && (11.1)\\ &= \exp\left(-\alpha_G \sum_{i=1}^{N} \|\mathbf{x}_i\|^2 + \sum_{i<j} \frac{\alpha_{PJ} r_{ij}}{1+\beta_{PJ} r_{ij}}\right), && (11.2)\end{aligned}$$

where requirements of wave functions fix $\alpha_{PJ} = 1$ [34], and α_G and β_{PJ} are the only two variational parameters.[‡]

11.1.1 Optimizing

We have run a simple optimization of the above wave function, using initial values of $\alpha_G = 0.5$ and $\beta_{PJ} = 1$. We used importance sampling and the

[†]We drop the constant factor from Eq. (2.30) because we have not normalized the wave function.

[‡]The specific requirement which fixes the value of α_{PJ} is called Kato's cusp condition, after Japanese mathematician T. Kato. It is a condition on how the wave function should behave around the singularity of the Coulomb potential, and ensures stability in our calculations.

ADAM optimization scheme. We used 2000 optimization steps, each with 5000 Monte Carlo (MC) cycles. These values are examples, and we get similar results for several other choices.

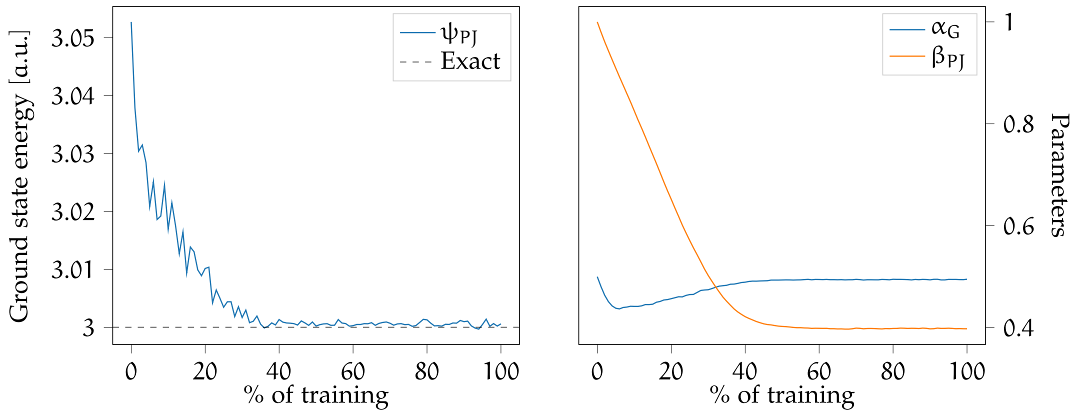


Figure 11.1: Left: Performance of the wave function in Eq. (11.1) as a function of training steps. Right: Corresponding progression of variational parameters as a function of training steps.
Source [1, `writing/scripts/QD-benchmark.py`]

Table 11.1: Energy benchmark using the Pade-Jastrow wave function, with 2^{22} samples and errors estimated by an automated blocking algorithm by Jonsson [15]. Energies in atomic units [a.u.]. The symbols CI_{-}^{95} and CI_{+}^{95} represent the lower and upper limits of a 95 % confidence interval, respectively. Std is the standard deviation of the energy and Var is the corresponding variance. See Fig. 11.1 for source code reference.

	$\langle E_L \rangle$	CI_{-}^{95}	CI_{+}^{95}	Std	Var
Φ	3.250(2)	3.246	3.254	$1.8 \cdot 10^{-1}$	$3.3 \cdot 10^{-2}$
ψ_{PJ}	3.000 66(6)	3.000 55	3.000 77	$5.0 \cdot 10^{-3}$	$2.5 \cdot 10^{-5}$

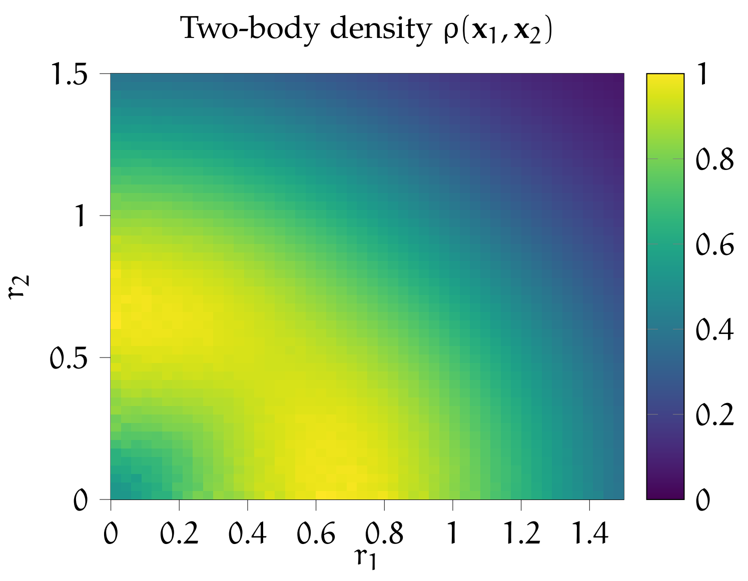
Figure 11.1 shows the optimization as function of percentage of training completed. We can observe that the optimizations quickly settles down to a set of optimal values, where they only oscillate slightly back and forth. Table 11.1 shows statistics for the energy estimate obtained with the final parameter values. Comparing to the analytical result of 3 a.u. these results are in good agreement. For reference we have also given the results obtained without the Pade-Jastrow term, i.e. the non-interacting ground state.

11.1.2 Two-body Density

Given that we now consider two interacting particles, a natural quantity to investigate is the two-body density. Recall Fig. 10.2 which showed the two-body density for the non-interacting system as a simple Gaussian. Computing $\rho(\mathbf{x}_1, \mathbf{x}_2)$ using ψ_{PJ} with the final parameters obtained previously we get Fig. 11.2.

This plot shows a very different behavior. The particles no longer prefer to (simultaneously) stay close to the center of the potential because of the Coulomb repulsion, and the density maxima is pushed outwards. Additionally, there is a small dent in the density when $r_1 \approx r_2$, not only in the center. This effect is made clearer by looking at the squared density. These results are quite intuitive, while showcasing the highly non-trivial behavior that the two particles exhibit.

If we integrate out one of the particle positions, i.e. compute the one-body density we get Fig. 11.3. This shows that the most probable position for any single particle is still in the center, but the density is much more spread out to larger values of r.



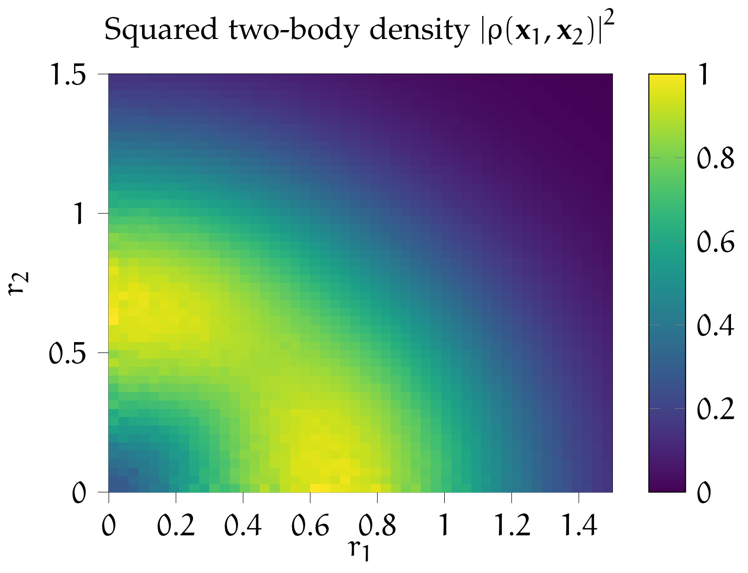


Figure 11.2: Top: Two-body density described by ψ_{PJ} with optimized parameters. Bottom: The squared equivalent of the top figure, included to emphasize the small decrease in density along the diagonal $r_1 = r_2$.
Source [1, `writing/scripts/QD-benchmark-density.py`]

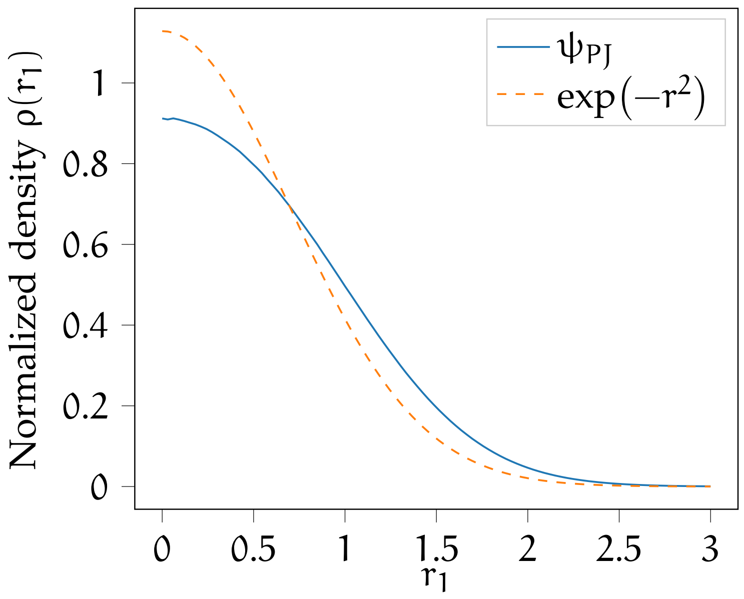


Figure 11.3: One-body density described by ψ_{PJ} with optimized parameters. For reference, the exact one-body density for the non-interacting system is indicated by the dotted line. As expected, the Coulomb interaction pushes the particles to a greater distance from the origin, flattening out the density. Source [1, `writing/scripts/QD-benchmark-onebody.py`]

