CLASSICAL MOLECULAR DYNAMICS USING NEURAL NETWORK REPRESENTATIONS OF ATOMIC POTENTIAL ENERGY SURFACES

by

Andreas Godø Lefdalsnes

THESIS

for the degree of

MASTER OF SCIENCE

Faculty of Mathematics and Natural Sciences
University of Oslo

August 25, 2019

Abstract

Artificial neural networks are fitted to molecular dynamics trajectories using the Behler-Parrinello method of atom-centered symmetry functions in order to obtain analytical interatomic potentials. Molecular dynamics trajectories are generated using the Atomic Simulation Environment (ASE) and the neural networks are initialized and trained using the Atomistic Machine-Learning Package (AMP). AMP is interfaced with ASE through the Calculator interface, which is a black box that accepts atomic numbers and atomic positions and calculates the energy and, if implemented, forces and stresses.

Neural network potentials are constructed for copper and silicon in equilibrium crystal structures, and are evaluated on the potential energy, energy conservation, radial distribution function and mean squared displacement, as well as the absolute errors of the potential energies and force components on the test trajectories. We find the neural networks are able to reproduce the crystal structures, but obtain negative results for the ability to conserve energy, leading to an increase in kinetic energy and translational momentum over time, with negative implications for long-term numerical stability. Recommendations for future work include better sampling algorithms for sampling likely configurations out of equilibrium, testing different numerical optimization algorithms and a more efficient implementation of the Behler-Parrinello symmetry functions for facilitating faster training and deployment of different architectures on available training data, as well as on new input data.

Contents

Chapter 1

Introduction

The field of artificial intelligence (AI) has experienced a tremendous growth in recent years, with a large growth in the number of published papers, and an influx in funding from both universities and commercial entities. The exponential growth of available and high quality data has brough about a demand for accurate, efficient and semi-automated algorithms, capable of building complicated mathematical models and making predictions without human supervision. Artificial neural networks (ANNs) in particular, partially inspired by simple models of human cognition, have found success in a large variety of tasks, largely owing to their ability to scale efficiently as the size of the dataset grows. Many trace this recent AI renaissance to AlexNet[1], which was a convolutional neural network that won the Imagenet Large Scale Visual Recognition Challenge in 2012[1]. Artificial intelligence, or machine learning (ML) as it is known within the literature, can now boast state-of-the-art performance in areas such as image recognition and analysis, computer vision, speech recognition and natural language processing[2].

Data analysis and machine learning has found application in many subfields of physics, such as observational astronomy, condensed matter, and subatomic particle physics [3][4][5]. Quantum chemistry, and electronic structure calculations in particular is an area which could be well suited for the adoption of machine learning methods, as large datasets can be produced on demand and relatively free of noise which obscures patterns.

[1] Imagenet Large Scale Visual Recognition Challenge

1.0.1 Electronic structure calculations

Electronic structure calculations - or numerical solutions to the many-body Schrödinger equation - are methods of modelling physical systems based on the principles of quantum mechanics. They involve very few parameters, and the results are found to be very accurate compared to laboratory measurements. However, exponential time complexity as the system grows in size greatly limits their applications, and we are in general limited to very small and simple systems. Approximations can be made which reduces the complexity to polynomial, but this also limits the accuracy of the method. The most succesful and widely applied methods are the Density Functional Theory (DFT) method and the Hartree-Fock (HF) family of methods. The Hartree-Fock method's time complexity scales nominally as $\mathcal{O}(N^4)$, where N is the number of particles in the system (i.e. electrons), owing to the number of two-electron integrals which must be computed. However, as the system grows in size many of these spatial integrals become vanishingly small and can be neglected, at the cost of introducing a small and adjustable error. A major weakness of Hartree-Fock methods is the neglect of electron correlation energies, which can lead to large deviations from experiment. The Density Functional Theory method scales in a similar manner, but often with a larger proportionality term. Density Functional Theory can treat both exchange and correlation interactions, but this must be done approximately, and developing accurate and numerically stable approximations remains an active area of research [6].

Molecular Dynamics (MD) freezes out the degrees of freedom of the electrons, and treats the atoms as point particles centered on the nucleus using a mean-field approach. The dynamics of the particles are treated as Newtonian, meaning the position and velocity can be integrated using a symplectic integrator such as Verlet integration (see for example **??**). This treatment allows for systems much larger than ab-initio methods, but at the cost of neglecting both nuclear and electronic phenomena. However, the development of molecular dynamics potentials involves a large amount of parameters for any realistic system, and determining their functional form is more of an art than an exact science.

An application of machine learning which has shown some promise is the development of molecular dynamics potentials or Potential Energy Surfaces (PES)[7]. Using for example artificial neural networks, the parameters (i.e. weights and biases) are fully determined from the training data, and the

amount of handcrafting involved can be greatly reduced. Under mild assumptions imposed on the activation functions in the hidden layers, ANNs have been proven to be able to approximate any continuous functions on compact subsets of $\mathbb{R}^N$[8]. This offers the potential of developing potential energy surfaces directly from ab-initio data or from mature and tested empirical potentials.

1.0.2 Atom-centered descriptors

For electronic structure calculations the potential energy, forces and other derived properties are determined by the cartesian coordinates. However, it is not sufficient to feed neural networks cartesian coordinates labeled with the energy. From physical theory we know that any molecular dynamics potential must be invariant w.r.t. translation, rotation, and permutation of like atoms. In order to apply machine learning methods we require a mapping from the cartesian coordinates to a 1D feature vector which conserves all of these properties.

In their article introducing the Atomistic Machine-learning Package (AMP), the authors [9, Khorshidi, Peterson] describe the general process. The idea is to approximate the potential energy with a regression model:

$$\{\boldsymbol{R}\} \overset{\text{regression}}{\longrightarrow} E = E\left(\{\boldsymbol{R}\}\right), \tag{1.1}$$

where $\{\boldsymbol{R}\}$ represents the cartesian coordinates of our system. We will refer to a mapping which satisfies the constraints of translation, rotation and permutation invariance as a *descriptor*. The descriptor is a multidimensional function $\boldsymbol{G}$ that serves as input to the regression method:

$$\{\boldsymbol{R}\} \rightarrow \boldsymbol{G}\left(\{\boldsymbol{R}\}\right) \overset{\text{regression}}{\longrightarrow} E = E\left(\boldsymbol{G}\left(\{\boldsymbol{R}\}\right)\right). \tag{1.2}$$

Once we have a descriptor and a regression model the dynamics can be readily obtained by taking derivatives. The force on atom i is calculated as:

$$\begin{aligned} \boldsymbol{F}_i &= -\nabla_i E \\ &= -\nabla_i E(\boldsymbol{G}(\{\boldsymbol{R}\})) \\ &= -\sum_j \frac{\partial E}{\partial G_j}\frac{\partial G_j}{\partial \boldsymbol{R}_i}, \end{aligned} \tag{1.3}$$

where we have applied the chain rule to break the gradient into derivatives with respect to the network inputs and derivatives of the network inputs with respect to the coordinates of atom i. The derivatives with respect to the network inputs are obtained through backpropagation [10], while the derivatives of the network inputs can be obtained analytically or numerically. The potential energy is typically also broken into atomic contributions:

$$E = \sum_{i=1}^{N} E_{\text{atom}} \left(\boldsymbol{R}_i, \{\boldsymbol{R}\} \right), \tag{1.4}$$

where each atomic contribution is determined by the atom's local environment. This allows us to treat systems with a varying amount of particles without retraining the machine learning method each time.

1.0.3 Goals

The goal of this thesis was to investigate machine learning as a tool for bridging the gap between ab-initio electronic structure calculations and molecular dynamics. Previous theses at the University of Oslo's Computational Physics group have investigated the Behler-Parrinello [11] method of atom-centered symmetry functions, in particular the theses of [12, Stende, John A,] and [13, Treider, Håkon Vikør]. In these theses they have evaluated the numerical speed and accuracy of potentials trained on data produced from the Lennard-Jones and Stillinger-Weber potentials. We sought to continue this work, and demonstrate and validate machine-learned potentials developed using the Behler-Parrinello method, but also apply the method to data obtained using ab-initio molecular dynamics and compare these potentials to empirical potentials. This was initially attemped using a combination of Tensorflow[2] and LAMMPS[3]. However, this was later abandoned, as connecting software packages written in different programming languages proved to be tedious and error-prone, and the Atomic Simulation Environment (ASE[4]) package provided a Python[5] interface to electronic structure calculations. Initially we sought to implement and test the Deep Potential Molecular Dynamics (DPMD) method by [14, Zhang et al.], but it proved difficult to obtain acceptable

[2] https://www.tensorflow.org/
[3] https://lammps.sandia.gov/
[4] https://wiki.fysik.dtu.dk/ase/
[5] https://www.python.org/

results for the energy and force root mean squared errors (RMSEs), and calculating the forces efficiently proved too difficult to implement. This was therefore abandoned in favor of the more tried and tested Behler-Parrinello method.

We finally settled on a combination of using the ASE package for producing molecular dynamics trajectories and the Atomistic Machine-learning Package (AMP[6]) for training them. The AMP package provides an interface for generating Behler-Parrinello "fingerprints" of atomic environments and training them using neural networks (and potentially other machine-learning models). This allowed us to focus on the accuracy, speed and scaling when training neural networks on empirical potentials and ab-initio data in molecular dynamics, and how accurately equilibrium properties can be replicated using trained neural networks. Unfortunately, we did not have time to generate ab-initio trajectories, as there are many details which have to be considered when performing density functional theory calculations, and the calculations are very time-consuming. Further work in this area could focus on generating and reproducing trajectories using either Velocity-Verlet dynamics and ground state DFT calculations or time-dependent density functional theory molecular dynamics described for example in the GPAW calculator documentation [7].

1.0.4 Contributions

Ideally we might have liked to make our own implementation of the Behler-Parrinello method, and I would have especially liked to build a neural network interface through the recently released Tensorflow 2.0 (beta). Unfortunately there was no time for this, and we settled for working through the AMP interface and making modifications as necessary. Developing code from scratch could have given insights into the structure and process of calculating fingerprints and feeding them through neural networks to produce energies and forces. However, the nature of AMP is modular and the code is well documented and readable, so settling on making modifications to the package can allow one to focus more on the process of generating and sampling data, training and deploying the trained neural networks on novel systems and evaluating the results. Suggestions for future work on the topic is included in our final conclusion. In this thesis we have:

[6] https://amp.readthedocs.io/en/latest/
[7] Ehrenfest theory

- Generated and sampled from classical molecular dynamics trajectories
- Trained and tested neural network potentials on data generated from molecular dynamics
- Evaluated the ability of neural network potentials to reproduce equilibrium properties of classical molecular simulations
- Written a set of analysis and post-processing scripts
- Minor extensions and modifications to the AMP library for personal use

1.0.5 Structure

This thesis is divided into four parts. First we build from first principles to connect classical molecular dynamics with the laws of quantum mechanics. We go through the basics of machine learning and connect it to molecular dynamics through the concept of atom-centered descriptors. Then we we discuss the implementation details of using ASE, AMP and Tensorflow. Third we train neural networks potentials on empirical potentials and generate new molecular dynamics trajectories using these potentials. Fourth we conclude the thesis and discuss future applications and work.

Part I

Theory

Chapter 2

Quantum Mechanics

In order to proceed to electronic structure calculations we require a solid foundation in the principles of quantum mechanics. This chapter will give a brief overview of the basic tenets of quantum mechanics and describe briefly how these rules lead to the Schrödinger equation, which is the equation governing all non-relativistic quantum mechanics. We will assume an undergraduate understanding of calculus and linear algebra, and some knowledge of mechanics is also helpful. The discussion in this chapter follows closely and summarizes the discussion in [Sakurai 15, pages 10-76], and the reader is referred there for more details.

2.0.1 Kets and bras

In quantum mechanics, the state of a quantum system is represented by a *state vector* in a complex vector space. Such a vector is called a *ket*, denoted by $|\alpha\rangle$, following the notation of Paul Dirac. The state ket is postulated to contain all information about the state of the quantum system, such as energy, angular momentum, mass and so on. Two kets can be added to produce a new ket:

$$|\alpha\rangle + |\beta\rangle = |\gamma\rangle \,. \tag{2.1}$$

They can also be multiplied by a complex number:

$$c\,|\alpha\rangle = |\alpha\rangle\, c = |\delta\rangle \,. \tag{2.2}$$

If c is zero the resulting ket is called a *null ket*. If c is non-zero it is postulated that the resulting ket contains the same information as the initial ket.

Observables such as momentum and spin are represented by operators acting on the vector space in question. Operators act on a ket from the left to produce a new ket:

$$A\left|\alpha\right\rangle = \left|\delta\right\rangle . \tag{2.3}$$

Of particular importance is when the action of an operator on a ket is the same as multiplication:

$$A\left|\alpha\right\rangle = c\left|\alpha\right\rangle = \left|\delta\right\rangle . \tag{2.4}$$

These kets are known as *eigenkets* and the corresponding complex numbers are known as *eigenvalues*. The physical state represented by an eigenket is known as an *eigenstate*. The eigenvalues of an operator A represent the only possible values of a measurement of the observable. For observables such as position and momentum, the operators will have a continuous spectrum of eigenvalues, whereas operators such as energy and spin have a discrete or *quantized* spectrum, whereby the term *quantum* mechanics is derived. The eigenkets of a physical observable form a complete orthogonal set, meaning any ket can be written as an expansion of eigenkets $\left|a'\right\rangle$:

$$\left|\alpha\right\rangle = \sum_{a'} c_{a'}\left|a'\right\rangle , \tag{2.5}$$

where $c_{a'}$ is a complex coefficient. In principle there are infinitely many linearly indepedent eigenkets, depending on the dimensionality of the vector space.