11.2 Restricted Boltzmann Machine

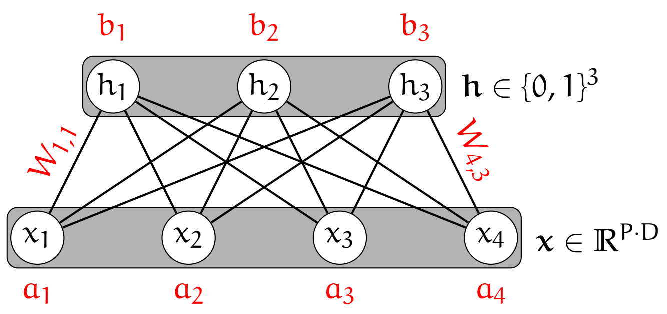


Figure 11.4: Example diagram of a Gaussian-binary restricted Boltzmann machine, showed with four visible nodes and three hidden nodes. The red values are the parameters, and consist of visible layer bias, $\mathbf{a}$, hidden layer bias, $\mathbf{b}$ and connection weights $\boldsymbol{W}$. $M = \#$ of particles $\times$ # of dimensions $= P \times D$ is the number of degrees of freedom in the system.
Source [1, `writing/illustrations/rbm-diagram.tex`]

The first machine learning (ML)-inspired model we have applied is a restricted Boltzmann machine (RBM). This type of model has seen a significant rise in usage since Carleo and Troyer [28] demonstrated the restricted Boltzmann machine (RBM)'s capacity to represent the wave function for some selected Hamiltonians. All the Hamiltonians for which they showed successful results, however, had discrete configuration spaces. The current system is continuous as the particles can have any real valued coordinates, and so the type of RBM must change as well.

While more than one choice exists, we have used a Gaussian-binary RBM. Figure 11.4 shows an example diagram of the network structure. Flugsrud [33] has a full introduction to RBMs, including all details needed for variational Monte Carlo (VMC). For our purposes it suffices to say that the resulting wave function looks as follows:

$$\psi_{RBM}(\mathbf{X}) = e^{-\sum_i^M \frac{(X_i - a_i)^2}{2\sigma^2}} \prod_j^N \left(1 + e^{b_j + \sum_i^M \frac{X_i W_{ij}}{\sigma^2}}\right), \tag{11.3}$$

where $M = \#$ of particles $\times$ # of dimensions is the number of degrees of freedom and N is the number of hidden nodes. Note also that X_i in the above refers to the i'th degree of freedom, counting through $\mathbf{X}$ in row major order. The parameters are $\mathbf{a}, \mathbf{b}$ and $\boldsymbol{W}$, and we hold $\sigma^2 = 1$ constant in this case.

If we set $\mathbf{a} = \mathbf{0}$ we can recognize the first factor of Eq. (11.3) as the non-interacting ground state. That way we can consider the second factor the

Jastrow factor introduced by the RBM structure. It has an unconventional form, as it is not a pure exponential.

11.2.1 Optimizing

We produced the following results using normally distributed random initial values for the parameters, running 60 000 optimization steps with 2000 MC cycles each. We have also once again used importance sampling and ADAM. A new addition (not strictly necessary for similar results) is the use of mild L2 regularization (Section 5.2.1), which serves to drive parameters that do not contribute towards zero. The results are similar for different hyperparameter choices, and the above is simply one such example.

Figure 11.5 shows the ground state energy as a function of training steps, along with the progression of the variational parameters. While we see a clear improvement in the initial stages, the RBM fails to converge as accurately as the benchmark. Table 11.2 shows the precise results of the final model. While slightly different results are possible with different training settings, we have never observed the RBM achieve energies below 3.07 a.u., which is an error of about two orders of magnitude larger than the benchmark.

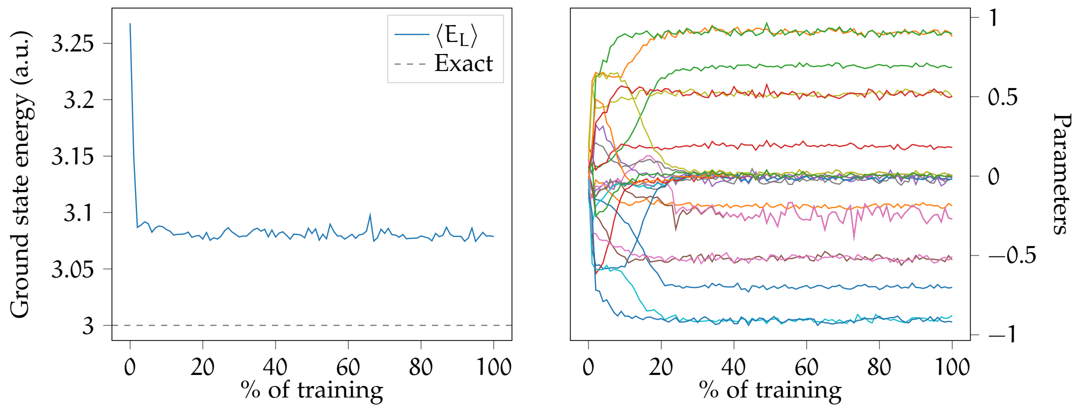


Figure 11.5: Left: Performance of the wave function in Eq. (11.3) as a function of training steps. Right: Progression of variational parameters as a function of training steps.
Source [1, `writing/scripts/QD-rbm.py`]

An important consideration that arises from this form of wave function model is that it has no guarantee of satisfying the required permutation symmetry. This is an attribute of most neural network based models. Because we know the true ground state must have the correct symmetry, we would hope that the RBM is able to realize that a symmetric form is best. To this end we have defined a metric $S(\psi)$, which has the property of being equal to 1 for

Table 11.2: Energy using the RBM wave function in Eq. (11.3), along with the same wave function using input sorting to impose symmetry. Results obtained from 2^{23} samples and errors estimated by an automated blocking algorithm by Jonsson [15]. Energies in atomic units [a.u.]. See Fig. 11.5 for source code reference.

	$\langle E_L \rangle$	CI_-^{95}	CI_+^{95}	Std	Var
ψ_{RBM}	3.0796(9)	3.0779	3.0813	$1.1 \cdot 10^{-1}$	$1.3 \cdot 10^{-2}$
ψ_{SRBM}	3.0805(9)	3.0788	3.0822	$1.6 \cdot 10^{-1}$	$2.5 \cdot 10^{-2}$

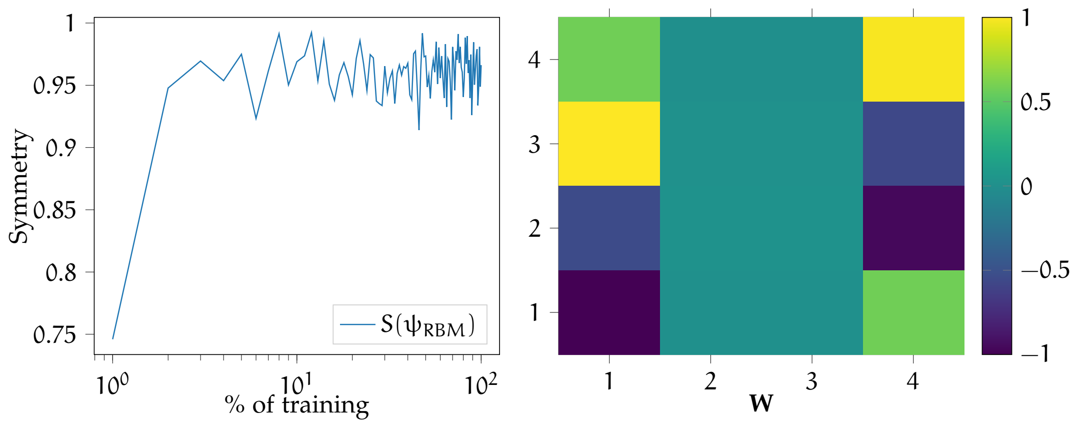


Figure 11.6: Left: Permutation symmetry of the RBM wave function in Eq. (11.3) as a function of training steps. Right: Color map of the weight matrix of the RBM after training. Weights have been scaled to $[-1, 1]$ for simplicity. See Fig. 11.5 for source code reference.

symmetric functions and 0 for anti-symmetric ones. See Appendix B for its definition and details. The left plot in Fig. 11.6 shows a plot of the symmetry metric of ψ_{RBM} during training. Luckily we see that the RBM, initially starting fully non-symmetric, tends rapidly towards $S(\psi_{RBM}) = 1$. Still it is not purely symmetric, and we see significant oscillations around the maximum value.