A *bra space* is a vector space "dual" to the ket space. We postulate that for every ket $\left|\alpha\right\rangle$ there exists a bra $\left\langle\alpha\right|$. The bra space is spanned by eigenbras $\left\langle a'\right|$ corresponding to the eigenkets $\left|a'\right\rangle$. The ket and bra spaces have a dual correspondence:

$$\begin{gathered} \left|\alpha\right\rangle \leftrightarrow \left\langle\alpha\right| \\ \left|\alpha'\right\rangle , \left|\alpha''\right\rangle , \ldots \leftrightarrow \left\langle\alpha'\right| , \left\langle\alpha''\right| , \ldots \\ \left|\alpha\right\rangle + \left|\beta\right\rangle \leftrightarrow \left\langle\alpha\right| + \left\langle\beta\right| . \end{gathered} \tag{2.6}$$

The bra dual to $c\left|\alpha\right\rangle$ is postulated to be $c^{*}\left|\alpha\right\rangle$, and more generally:

$$c_{\alpha}\left|\alpha\right\rangle + c_{\beta}\left|\beta\right\rangle \leftrightarrow c_{\alpha}^{*}\left\langle\alpha\right| + c_{\beta}^{*}\left\langle\beta\right| . \tag{2.7}$$

The *inner product* of a bra and a ket is a complex number written as a bra on the left and a ket on the right. It has the fundamental property:

$$\left\langle\alpha|\beta\right\rangle = \left\langle\beta|\alpha\right\rangle^{*} , \tag{2.8}$$

meaning they are complex conjugates. For this to satisfy the requirements of an inner product we must have

$$\langle\alpha|\alpha\rangle \geq 0, \tag{2.9}$$

with equality if and only if $|\alpha\rangle$ is a null ket. We define the *norm* of a ket as

$$\sqrt{\langle\alpha|\alpha\rangle}, \tag{2.10}$$

which can be used to form normalized kets

$$|\widetilde{\alpha}\rangle = \frac{1}{\sqrt{\langle\alpha|\alpha\rangle}}\,|\alpha\rangle\,, \tag{2.11}$$

with the property

$$\langle\widetilde{\alpha}|\widetilde{\alpha}\rangle = 1. \tag{2.12}$$

Two kets are said to be *orthogonal* if

$$\langle\alpha|\beta\rangle = 0. \tag{2.13}$$

2.0.2 Operators

As we mentioned briefly above, operators act on kets from the left to produce a new ket. Two operators A and B are equal $A = B$ if

$$A\,|\alpha\rangle = B\,|\alpha\rangle\,, \tag{2.14}$$

for an arbitrary ket in the relevant ket space. An operator A is said to be the *null operator* if

$$A\,|\alpha\rangle = 0. \tag{2.15}$$

Operators can be added, and addition operations are commutative and associative.

$$X + Y = Y + X, \tag{2.16}$$

$$(X + Y) + Z = X + (Y + Z). \tag{2.17}$$

Operators act on bras from the right to produce a new bra

$$\langle\alpha|\,A = \langle\beta|\,. \tag{2.18}$$

The ket $A|\alpha\rangle$ and the bra $\langle\alpha|A$ are in general not dual to each other. We define the *hermitian adjoint* $A^\dagger$ through the dual correspondence:

$$A|\alpha\rangle \leftrightarrow \langle\alpha|A^\dagger. \tag{2.19}$$

An operator is said to be *hermitian* if

$$A = A^\dagger. \tag{2.20}$$

Hermitian operators have real eigenvalues, and since the result of any measurement must be a real number any operator that represents a physical observable must be Hermitian.
Operators can be multiplied. Multiplication is associative, but non-commutative:

$$XY \neq YX, \tag{2.21}$$

$$X(YZ) = (XY)Z. \tag{2.22}$$

The left product of a ket and a bra is known as the *outer product*:

$$|\alpha\rangle\langle\beta|. \tag{2.23}$$

The outer product should be treated as an operator, while the inner product $\langle\alpha|\beta\rangle$ is a complex number. If an operator is to the left of a ket $|\alpha\rangle A$ or to the right of a bra $A\langle\beta|$ these are illegal products, in other words not defined within the ruleset of quantum mechanics. The associative properties of operators are postulated to hold true as long as we are dealing with legal multiplications among kets, bras and operators. As an example, the outer product acting on a ket:

$$(|\alpha\rangle\langle\beta|)|\gamma\rangle, \tag{2.24}$$

can be equivalently regarded as scalar multiplication

$$|\alpha\rangle(\langle\alpha|\gamma\rangle) = |\alpha\rangle c = c|\alpha\rangle, \tag{2.25}$$

where $c = \langle\alpha|\gamma\rangle$ is just a complex number.

2.0.3 Time evolution

In quantum mechanics, time is treated not as an observable, but as a parameter. Relativistic quantum mechanics treats space and time on the same footing, but only by demoting position to a parameter.

Suppose we have a physical system $|\alpha\rangle$ at a time t_0. Denote the ket at a later time $t > t_0$ by

$$|\alpha, t; t_0\rangle \,. \tag{2.26}$$

Time evolution is assumed to be continuous and symmetric, meaning that if we evolve the system backwards in time we should arrive at the initial state:

$$\lim_{t \to t_0} |\alpha, t; t_0\rangle = |\alpha\rangle \,. \tag{2.27}$$

The kets separated by a time $\Delta t = t - t_0$ are related by the *time-evolution operator* $\mathcal{U}$:

$$|\alpha, t; t_0\rangle = \mathcal{U}(t, t_0) \, |\alpha, t_0\rangle \,. \tag{2.28}$$

If the state ket is normalized to unity at a time t_0, it must remain normalized at a later time:

$$\langle \alpha, t_0 | \alpha, t_0 \rangle = \langle \alpha, t; t_0 | \alpha, t; t_0 \rangle = 1. \tag{2.29}$$

This is guaranteed if the time evolution operator $\mathcal{U}$ is a *unitary* operator:

$$\mathcal{U}^\dagger \mathcal{U} = 1. \tag{2.30}$$

We also require that the time evolution operator exhibits a composition property:

$$\mathcal{U}(t_2, t_0) = \mathcal{U}(t_2, t_1) \mathcal{U}(t_1, t_0), \quad (t_2 > t_1 > t_0), \tag{2.31}$$

meaning that the time evolution between two points t_0 and t_2 remains the same if we first evolve the system to an intermediate time t_1.

If we consider an infinitesimal time-evolution operator

$$|\alpha, t_0 + dt; t_0\rangle = \mathcal{U}(t_0 + dt, t_0) \, |\alpha, t_0\rangle \,, \tag{2.32}$$

it must reduce to the identity operator as the infinitesimal time interval dt goes to zero:

$$\lim_{dt \to 0} \mathcal{U}(t_0 + dt, t_0) = 1, \tag{2.33}$$

and we expect the difference between the operators to be of first order in dt. These requirements are all satisfied by the operator

$$\mathcal{U}(t_0 + dt, t_0) = 1 - i\Omega dt, \tag{2.34}$$

where Ω is a Hermitian operator:

$$\Omega^\dagger = \Omega. \tag{2.35}$$

The operator Ω has the dimension inverse time. Frequency or inverse time is related to energy through the Planck-Einstein relation:

$$E = \hbar\omega. \tag{2.36}$$

In classical mechanics the Hamiltonian is the generator of time evolution, so we postulate that Ω is related to the Hamiltonian operator H:

$$\Omega = \frac{H}{\hbar}. \tag{2.37}$$

The Hamiltonian operator represents the energy of our system, which is a physical observable and must therefore be Hermitian.

2.0.4 The Schrödinger equation

The Schrödinger equation is the fundamental equation governing non-relativistic quantum mechanics. It can be assumed as a postulate, but is usually derived from more fundamental principles. By exploiting the composition property of the time-evolution operator we find that:

$$\mathcal{U}(t + dt, t_0) = \mathcal{U}(t + dt, t)\mathcal{U}(t, t_0) = (1 - \frac{iHdt}{\hbar})\mathcal{U}(t, t_0), \tag{2.38}$$

where the time difference $t - t_0$ is not required to be infinitesimal. Subtracting from both sides of this equation:

$$\mathcal{U}(t + dt, t_0) - \mathcal{U}(t, t_0) = -\frac{iHdt}{\hbar}\mathcal{U}(t, t_0). \tag{2.39}$$

Rearranging this equation and taking the limit $dt \to 0$ leads to the equation:

$$i\hbar\frac{\partial}{\partial t}\mathcal{U}(t, t_0) = H\mathcal{U}(t, t_0). \tag{2.40}$$

This is known as the Schrödinger equation for the time-evolution operator. We multiply both sides by a ket $|\alpha, t_0\rangle$:

$$i\hbar\frac{\partial}{\partial t}\mathcal{U}(t,t_0)\,|\alpha,t_0\rangle = H\mathcal{U}(t,t_0)\,|\alpha,t_0\rangle\,. \tag{2.41}$$

This ket does not depend on t, leading us to the famous equation:

$$i\hbar\frac{\partial}{\partial t}\,|\alpha,t;t_0\rangle = H\,|\alpha,t;t_0\rangle\,. \tag{2.42}$$

This is known as the time-dependent Schrödinger equation, and gives the description for how a quantum system evolves with time. It is possible to show that in the classical limit $\hbar \to 0$, the expectation value of the operator H takes on the role of the energy in classical mechanics. The Schrödinger equation takes on the role of Newton's laws in quantum mechanics. However, it is not the only way to study quantum systems, as it has been shown to be an equivalent interpretation to the matrix mechanics of Werner Heisenberg and the path-integral formulation developed by Richard Feynman.

Chapter 3

Many-body Quantum Mechanics

The Schrödinger equation only offers exact solutions for very small and simple systems, such as the hydrogen atom or a system of non-interacting harmonic oscillators. If we want to study most systems of interest we must turn instead to numerical methods and solutions. The most prominent methods in the field of many-body quantum mechanics are the Hartree-Fock (HF) and the Density Functional Theory (DFT) families of solvers. This chapter will give a brief overview of both methods, focused more on understanding and intuition than rigor. Eventually we will take these methods to be black boxes of electronic structure calculations, given some cartesian coordinates and outputting energies and forces. This section will give a brief overview of the discussion of the electronic Hamiltonian in [Szabo 16, pages 39-89], which also covers the Hartree-Fock theory which will be expanded upon in the next section.

We want to find solutions to the non-relativistic time-independent Schrödinger equation:

$$\hat{H} \left|\Psi\right\rangle = E \left|\Psi\right\rangle, \tag{3.1}$$

with the Hamiltonian $\hat{H}$ describing a system of nuclei and electrons with cartesian coordinates $\boldsymbol{R}_a,\ a = 1, 2, \ldots, A$ and $\boldsymbol{r}_i,\ i = 1, 2, \ldots, N$ respectively. The distance between nuclei a and electron i is given as the euclidean distance $R_{ai} = |\boldsymbol{R}_a - \boldsymbol{r}_i|$ and correspondingly for the nuclei-nuclei and electron-electron distances. The full Hamiltonian for a set of N electrons and A nuclei in atomic

units is:

$$\begin{aligned}\hat{H} = &-\sum_{i=1}^{N}\frac{1}{2}\nabla_i^2 - \sum_{a=1}^{A}\frac{1}{M_a}\nabla_a^2 - \sum_{i=1}^{N}\sum_{a=1}^{A}\frac{Z_a}{R_{ia}} \\ &+ \sum_{i=1}^{N}\sum_{j=i+1}^{N}\frac{1}{r_{ij}} + \sum_{a=1}^{A}\sum_{b=a+1}^{A}\frac{Z_a Z_b}{R_{ab}}.\end{aligned} \tag{3.2}$$

The first two terms describe the kinetic energy operators of the electrons and nuclei, with M_a the ratio of the mass of nuclei a to the electron mass. The third term describes the Coulomb attraction between electrons and nuclei, while the fourth and fifth terms describe the repulsion between electrons and nuclei respectively.

Since the nucleons are approximately 2000 times heavier than the electrons, the electrons can to a good approximation be described as moving in the field of fixed nuclei. In practice we neglect the kinetic energy terms of the nuclei, while considering an averaged effect from the nuclei-nuclei repulsion. The nuclei-nuclei repulsion energy averaged over adds a constant to the energy eigenvalues, but has no effect on the energy eigenfunctions. The remaining terms are known as the electronic Hamiltonian:

$$\hat{H}_e = -\sum_{i=1}^{N}\frac{1}{2}\nabla_i^2 - \sum_{i=1}^{N}\sum_{a=1}^{A}\frac{Z_A}{R_{ia}} + \sum_{i=1}^{N}\sum_{j=i+1}^{N}\frac{1}{r_{ij}}. \tag{3.3}$$

The electronic wavefunction $\Psi_e = \Psi_e(\{r_i\};\{R_a\})$ is a function of the electronic coordinates with a parametric dependence on the fixed nuclear coordinates. The electronic energy is obtained in the usual way $E_e = \langle\Psi_e|\hat{H}_e|\Psi_e\rangle$. The total energy of our system must now include the constant nuclear repulsion:

$$E_{tot} = E_e + \sum_{a=1}^{A}\sum_{b=a+1}^{A}\frac{Z_a Z_b}{R_{ab}}. \tag{3.4}$$

If one has solved the Schrödinger equation for the electronic Hamiltonian, one can subsequently solve for the nuclear motion using the same trick, i.e. substituting the electronic coordinates for their average values, averaged over

the electronic wave function. We are then left with a nuclear Hamiltonian $\hat{H}_n$:

$$\begin{aligned}\hat{H}_n &= -\sum_{a=1}^{A}\frac{1}{2M_a}\nabla_a^2 + \left\langle -\sum_{i=1}^{N}\frac{1}{2}\nabla_i^2 - \sum_{i=1}^{N}\sum_{a=1}^{A}\frac{Z_a}{R_{ia}} + \sum_{i=1}^{N}\sum_{j=i+1}^{N}\frac{1}{r_{ij}}\right\rangle \\ &\quad + \sum_{a=1}^{A}\sum_{b=a+1}^{A}\frac{Z_a Z_b}{R_{ab}} \\ &= -\sum_{a=1}^{A}\frac{1}{2M_a}\nabla_a^2 + E_{tot}.\end{aligned} \tag{3.5}$$

Under this approximation the nuclei move on a potential energy surface obtained by solving the electronic Hamiltonian. We will however remain focused on the electronic structure problem. For the electronic Hamiltonian we would like to make one more approximation. The electronic Hamiltonian further simplifies if instead of considering the relative positions of the nuclei, we fix the coordinate system in the center of mass of the nucleus and neglect the coordinates of the individual nuclear constituents. This gives us the final expression we want for the electronic Hamiltonian:

$$\hat{H} = -\sum_{i=1}^{N}\frac{1}{2}\nabla_i^2 - \sum_{i=1}^{N}\frac{Z}{r_i} + \sum_{i=1}^{N}\sum_{j=i+1}^{N}\frac{1}{r_{ij}} \tag{3.6}$$

3.0.1 Hartree-Fock

The Hartree-Fock method is a method for finding solutions to the electronic Hamiltonian assuming the electron-electron repulsion can be approximated with a set of single-particle functions or *orbitals*, moving in a mean field generated by the presence of other electrons. The theory in this section is based upon the [17, Sherrill] lecture notes from the Georgia Institute of Technology and the lecture notes on Computational Physics II [18, Hjorth-Jensen] from the University of Oslo. Assuming that the electrons do not interact the Hamiltonian is separable and the wavefunction is simply a product of orbitals ψ which are solutions to a onebody Hamiltonian. This gives us an ansatz for the many-body wavefunction Ψ known as the *Hartree product*:

$$\Psi(\boldsymbol{r}_1, \ldots, \boldsymbol{r}_N) = \psi(\boldsymbol{r}_1) \cdot \cdots \cdot \psi(\boldsymbol{r}_N). \tag{3.7}$$

Since we are dealing with fermions this ansatz fails to satisfy the antisymmetry principle, i.e. the wavefunction is not antisymmetric with respect to the interchange of any two particles. Fermions in addition to three spatial degrees of freedom also have a spin degree of freedom σ which means the fermion can be described by the space-spin coordinate $\boldsymbol{x} = (\boldsymbol{r}, \sigma)$ with $\boldsymbol{x} \in \mathbb{R}^3 \otimes \sigma$. The problem of antisymmetry in a system of N fermions is satisfied by the introduction of *Slater determinants*

$$\Psi(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N) = \frac{1}{\sqrt{N!}} \begin{vmatrix} \chi_1(\boldsymbol{x}_1) & \chi_2(\boldsymbol{x}_1) & \ldots & \chi_N(\boldsymbol{x}_1) \\ \chi_1(\boldsymbol{x}_2) & \chi_2(\boldsymbol{x}_2) & \ldots & \chi_N(\boldsymbol{x}_2) \\ \vdots & \vdots & \ddots & \vdots \\ \chi_1(\boldsymbol{x}_N) & \chi_2(\boldsymbol{x}_N) & \ldots & \chi_N(\boldsymbol{x}_N) \end{vmatrix}, \tag{3.8}$$

with $\chi(\boldsymbol{x})$ spin orbitals and a normalization factor $(N!)^{-1/2}$. The introduction of this ansatz is equivalent to assuming that all electrons move independently of each other in a mean field generated by the electron-electron repulsion. Define the one-electron operator of the electronic Hamiltonian as:

$$\hat{h}_1(\boldsymbol{x}_i) = -\frac{1}{2}\nabla_i^2 - \frac{Z}{r_i}, \tag{3.9}$$

with a twobody interaction term

$$\hat{v}(\boldsymbol{x}_i, \boldsymbol{x}_j) = \frac{1}{r_{ij}}, \tag{3.10}$$

which allows us to write the electronic Hamiltonian more compactly as:

$$\hat{H} = \sum_i \hat{h}_1(\boldsymbol{x}_i) + \sum_{i<j} \hat{v}(\boldsymbol{x}_i, \boldsymbol{x}_j). \tag{3.11}$$