In attempts to properly fix the symmetry of ψ_{RBM}, we have made two significant attempts. The first was reducing the biases $\mathbf{a}$ to be $\mathbf{a} \in \mathbb{R}^D$ and reducing the weights to $\mathbf{W} \in \mathbb{R}^{D\times N}$, i.e. have parameters per dimension and not per degree of freedom. While this ensured $S(\psi_{RBM}) = 1$, and the RBM could still learn the ground state if the particles where non-interacting (i.e. the pure Gaussian), it failed in the full system. The next section looks closer at the weights, and explains why this reduced form failed.

More successfully we imposed symmetry by sorting the inputs prior to feeding them through ψ_{RBM}. While this was able to learn something, it stalled out at > 3.1 a.u.. In general it is our experience that imposing the symmetry from the beginning of the training stops the RBM from getting anywhere. This is a manifestation of the classical problem in learning of balancing exploitation and exploration, with this case suffering from a lack of exploration.

The only truly successful way we found was to train the RBM as before, and then apply sorted inputs on it once it had been fully trained. The results of this is also shown in Table 11.2 by ψ_{SRBM} (symmetric RBM). This lead to only a marginal, not statistically significant increase in energy, and as such was a successful way to impose the symmetry.

11.2.2 Interpreting the Network

The right plot in Fig. 11.6 shows a peculiar pattern emerging from the weights $\boldsymbol{W}$ of the final model. Half of the weights appear inconsequential, while the remaining come in pairs of equal value. If we increase the number of hidden nodes, the result is always two similar columns with the extra elements zeroing out. Exactly *which* columns get the non zero values seems random, and varies based on the random initialization of the weights.

Although not immediately obvious from the plot, there are only two unique absolute values, i.e. $\pm v_1$ and $\pm v_2$. That is, in this particular case:

$$\boldsymbol{W} = \begin{pmatrix} W_{1,1} & W_{1,2} & W_{1,3} & W_{1,4} \\ W_{2,1} & W_{2,2} & W_{2,3} & W_{2,4} \\ W_{3,1} & W_{3,2} & W_{3,3} & W_{3,4} \\ W_{4,1} & W_{4,2} & W_{4,3} & W_{4,4} \end{pmatrix} \simeq \begin{pmatrix} v_1 & 0 & 0 & v_2 \\ v_2 & 0 & 0 & -v_1 \\ -v_1 & 0 & 0 & -v_2 \\ -v_2 & 0 & 0 & v_1 \end{pmatrix} \tag{11.4}$$

This relation is only approximately true, as the RBM has not learned to set the weights exactly equal to each other, but it is close enough to show a clear pattern.

This pattern explains why the attempt at imposing symmetry with a reduced form of the RBM failed. This pattern is simply not possible to express with one weight per degree of freedom.

Having now observed this phenomena, we can seek to understand why the RBM learns this particular weight matrix. If we ignore the Gaussian factor of ψ_{RBM} and substitute in the above matrix for $\boldsymbol{W}$, we get:

$$\begin{aligned} \psi_{\mathrm{RBM}} \propto & \left\{ 1 + \exp\left[b_1 + \frac{1}{\sigma^2}(v_1(x_1 - x_2) + v_2(y_1 - y_2)) \right] \right\} \\ \times & \left\{ 1 + \exp\left[b_4 + \frac{1}{\sigma^2}(v_2(x_1 - x_2) - v_1(y_1 - y_2)) \right] \right\} \end{aligned} . \tag{11.5}$$

If $\mathbf{a} = a\mathbf{1}$ and $\mathbf{b} = b\mathbf{1}$ (i.e. biases are all equal in value), this leads to an RBM that is deceptively close to a fully permutation symmetric function, broken only by the sign on $(y_1 - y_2)$. The RBM has, without direct instructions, learned that the differences between the spacial coordinates of the two particles are important. In contrast to ψ_{PJ}, which has the distance r_{12} explicitly incorporated, the RBM has learned the closest equivalent it has.

The fact that the RBM is able to pick up on the importance of relative differences in position is encouraging for further work with less constrained neural networks. In this case, the RBM could not learn to square the differences (e.g. $(x_1 - x_2)^2$ instead of $(x_1 - x_2)$), simply because it had no means of expressing this.

11.3 Neural Network

Finally we turn our attention to the new contribution of this thesis – applying general neural networks as wave functions.

We have not been able to simply replace the entire wave function with a neural network from the start. In order for VMC optimization to work we need a stable enough starting point to avoid immediate divergence. The better approach is to bootstrap the network with an existing wave function as a starting point. In our case we use the benchmark wave function ψ_{PJ}. There are two options for how to do this:

1. Pre-train the network to emulate ψ_{PJ} before starting VMC optimization.
2. Use the network as an extra correlation factor multiplied with ψ_{PJ}.

While there is a certain appeal to the first alternative, in the sense that we would end up with a pure neural network wave function, the downside is that pre-training will not be as accurate, as well as taking some time. Small inaccuracies could also lead to instabilities from non-satisfied cusp conditions/symmetry requirements. More importantly, we might not know the correct form to pre-train until we have performed a VMC optimization to begin with. By using the second alternative, we are able to train both network and base wave function simultaneously.

Because of this, we opted to simply treat the network as a correction factor, aimed to fix the small discrepancy between the benchmark and the true ground state. This approach allows us to bootstrap learning and start where existing techniques have already taken us, and improve from there. It is also trivial to implement in QFLOW, because of built in support for composing wave functions by multiplication.

The proposed wave function is:

$$\psi_{\mathrm{DNN}}(\mathbf{X}) = \psi_{\mathrm{PJ}}(\mathbf{X})\, f(\mathbf{X}), \tag{11.6}$$

where f represents some arbitrary neural network.

11.3.1 Network Architecture

We consider here only simple feed-forward neural networks (FFNNs) and leave other species of neural networks (NNs) for future research.

The size of the system determines the number of inputs. The output layer is also essentially forced as we want a scalar output from our network.* In addition, all existing research implies that such correlation factors should be exponential.

For the hidden layers we have much more of a choice. By simple trail and error we found tanh to be the most suitable activation function, and we have used this in all cases. The number of nodes needs to be sufficiently large compared to the number of inputs. This is to allow the network to learn a large amount of different correlations, and not force it to settle right away. The specific choices are again empirically motivated, and stem from observing worse results with much smaller values and diminishing returns for larger choices.

The later hidden layers should decrease in width as we approach the output, with the idea that the network should gradually attempt to compact what it learns into a smaller space. We have used powers of two for no particular reason other than that it allows for even divisions.

While a large family of architectures are likely to perform well in this case, we have settled on the following setup:

Layer	Nodes	Activation
Input	4	—
Hidden 1	32	tanh
Hidden 2	16	tanh
Output	1	exp

Finally, we have also tested the effect of imposing permutation symmetry on the network by sorting the inputs, as discussed in Section 6.2.2. The network remains unchanged, and the symmetric version of ψ_{DNN} is labeled ψ_{SDNN} in all of the following.

*For applications that require complex wave functions, this can be easily modeled by having two outputs and interpret them as the real and imaginary component.

11.3.2 Optimization

We produced the following results by optimizing $\psi_{\mathrm{PJ}}, \psi_{\mathrm{DNN}}$ and ψ_{SDNN} with all the same hyperparameters. We also initialized the parameters of the two network wave functions equally for better comparison. We used 30 000 iterations of 1000 MC cycles each. We did not use any regularization in this case, and as usual we used importance sampling and the ADAM optimizer. Again, this is an example of hyperparameters, and we can achieve good results with a range of other choices.

Figure 11.7 shows a graph of the absolute error in energy during the course of training, for all three wave functions. To begin with, ψ_{PJ} and ψ_{DNN} closely follow each other, likely because the Pade-Jastrow factor contributes most of the improvements at this point. Additionally, we have initialized the networks with rather small weights, which results in the network in ψ_{DNN} having a weak effect to start. This was done to ensure stability.

While ψ_{PJ} plateaus after a short amount of time, we see ψ_{DNN} continues to improve beyond this point once the weights reach a critical magnitude. In the end ψ_{DNN} seems to converge at a little more than halfway through the training, and we achieve an order of magnitude better accuracy compared to ψ_{PJ}, with a corresponding reduction in the variance.

The symmetric counterpart, ψ_{SDNN}, acts quite differently. Ignoring the random spike in the beginning, we still see improvement compared to ψ_{PJ}, but distinctly worse than the network without input sorting. It also appears to have significantly more variance compared to ψ_{DNN}, however, the variance estimates for the final model (Table 11.3) disagrees. It is possible that the variance was larger during training, and that the effect is amplified by the logarithmic scale on the axis. Regardless, we again conclude that imposing symmetry in this system seem to only hamper the learning. The symmetry metric for ψ_{DNN} is very close to 1 throughout training, so it seems the network learns the symmetry sufficiently well on its own.

Table 11.3 shows the final energies produced from the three wave functions after training. The results solidify the graphical impression of Fig. 11.7, showing a clear improvement by both networks. The result from ψ_{DNN} approaches the accuracy of diffusion Monte Carlo (DMC), a technique that can obtain exact numerical results. Pedersen Lohne et al. [35] lists $3.000\,00(1)$ a.u. as the DMC result, which has an overlapping confidence interval with our results.

Also included is the energy obtained by using the parameters from ψ_{DNN} in ψ_{SDNN}. Unlike for the RBM, where this strategy barely changed the energy, the effect was much worse for the deep neural network (DNN). In this small two-body system it seems the best strategy is to simply let the network learn the symmetry, despite that it never learns a perfectly symmetric set of parameters.

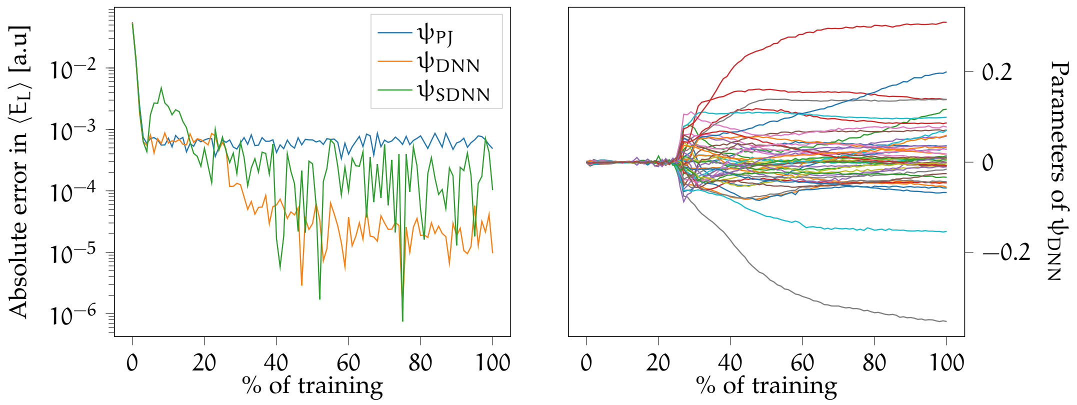


Figure 11.7: Left: Performance of the wave function in Eq. (11.6) as a function of training steps. Right: Progression of variational parameters as a function of training steps. We only show a small selection of the total number of parameters.
Source [1, `writing/scripts/QD-pade-dnn.py`]

11.3.3 Interpreting the Networks

Given the apparent ability of the networks to improve upon our theoretically motivated ansatz, it would be interesting to see if we can learn something from the structure of the network. This is, however, notoriously difficult to do with neural networks. In most cases the interconnections that make up the network are treated as a black box, simply because we lack the understanding to do anything else.

Still, we can make an attempt. Figure 11.8 shows a color map representation of all the weights and biases of ψ_{DNN} after training. Arguably, we should not attempt to assign physical meanings to any of the later layers. If anything, the weight matrix of the first layer could be insightful, as this is directly multiplied with the coordinates $\mathbf{X}$. Interestingly, only a few columns appear to have significant values. This is very similar to the effect seen in Fig. 11.6 for the RBM. In fact, a similar symmetry between two of the columns is present, hinting that the network has realized the importance of relative differences in coordinates. The remaining columns are not as clearly zeroed out, partly due to the lack of regularization, but also indicating that more is happening here compared to the RBM. While non-trivial to interpret, this points to the networks learning, completely by themselves, some underlying physical structure in the inputs.

Table 11.3: Energy using the neural networks ψ_{DNN} and ψ_{SDNN}, along with the benchmark wave function after the same amount of optimization. $\hat{\psi}_{\text{SDNN}}$ shows the result of using the parameters from ψ_{DNN} with input sorting applied only after training. Results obtained from 2^{22} samples and errors corrected for autocovariance by blocking. Energies in atomic units [a.u.]. See Fig. 11.7 for source code reference.

	$\langle E_L \rangle$	CI_{-}^{95}	CI_{+}^{95}	Std	Var
ψ_{PJ}	3.000 64(4)	3.000 57	3.000 72	$4.9 \cdot 10^{-3}$	$2.4 \cdot 10^{-5}$
ψ_{DNN}	3.000 023(4)	3.000 014	3.000 031	$3.8 \cdot 10^{-4}$	$1.5 \cdot 10^{-7}$
ψ_{SDNN}	3.000 087(3)	3.000 081	3.000 092	$3.7 \cdot 10^{-4}$	$1.3 \cdot 10^{-7}$
$\hat{\psi}_{\text{SDNN}}$	3.000 451(3)	3.000 445	3.000 457	$4.0 \cdot 10^{-4}$	$1.6 \cdot 10^{-7}$

11.3.4 Complexity

While the neural network has demonstrated its capacity to learn an improved correlation function, this does come at a considerable computational cost.

The benchmark function has complexity $\mathcal{O}(N^2)$ for N particles, rooted in the double sum in Eq. (11.1). Thanks to analytic expressions for the derivatives, the otherwise computationally expensive Laplace operator is also calculated in $\mathcal{O}(N^2)$ time.

The complexity of the neural network is a little harder to pin down, because of the inherent flexibility in how the network can be structured. It computes a series of matrix-vector multiplications, both during evaluation and for its derivatives. The input layer needs to scale equal to N, while the other layers could in principle be constant. This means we could say that the network only scales linearly, $\mathcal{O}(N)$, with increasing number of particles. Still, the "constant" part of the computation time is very large, and this kind of complexity analysis is slightly unfair.

In practice we found that we needed the second layer to have more nodes than we have inputs. We have not determined a strict relationship, but either $\mathcal{O}(N)$ or $\mathcal{O}(N^2)$ seems reasonable for the number of nodes in the second layer. Assuming the network shrinks from there on, the dominant operation is a matrix-vector operation with matrix dimensions of order $(N \times N)$ or $(N \times N^2)$, which works out to a complexity of $\mathcal{O}(N^2)$ or $\mathcal{O}(N^3)$. In this more realistic view, the networks are not improving upon the complexity of the benchmark, but rather the opposite is true.

The networks also depend highly on the depth (number of layers) as well as obviously the sizes of subsequent layers. We should also note that while asymptotic cost might be equal to the benchmark, a more precise count of

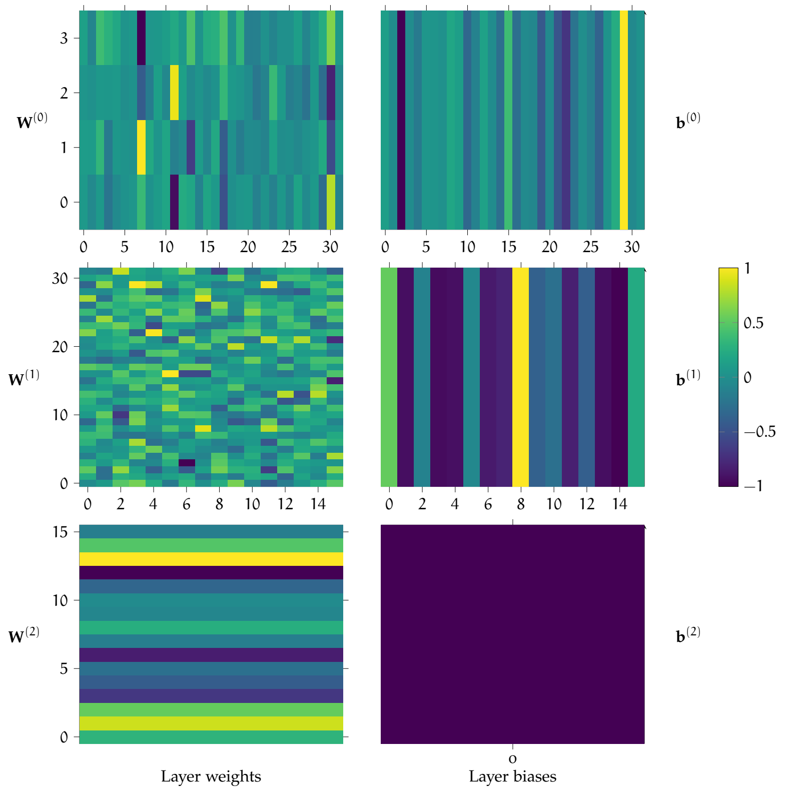


Figure 11.8: Color map representation of the weights and biases in the network of ψ_{DNN}. The left column of plots shows the weight matrices in each layer, and the corresponding biases are in the right column. The bottom right plot has only one color because there is only one output layer bias. Each rectangle represents a parameter, and they have been scaled to have the same total size. The size of the rectangles are not important. The parameters of ψ_{SDNN} are largely different, but show similar patterns. See Fig. 11.7 for source code reference.

the required operations would show a substantial disparity. Furthermore, we have found that the NNs generally require more training iterations compared to the benchmark, which further compounds the computational cost.