The expectation value of the energy is given as the usual inner product:

$$E = \langle \Psi | \hat{H} | \Psi \rangle \,. \tag{3.12}$$

The *variational theorem* states that the expectation value of any normalized wavefunction with respect to the energy represents an upper bound to the ground state energy. This suggests a procedure wherein we vary the parameters of a set of approximate wave functions Ψ_T until an energy minimum

is reached. The Hartree-Fock energy can be written in terms of integrals over the onebody and interaction terms:

$$E_{HF} = \langle \Psi | \hat{H} | \Psi \rangle = \sum_i \langle i | \hat{h} | i \rangle + \sum_{i<j} \langle ij | \hat{v} | ij \rangle_{AS} \,, \tag{3.13}$$

where we have introduced an antisymmetrized matrix element:

$$\langle ij | \hat{v} | ij \rangle_{AS} = \langle ij | \hat{v} | ij \rangle - \langle ij | \hat{v} | ji \rangle \,, \tag{3.14}$$

and the shorthand integrals:

$$\langle i | \hat{h}_1 | i \rangle = \int d\boldsymbol{r} \chi_i^*(\boldsymbol{r}) \hat{h}_1 \chi_i(\boldsymbol{r}), \tag{3.15}$$

and

$$\langle ij | \hat{v} | ij \rangle = \int d\boldsymbol{r}_i d\boldsymbol{r}_j \chi_i^*(\boldsymbol{r}_i) \chi_j^*(\boldsymbol{r}_j) \hat{v} \chi_i(\boldsymbol{r}_i) \chi_j(\boldsymbol{r}_j). \tag{3.16}$$

In order to derive the Hartree-Fock equations we perform a linear expansion of the spin orbitals χ in terms of a fixed orthogonal basis ϕ:

$$\chi_i = \sum_\nu C_{i\nu} \phi_\nu, \tag{3.17}$$

in principle an infinite sum, but in practice truncated. This basis is usually obtained as the eigenfunctions of parts of the electronic Hamiltonian, e.g. solutions to the Schrödinger equation with a harmonic oscillator or Coulomb potential. If the coefficients belong to an orthogonal or unitary matrix, the resulting basis will preserve orthogonality. This expansion allows us to rewrite the Hartree-Fock energy as:

$$E_{HF} = \sum_i \sum_{\alpha\beta} C_{i\alpha}^* C_{i\beta} \langle \alpha | \hat{h}_1 | \beta \rangle + \sum_{i<j} \sum_{\alpha\beta\delta\eta} C_{i\alpha}^* C_{j\beta}^* C_{i\delta} C_{j\eta} \langle \alpha\beta | \hat{v} | \delta\eta \rangle \,. \tag{3.18}$$

Using the method of Lagrangian multipliers we can define a functional to be minimized:

$$\begin{aligned} F[\Psi] &= E_{HF}[\Psi] - \sum_i \epsilon_i \langle i | j \rangle \\ &= E_{HF}[\Psi] - \sum_i \epsilon_i \sum_\alpha C_{i\alpha}^* C_{i\alpha}, \end{aligned} \tag{3.19}$$

with Lagrange multipliers ϵ_i, where we have exploited the orthogonality of the basis functions to introduce the coefficients. The Lagrange multipliers are identified with the energy eigenvalues of the single-particle orbitals. Minimizing with respect to $C^*_{i\alpha}$ yields the eigenvalue equation:

$$\sum_\beta \hat{h}^{HF}_{\alpha\beta} C_{i\beta} = \epsilon_i C_{i\alpha}, \tag{3.20}$$

where we have introduced the Fock matrix elements:

$$\hat{h}^{HF}_{\alpha\beta} = \langle\alpha|\hat{h}_1|\beta\rangle + \sum_j^N \sum_{\delta\eta} C^*_{j\delta} C_{j\eta} \langle\alpha\delta|\hat{v}|\beta\eta\rangle_{AS}\,. \tag{3.21}$$

The single-particle integrals are usually tabulated in advance, and depend upon the choice of basis functions. Often the single-particle integrals $\langle\alpha|\hat{h}_1|\beta\rangle$ have analytical expressions, while the antisymmetric matrix elements must be evaluated using numerical integration. The eigenvalue problem can be written more compactly as:

$$FC = C\epsilon, \tag{3.22}$$

where F is the Fock matrix defined above, C is the matrix of coefficients and ϵ is now a vector of single-particle energies. The Hartree-Fock equations are solved in an iterative way, starting with an initial guess for the coefficients C. Solving the eigenvalue problem yields new eigenvectors and eigenvalues. The process continues until the change in eigenvalues is within some tolerance ν:

$$\frac{\sum_p \left|\epsilon^n_i - \epsilon^{n-1}_i\right|}{m} \leq \nu, \tag{3.23}$$

where p runs over all the single-particle energies and m is the number of single-particle states.

3.0.2 Density-functional theory

Density-functional theory (DFT) is a method for investigating the electronic structure of a many-body system by finding approximations to the ground state density $n(\boldsymbol{r})$. It holds many similarities to the Hartree-Fock method,

since the usual method of obtaining the ground-state density involves constructing a single-determinant wave function from a set of orthonormal single-particle states, and expanding these single-particle states in terms of a known basis. However, DFT methods offer the explicit treatment of both the electron exchange and electron correlation interactions, while Hartree-fock only includes the exchange energies.

This section is a brief summary of the material covered in the [Toulouse 19, pages 1-12] lecture notes from the Université Pierre et Marie Curie, which the reader is encouraged to check out for more detail. Our starting point is again the electronic Hamiltonian:

$$\hat{H} = -\sum_{i=1}^{N} \frac{1}{2}\nabla_i^2 - \sum_{i=1}^{N} \frac{Z}{r_i} + \sum_{i=1}^{N} \sum_{j=i+1}^{N} \frac{1}{r_{ij}}. \tag{3.24}$$

Any electronic wavefunction Ψ which solves this equation is in principle a function of $4N$ coordinates $\boldsymbol{x}_i = (\boldsymbol{r}_i, \sigma_i),\ i = 1, \ldots, N$ where N is the number of electrons. Once we have obtained a solution to the Schrödinger equation we can obtain the one-electron density $n(\boldsymbol{r})$ as:

$$n(\boldsymbol{r}) = N \int |\Psi(\boldsymbol{r}, \sigma, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N)|^2 \, d\sigma d\boldsymbol{x}_2 \ldots d\boldsymbol{x}_N. \tag{3.25}$$

Since the wavefunction is a unique functional of the Hamiltonian $\hat{H}$, the one-electron density is uniquely determined by the Hamiltonian. Hohenberg and Kohn showed in 1964 [20] in their first theorem that this mapping can be inverted, i.e. that the one-electron density uniquely determines the Hamiltonian of our system (up to an arbitrary constant). Taken altogether, this means that all properties of our system, including the Hamiltonian and the many-body wavefunction are fixed by a one-electron density carrying a dependency on only 3 spatial coordinates. The electronic Hamiltonian can be rewritten as:

$$\hat{H} = \hat{F} + \hat{V}_{ne}, \tag{3.26}$$

where $\hat{F}$ is an operator consisting of the kinetic energy and electron-electron operators and $\hat{V}_{ne}$ is the electron-nuclei interaction. For their second theorem, Hohenberg and Kohn defined the universal density-functional:

$$F[n] = \langle \Psi[n] | \hat{F} | \Psi[n] \rangle \,, \tag{3.27}$$

and the total electronic energy functional:

$$E[n] = F[n] + \int \hat{V}_{ne} n(\boldsymbol{r}) d\boldsymbol{r}. \tag{3.28}$$

Hohenberg and Kohn showed that the energy functional with respect to one-electron densitites $n(\boldsymbol{r})$ is an upper bound to the ground state energy:

$$E_0 \leq F[n] + \int \hat{V}_{ne} n(\boldsymbol{r}) d\boldsymbol{r}, \tag{3.29}$$

with equality if and only if the one-electron density is the one-electron density corresponding to the Hamiltonian $\hat{H}$. This suggests a variational procedure, wherein we vary the total electronic energy functional until we reach an energy minimum:

$$E_{min} = \min_{n} E[n], \tag{3.30}$$

which serves as our best estimate for the ground state one-electron density $n(\boldsymbol{r})$. Levy and Lieb [21, 22, 23] proposed to redefine the universal density-functional in terms of normalized antisymmetric wavefunctions Ψ which yield a fixed density n:

$$F[n] = \min_{\Psi \to n} \langle \Psi | \hat{F} | \Psi \rangle = \langle \Psi[n] | \hat{F} | \Psi[n] \rangle \,, \tag{3.31}$$

wherein the minima search is performed over wavefunctions which yield the fixed density n. A search is then performed over densitities n until we reach an energy minimum. This method is known as the *constrained search formulation*. Kohn and Sham [24] proposed to decompose $F[n]$ as:

$$F[n] = T_s[n] + E_{\text{Hxc}}[n], \tag{3.32}$$

where $T_s[n]$ is a non-interacting kinetic-energy functional which can be defined through the constrained-search formulation:

$$T_s[n] = \min_{\Phi \to n} \langle \Phi | \hat{T} | \Phi \rangle = \langle \Phi[n] | \hat{T} | \Phi[n] \rangle \,, \tag{3.33}$$

wherein the minima search is now performed over normalized single-determinant wavefunctions Φ which yield the fixed density n. The functional

$E_{\mathrm{Hxc}}[n]$ is known as the Hartree-exchange-correlation functional. The variational procedure is now performed over single-determinant wavefunctions which yield a fixed density n and then minimized over densities:

$$\begin{aligned} E_0 &= \min_n \left\{ F[n] + \int \hat{V}_{ne} n(\boldsymbol{r}) d\boldsymbol{r} \right\} \\ &= \min_n \left\{ \min_{\Phi \to n} \langle \Phi | \hat{T} | \Phi \rangle + E_{\mathrm{Hxc}}[n] + \int \hat{V}_{ne} n(\boldsymbol{r}) d\boldsymbol{r} \right\} \\ &= \min_n \min_{\Phi \to n} \left\{ \langle \Phi | \hat{T} + \hat{V}_{ne} | \Phi \rangle + E_{\mathrm{Hxc}}[n_\Phi] \right\} \\ &= \min_\Phi \left\{ \langle \Phi | \hat{T} + \hat{V}_{ne} | \Phi \rangle + E_{\mathrm{Hxc}}[n_\Phi] \right\}. \end{aligned} \tag{3.34}$$

These equations now involve only a single-determinant wave function, which is a large simplification over a variational method involving multi-determinant wave functions. Now a major part of the kinetic energy contribution can be treated through the single-determinant wave function, while only the Hartree-exchange-correlation needs to be approximated as a functional of the density. As with the Hartree-Fock method, the single determinant wavefunctions are constructed from an orthonormal basis of spin orbitals $\chi_i(\boldsymbol{x}),\ i = 1, \ldots, M$ with spin-spatial coordinates $\boldsymbol{x}_i = (\boldsymbol{r}_i, \sigma_i)$. The total electronic energy can be expressed in terms of spatial orbitals $\phi_i(\boldsymbol{r})$ after integrating out the spin variables:

$$E[\{\phi_i\}] = \sum_i \int \phi_i^*(\boldsymbol{r}) (-\frac{1}{2}\nabla^2 + \hat{V}_{ne}) \phi_i(\boldsymbol{r}) d\boldsymbol{r} + E_{\mathrm{Hxc}}[n], \tag{3.35}$$

with the density expressed as:

$$n(\boldsymbol{r}) = \sum_i |\phi_i(\boldsymbol{r})|^2 . \tag{3.36}$$

The energy minimum is obtained using the method of Lagrangian multipliers, with the constraint that the spatial orbitals be normalized we introduce the following Lagrangian:

$$\mathcal{L}[\{\phi_i\}] = E[\{\phi_i\}] - \sum_i \epsilon_i \left(\int \phi_i^*(\boldsymbol{r}) \phi_i(\boldsymbol{r}) d\boldsymbol{r} - 1 \right), \tag{3.37}$$

with ϵ_i the associated Lagrangian multiplier. The energy minimum is where the Lagrangian is stationary with respect to the spatial orbitals:

$$\frac{\partial \mathcal{L}}{\partial \phi_i^*(\boldsymbol{r})} = 0, \tag{3.38}$$

which gives us the functional derivative:

$$\left(-\frac{1}{2}\nabla^2 + \hat{V}_{ne}\right)\phi_i(\boldsymbol{r}) + \frac{\partial E_{\text{Hxc}}[n]}{\partial \phi_i^*} = \epsilon_i \phi_i(\boldsymbol{r}). \tag{3.39}$$

The second term can be expressed through the chain rule as:

$$\frac{\partial E_{\text{Hxc}}[n]}{\partial \phi_i^*} = \int \frac{\partial E_{\text{Hxc}}}{\partial n(\boldsymbol{r}')}\frac{\partial n(\boldsymbol{r}')}{\partial \phi_i^*(\boldsymbol{r})} d\boldsymbol{r}'. \tag{3.40}$$

The second factor can be expressed as:

$$\frac{\partial n(\boldsymbol{r}')}{\partial \phi_i^*(\boldsymbol{r})} = \phi_i(\boldsymbol{r})\delta(\boldsymbol{r} - \boldsymbol{r}'). \tag{3.41}$$

Defining the Hartree-exchange-correlation potential as the functional derivative:

$$\hat{V}_{\text{Hxc}} = \frac{\partial E_{\text{Hxc}}}{\partial \phi_i^*(\boldsymbol{r})}, \tag{3.42}$$

we arrive at the Kohn-Sham eigenvalue equations:

$$\left(-\frac{1}{2}\nabla^2 + \hat{V}_{ne} + \hat{V}_{\text{Hxc}}\right)\phi_i(\boldsymbol{r}) = \epsilon_i \phi_i(\boldsymbol{r}). \tag{3.43}$$

The eigenfunctions satisfying these equations are known as the Kohn-Sham orbitals, and are eigenfunctions of the Kohn-Sham one-electron Hamiltonian:

$$\hat{h}_{KS} = -\frac{1}{2}\nabla^2 + \hat{V}_{KS}, \tag{3.44}$$

with the Kohn-Sham potential defined as:

$$\hat{V}_{KS} = \hat{V}_{ne} + \hat{V}_{\text{Hxc}}. \tag{3.45}$$

The Kohn-Sham one-electron Hamiltonian defines a system of N non-interacting electrons in an effective external potential $\hat{V}_{KS}$ ensuring that the one-electron density is the same as that of the ground-state one-electron

density of a system of interacting electrons. The Hartree-exchange-correlation potential is further decomposed as:

$$\hat{V}_{\mathrm{Hxc}} = \hat{V}_{\mathrm{H}} + \hat{V}_{\mathrm{XC}}, \tag{3.46}$$

where $\hat{V}_{\mathrm{H}}$ is the Hartree potential and $\hat{V}_{\mathrm{XC}}$ is the exchange-correlation potential. The Hartree potential can be expressed through the density:

$$\hat{V}_{\mathrm{H}} = \int \frac{n(\boldsymbol{r})}{|\boldsymbol{r} - \boldsymbol{r}'|} d\boldsymbol{r}'. \tag{3.47}$$

The exchange-correlation potential is decomposed into the exchange potential and the correlation potential:

$$\hat{V}_{\mathrm{XC}} = \hat{V}_{\mathrm{X}} + \hat{V}_{\mathrm{C}}. \tag{3.48}$$

For practical calculation the spatial orbitals are expanded as a linear combination of a known basis, such as hydrogen-like functions or Gaussian-type orbitals:

$$\phi_i(\boldsymbol{r}) = \sum_{\nu} C_{\nu i} \chi_i(\boldsymbol{r}). \tag{3.49}$$