Nevertheless, a constant cost difference can always be brought down with optimized code, more and better hardware, introducing graphics processing units (GPUs) etc. And while computing power may no longer increase exponentially each year, it continues to grow. Large scale use of networks for quantum mechanics might therefore be well within reach.

11.3.5 Other Observables

So far we have only investigated the energies produced by the different wave functions. Other observables of interest could be the mean distance from the origin, $\langle r \rangle$, the mean squared distance from the origin, $\langle r^2 \rangle$, and the mean inter-particle distance, $\langle r_{12} \rangle$. These are all quantities that should depend on the correlation between the particles. Table 11.4 shows the three observables as predicted from the three wave functions trained previously.

Table 11.4: Average distances predicted by the different wave functions. Results obtained by Monte Carlo integration (MCI) using importance sampling (IS) and 2^{24} samples. The first row shows the corresponding values for a single particle in an ideal harmonic oscillator, with the values coming from the analytic expressions $\langle r \rangle = \sqrt{\pi}/2\sqrt{\omega}$ and $\langle r^2 \rangle = \omega^{-1}$. While the differences between ψ_{PJ} and the networks are small, the inter-particle distance shows the largest difference. Distances in dimensionless units of a_{ho}.
Source [1, `writing/scripts/QD-other-metrics.py`]

	$\langle r \rangle$	$\langle r^2 \rangle$	$\langle r_{12} \rangle$
Ideal HO	0.886 23	1.000 00	—
ψ_{PJ}	1.0192(2)	1.2952(5)	1.6337(5)
ψ_{DNN}	1.0197(2)	1.2974(5)	1.6393(5)
ψ_{SDNN}	1.0200(2)	1.2967(5)	1.6393(5)

We see first that, as one would expect, the predicted radii are larger when we have multiple interacting particles, as opposed to the ideal non-interacting case. Less clear is the differences between the benchmark wave function ψ_{PJ} and the networks. In general, the results indicate that ψ_{PJ} slightly underestimates the effect of the Coulomb repulsion with smaller estimates for all three quantities. While $\langle r \rangle$ and $\langle r^2 \rangle$ are only borderline significantly different from the networks, we see a larger difference in $\langle r_{12} \rangle$. This might be because $\langle r_{12} \rangle$ is the quantity which is most directly connected to the strength of the interaction

The two network types seem to be equivalent, with the observed differences well within statistical expectations.

Chapter 12

Liquid ^{4}He

We turn now to the much more challenging system of liquid helium, presented in Section 2.6.2. As before, we will first present the traditional benchmark result followed by the neural networks.

12.1 Benchmark

We will use one of the simpler benchmark wave functions for this system, known as the McMillan form wave function [36]:

$$\psi_M = \exp\left(-\frac{1}{2}\sum_{i<j}\left(\frac{\beta}{r_{ij}}\right)^5\right). \tag{12.1}$$

where β is the only variational parameter and has units of angstrom. An important observation now is the lack of any single particle wave function factor. In the case of quantum dotss (QDs) we had a Gaussian localized at the origin as a result of the potential well. This system, however, is infinite and periodic without any such influence driving it towards particular points in space. Furthermore, because of the lack of an external field the single particle solutions are just free particles, and does not help us understand the many-body system.

12.1.1 Finite Size Dependency

An important aspect of all the results that we will present is that they are highly dependent on the number of particles used in the simulation box, as well as the size of the box it self. We will hold the number density of particles constant, $\rho = 0.365/\sigma^3$ (where $\sigma = 2.556\,\text{Å}$ as defined in Eq. (2.37)), and set the side lengths of the simulation box, L, depending on the number of particles N:

$$L = \sqrt[3]{\frac{N}{\rho}}. \tag{12.2}$$

As the assumption of periodicity is a simplifying approximation, we introduce some erroneous effects because of it. These generally disappear as we increase the number of particles (and hence the size of box), but the computation time needed to run the simulations increases significantly with increasing numbers. The purpose of the following analysis is to test the *relative* accuracy of different wave functions. With that in mind we have used a small number of particles in the main results, which introduces a significant error with respect to reality. The number of particles should nevertheless be large enough to introduce all the relevant effects and allow for a valid comparison of methods. With the same motivation, we limit our Hamiltonian to use only the Lennard-Jones potential (Eq. (2.37)), and leave tests with HFDHE2 for future research.

12.1.2 Optimizing

We have optimized the parameters of Eq. (12.1) using 10 000 iterations of 5000 Monte Carlo (MC) cycles each. We used standard Metropolis sampling along with the ADAM optimizer. Fig. 12.1 shows the progression of both the energy and variational parameter during training. Because of the strong correlations involved, there is significantly more variance in these results compared to the benchmark used for QDs. We see the value for β oscillating without any signs of converging to a fixed value.

Table 12.1 shows the energy estimates from the final model. The same wave function (i.e. same value for β) has also been used on different numbers of particles to illustrate how the energy increases for larger simulation boxes. Based on computations with larger and larger systems, the value for $N = 256$ is quite close to the apparent convergence for $N \to \infty$.

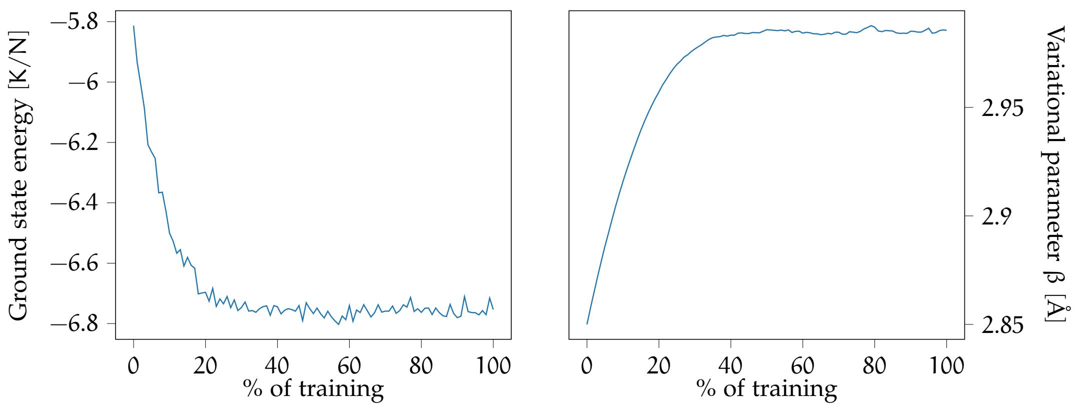


Figure 12.1: Left: Ground state energy produced by Eq. (12.1) as a function of training steps. Right: Progression of the variational parameter β of Eq. (12.1) as a function of training steps. A convergence is clearly visible, although there is more variance in the value for β compared to the parameters of the benchmark used for quantum dots.
Source [1, `writing/scripts/He-benchmark.py`]

Table 12.1: Predicted ground state energy of helium atoms at density $\rho = 0.365/\sigma^3$ using ψ_M with β as determined after training. The number of particles used in the simulation box is indicated by the superscript on ψ_M. Values obtained using 2^{23} samples. Energies in units of Kelvin per particle [K/N]. The symbols CI_-^{95} and CI_+^{95} represent the lower and upper limits of a 95 % confidence interval, respectively. Std is the standard deviation of the energy and Var is the corresponding variance. See Fig. 12.1 for source code reference.

	$\langle E_L \rangle$	CI_-^{95}	CI_+^{95}	Std	Var
$\psi_M^{(32)}$	$-6.76(1)$	-6.79	-6.74	$4.6 \cdot 10^{-1}$	$2.1 \cdot 10^{-1}$
$\psi_M^{(64)}$	$-6.15(1)$	-6.18	-6.12	$3.0 \cdot 10^{-1}$	$9.1 \cdot 10^{-2}$
$\psi_M^{(256)}$	$-5.85(1)$	-5.87	-5.83	$2.2 \cdot 10^{-1}$	$4.7 \cdot 10^{-2}$

12.2 Neural Networks

We again attempt to apply a neural network to our wave function representation. Liquid ^{4}He is a far more challenging problem compared to the two-particle quantum dot, and as such it will serve as an important test of the capabilities of neural networks for general quantum mechanical many-body problems.

As before, we have not simply removed the original wave function, but rather added the network as another correlation factor:

$$\psi_{\text{DNN}}(\mathbf{X}) = \psi_M(\mathbf{X})\, f(\mathbf{X}), \tag{12.3}$$

where once again $f : \mathbb{R}^{N\times D} \to \mathbb{R}$ represents an arbitrary neural network.

Lastly, we limit our investigation to $N = 32$ particles and still with density $\rho = 0.365/\sigma^3$. This is done from a practical standpoint, considering the substantial computational cost.

12.2.1 Network Architecture

The proposed network structure is one of many that are likely to perform well, and in all likelihood this is not the optimal form. A more in depth analysis of which structures perform best, including other types of networks (e.g. recurrent, convolutional, residual etc.) is left for future research.

Based on the experimental results observed for quantum dots, in conjunction with trail and error, we have used the following network structure:

Layer	Nodes	Activation
Input	96	—
Hidden 1	144	tanh
Hidden 2	36	tanh
Output	1	exp

Once again the number of nodes in the input layer is fixed by the number of degrees of freedom ($N \times D$ for $N = 32$ particles and $D = 3$ dimensions). The number of nodes in the second layer is somewhat arbitrary, and could be changed. Importantly it is larger than $N \times D$, allowing the network to learn more features than it has inputs. Still, it does not have enough nodes to fully encode the relative difference between every pair of coordinates, as would be required to reproduce ψ_M fully. This number (144) was found to be large enough to allow learning, but still be reasonably efficient. Lastly, the second hidden layer was set to 36 nodes. This is also somewhat arbitrary, but importantly it is smaller than the last layer, allowing the gradual narrowing down to the single output. The activation functions are the same as those used for quantum dots.

Finally, we train two instances of the network, with and without input sorting applied to impose symmetry, denoted once again by ψ_{DNN} and ψ_{SDNN}, respectively.

12.2.2 Optimization

We produced the following results by optimizing ψ_M, ψ_{DNN} and ψ_{SDNN} with similar hyperparameters. We initialized $\beta = 2.85$ for all wave functions for a fair comparison. We used 250 000 training iterations of 5000 MC cycles each. Contrary to the quantum dot system, where a large set of hyperparameters yielded good results, we have found this problem to be highly sensitive to good hyperparameters. We had to reduce the learning rate to $\eta = 0.0001$ to avoid divergence during training. We also found that the networks performed better with a small degree of L2 regularization, $\gamma = 0.000\,01$ (Section 5.2.1). While other choices might still be better, there were significantly less room for error in this much more challenging problem.

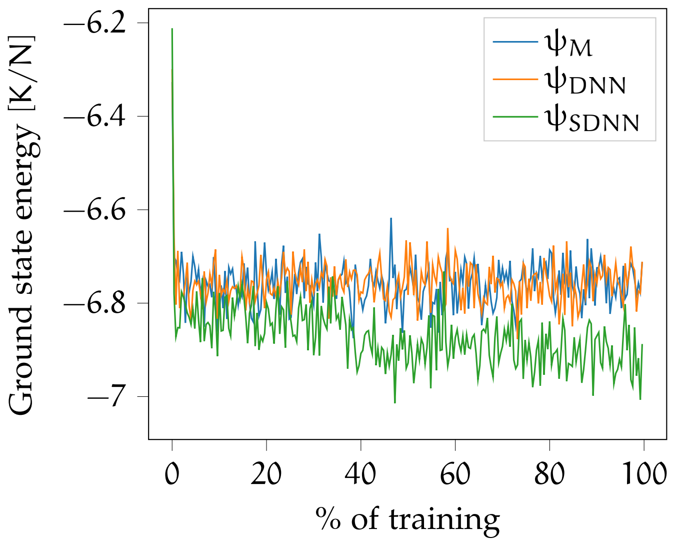


Figure 12.2: Performance of ψ_M, ψ_{DNN} and ψ_{SDNN} as a function of training steps. The network with input sorting, ψ_{SDNN} shows a clear energy reduction compared to the traditional benchmark, albeit not as significant as the results of Chapter 11. The non-symmetric network, ψ_{DNN}, fails to learn anything beyond the benchmark.
Source [1, `writing/scripts/He-mcmillan-dnn.py`]

Fig. 12.2 shows a graph of the ground state energy during the course of training for all three wave functions. On this time scale (i.e. 25 times more training than used for the benchmark alone) we see ψ_M converge immediately to the same energy level as we saw in Fig. 12.1. For ψ_{SDNN}, however, there is an apparent immediate lowering of energy, followed by a gradual decrease throughout the remaining training time. It is not clear if ψ_{SDNN} has completely converged, or if small reductions might still be achieved if we train for even longer. Nevertheless, such gains are likely small and not necessary to demonstrate an improvement.

In contrast to the results seen for QDs, the network without symmetry imposed (ψ_{DNN}) fails to learn anything beyond the benchmark. It appears that for this larger system, imposing the symmetry is vitally important in helping the networks learn the correct correlations. We hypothesize that for only two particles the network could learn the symmetry, allowing more flexible learning, where as in this case the inputs are too complex for the network to learn on its own.

Table 12.2 shows the final energies produced from ψ_M, ψ_{DNN} and ψ_{SDNN} after training. As indicated by the plot, we see a statistically significant lowering of the total energy. Unlike the results for quantum dots, however, we do not get a lowering of the variance, but rather a negligible increase. It is hard to say exactly how much of the total error in energy is corrected for by the inclusion of the networks like we could for QD, due to the lack of existing results using $N = 32$ particles with the Lennard-Jones potential with the same potential corrections that we have included. Nevertheless, the networks continue to show their ability to improve upon the results of all the benchmark wave functions we have paired them with. As our main interest in this work is to demonstrate this general ability, as opposed to producing results closest to experimental results, we view these results as further positive encouragement to continue this line of research.

Table 12.2: Predicted ground state energy of ψ_M, ψ_{DNN} and ψ_{SDNN} after the same amount of optimization. Results obtained from 2^{23} samples and with errors estimated using blocking. Energies in units of Kelvin per particle [K/N]. See Fig. 12.2 for source code reference.

	$\langle E_L \rangle$	CI_-^{95}	CI_+^{95}	Std	Var
ψ_M	$-6.75(2)$	-6.78	-6.72	$3.6 \cdot 10^{-1}$	$1.3 \cdot 10^{-1}$
ψ_{DNN}	$-6.75(1)$	-6.78	-6.72	$4.5 \cdot 10^{-1}$	$2.1 \cdot 10^{-1}$
ψ_{SDNN}	$-6.91(2)$	-6.95	-6.88	$3.8 \cdot 10^{-1}$	$1.4 \cdot 10^{-1}$

Part IV

Conclusion and Outlook

Chapter 13

Conclusion

13.1 Summary

In this work we have presented the underlying theory of variational Monte Carlo (VMC) for the study of many-body quantum mechanical systems, along with fundamental machine learning techniques and the necessary glue that brings the two together. From this theory we have developed a code library suitable for rapid development and experimentation with variations of Hamiltonians, wave functions, sampling strategies, optimization schemes etc., which still performs well enough to handle large scale computations running on supercomputing facilities. The code was thoroughly tested and verified by reproducing known analytical results. Finally we applied our techniques on two different test systems; quantum dots and liquid helium. In both cases we were able to outperform the traditional benchmarks by a significant margin, at the expense of increased computational cost.

13.2 Related Works

The initial motivation for this topic sprung out of the article by Carleo and Troyer [28] which first showed the promise of neural networks (restricted Boltzmann machines (RBMs) in particular) for quantum many-body wave functions. Following this, in a thesis by Flugsrud [33], attempts were made to apply RBMs to quantum dots. This saw some success, but the particular RBM used proved to be insufficient for accurate reproduction of the ground state. At this point the idea of applying more general neural networks started to develop and the work on this thesis started around mid 2018. Since then we have seen a few articles arise with similar ideas. Saito [37] used a simple neural network with one hidden layer to represent the ground state wave function of selected few-body bosonic systems. The approach used here is very comparable to our, except that Saito chose to pre-train the network instead of simply adding it as another correlation factor. Han, Zhang, and E [38] applied a much more complex network to other systems, including systems of many fermions with a kinetic energy term, where part of the complexity was added to account for the Pauli exclusion principle. Others have used so called backflow transformations expressed as neural networks, also with encouraging results [9, 39].

13.3 Our Contributions

The main contributions of this work are as follows:

1. We proposed a new formalism for using arbitrary neural networks as correction factors which can be used in conjunction with established methods.
2. We developed a flexible Python library which enabled rapid development and experimentation with different algorithms and network architectures.
3. We demonstrated the validity of our approach by improving upon the accuracy of the benchmarks for both few-body and many-body systems.
4. In particular, these where continuous space Hamiltonians with a kinetic energy term, which necessitates tractable computation of second order derivatives of the networks. To our knowledge, few authors have selected to work with such terms in the Hamiltonians, possibly due to this very issue.

The obvious downside to applying the techniques developed here is the increased computational cost. Still, we have argued that the complexity of the operations need not scale worse than quadratic in the number of particles, $\mathcal{O}(N^2)$. Of course, asymptotic scaling is not always the most important, as there can be a preemptively large constant increase. However, we have demonstrated that it is more than feasible for many-body problems already. With proper integration of graphics processing units (GPUs) we expect that this barrier becomes even smaller. After all, a constant cost increase is just that - constant.