We insert these into the Kohn-Sham equations, multiply by $\chi_i^*(\boldsymbol{r})$ and integrate over $\boldsymbol{r}$ to arrive at the eigenvalue equations:

$$\sum_{\nu} F_{\mu\nu} C_{\nu i} = \epsilon_i \sum_{\nu} S_{\mu\nu} C_{\nu i}, \tag{3.50}$$

where $F_{\mu\nu} = \int \chi_\mu^*(\boldsymbol{r}) \hat{h}_{KS} \chi_\nu^*(\boldsymbol{r}) d\boldsymbol{r}$ are the elements of the Kohn-Sham Fock matrix and $S_{\mu\nu} = \int \chi_\mu^*(\boldsymbol{r}) \chi_\nu(\boldsymbol{r}) d\boldsymbol{r}$ are the elements of the overlap matrix of basis elements. The Fock matrix is decomposed into its constituent parts:

$$F_{\mu\nu} = H_{\mu\nu}\ J_{\mu\nu} + V_{XC,\mu\nu}, \tag{3.51}$$

where $H_{\mu\nu}$ are the one-electron integrals:

$$H_{\mu\nu} = \int \chi_\mu^*(\boldsymbol{r}) \left(-\frac{1}{2} \nabla^2 + \hat{V}_{ne}(\boldsymbol{r}) \right) \chi_\mu(\boldsymbol{r}) d\boldsymbol{r}, \tag{3.52}$$

$J_{\mu\nu}$ is the Hartree-potential contribution:

$$J_{\mu\nu} = \sum_{\lambda} \sum_{\gamma} P_{\lambda\gamma} \left(\chi_\mu \chi_\nu | \chi_\lambda \chi_\gamma \right), \tag{3.53}$$

where we have defined the density matrix $P_{\lambda\gamma}$ as:

$$P_{\gamma\lambda} = \sum_i C_{\gamma i} C^*_{\lambda i}, \tag{3.54}$$

and the two-electron integrals are defined as:

$$(\chi_\mu \chi_\nu | \chi_\lambda \chi_\gamma) = \int\int \frac{\chi^*_\mu(\boldsymbol{r}_1)\chi_\nu \chi^*_\lambda(\boldsymbol{r}_2)\chi_\gamma}{|\boldsymbol{r}_1 - \boldsymbol{r}_2|} d\boldsymbol{r}_1 d\boldsymbol{r}_2. \tag{3.55}$$

and finally the exchange-correlation potential contribution:

$$\hat{V}_{\mathrm{XC},\mu\nu} = \int \chi^*_\mu(\boldsymbol{r}) \hat{V}_{\mathrm{XC}}(\boldsymbol{r}) \chi_\mu(\boldsymbol{r}) d\boldsymbol{r}. \tag{3.56}$$

The Kohn-Sham equations can be written more compactly as the matrix equation:

$$FC = SC\epsilon, \tag{3.57}$$

where ϵ is a vector containing the energy eigenvalues of the basis elements. The equations are then solved iteratively through diagonalization to obtain the matrix of coefficients C. Once we have obtained the coefficients the density $n(\boldsymbol{r})$ is calculated as:

$$n(\boldsymbol{r}) = \sum_\gamma \sum_\lambda P_{\gamma\lambda} \chi_\gamma(\boldsymbol{r}) \chi^*_\lambda(\boldsymbol{r}), \tag{3.58}$$

where the summation is over the number of basis functions M. The total electronic energy can now be expressed as:

$$E = \sum_\mu \sum_\nu P_{\nu\mu} H_{\mu\nu} + \frac{1}{2} \sum_\mu \sum_\nu P_{\nu\mu} J_{\mu\nu} + E_{\mathrm{XC}}. \tag{3.59}$$

Typically the energy contribution from the exchange-correlation E_{XC} has a complicated non-linear dependence on the density and must therefore be evaluated through numerical integration.

Chapter 4

Molecular Dynamics

The ab-initio methods discussed in the previous chapter are appropriate for systems with a relatively small amount of electrons, but suffer from cubic time complexity as the number of electrons grows. If we would like to study systems of molecules, nanoscale structures or calculate transport coefficients we have to make further approximations limiting the degrees of freedom of our system. Instead of solving Schrödinger's equation to obtain the wave function, classical approximations are made to treat the atoms as point particles, with their interactions governed by a classical potential. This eliminates both the nucleonic and electronic degrees of freedom and allows for much larger systems to be simulated, though at the cost of quantum effects exhibited by the electrons. In addition, the construction of molecular dynamics potentials involves guessing at a functional form, and typically a large amount of parameters to be determined from experiments and simulations. In this chapter we will show how quantum mechanics and classical mechanics can be bridged semi-rigorously, and also give an introduction to central concepts of molecular dynamics, such as initialization, integration of the equations of motion and construction and evaluation of potentials.

4.0.1 From quantum mechanics to molecular dynamics

We will follow the route described in the lecture notes [Marx, Dominik and Hutter, Jörg 25, pages 1-10] given at the Winterschool 2000 at the John von Neuman Institute for Computing, Jölich. We will be using SI units as we wish to study the equations of motion derived from quantum mechanics but in the classical limit. Our starting point is the full Hamiltonian for a set of

N electrons and A nuclei:

$$\begin{aligned}\hat{H} = &-\sum_{i=1}^{N}\frac{\hbar^2}{2m_e}\nabla_i^2 - \sum_{a=1}^{A}\frac{\hbar^2}{2M_a}\nabla_a^2 - \sum_{i=1}^{N}\sum_{a=1}^{A}\frac{e^2 Z_a}{R_{ia}} \\ &+\sum_{i=1}^{N}\sum_{j=i+1}^{N}\frac{e^2}{r_{ij}} + \sum_{a=1}^{A}\sum_{b=a+1}^{A}\frac{e^2 Z_a Z_b}{R_{ab}}\end{aligned}. \tag{4.1}$$

We want to find solutions to the time-dependent non-relativistic Schrodinger equation:

$$i\hbar\frac{\partial}{\partial t}\Psi = \hat{H}\Psi. \tag{4.2}$$

The wave function is separated in terms of the electronic and nuclear coordinates with the ansatz:

$$\Psi\left(\{\boldsymbol{r}\}_i, \{\boldsymbol{R}\}_a, t\right) \approx \Phi(\{\boldsymbol{r}\}_i)\chi(\{\boldsymbol{R}\}_a)\exp\left[\frac{i}{\hbar}\int_{t_0}^{t} dt' E_e(t')\right], \tag{4.3}$$

with the electronic and nuclear wave functions normalized to unity at every instance of time. A phase factor is introduced to make the equations look nice:

$$E_e = \int d\boldsymbol{r} d\boldsymbol{R}\Phi^*\chi^*\hat{H}\Phi\chi, \tag{4.4}$$

where the integration occurs over all spatial coordinates $\{\boldsymbol{r}\}_i, \{\boldsymbol{R}\}_a$. This is a single determinant ansatz which must lead to a mean-field description of the dynamics. Inserting this ansatz into the Schrodinger equation reveals the following set of equations:

$$i\hbar\frac{\partial\Phi}{\partial t} = -\sum_i \frac{\hbar^2}{2m_e}\nabla_i^2\Phi + \left\{\int d\boldsymbol{R}\chi^* V_{ne}\chi\right\}\Phi, \tag{4.5}$$

$$i\hbar\frac{\partial\chi}{\partial t} = -\sum_i \frac{\hbar^2}{2M_a}\nabla_a^2\chi + \left\{\int d\boldsymbol{r}\Phi^*\hat{H}\Phi\right\}\chi. \tag{4.6}$$

These coupled equations form the framework for the time-dependent self-consistent field (TDSCF) method. The electrons and nucleons move on a

potential energy surface obtained from averages over the opposite class of degrees of freedom (the nuclear and electronic wave functions respectively). In the framework of classical molecular dynamics we approximate the nuclei as classical point particles. This can be done by rewriting the nuclear wave function as

$$\chi = A \exp[iS/\hbar], \tag{4.7}$$

with an amplitude factor A and a phase S which are both considered to be real. The TDSCF equations are rewritten in terms of these variables

$$\frac{\partial S}{\partial t} + \sum_a \frac{1}{2M_a}(\nabla_a S)^2 + \int d\boldsymbol{r} \Phi^* \hat{H} \Phi = \hbar^2 \sum_a \frac{1}{2M_a} \frac{\nabla_a^2 A}{A}, \tag{4.8}$$

$$\frac{\partial A}{\partial t} + \sum_a \frac{1}{M_a}(\nabla_a A)(\nabla_a S) + \sum_a \frac{1}{2M_a} A(\nabla_a^2 S) = 0. \tag{4.9}$$

This set of equations is known as the "quantum fluid dynamical representation". The term for S contains a term for $\hbar$ which vanishes in the classical limit $\hbar \to 0$:

$$\frac{\partial S}{\partial t} + \sum_a \frac{1}{2M_a}(\nabla_a S)^2 + \int d\boldsymbol{r} \Phi^* \hat{H} \Phi = 0. \tag{4.10}$$

This formulation of the nuclear dynamics is isomorphic to the Hamilton-Jacobi formulation:

$$\frac{\partial S}{\partial t} + \hat{H} = 0, \tag{4.11}$$

with the classical Hamilton function

$$\hat{H} = T(\{P_a\}) + V(\{R_a\}), \tag{4.12}$$

with coordinates $\{R_a\}$ and conjugate momenta $\{P_a\}$. If we identify the conjugate momenta with the phase S as:

$$\boldsymbol{P}_a = \nabla_a S, \tag{4.13}$$

we obtain the following Newtonian equations of motion:

$$\begin{aligned} \frac{d\boldsymbol{P}_a}{dt} &= -\nabla_a V = -\nabla_a \int d\boldsymbol{r} \Phi^* \hat{H} \Phi \quad \text{or} \\ M_a \frac{d^2 \boldsymbol{R}_I}{dt^2} &= -\nabla_a \int d\boldsymbol{r} \Phi^* \hat{H} \Phi \\ &= -\nabla_a V_e^E \left(\{R_a(t)\} \right). \end{aligned} \tag{4.14}$$

Under this formulation of nuclear dynamics the nuclei move according to the laws of classical mechanics in an effective potential V_e^E generated by the electrons. After averaging out the electronic degress of freedom this potential is now only a function of the nuclear coordinates. For consistency the nuclear wave function appearing in the TDSCF equation for the electronic degrees of freedom has to be replaced by the positions of the nuclei. This is accomplished by replacing the nuclear density $|\chi|^2$ in the limit $\hbar \to 0$ by a product of delta functions $\prod_a \delta(\boldsymbol{R}_a - \boldsymbol{R}_a(t))$ centered at the instantaneous positions $\{\boldsymbol{R}_a(t)\}$ of the classical nuclei. This leads to a time-dependent wave equation for the electrons:

$$i\hbar \frac{\partial \Phi}{\partial t} = -\sum_i \frac{\hbar}{2m_e} \nabla_i^2 \Phi + \hat{V}_{ne} \Phi, \tag{4.15}$$

which evolve quantum mechanically as the nuclei propagate classically. This mixed approach is commonly referred to as *Ehrenfest molecular dynamics*. Under this formulation of nuclear dynamics the nuclei evolve classically while the electrons evolve according to the laws of quantum mechanics. Although the underlying equations describe a mean-field theory, the Ehrenfest approach includes transitions between electronic states. In order to arrive at a purely classical description of the dynamics of both the nuclei and the electrons we need to make further simplifications. Firstly we restrict the electronic wave function Φ to the ground state wave function Φ_0 at every instant of time. This means the nuclei move on a single potential energy surface:

$$V_e^E = \int d\boldsymbol{r} \Phi_0^* \hat{H} \Phi_0 = E_0 \left(\{R_a\} \right), \tag{4.16}$$

that is obtained by solving the Schrodinger equation for the ground state electron wave function:

$$\hat{H} \Phi_0 = E_0 \Phi_0. \tag{4.17}$$

Since we are now dealing with a single potential energy surface, the problem of computing the energy surface can be decoupled from computing the expectation values from the electronic wave function. First one produces an appropriate set of nuclear configurations by solving the time-independent Schrodinger equation. Second, these configurations are fitted to an analytical functional form to produce a global potential energy surface. Finally the Newtonian equations of motions are solved on this energy surface, producing a set of classical trajectories. To deal with the large number of degrees of freedom as the number of nuclei in the system increases, the global potential energy surface is approximated as an expansion of manybody contributions:

$$V_e^E \approx V_e^{\text{approx}} = \sum_a v_1(R_a) + \sum_{a<b} v_2(R_a, R_b) + \sum_{a<b<c} v_3(R_a, R_b, R_c), \quad (4.18)$$

and is typically truncated at 2, 3 or 4-body interactions depending on the complexity of the atoms and molecules in the system. This renders the problem of computing dynamics purely classical:

$$M_a \frac{d^2 R_a}{dt^2} = -\nabla_a V_e^{\text{approx}}. \quad (4.19)$$

This reduction in the number of degrees of freedom is a huge simplification which allows us to study much larger and more complex systems than ab-initio methods. However, many approximations have to be made to get to this formulation of atomic/molecular mechanics, and neglecting the electronic degrees of freedom effectively precludes chemical transformations from appearing in the simulations. In addition, the analytical functional forms of the potentials usually include many parameters to be determined, and they often have to be tailored to the quantities one is trying to compute.

4.0.2 Molecular dynamics simulations

The theory in this and the following sections is based partly on [Frenkel, Daan and Smit, Berend 26, pages 63-107], which explains the physics behind many popular methods for computer calculation and simulation. Classical molecular dynamics is a method for computing equilibrium and transport properties of manybody systems obeying classical laws of motion. While a large number of simplifications have to be made in order to describe quantum mechanical systems classically, the approximation works surprisingly well

except for atoms which are quite light (He, H^2) or for atoms with a vibrational energy which is substantially larger than the thermal energy of the system ($h\nu > k_B T$).

In order to calculate properties of the system they have to be expressed in terms of the positions and velocities of the constituent nuclei. For instance the temperature can be related to the average kinetic energy of the system:

$$\langle \frac{1}{2} m v^2 \rangle = \frac{N_f}{2} k_B T, \tag{4.20}$$

where N_f is the number of degrees of freedom in our system. At every instant of time the total kinetic energy of our system defines an instantaneous temperature, which has to averaged over a large number of timesteps in order to produce the equilibrium property. In practice, one is satisfied when the fluctuations in the instantaneous temperature appear reasonably small.

To run a molecular dynamics simulation one requires a set of initial conditions, i.e. a set of initial positions and velocities for every atom in the system. Typically the atoms are placed by replicating a unit cells a number of times in every dimension. A unit cell consists of a set of lattice vectors which define the placement of every atom in the unit cell. For instance the face-centered cubic cell (FCC) contains 4 atoms:

$$\begin{aligned} \boldsymbol{r}_1 &= (0, 0, 0) \\ \boldsymbol{r}_2 &= (\frac{b}{2}, \frac{b}{2}, 0) \\ \boldsymbol{r}_3 &= (0, \frac{b}{2}, \frac{b}{2}) \\ \boldsymbol{r}_4 &= (\frac{b}{2}, 0, \frac{b}{2}), \end{aligned} \tag{4.21}$$

where b is known as the lattice constant and defines the size of the unit cells. In figure 4.1 we have an image of a box of atoms visualized using the Visual Molecular Dynamics (VMD[1]) software. In this system we have a group of water molecules over a Self-Assembled Monolayer (SAM) surface, consisting of C12 molecules attached to a sulfhydryl group. In this case the SAM surface is placed regularly while the water molecules are placed randomly, ensuring no molecules are too close.

[1] Visual Molecular Dynamics (VMD)

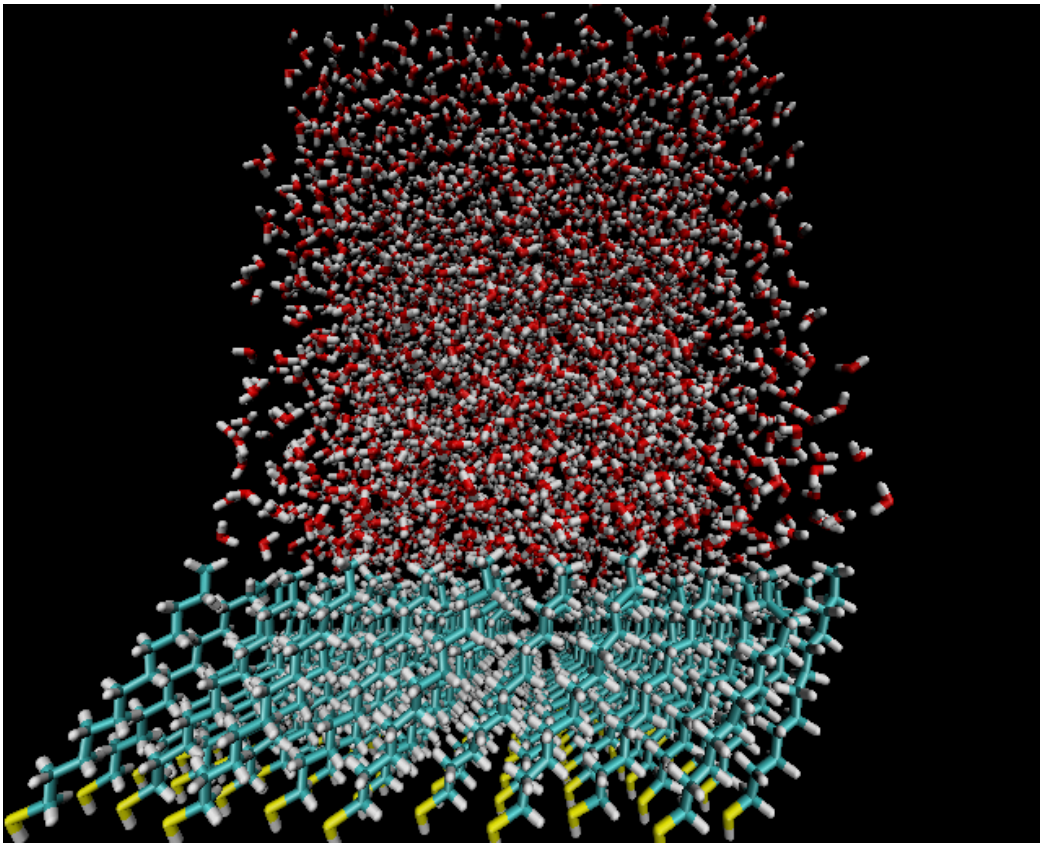


Figure 4.1: Water molecules over a SAM surface. Hydrogen atoms in white, oxygen in red, carbon in blue and sulfur in yellow.

The velocities are typically initialized with a random uniform distribution or the Maxwell-Boltzmann distribution. The Maxwell-Boltzmann distribution is the one most often used, since the equilibrium distribution tends towards this distribution. The exact form however will differ from the one we started with.

Given these initial conditions, the system will not be in an equilibrium state at $t = 0$. To evolve the system to an equilibrium state one most commonly evolves the system for a number of timesteps until fluctuations in dynamic properties such as the total potential energy or the temperature settle down. Once we are in equilibrium we can start calculating thermodynamic averages.

As we mentioned before, the global energy surface as a function of nuclear coordinates $\{\boldsymbol{r}\}$ is approximated as an expansion of manybody contributions:

$$V_e^E\left(\{\boldsymbol{r}\}\right) = \sum_i v_1(r_i) + \sum_{i<j} v_2(r_i, r_j) + \sum_{i<j<k} v_3(r_i, r_j, r_k), \tag{4.22}$$

wherein each n-body term is an analytical function of n coordinates. As an atom moves on the energy surface it feels a force which is the gradient of the potential energy surface. This means atom i feels an acceleration:

$$\boldsymbol{F}_i = m_i \frac{d^2\boldsymbol{r}_i}{dt^2} = -\nabla_i V_e^E\left(\{\boldsymbol{r}\}\right). \tag{4.23}$$

For a system of N atoms with only pairwise interactions this means the forces must be calculated $N(N-1)/2$ times for every timestep which means we have a time complexity of order $\mathcal{O}(N^2)$. The force calculation is by far the most important part of any molecular dynamics simulation, and the most time consuming. A number of techniques are employed in order to reduce the time usage, perhaps the most common is the use of neighbor lists. Using neighbor lists, each atom carries a list of neighbors within a cut-off radius r_{cut} and interactions beyond this cut-off are neglected. This reduces the time-complexity to merely $\mathcal{O}(N)$, with a proportionality constant dependent upon the average number of neighbors in the system within a cutoff r_{cut}. For a large system this can be a huge reduction in complexity, but the choice of cut-off can obviously massively impact the dynamics of the system.

In order to simulate the dynamics of a system governed by a conservative force $\boldsymbol{F} = -\nabla V_e^E$ we need to integrate the Newtonian equations of motion. The equations of motion are typically not solvable analytically, which means we require an effective numerical method for integration. Some important considerations for molecular dynamics are conservation of energy and accuracy for large time steps. The most common method used is the Velocity-Verlet algorithm. At any given time step t, the position $\boldsymbol{r}(t+\Delta t)$ and velocity $\boldsymbol{v}(t+\Delta t)$ at the next time step $t+\Delta t$ is calculated as:

$$\begin{aligned}
\boldsymbol{r}(t+\Delta t) &= \boldsymbol{r}(t) + \boldsymbol{v}(t)\Delta t + \frac{1}{2}\boldsymbol{a}(t)\Delta t^2, \\
\boldsymbol{a}(t+\Delta t) &= -\frac{1}{m}\nabla V_e^E(\boldsymbol{r}(t+\Delta t)), \\
\boldsymbol{v}(t+\Delta t) &= \boldsymbol{v}(t) + \frac{1}{2}(\boldsymbol{a}(t) + \boldsymbol{a}(t+\Delta t))\Delta t.
\end{aligned} \tag{4.24}$$

The error in the Velocity-Verlet method is of order $\mathcal{O}(\Delta t^2)$, which means that it is not particularly accurate for large time steps over a long time. However, the long term energy drift of the method is small, which is very desirable. It is also not very memory-intensive, which matters for simulating very large systems.

Molecular dynamics is usually performed within a cubic box of fixed volume $V = L_x \cdot L_y \cdot L_z$, where L_i is the length of the box in direction i. Molecular dynamics is typically limited by the number of particles we are able to simulate, of the order $10^6 - 10^8$, which means the size of the box is often decided by the desired density $\rho = N/V$. Since the number of particles is always much smaller than the number of particles in realistic systems, approximations are required. Periodic boundary conditions can be applied to the box in order to approximate an infinite system. Typically particle coordinates are restricted to the simulation box, which can be expressed in pseudocode as:

Algorithm 1 Continuity

```
if x < −L_x/2 then
    x += L_x
end if
if x > L_x/2 then
    x −= L_x
end if
```

Distance and distance vectors between particles should also obey the minimum image convention:

Algorithm 2 Minimum image

```
dx = x_j − x_i
if dx < −L_x/2 then
    dx += L_x
end if
if dx > L_x/2 then
    dx −= L_x
end if
```

These conditions should be applied in every dimension. This approach runs the risk of introducing nonphysical artifacts of the simulation, such as a

macromolecule interacting with its own image, and for coulombic interactions the system must be charge neutral to avoid summing to an infinite charge. The optimal system size with periodic boundary conditions will therefore depend on the intended simulation length, the desired accuracy and the dynamics which are being studied.

Thus far we have discussed molecular dynamics for a system of N particles within a fixed volume V and constant energy E, i.e. in the microcanonical ensemble NVE. Other ensembles are also impossible, such as the canonical ensemble NVT and the isothermal-isobaric ensemble NPT.

Simulations in the canonical ensemble can be achieved by modifying the Verlet integration algorithm. The simplest thermostat possible follows from the equipartion theorem:

$$T \propto \langle mv^2 \rangle, \tag{4.25}$$

meaning some amount of kinetic energy i.e. velocity can be added or subtracted to every atom in order to maintain a constant temperature at every timestep. Multiplying every velocity by a factor $\lambda = \sqrt{T_0/T(t)}$ where $T(t)$ is the instantaneous temperature and T_0 is the desired temperature will achieve desired effect. This approach significantly alters the trajectories of the system however, which means this thermostat should only be applied for adjusting the temperature of the system and not for taking ensemble averages in equilibrium.

The most common thermostat used is the Nosé-Hoover thermostat, which is one of the most accurate and efficient algorithms for achieving realistic constant-temperature conditions (see the lecture notes [27, Shell, M. Scott]). Nosé introduced an extended Hamiltonian with two additional degrees of freedom:

- s - the position of an imaginary coupled heat reservoir
- p_s - the conjugate momentum of the heat reservoir

In addition it introduces an effective mass Q such that $p_s = Q\frac{ds}{dt}$.
Hoover modified Nosé's approach by introducing the Hamiltonian:

$$H = \frac{1}{2}\sum m_i \left|\boldsymbol{p}_i\right|^2 + U(\boldsymbol{r}) + \frac{\xi Q}{2} + 3Nk_BT\ln s, \tag{4.26}$$

where ξ is a friction coefficient and $\boldsymbol{p}_i = m_i \boldsymbol{v}_i \times s$ are the particle momenta. This leads to a new set of Newtonian equations of motions with an additional force that is proportional to the velocity:

$$\begin{aligned} \frac{d\boldsymbol{r}_i}{dt} &= \boldsymbol{v}_i \\ \frac{d\boldsymbol{v}_i}{dt} &= -\frac{1}{m_i}\frac{\partial U(\boldsymbol{r})}{\partial \boldsymbol{r}_i} - \xi \boldsymbol{v}_i \\ \frac{d\xi}{dt} &= \left(\sum m_i \left|\boldsymbol{v}_i\right|^2 - 3Nk_bT\right)/Q \\ \frac{d\ln s}{dt} &= \xi. \end{aligned} \tag{4.27}$$

These can then be solved with a numerical integration scheme such as the velocity-Verlet algorithm.

4.0.3 Molecular dynamics potentials

The dynamics of an ensemble of particles is governed by their interactions. In molecular dynamics we stipulate that the interactions are decided only by the relative positions of the particles, i.e. only conservative forces act on the atoms. This means that the force on atom i is fully described by a potential energy U:

$$\boldsymbol{F}_i = -\nabla_i U\left(\{\boldsymbol{r}\}\right), \tag{4.28}$$

which in principle depends on the position of atom i and all other atoms in the system. Finding an appropriate potential for the system of atoms which you intend to study can be ardous work, and usually involves fitting an analytical functional form with a large set of parameters to a potential energy surface from ab initio quantum mechanical calculations.

Potentials can be classified as either bonded or non-bonded. Bonded potentials compute the interactions for a predefined set of atoms and molecules in the simulations, while non-bonded potentials compute the interactions between *all* pairs, triplets etc. of atoms (usually within a certain radius). Larger, more complex systems typically contain a mix of bonded and non-bonded potentials, for example a system of rigid water molecules interacting with a surface of carbon atoms.

One of the simplest potentials meant to simulate a realistic system is the Lennard-Jones potential:

$$U(r_{ij}) = 4\epsilon \left(\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right), \tag{4.29}$$

with $r_{ij} = |\boldsymbol{r}_j - \boldsymbol{r}_i|$. For the Lennard-Jones potential there are only two parameters to be decided, a characteristic energy ϵ and a characteristic length σ. This potential is meant to emulate the relatively weak interactions between noble gas atoms such as Argon. It can be separated into two terms:

- $-\left(\frac{\sigma}{r}\right)^6$ - owing to the long-term attraction from van der Waals interactions
- $+\left(\frac{\sigma}{r}\right)^{12}$ - owing to the short-term repulsion from the Pauli principle

While the van der Waals term is justified by theory, the repulsion term is justified by numerical efficiency - as it contains the square of the van der Waals term - and because it models the Pauli repulsion accurately. In figure 4.2 we have plotted the Lennard-Jones potential as a function of interatomic distance. This form of the potential, with close repulsion and distant attraction is very common among molecular dynamics potentials, though often with modifications.

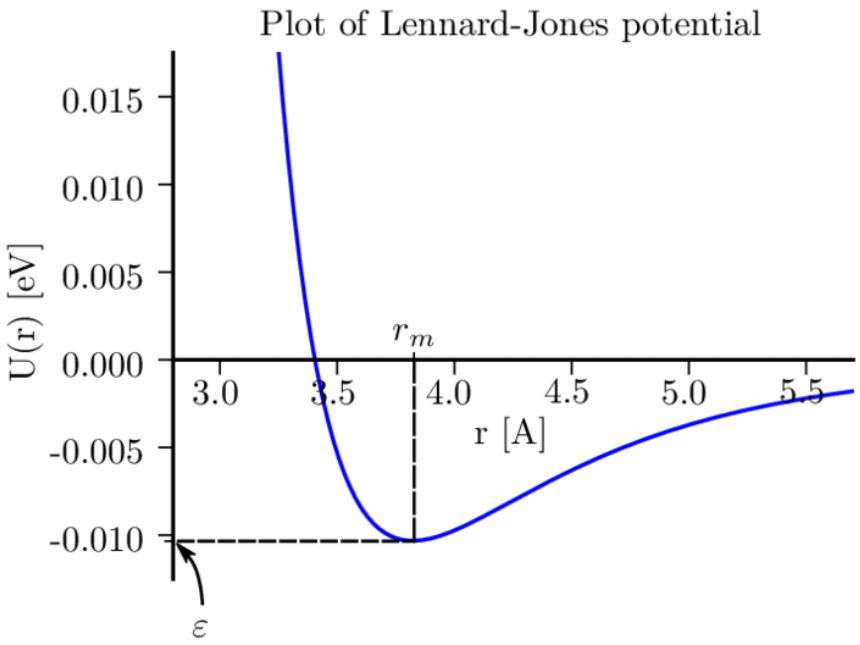


Figure 4.2: Lennard-Jones potential as a function of interatomic distance. Units in Angstrom and electronvolts. Reprinted from [28, Molecular dynamics modelling of clay-fluid interfaces].

While the Lennard-Jones potential is very simple, requiring only two parameters to be determined, it has shown to be effective at modelling noble gas atoms and is commonly used as a building block for more complicated interactions.

Another simple and common potential is the Stillinger-Weber potential, which is meant to model the interactions between silicon atoms (see the lecture notes [29, Abrams, Cameron]). Silicon forms tetrahedral bonded structures as well as pairwise interactions which means the potential includes a twobody and a threebody interaction:

$$U = \sum_{i<j} v_2(r_{ij}) + \sum_{i<j<k} v_3(\boldsymbol{r}_i, \boldsymbol{r}_j, \boldsymbol{r}_k). \tag{4.30}$$

The twobody interaction models the pairwise interaction:

$$v_2(r) = \begin{cases} \epsilon A(Br^{-p} - r^{-q}) \exp\left[(r-a)^{-1}\right], & r < a \\ 0 & r \geq a \end{cases} \tag{4.31}$$

This twobody term resembles the Lennard-Jones potential, but with an exponential cutoff. The threebody term models the tetrahedral angles, and is a sum over three triplets:

$$v_3(\boldsymbol{r}_i, \boldsymbol{r}_j, \boldsymbol{r}_k) = h_{jik} + h_{ijk} + h_{ikj}, \tag{4.32}$$

with the angular interaction $h_{jik} = h(r_{ij}, r_{ik}, \theta_{jik})$ and

$$h_{jik} = \begin{cases} \epsilon\lambda \exp\left[\dfrac{\gamma}{r_{ij}-a} + \dfrac{\gamma}{r_{ik}-a}\right] \left(\cos\theta_{jik} - \cos\theta^0_{jik}\right)^2 & r_{ij} < a \\ 0 & r_{ij} \geq a \end{cases} \tag{4.33}$$

where θ^0_{jik} is an "equilibrium" angle. The terms $\epsilon, A, B, p, q, \lambda, \gamma$ are parameters to be decided and a is a cutoff radius. The inclusion of the threebody terms prove to be quite important in silicon maintaining its equilibrium crystal structure. The Stillinger-Weber potential agrees quite well with experiment, and its relatively simple form makes it well suited for the testing and evaluation of new potentials.