The most promising aspect of the results of this work is the indication that we can apply a neural network on top of existing techniques and systematically obtain improved results. If proved to be universally true, this means that our technique is not one which can be outperformed by later advancements. Rather, it would be a tool to be combined with any other method, always providing increased levels of accuracy. This could serve particularly useful for problems where we have limited theoretical insight and only approximate results. Neural networks have many times demonstrated their ability to learn without prior knowledge, so why not apply them where there is no prior knowledge to be found?

13.4 Future Prospects

In addition to the many areas of this work alone where there are much room for further research, a myriad of articles are surfacing each year with new and novel applications of machine learning within quantum mechanics. As said by Melko et al. in a recent article in Nature: "[...] the exploration of machine learning in many-body physics has just begun" [40].

We would like to emphasize a few of the most compelling areas to continue with this work:

1. Incorporate GPU acceleration. This is perhaps the most important point, if large scale VMC calculations are ever going to utilize neural networks in a meaningful way. How to optimally combine message passing interface (MPI) acceleration on thousands of central processing units (CPUs) together with a (due to costs) much smaller number of GPUs remains a challenging problem.

2. Investigate systems of many fermions. We hypothesize that our approach should still prove useful as a correction factor on Slater determinants, and verifying this should be a high priority.

3. Apply different types of neural networks. We limited our work to feed-forward neural networks, but there are more exotic types which might prove equally or better suited. In particular, convolutional networks might better learn spacial correlations for certain systems, and recurrent networks might be especially suitable for time-dependent VMC extensions.

4. Go deeper into optimization techniques. The methods we have applied are fairly simple, at least compared to the current state of the art. For example, batch normalization, which consistently improves training results, has not been applied. We have not placed much emphasis on this point simply because we wanted to demonstrate *an* improvement, as opposed to obtaining the very best results possible.

Chapter 14

Appendices

A Notation Reference

This thesis strives to use a consistent set of rules for notation throughout. Where possible, standard notation choices have been made so as to be easily comparable to other works. Sometimes, however, we cannot all agree on how things should be written, and this thesis inevitably diverges from some part of the readers preferences. In an attempt to soften the inevitable rage of notation disagreements, this section serves as a reference for all standard notation used in this thesis, such that the meaning of every expression should be unambiguous and clear.

A.I Symbols

l

Symbol	Reads as
$=$	is equal to
$:=$	is defied as
$\equiv$	is equivalent to
$\overset{d}{=}$	is distributed equal to

A.II Scalars, Vectors and Matrices

Scalars: x
Use lowercase symbols.

Vectors: $\mathbf{x}$
All vectors are column vectors, and use lowercase, bold-face symbols.

Matrices: $\mathbf{X}$

Use uppercase, bold-face symbols.

Example: The matrix $\mathbf{X} \in \mathbb{R}^{m\times n}$ is defined by its column vectors $\{\mathbf{x}_i\}_{i=1}^{n}$ (with $\mathbf{x}_i \in \mathbb{R}^m$) such that $\mathbf{X} = (\mathbf{x}_1\ \mathbf{x}_2\ \dots\ \mathbf{x}_n)$ column vectors, such that the scalar X_{ij} is the i'th component of $\mathbf{x}_j$.

In particular, $\mathbf{X}$ is often used throughout this thesis. If not explicitly defined otherwise, $\mathbf{X}$ denotes a matrix of particle coordinates, $\mathbf{X} := (\mathbf{x}_1\ \mathbf{x}_2\ \dots\ \mathbf{x}_N)^T \in \mathbb{R}^{N\times D}$ for N particles in D dimensions. In three dimensions that is:

$$\mathbf{X} := \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_N \end{pmatrix} := \begin{pmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ \vdots & \vdots & \vdots \\ x_N & y_N & z_N \end{pmatrix}. \tag{14.1}$$

A.III Indices

Some indices are meant to be summed over, while others remain fixed throughout a calculation. To help make the distinction between the two, the following sequences of indices are used for each class, in decreasing order of precedence:

Summation indices: i, j, a, b, c, d, e, f

Fixed indices: k, l

A.IV Summation

Scope

The summation symbol, $\sum$, effects only the term imediately after it. That is,

$$\sum_{i=1}^{3} i+1 = \left(\sum_{i=1}^{3} i\right) + 1 = 7, \quad \text{not} \quad \cancel{\sum_{i=1}^{3} i+1 = \sum_{i=1}^{3} (i+1) = 9} \tag{14.2}$$

Implied Summations

For notational brevity (and to avoid visual clutter), summations are not always stated explicitly as in the above example. In these cases the limits of the summation are determined by their context.

Explicit limits:
In $\sum_{i=1}^{n}$, the sum goes from 1 to n, inclusively.

Implicit limits:
When limits are clearly defied by their context, the following is equivalent: $\sum_{i=1}^{n} \equiv \sum_{i}^{n} \equiv \sum_{i}$.

Implicit summation symbols:
We make heavy use of the Einstein summation convention when this is appropriate. Any summation index (see above) which appears more than once *within a single term* is implicitly summed over all its possible values. Superscripts in parenthesis are excluded from this rule, i.e. $a^{(k)}b^{(k)}$ does not have an implied sum.

Example: Consider the matrix equation $\mathbf{x} = \mathbf{A}\mathbf{b}$, for $\mathbf{A} \in \mathbb{R}^{m\times n}$ and $\mathbf{x}, \mathbf{b} \in \mathbb{R}^{n}$. Written explicitly, all of the following statements about the elements of $\mathbf{x}$ are equivalent:

$$\begin{aligned} x_k &= \sum_{i=1}^{n} A_{ki} b_i, & x_k &= \sum_{i}^{n} A_{ki} b_i, \\ x_k &= \sum_{i} A_{ki} b_i, & x_k &= A_{ki} b_i \end{aligned} \tag{14.3}$$

A.V Statistics

Common statistical quantities use the following notation:

Quantity	Notation
Expectation value	$\langle\cdot\rangle$
Standard deviation	$\mathrm{Std}[\cdot]$
Variance	$\mathrm{Var}[\cdot]$
Covariance	$\mathrm{Cov}[\cdot]$
Standard error of the mean (SEM)	$\mathrm{Se}[\cdot]$

B Symmetry Metric

In the context of our variational Monte Carlo (VMC) framework, we should in general take care that the ansatz we make for the functional form of Ψ is such that it obeys Theorem 1. Not all proposed wavefunctions will necessary *guarantee* this property, however, and optimization might find a minimum in which this is not strictly true. A such, we shall define a metric to measure the *symmetry-ness* of a wavefunction.

Before giving a definition, there are a couple of desired properties such a metric should have.

1. Fully symmetric functions should yield a distinct, finite value.
2. Fully anti-symmetric functions should yield a distinct, finite value, which must be different from fully symmetric functions.
3. Scale invariant, i.e. scaling the function by some constant factor does not change its symmetry metric.
4. Computationally feasible to evaluate.

Definition B.1. *Permutation Operator.*

Let $\mathcal{P}_n$ denote the set of all permutations of the first n natural numbers, $\{1, 2, \ldots, n\}$, and let $\boldsymbol{\alpha} \in \mathcal{P}_n$ be one of these. Given a function $\Psi(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n)$ of n vectors $\mathbf{x}_i \in \mathbb{R}^d$, we let P_α be an operator with the following property:

$$P_\alpha \Psi(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n) := \Psi(\mathbf{x}_{\alpha_1}, \mathbf{x}_{\alpha_2}, \ldots, \mathbf{x}_{\alpha_n}). \tag{14.4}$$

Definition B.2. *Symmetry Metric.*

Given a function $\Psi(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n)$ of n vectors $\mathbf{x}_i \in \mathbb{R}^d$. We use the notation $d\mathbf{X} := d\mathbf{x}_1 \ldots d\mathbf{x}_n$. The Symmetry Metric of Ψ is then defined as the following:

$$S(\Psi) := \frac{\displaystyle\int_{-\infty}^{\infty} d\mathbf{X} \left| \frac{1}{n!} \sum_{\alpha \in \mathcal{P}_n} P_\alpha \Psi \right|^2}{\displaystyle\int_{-\infty}^{\infty} d\mathbf{X} \max_{\alpha \in \mathcal{P}} |P_\alpha \Psi|^2} \tag{14.5}$$

Corollary 2.1. *The symmetry metric is bounded to the interval $[0, 1]$ for all functions where the integrals are defined.*

Proof. Both the numerator and denominator of Equation 14.5 are integrals of absolute values. Hence they cannot be negative, which proves the lower bound $S(\Psi) \geqslant 0$ for all Ψ where the integrals are defined. Further, the absolute value of the average of a set must be less than or equal to the maximum absolute value of the set, and equal iff. all values are equal. That is,

$$\left| \frac{1}{n!} \sum_{\alpha \in \mathcal{P}_n} P_\alpha \Psi \right|^2 \leqslant \max_{\alpha \in \mathcal{P}} |P_\alpha \Psi|^2. \tag{14.6}$$

From this it follows that $S(\Psi) \leqslant 1$. □

Corollary 2.2. *The symmetry metric is scale invariant, i.e.*

$$S(c\Psi) = S(\Psi), \tag{14.7}$$

for any scalar $c \in \mathbb{C}$.

Proof. Obvious. □

Corollary 2.3. *For a symmetric function* Ψ *we have*

$$S(\Psi) = 1. \tag{14.8}$$

Proof. For a symmetric function we have $P_\alpha \Psi = \Psi$ for all $\boldsymbol{\alpha} \in \mathcal{P}_n$ by definition. The result then follows directly. □

Corollary 2.4. *For a anti-symmetric function* $\overline{\Psi}$ *we have*

$$S(\overline{\Psi}) = 0. \tag{14.9}$$

Proof. Of the $n!$ possible permutations, half can be written as an even number of pairwise exchanges, and half as an odd number. This implies

$$\sum_{\alpha \in \mathcal{P}_n} P_\alpha \Psi = 0, \tag{14.10}$$

and the result follows directly. □

This definition of the symmetry metric has all the attributes we wanted, except the computational cost. The asymptotic complexity of S is $\mathcal{O}(n!)$, which is about as bad as asymptotic complexities get. In addition, the integrals are many-dimensional and intractable for any non-trivial n. In practice we will therefore approximate S by only considering a random subset of all the permutations, and of course approximating the integrals with Monte Carlo integration (MCI) as before. In cases where n is small we may compute the full set.

This now provides a consistent metric for comparing potential wavefunctions in terms of their symmetry. This could be helpful when considering trail wave functions whose functional form does not guarantee symmetry, and to observe to what extent optimizing them changes the symmetry metric favorably.

List of Accronyms

References

[1] B. Samseth. *QFLOW: Quantum Variational Monte Carlo Framework with Neural Networks*. https://github.com/bsamseth/qflow. (2019) (cit. on pp. 16, 30, 31, 47, 48, 49, 61, 69, 72, 87, 88, 91, 99, 101, 107, 108, 109, 110, 111, 112, 113, 115, 120, 122, 123, 124, 125, 131, 134, 137, 139).

[2] W. Pauli. "The Connection Between Spin and Statistics". *Physical Review*. **58**, 716 (1940). (Cit. on p. 24).

[3] M. Hjorth-Jensen. *Computational Physics 2: Variational Monte Carlo methods, Lecture Notes*. (2018). URL: http://compphysics.github.io/ComputationalPhysics2/doc/web/course (cit. on p. 27).

[4] M. Taut. "Two electrons in an external oscillator potential: Particular analytic solutions of a Coulomb correlation problem". *Physical Review A*. **48**, 3561 (1993). (Cit. on p. 27).

[5] D. J. Griffiths and D. F. Schroeter. *Introduction to Quantum Mechanics*. 3rd ed. Cambridge University Press, (2018) (cit. on p. 27).

[6] J. Katriel and H. Montgomery. "Hund's rule in the two-electron quantum dot". *European Physical Journal B*. **85**, 394 (2012). (Cit. on p. 29).

[7] M. H. Kalos et al. "Modern potentials and the properties of condensedHe4". *Physical Review B*. **24**, 115 (1981). (Cit. on pp. 29, 31).

[8] R. A. Aziz et al. "An accurate intermolecular potential for helium". *The Journal of Chemical Physics*. **70**, 4330 (1979). (Cit. on pp. 30, 31).

[9] M. Ruggeri, S. Moroni, and M. Holzmann. "Nonlinear Network Description for Many-Body Quantum Systems in Continuous Space". *Physical Review Letters*. **120**, 205302 (2018). (Cit. on pp. 31, 144).

[10] J. C. Slater. "The Theory of Complex Spectra". *Physical Review*. **34**, 1293 (1929). (Cit. on p. 38).

[11] P. A. M. Dirac. "On the Theory of Quantum Mechanics". *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*. **112**, 661 (1926). (Cit. on p. 38).

[12] R. Jastrow. "Many-Body Problem with Strong Forces". *Physical Review*. **98**, 1479 (1955). (Cit. on p. 39).

[13] N. D. Drummond, M. D. Towler, and R. J. Needs. "Jastrow correlation factor for atoms, molecules, and solids". *Physical Review B*. **70**, 235119 (2004). (Cit. on p. 39).

[14] W. H. Press et al. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. 3rd ed. Cambridge University Press, (2007) (cit. on pp. 45, 51).

[15] M. Jonsson. "Standard error estimation by an automated blocking method". *Physical Review E*. **98**, 043304 (2018). (Cit. on pp. 46, 47, 107, 114, 120, 126).

[16] D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. 3rd ed. Addison-Wesley Longman Publishing, (1997) (cit. on p. 51).

[17] D. S. Lemons and A. Gythiel. "Paul Langevin's 1908 paper "On the Theory of Brownian Motion" ["Sur la théorie du mouvement brownien," C. R. Acad. Sci. (Paris) 146, 530–533 (1908)]". *American Journal of Physics*. **65**, 1079 (1997). (Cit. on p. 55).

[18] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. "Learning representations by back-propagating errors". *Nature*. **323**, 533 (1986). (Cit. on p. 65).

[19] B. T. Polyak and A. B. Juditsky. "Acceleration of Stochastic Approximation by Averaging". *SIAM Journal on Control and Optimization*. **30**, 838 (1992). (Cit. on p. 65).

[20] D. P. Kingma and J. Ba. "Adam: A Method for Stochastic Optimization". *arXiv e-prints*, arXiv:1412.6980 (2014). (Cit. on p. 66).

[21] J. Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". *arXiv e-prints*, arXiv:1810.04805 (2018). (Cit. on p. 68).

[22] Y. Huang et al. "GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism". *arXiv e-prints*, arXiv:1811.06965 (2018). (Cit. on p. 68).

[23] D. Silver et al. "Mastering the game of Go without human knowledge". *Nature*. **550**, 354 (2017). (Cit. on p. 68).

[24] D. Silver et al. "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play". *Science*. **362**, 1140 (2018). (Cit. on p. 68).

[25] O. Vinyals et al. *AlphaStar: Mastering the Real-Time Strategy Game StarCraft II*. https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/. (2019). (Visited on 06/01/2019) (cit. on p. 68).

[26] G. Cybenko. "Approximation by superpositions of a sigmoidal function". *Mathematics of Control, Signals, and Systems*. **2**, 303 (1989). (Cit. on p. 71).

[27] B. Hanin. "Universal Function Approximation by Deep Neural Nets with Bounded Width and ReLU Activations". *arXiv e-prints*, arXiv:1708.02691 (2017). (Cit. on p. 71).

[28] G. Carleo and M. Troyer. "Solving the quantum many-body problem with artificial neural networks". *Science*. **355**, 602 (2017). (Cit. on pp. 75, 76, 124, 143).

[29] A. Y. Ng and M. I. Jordan. "On Discriminative vs. Generative Classifiers: A Comparison of Logistic Regression and Naive Bayes". In: *Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic*. NIPS'01. Vancouver, British Columbia, Canada: MIT Press, (2001), 841–848 (cit. on pp. 76, 77).

[30] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. (2015). URL: https://www.tensorflow.org/ (cit. on p. 93).

[31] H. Berland. *Automatic Differentiation*. http://www.robots.ox.ac.uk/~tvg/publications/talks/autodiff.pdf. (2006). (Visited on 05/01/2019) (cit. on p. 96).

[32] M. Wang and A. Pothen. *An Overview of High Order Reverse Mode*. https://openreview.net/pdf?id=Hkmj6tzRZ. (2017). (Visited on 06/23/2019) (cit. on p. 97).

[33] V. M. Flugsrud. *Solving Quantum Mechanical Problems with Machine Learning*. (2018). URL: http://urn.nb.no/URN:NBN:no-68503 (cit. on pp. 106, 124, 143).

[34] T. Kato. "On the eigenfunctions of many-particle systems in quantum mechanics". *Communications on Pure and Applied Mathematics*. **10**, 151–177 (1957). (Cit. on p. 119).

[35] M. Pedersen Lohne et al. "Ab initio computation of the energies of circular quantum dots". *Physical Review B*. **84**, 115302 (2011). (Cit. on p. 130).

[36] W. L. McMillan. "Ground State of LiquidHe4". *Physical Review*. **138**, A442 (1965). (Cit. on p. 135).

[37] H. Saito. "Method to Solve Quantum Few-Body Problems with Artificial Neural Networks". *Journal of the Physical Society of Japan*. **87**, 074002 (2018). (Cit. on pp. 143, 144).

[38] J. Han, L. Zhang, and W. E. “Solving Many-Electron Schrödinger Equation Using Deep Neural Networks”. *arXiv e-prints*, arXiv:1807.07014 (2018). (Cit. on p. 144).

[39] D. Luo and B. K. Clark. “Backflow Transformations via Neural Networks for Quantum Many-Body Wave Functions”. *Physical Review Letters*. **122**, 226401 (2019). (Cit. on p. 144).

[40] R. G. Melko et al. “Restricted Boltzmann machines in quantum physics”. *Nature Physics*. **15**, 887 (2019). (Cit. on p. 145).