A more complicated set of potentials is the Effective Medium Theory (EMT) family of potentials, developed to describe the late transition metals in the FCC crystal structure. The potential was first described by [30, Jacobsen, K.W., Nørskov, J.K. and Puska, M.J.], but the most common set of parameters was published in the later article by [31, Jacobsen, K.W., Stoltze, P. and Nørskov, J.K.]. These potentials are based on the effective medium theory concept of an electron density with a volume-dependent contribution to the total energy. Effective Medium Theory computes the energy of an atom in an arbitrary environment by computing it first in a reference system and then estimating the energy difference between the real system and the reference system. The total energy is written as:

$$E = \sum_i E_{c,i} + \left(E - \sum_i E_{c,i}\right), \tag{4.34}$$

where $E_{c,i}$ is the energy of atom i in the reference system. The correction $E - \sum_i E_{c,i}$ is made small enough so that it can be estimated using approximate methods such as perturbation theory. From Density Functional Theory the total energy can be written as:

$$E = \sum_i E_{c,i}(n_i) + \Delta E_{AS} + \Delta E_{1\text{el}}, \tag{4.35}$$

where E_c, ΔE_{AS} and $\Delta E_{1\text{el}}$ are known as the cohesive function, atomic-sphere correction and one-electron correction respectively. The density argument n_i is known as the embedding density, which connects the surroundings of atom i to the reference system of atom i. The atomic-sphere correction is the difference in electrostatic and exchange-correlation energy for the atoms in the system of interest and the reference system. The one-electron correction is the sum of one-electron energies in the two systems. If the one-electron correction is small and can be neglected a pair-potential approximation leads to the expression:

$$\begin{aligned} E &= \sum_i \left[E_{c,i}(n_i) + \Delta E_{AS} \right] \\ &= \sum_i \left\{ E_{c,i}(n_i) + \frac{1}{2} \left[\sum_{j \neq i} V_{ij}(r_{ij}) - \sum_{j \neq i}^{\text{ref}} V_{ij}(r_{ij}) \right] \right\} \end{aligned} \tag{4.36}$$

The one-electron correction is often not small, so this is usually referred to as just the atomic-sphere correction. The embedding density n_i is computed by superimposing density contributions from the neighboring atoms:

$$n_i = \sum_{j \neq i} \Delta n_j(s_i, r_{ij}), \tag{4.37}$$

where the density tail $\Delta n(s, r)$ from a neighboring atom a distance r averaged over a sphere of radius s has an exponential form:

$$\Delta n(s, r) = \Delta n_0 \exp\left[\eta_1 (s - s_0) - \eta_2 (r - \beta s_0) \right]. \tag{4.38}$$

The size of the sphere s is chosen so that the total charge within in zero. The geometric factor $\beta = (16\pi/3)^{1/3} / \sqrt{2}$ is related to the nearest-neighbor distance $d_{nn} = \beta s$. For this set of electron densities there is a one-to-one

correspondence between the average electron density and its neutral-sphere radius. This relationship has been shown to be approximately exponential:

$$\bar{n}(s) = n_0 \exp\left[-\eta(s - s_0)\right]. \tag{4.39}$$

For an FCC crystal of varying nearest-neighbor distance $r = \beta s$ the neutral-sphere radius can be expressed as:

$$s_i = s_0 - \frac{1}{\beta\eta_2} \log\left(\frac{\sigma_{1,i}}{12}\right), \tag{4.40}$$

where

$$\sigma_{1,i} = \sum_{j \neq i} \exp\left[-\eta_2 \left(r_{ij} - \beta s_0\right)\right]. \tag{4.41}$$

The cohesive function is parameterized as a function of the neutral-sphere radius using the functional form:

$$\begin{aligned} E_c(s) &= E_0 f\left[\lambda\left(s - s_0\right)\right], \\ f\left(x\right) &= \left(1 + x\right)\exp\left(-x\right) \end{aligned} \tag{4.42}$$

where s_0 is the equilibrium (zero pressure) neutral-sphere radius. Finally the atomic sphere correction is written as:

$$\Delta E_{AS}(i) = \frac{1}{2}\left[\sum_{j \neq i} V(r_{ij}) - 12V(\beta s_i)\right], \tag{4.43}$$

where the factor 12 comes from the 12 nearest neighbors at a distance $r = \beta s_i$ in the FCC reference system. The pair potential is parameterized as:

$$V(r) = -V_0 \exp\left[-\kappa\left(r/\beta - s_0\right)\right]. \tag{4.44}$$

This has been found to give reasonable descriptions of the six fcc metals Cu, Ag, Au, Ni, Pd, Pt and their alloys. Further description and parameters can be found in the article [31, Jacobsen, K.W., Stoltze, P. and Nørskov, J.K.].

Chapter 5

Machine learning

Machine learning is the study of algorithms and statistical models employed by computing systems capable of performing tasks without explicit instruction. While traditional algorithms rely on some specified input and a ruleset for determining the output, machine learning is instead concerned with a set of generic algorithms which can find patterns in a broad class of data sets. This section will give a brief overview of machine learning, and more specifically the class of algorithms known as neural networks, and will follow closely the review by [Mehta et al. 32, pages 1-64] which the reader is encouraged to seek out for further information.

Examples of machine learning problems include identifying objects in images, transcribing text from audio and making film recommendations to viewers based on their watch history. Machine learning problems are often subdivided into estimation and prediction problems. In both cases, we choose some observable $\boldsymbol{x}$ (e.g. the period of a pendulum) related to some parameters $\boldsymbol{\theta}$ (e.g. the length and the gravitational constant) through a model $p(\boldsymbol{x}|\boldsymbol{\theta})$ that describes the probability of observing $\boldsymbol{x}$ given $\boldsymbol{\theta}$. Subsequently we perform an experiment to obtain a dataset $\boldsymbol{X}$ and use these data to fit the model. Fitting the model means finding the parameters $\hat{\boldsymbol{\theta}}$ that provide the best explanation for the data. *Estimation* problems are concerned with the accuracy of $\hat{\boldsymbol{\theta}}$, whereas prediction problems are concerned with the ability of the model $p(\boldsymbol{x}|\boldsymbol{\theta})$ to make new predictions. Physics has traditionally been more concerned with the estimation of model parameters, while in this thesis we will be focused on the accuracy of the model.

Many problems in machine learning are defined by the same set of ingredients. The first is the dataset $\mathcal{D} = (\boldsymbol{X}, \boldsymbol{Y})$, where $\boldsymbol{X}$ is a matrix containing

observations of the independent variables $\boldsymbol{x}$, and $\boldsymbol{Y}$ is a matrix containing observations of dependent variables $\boldsymbol{y}$. Second is a model $\boldsymbol{F} : \boldsymbol{x} \rightarrow \boldsymbol{y}$ which is a function with parameters $\boldsymbol{\theta}$. Finally we have a cost function $\mathcal{C}\left(\boldsymbol{Y}, \boldsymbol{F}\left(\boldsymbol{X}; \boldsymbol{\theta}\right)\right)$ that judges the ability of our model to make predictions.
In the case of linear regression we consider a set of independent observations $\boldsymbol{X} = \begin{bmatrix} \boldsymbol{x}_1 & \boldsymbol{x}_2 & \dots & \boldsymbol{x}_N \end{bmatrix}$ related to a set of dependent observations $\boldsymbol{y} = (y_1, y_2, \dots, y_N)$ through a linear model $f(\boldsymbol{x}; \boldsymbol{\theta}) = x_1 \cdot w_1 + x_2 \cdot w_2 + \dots + x_P \cdot w_P$, with parameters $\boldsymbol{\theta} = (w_1, w_2, \dots, w_P)$. The cost function is the well known sum of least squares $\mathcal{C}(\boldsymbol{y}, f(\boldsymbol{X}; \boldsymbol{\theta})) = \sum_i^N (y_i - f(\boldsymbol{x}_i; \boldsymbol{\theta}))^2$ and the best fit is chosen as the set of parameters which minimize this cost function:

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\ \mathcal{C}(\boldsymbol{Y}, f(\boldsymbol{X}; \boldsymbol{\theta})) \tag{5.1}$$

In figure 5.1 we have plotted an example of simple linear regression, wherein a single value x is mapped to a response $y = b_0 + b_1 \cdot x$. The term b_0 is known as the offset and b_1 the slope and they are parameters which are to be determined from the training data.

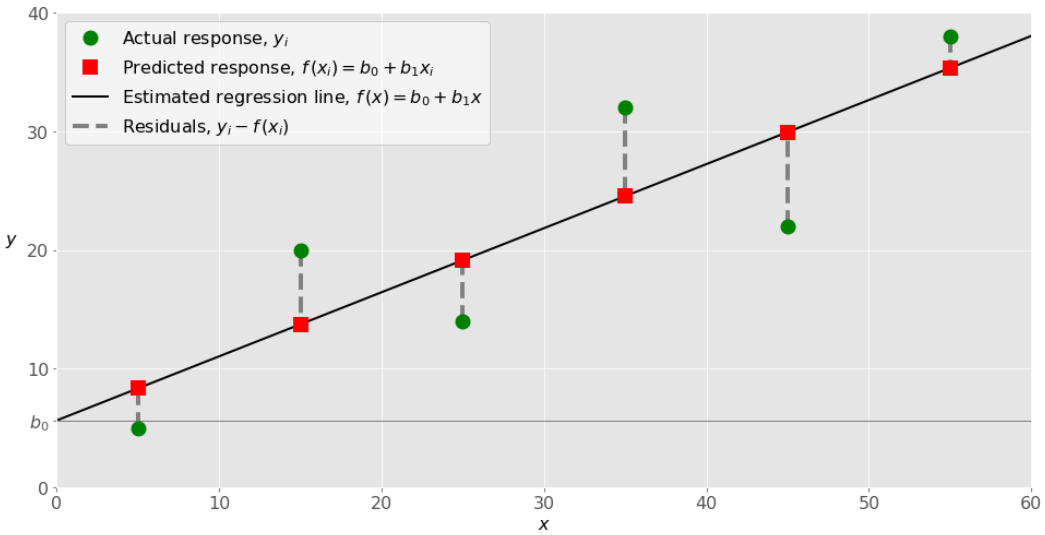


Figure 5.1: Example of simple one-dimensional linear regression. Reprinted from www.realpython.com: Linear Regression in Python.

5.0.1 Basics of statistical learning

Statistical learning theory is a field of statistics dealing with the problem of making predictions from data. This section is covered in the slides from the MIT course [33, 9.2520: Statistical Learning Theory and applications] from 2012, and then the remainder of the section and chapter is a summary of the most relevant parts of [32, Mehta et al.] as we mentioned before. We start with an unknown function $y = f(x)$ and our goal is to develop a function $h(x)$ such that $h \sim f$. We fix a hypothesis set $\mathcal{H}$ that the algorithm is willing to consider. The expected error of a particular h over all possible inputs x and outputs y is:

$$E[h] = \int_{X \times Y} \mathcal{C}(h(x), y)\rho(x, y)dxdy, \tag{5.2}$$

where $\mathcal{C}$ is a cost function and $\rho(x, y)$ is the joint probability distribution for x and y. This is known as the *expected error*. Since this is impossible to compute without knowledge of the probability distribution ρ, we instead turn to the *empirical error*. Given n data points the empirical error is given as:

$$E_E[h] = \frac{1}{n}\sum_{i}^{n} \mathcal{C}(h(x_i), y_i). \tag{5.3}$$

The *generalization error* is defined as the difference between the expected and empirical errors:

$$G = E[h] - E_E[h]. \tag{5.4}$$

We say an algorithm is able to learn from data or *generalize* if

$$\lim_{n\to\infty} G = 0. \tag{5.5}$$

We are in general unable to compute the expected error, and therefore unable to compute the generalization error. The most common approach known as *cross-validation* is to estimate the generalization error by subdividing our dataset into a *training* set and a *test* set. The value of the cost function on the training set is called the *in-sample* error and the value of the cost function on the test set the *out-of-sample* error. Assuming the dataset is sufficiently large and representative of f, and the subsampling into train and test datasets is unbiased, the in-sample error can serve as an appropriate proxy for the generalization error.

In figure 5.2 we show the typical evolution of the errors as the number of data points increase. It is assumed that the function being learned is sufficiently complicated that we cannot learn it exactly, and that we have a sizeable number of data points available. The in-sample error will decrease monotonically, as our model is not able to learn the underlying data exactly. In contrast, the out-of-sample error will decrease, as the sampling noise decreases and the training data set becomes more representative of the underlying probability distribution. In the limit, these errors both approach same value, which is known the model *bias*. The bias represents the best our model could do in the infinite data limit. The out-of-sample error produced from the sampling noise is known as *variance*, and will vanish completely given an infinite representative data set.

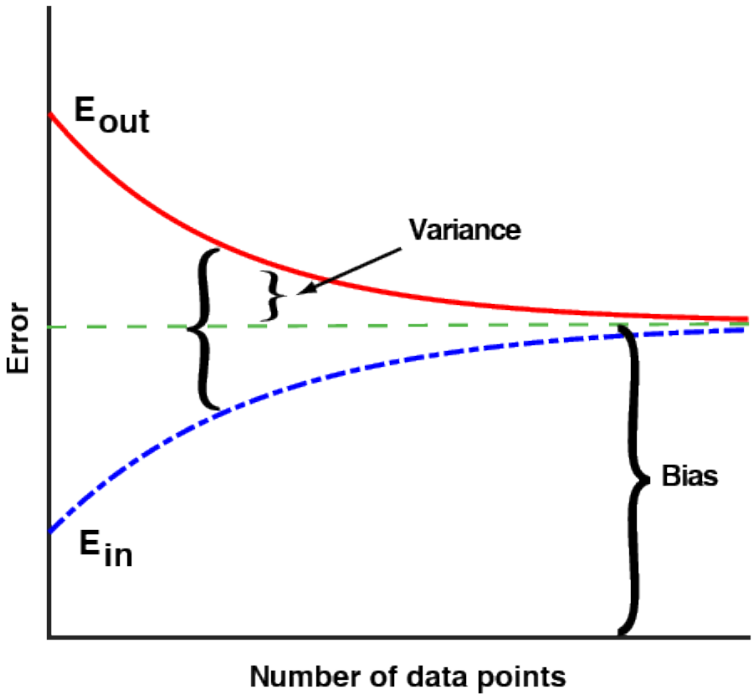


Figure 5.2: Typical in-sample and out-of-sample error as a function of the number of data points. It is assumed that the number of data points is not small, and that the true function cannot be exactly fit. Reprinted from [32, Mehta et al. page 11].

In figure 5.3 we show the typical evolution of the out-of-sample error

as the model *complexity* increases. Model complexity is a measure of the degrees of freedom in the model space, for example the number of coefficients in a polynomial regression. In the figure we can see that bias decreases monotonically as model complexity increases, as the model is able to fit a larger space of functions. However, the variance will also increase as the model becomes more susceptible to sampling noise. In general the lowest out-of-sample error, and therefore generalization error, is achieved at an intermediate model complexity. We also find that as model complexity increases, a larger amount of data points is required to be able to reasonably fit the true function.

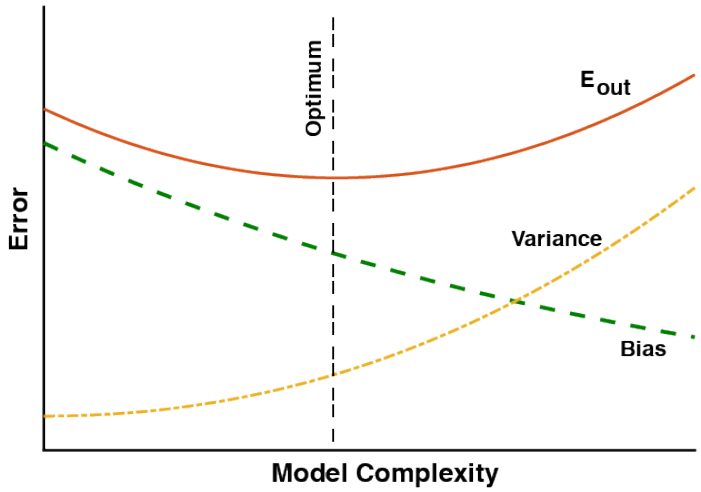


Figure 5.3: Typical out-of-sample error as a function of model complexity for a fixed dataset. Bias decreases monotonically with model complexity, while variance increases as a result of sampling noise. Reprinted from [32, Mehta et al. page 11].

5.0.2 Bias-variance decomposition

Consider a dataset $\mathcal{D}(\boldsymbol{X}, \boldsymbol{y})$ of n pairs of independent and dependent variables. Assume the true data is generated from a noisy model:

$$y = f(\boldsymbol{x}) + \epsilon, \tag{5.6}$$

where ϵ is normally distributed with mean μ and standard deviation σ.

Assume that we have an estimator $h(\boldsymbol{x};\boldsymbol{\theta})$ trained by minimizing a cost function $\mathcal{C}(\boldsymbol{y}, h(\boldsymbol{x}))$ which we take to be the sum of squared errors:

$$\mathcal{C}(\boldsymbol{y}, h(\boldsymbol{x})) = \sum_{i}^{n} (y_i - h(\boldsymbol{x}_i;\boldsymbol{\theta}))^2. \tag{5.7}$$

Our best estimate for the model parameters

$$\boldsymbol{\theta}_{\mathcal{D}} = \arg\min_{\boldsymbol{\theta}} \mathcal{C}(\boldsymbol{y}, h(\boldsymbol{x};\boldsymbol{\theta})), \tag{5.8}$$

is a function of the dataset $\mathcal{D}$. If we imagine we have a set of datasets $\mathcal{D}_j = (\boldsymbol{y}_j, \boldsymbol{X}_j)$, each with n samples, we would like to calculate the expectation value of the cost function over all these datasets $E_{\mathcal{D},\epsilon}$. We would also like to calculate the expectation value over different instances of the noise ϵ. The expected generalization error can be decomposed as:

$$\begin{aligned} E_{\mathcal{D},\epsilon}[\mathcal{C}(\boldsymbol{y}, h(\boldsymbol{X};\boldsymbol{\theta}_{\mathcal{D}}))] &= E\left[\sum_i (y_i - h(\boldsymbol{x}_i;\boldsymbol{\theta}_{\mathcal{D}}))^2\right] \\ &= \sum_i \sigma_\epsilon^2 + E_{\mathcal{D}}[(f(\boldsymbol{x}_i) - f(\boldsymbol{x}_i;\boldsymbol{\theta}_{\mathcal{D}}))^2]. \end{aligned} \tag{5.9}$$

The second term can be further decomposed as:

$$\begin{aligned} E_{\mathcal{D}}[(f(\boldsymbol{x}_i) - f(\boldsymbol{x}_i;\boldsymbol{\theta}_{\mathcal{D}}))^2] &= (f(\boldsymbol{x}_i) - E_{\mathcal{D}}[h(\boldsymbol{x}_i;\boldsymbol{\theta}_{\mathcal{D}})])^2 \\ &\quad + E[(h(\boldsymbol{x}_i;\boldsymbol{\theta}_{\mathcal{D}}) - E[h(\boldsymbol{x}_i;\boldsymbol{\theta}_{\mathcal{D}}])^2] \end{aligned} \tag{5.10}$$

The first term is what we have referred to as the bias:

$$\text{Bias}^2 = \sum_i (f(\boldsymbol{x}_i) - E_{\mathcal{D}}[h(\boldsymbol{x}_i;\boldsymbol{\theta}_{\mathcal{D}})])^2. \tag{5.11}$$

The bias measures the expectation value of the deviation of our model from the true function, i.e. the best we can do in the infinite data limit. The second term is what we have referred to as the variance:

$$\text{Var} = \sum_i E[(h(\boldsymbol{x}_i;\boldsymbol{\theta}_{\mathcal{D}}) - E[h(\boldsymbol{x}_i;\boldsymbol{\theta}_{\mathcal{D}}])^2] \tag{5.12}$$

The variance measures the deviation of our model due to finite-sampling effects. Combining these effects we can decompose the out-of-sample error into:

$$E_{\text{out}} = \text{Bias}^2 + \text{Var} + \text{Noise}, \tag{5.13}$$

with Noise $= \sum_i \sigma_\epsilon^2$. In general it can be much more difficult to obtain sufficient good data than to train a very complex model. Therefore it is often useful in practice to use a less complex model with higher bias, because it is less susceptible to finite-sampling effects.

5.0.3 Neural networks

Artificial Neural Networks (ANN) or Deep Neural Networks (DNN) are supervised learning models vaguely inspired by biological neural networks. The building blocks of neural networks are neurons that take a vector input of d features $\boldsymbol{x} = (x_1, \ldots, x_d)$ and produce a scalar output $a(\boldsymbol{x})$. A neural network consists of layers of these neurons stacked together with the output of one layer serving as input for another. The first layer is typically known as the *input layer*, the middle layers as *hidden layers* and the final layer the *output layer*. The basic architecture is shown in figure 5.4. In almost all cases the output $a_i(\boldsymbol{x})$ of neuron i can be decomposed into a linear operation on the inputs passed through a non-linear activation function:

$$a_i(\boldsymbol{x}) = \sigma_i(z_i), \tag{5.14}$$

where σ_i is a non-linear function and z_i is the dot product between the inputs $\boldsymbol{x}$ and a set of neuron-specific weights $\boldsymbol{w}_i$:

$$z_i = \boldsymbol{x}^T \boldsymbol{w}_i + b_i. \tag{5.15}$$

The term b_i is a neuron-specific re-centering of the input. Typical choices of non-linearities/activation functions include the sigmoid and hyperbolic tangent functions, and Rectified Linear Units (ReLU). When the activation function is non-linear, the neural network with a single hidden layer can be proven to be a *universal function approximator*[8]; given an arbitrarily large number of neurons it can reproduce any continuous function on compact subsets of $\mathbb{R}^N$. We typically also want activation functions that are monotonic and smooth with a monotonic derivative.

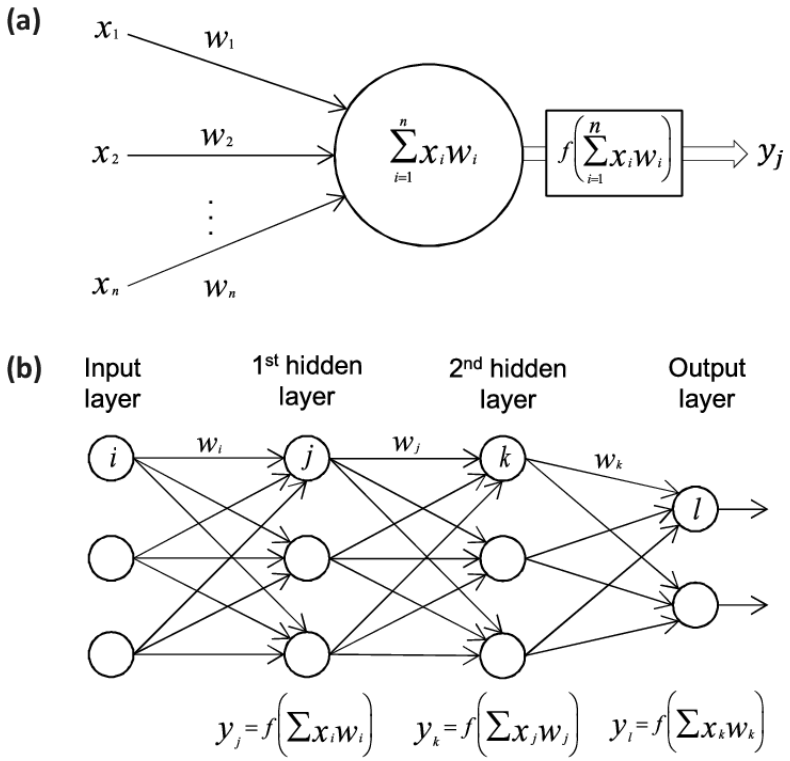


Figure 5.4: Neural network building blocks. Reprinted from [34, Vieira, Pinaya, Mechelli].

The simplest neural networks are known as *feed-forward* neural networks (FNN). The input layer is the vector $\boldsymbol{x}$ of inputs, while each neuron in the first hidden layer performs a dot product between its weights $\boldsymbol{w}_i$ and the inputs and passes it through a non-linearity σ_i. The activation function is typically shared across one or multiple layers $\sigma_i = \sigma$. The vector of neuron outputs $\boldsymbol{a}$ serves as input to the next hidden layer until we reach the final layer. In the final layer the choice of activation function is dependent on the problem we are trying to solve. If we are performing non-linear regression the final activation function is often the identity $\sigma_i(z) = z$, or if we are doing classification the soft-max function is often employed.

Let $\boldsymbol{x}$ be a vector of $d = 1, \ldots, D$ inputs or *features*. Let $a_i^{(l)}$ denote the output of neuron $i = 1, \ldots, N_l$ in layer $l = 1, \ldots, L$. The output of neuron i

in the first hidden layer $a_i^{(1)}$ is thus:

$$\begin{aligned} z_i^{(1)} &= \boldsymbol{x}^T \boldsymbol{w}_i^{(1)} + b_i^{(1)}, \\ a_i^{(1)} &= \sigma_i^{(1)}(z_i^{(1)}). \end{aligned} \tag{5.16}$$

The inputs are iterated through each hidden layer until we reach the final layer. Denote the vector of outputs $\boldsymbol{o} = (o_1, \ldots, o_O)$:

$$\begin{aligned} z_i^{(L)} &= (\boldsymbol{a}_{L-1})^T \boldsymbol{w}_i^{(L)} + b_i^{(L)}, \\ a_i^{(L)} &= o_i \\ &= \sigma_i^{(L)}(z_i^{(L)}) \\ &= \sigma_i^{(L)}\left((\boldsymbol{a}_{L-1})^T \boldsymbol{w}_i^{(L)} + b_i^{(L)}\right) \\ &= \sigma_i^{(L)}\left(\left(\sigma_1^{(L-1)}, \ldots, \sigma_{N_{L-1}}^{(L-1)}\right)^T \boldsymbol{w}_i^{(L)} + b_i^{(L)}\right). \end{aligned} \tag{5.17}$$

This allows us to compose a complicated function $\boldsymbol{F} : \mathbb{R}^D \to \mathbb{R}^O$, with D the number of inputs and O the number of outputs. The *universal approximation theorem*[8] tells us that this simple architecture can approximate any of a large set of continuous functions given appropriate choice of weights $\boldsymbol{w}_i^l$ and mild assumptions on the activation functions. The theorem requires only a single hidden layer, where the strength of the approximation relies on the number of neurons. In practice it has been found that adding more layers produces faster convergence and higher accuracy, which has given rise to the field of *deep learning*.

5.0.4 Backpropagation

Given a set of datapoints $(\boldsymbol{x}_i, y_i),\ i = 1, \ldots, n$, the value of the cost function is entirely determined by the weights and biases of each neuron in the network. We define learning narrowly as adjusting the parameters of the network in order to minimize the cost function. *Gradient descent* is a simple, but powerful method of finding the minima of differentiable functions. Given a function $F : \mathbb{R}^d \to \mathbb{R}$, and an initial value $\boldsymbol{x}_0$ we define an iterative procedure:

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n - \eta \nabla F(\boldsymbol{x}_n), \tag{5.18}$$

where η is known as the *learning rate*. The procedure terminates when the norm $|\nabla F(\boldsymbol{x}_n)|$ or alternatively $|\boldsymbol{x}_{n+1} - \boldsymbol{x}_n|$ is appropriately small. The learning rate is not necessarily fixed throughout the procedure, and proves crucial to the convergence of the method. If f is convex, and η is reasonably small, convergence is guaranteed. Convergence may be very slow however, and if f is not convex you are only guaranteed to find local minima, and this makes the method very sensitive to initial conditions. In figure 5.5 we can see how gradient descent finds minima in a function of two variables by always walking in the steepest direction.

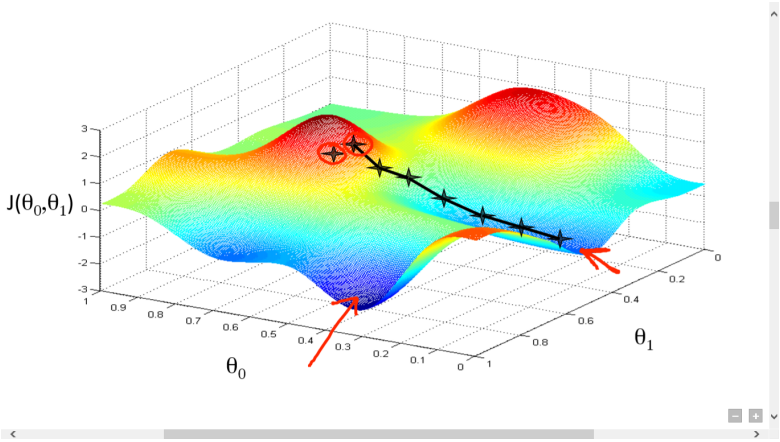


Figure 5.5: How gradient descent discovers extrema points. Reprinted from [Gradient Descent: All You Need to Know] (Hacker Noon).

In order to train the model, we need to calculate the derivative of the cost function with respect to a very large number of parameters multiple times. However, numerical calculation of gradients is very time consuming. The *backpropagation* algorithm is a clever use of the chain rule that allows us to calculate gradients efficiently. Assume that there are L layers in our network with $l = 1, 2, ..., L$ indexing the layers, including the output layer and all the hidden layers. Let w_{ij}^l denote the weight for the connection from the i-th neuron in layer $l-1$ to the j-th neuron in layer l. Let b_j^l denote the bias of this j-th neuron. The activation a_j^l of the j-th neuron in the l-th layer is related to the activities of the neurons in the layer $l-1$ by:

$$a_j^l = f\left(\sum_i w_{ij}^l a_i^{l-1} + b_j^l\right) = f\left(z_j^l\right), \tag{5.19}$$

where f is some activation function. The cost function $\mathcal{C}$ depends directly on the activations in the output layer, and indirectly on the activations in all the lower layers. Define the error Δ_j^L of the j-th neuron in the L-th (final) layer as the change in cost function with respect to the weighted input z_j^L:

$$\Delta_j^L = \frac{\partial \mathcal{C}}{\partial z_j^L}. \tag{5.20}$$

Define analogously the error Δ_j^l of neuron j in the l-th layer as the change in cost function with respect to the weighted input z_j^l:

$$\Delta_j^l = \frac{\partial \mathcal{C}}{\partial z_j^l}. \tag{5.21}$$

This can also be interpreted as the change in cost function with respect to the bias b_j^l:

$$\Delta_j^l = \frac{\partial \mathcal{C}}{\partial z_j^l} = \frac{\partial \mathcal{C}}{\partial b_j^l}\frac{\partial b_j^l}{\partial z_j^l} = \frac{\partial \mathcal{C}}{\partial b_j^l}, \tag{5.22}$$

since $\partial b_l^j / \partial z_j^l = 1$. The error depends on neurons in layer l only through the activation of neurons in layer $l+1$, so using the chain rule we can write:

$$\begin{aligned}
\Delta_j^l = \frac{\partial \mathcal{C}}{\partial z_j^l} &= \sum_i \frac{\partial \mathcal{C}}{\partial z_i^{l+1}}\frac{\partial z_i^{l+1}}{\partial z_j^l} \\
&= \sum_i \Delta_i^{l+1}\frac{\partial z_i^{l+1}}{\partial z_j^l} \\
&= \sum_i \Delta_i^{l+1} w_{ij}^{l+1} f'(z_j^l) \\
&= \left(\sum_i \Delta_i^{l+1} w_{ij}^{l+1}\right) f'(z_j^l).
\end{aligned} \tag{5.23}$$

The sum comes from the fact that any error in neuron j in the l-th layer propagates to all the neurons in the layer $l+1$, so we have to sum up these errors. This gives us the equations we need to update the weights and biases of our network:

$$\frac{\partial \mathcal{C}}{\partial w_{ij}^l} = \frac{\partial \mathcal{C}}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{ij}^l} = \Delta_j^l a_i^{l-1}. \tag{5.24}$$

$$\frac{\partial \mathcal{C}}{\partial b_j^l} = \Delta_j^l. \tag{5.25}$$

Now, if we have the error of every neuron j at the output layer, Δ_j^L, equation 5.23 gives us the recipe for calculating the error in the preceding layer until we reach the first hidden layer, and we are done. All we are missing is the error at the output layer:

$$\Delta_j^L = \frac{\partial \mathcal{C}}{\partial z_j^L} = \frac{\partial}{\partial z_j^L}\left(\sum_o \frac{1}{2}\left(z_o^L - y_o\right)^2\right) = z_j^L - y_j. \tag{5.26}$$

Now, since the derivatives for the weights depend on the activations in the preceding layer, this suggests an iterative procedure for training the network. First we feed the input data through the network and obtain activations and an output, then the output is backpropagated through the network to update the weights and biases. This is then repeated for some number of steps or until the network achieves acceptable accuracy.

5.0.5 Optimization

In order to begin the minimization procedure and find an optimal set of weights and biases for our network we first need some initial values. Often weights are initialized with small values distributed around zero, drawn from a uniform or normal distribution. The bias can be initalized to zero, but enforcing that all biases have some small value ensures that every neuron has output which can be backpropagated in the first training cycle. In recent years a technique known as Xavier initialization has become the default technique for initializing the weights of deep neural networks. For more information the reader is encouraged to read the paper of [35, Glorot, Xavier and Bengio, Yoshua], though the lessons in this paper do not necessarily generalize to deep convolutional neural networks using ReLU activations or other architectures which we will not discuss here.

A critical choice for building neural networks is the choice of activation function. As we have mentioned briefly above, we have a small set of requirements in order for the neural network to be an universal function approximator, namely that the function be nonconstant, bounded, continuous and monotonically increasing. We also desire that the function be quick to

evaluate, with a derivative that is simple to calculate. A function that fulfills all of these criteria is the sigmoid function:

$$\begin{aligned} \sigma(x) &= \frac{1}{1+e^{-x}}, \\ \sigma'(x) &= \sigma(x)(1-\sigma(x)). \end{aligned} \tag{5.27}$$

The sigmoid function is defined on the entire real number line, and outputs a number between 0 and 1. It also has a continuous derivative which is simple to calculate. The sigmoid function is well suited for binary classifiers, since it easily creates a decision boundary between two categories 0 and 1, and is often the first goto for AI programmers. Another simple and commonly used activation function is the hyperbolic tangent

$$\begin{aligned} \tanh(x) &= \frac{e^{2x}-1}{e^{2x}-1}, \\ \tanh'(x) &= (1-\tanh(x)^2). \end{aligned} \tag{5.28}$$

The hyperbolic tangent is defined on the real number line and outputs a number between -1 and 1. Both the sigmoid and the hyperbolic tangent functions exhibit the problem of vanishing gradients. In deep learning, you typically have several layers of neurons outputting some linear combination of the activation functions. Through the backpropagation algorithm, each neuron receives a weight update proportional to the partial derivative of the loss with respect to its weights. For the sigmoid and hyperbolic tangent activations these gradients will be in the range $(-1, 1)$, and therefore often exceedingly small. This means that many neurons may receive effectively no weight update in any given training epoch, which severely impedes training. The Rectified Linear Unit (ReLU) has gained popularity in recent years for its effectiveness in the field of convolutional neural networks. The ReLu function and its derivative is defined as:

$$\begin{aligned} \mathrm{ReLU}(x) &= \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases} \\ \mathrm{ReLU}'(x) &= \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases} \end{aligned} \tag{5.29}$$

Since the ReLU function in principle has no upper bound, it does not suffer the effect of vanishing gradients. Instead, the ReLU function can produce exploding gradients, if one or more activations becomes very large. In practice the gradients are often "clipped", i.e. truncated above a certain upper bound. With the ReLU function one can also encounter "dying" neurons, since the input to any neuron that does not exceed zero is set to zero, which means that many neurons may receive no weight update. From the point of view of the neural network, the inputs are just dimensionless numbers, which are passed through layers until some output is produced. It is therefore often useful to standardize the inputs. A common method is to shift the inputs by their mean and normalize by their standard deviation:

$$x_i' = \frac{x_i - \bar{x}}{\sigma_x}. \tag{5.30}$$

Another method is to rescale the inputs to the range of activation function, meaning $[0, 1]$ for the sigmoid and $[-1, 1]$ for the hyperbolic tangent. This is often useful in speeding up the training of neural networks. In theory any shifting and rescaling can be reproduced simply by updating the weights and biases of the network. In practice however, as the weights often start as small numbers, it can take quite a few training cycles before the weights are of appropriate size. Standardizing or rescaling the inputs also imposes a penalty on the weights which in turn reduces overfitting.

The most common methods for training neural networks are variations on the simple gradient descent scheme as mentioned above:

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n - \eta \nabla F(\boldsymbol{x}_n). \tag{5.31}$$

Since we are interested in training neural networks, $\boldsymbol{x}_n$ represents the weights and biases of the network after n training cycles, η is our learning rate and $\nabla F(\boldsymbol{x}_n)$ is the gradient of the loss function. The learning rate η controls how fast we reach a given minimum. A small learning rate will require more training cycles and a larger learning rate will require fewer. The learning rate also controls the numerical stability of the method, if it is too large the weights may diverge and if it is too small the weight may never converge. Gradient descent also tends to exhibit oscillating behaviour around the minimum. These basic intuitions can be shown in figure 5.6. Often a schedule is imposed on the learning rate, such as reducing it by a fixed amount every few epochs or incorporating an exponentially decaying learning rate. As

discussed in [32, Mehta et. al] using gradient descent to optimize our neural network imposes some limitations:

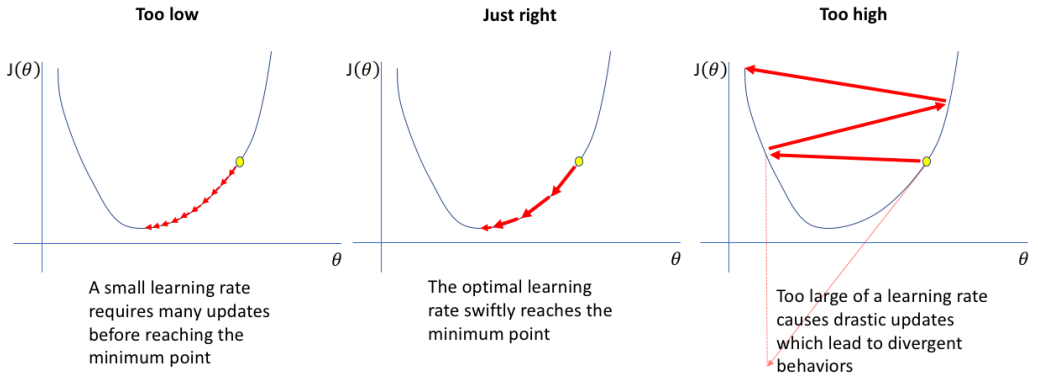


Figure 5.6: How gradient descent behaves with different (fixed) values of the learning rate. Reprinted from [www.jeremyjordan.me: Setting the learning rate of your neural network] (author Jeremy Jordan).

- *Gradient descent finds local minima* - if the GD algorithm converges, it will converge to a local minimum. This can lead to poor performance in complicated cost function landscapes
- *Gradients are expensive to compute* - the loss function often includes a term for each data point, which means the gradient also does.
- *GD is sensitive to learning rate* - as mentioned above, GD is very sensitive to learning rate
- *GD is sensitive to initial conditions* - gradient descent can take two different initial values and converge at two drastically different values
- *GD does not take curvature into account* - the learning rate for gradient descent is the same in all directions of parameter space, which means the maximum learning rate is set by the behaviour of the steepest direction

A common method for ameliorating some of these concerns is minibatching the inputs. Instead of calculating the gradient on the entire dataset for every epoch, we instead calculate the gradient on a subset of the data called a minibatch. If there are n datapoints and a minibatch size of m the total

number of batches is n/m. If we denote each minibatch $b_k,\ k = 1, 2, \ldots, n/m$ the gradient becomes:

$$\nabla \mathcal{C} = \frac{1}{n} \sum_{i=1}^{N} \nabla \mathcal{L}_i \rightarrow \frac{1}{m} \sum_{i \in b_k} \nabla \mathcal{L}_i, \tag{5.32}$$

i.e. instead of averaging the loss over the entire dataset we instead average over a minibatch. The cost function $\mathcal{C}$ is now decomposed as a sum over the loss function $\mathcal{L}$ at every datapoint i, $\mathcal{L}_i$. This significantly speeds up the calculation, since we do not use the entire dataset for every update. In addition, introducing stochasticity in the division of the dataset into minibatches introduces stochasticity to the weights update, which should help gradient descent in overcoming local minima. It is also common to introduce regularization to the weights and biases of the networks. Regularization in the context of neural networks means adding a term to the cost function proportional to the L_p norm of the weights and biases. For example using the L2-norm our cost function becomes:

$$\nabla \mathcal{C} = \frac{1}{n} \sum_{i} \nabla \mathcal{L}_i \rightarrow \frac{1}{n} \sum_{i} \nabla \mathcal{L}_i + \lambda \left|\left|\boldsymbol{w}\right|\right|^2 = \frac{1}{n} \sum_{i} \nabla \mathcal{L}_i + \lambda \sum_{ij} w_{ij}^2, \tag{5.33}$$

i.e. we sum up all the weights squared. The parameter λ is known as a regularization parameter. This extra term adds a penalty on the size of the weights dependent on the regularization parameter. This has been shown to reduce overfitting, as the weights cannot be adjusted to arbitrary size in order to fit the training dataset. Gradient descent is often paired with a momentum parameter that serves as a "memory" of the direction we are moving. This can be implemented as a modification to the gradient descent update:

$$\begin{aligned} \boldsymbol{v}_n &= \gamma \boldsymbol{v}_{n-1} + \eta \nabla F(\boldsymbol{x}_n), \\ \boldsymbol{x}_{n+1} &= \boldsymbol{x}_n - \boldsymbol{v}_n, \end{aligned} \tag{5.34}$$

where $0 < \gamma < 1$ is a memory parameter that controls the time scale for the memory. It is termed a momentum parameter because the equations for updating the parameters $\boldsymbol{x}$ are analogous to the equations of motion for a particle moving in a viscous medium. Momentum helps the gradient descent algorithm gain speed in directions where the curvature is flat while dampening speed in high curvature directions.

So called first-order gradient descent methods differ from quasi-Newton methods in that they do not keep track of the curvature which is encoded in the so-called Hessian matrix of second order derivatives. Second-order methods accomplish this by calculating or approximating the Hessian and normalizing the learning rate by the curvature. The RMSprop algorithm keeps a running average of both the first and second moment of the gradient. If we term the gradient as $\boldsymbol{g} = \nabla F(\boldsymbol{x})$ and the first and second orders as $\boldsymbol{m} = \mathbb{E}[\boldsymbol{g}]$ and $\boldsymbol{s} = \mathbb{E}[\boldsymbol{g}^2]$ we can write the update rule as:

$$\begin{aligned} \boldsymbol{g}_n &= \nabla F(\boldsymbol{x}_n) \\ \boldsymbol{s}_n &= \beta \boldsymbol{s}_{n-1} + (1-\beta)\boldsymbol{g}_n^2 \\ \boldsymbol{x}_{n+1} &= \boldsymbol{x}_n - \eta \frac{\boldsymbol{g}_n}{\sqrt{\boldsymbol{s}_n + \epsilon}} \end{aligned} \tag{5.35}$$

where β is a parameter which controls the time scale of the averaging and ϵ is a small regularization constant to prevent divergences. Operations involving vectors are understood to be element-wise. This results in a dampening of the learning rate in directions where the gradient is consistently large, and allows us to use a larger learning rate for areas of flat curvature. Perhaps the most commonly used optimizer today is the closely related ADAM optimizer. In addition to keeping a running average of the first and second-order moments ADAM performs an additional bias correction to account for the fact that we are estimating first and second order moments using a running average. The update rule is given as:

$$\begin{aligned} \boldsymbol{g}_n &= \nabla F(\boldsymbol{x}_n) \\ \boldsymbol{m}_n &= \beta_1 \boldsymbol{m}_{n-1} + (1-\beta_1)\boldsymbol{g}_n \\ \boldsymbol{s}_n &= \beta_2 \boldsymbol{s}_{n-1} + (1-\beta_2)\boldsymbol{g}_n^2 \\ \hat{\boldsymbol{m}}_n &= \frac{\boldsymbol{m}_n}{1-\beta_1^n} \\ \hat{\boldsymbol{s}}_n &= \frac{\boldsymbol{s}_n}{1-\beta_2^n} \\ \boldsymbol{x}_{n+1} &= \boldsymbol{x}_n - \eta \frac{\hat{\boldsymbol{m}}_n}{\sqrt{\hat{\boldsymbol{s}}_n} + \epsilon}, \end{aligned} \tag{5.36}$$

with β_1 and β_2 as memory parameters for the first and second moments respectively. This update rule effectively normalizes the learning rate by the

standard deviation of the gradient, which is a natural measurement scale. This also has the effect of cutting off directions where the gradient varies by a lot.

Quasi-Newton methods differ from the methods discussed above in that they make explicit approximations to the Hessian matrix of second order derivatives. These methods are discussed in great detail in the book [36, Numerical optimization: theoretical and practical aspects]. Let $W \approx (H^{-1})$ be an approximation to the inverse of the Hessian. A schematic algorithm is described in the book (p.53):

1. Step 0: Choose an initial $\boldsymbol{x}_0$, stopping tolerance ϵ and matrix W, positive definite. Compute the gradient $\boldsymbol{g} = \nabla F(\boldsymbol{x}_0)$.

2. Step 1: Compute $\boldsymbol{d} = -W\boldsymbol{g}$.

3. Step 2: A line search is performed with a step size $t = 1$, to compute the next iteration $\boldsymbol{x}_+ = \boldsymbol{x} + t\boldsymbol{d}$ and its gradient $\boldsymbol{g}(\boldsymbol{x}_+)$.

4. Compute the new matrix $\boldsymbol{W}_+$ and we return to step 1.

A full discussion of these methods is beyond the scope of this thesis, and the reader is directed to the book[36] for further details.

Chapter 6

Atom-centered descriptors

For small-scale systems, we have discussed the Hartree-Fock and Density Functional Theory methods as the primary workhorses of ab-initio electronic structure calculations. However, these methods suffer from very poor scaling as the system size increases, with Hartree-Fock naively scaling as $\mathcal{O}(N^4)$ with N the number of electrons and Density Functional Theory scaling as $\mathcal{O}(N^4)$, but with a larger proportionality scaling. In many cases the exact details of the electronic structure are less important than the long-time behaviour of the atoms and molecules involved in the simulation, and classical approximations can be made as in molecular dynamics, which comes close to linear scaling. This allows us to simulate systems of up to millions or hundreds of millions of atoms, which can approximate nano- or micro-scale systems if periodic boundary conditions are applied. However, the question remains as to how you develop an accurate classical potential which can accurately reproduce fundamentally quantum systems, with a speed that allows us to enter into realistic timescales (i.e. nano or microseconds).

The most common approach to developing molecular dynamics potentials is to guess a functional form based on your physical intuition and experience with the systems and calculate appropriate parameters from data obtained from DFT calculations. The number of parameters involved can range from two in the case of Lennard-Jones or hundreds of parameters in the case of complex, many-atom potentials such as the AMBER and CHARMM force fields. The imposition of functional forms to quantum data is an artform, and the potential must often be tailored to not only the chemical species and number of atoms in your system, but also the specific experimental quantity you are trying to extract, such as the energy, radial distribution

function or transport coefficients. Notably there are dozens of MD potentials describing different models of water (H2O), each fine-tuned for a specific system structure or parameter.

Due to recent developments in the field of machine learning, the question has been raised as to how it may be possible to automate the process of developing potentials. In their article introducing the Atomistic Machine-Learning Package (AMP) [9, Khorshidi, Alireza and Peterson, Andrew A.] outline one potential way forward. The idea is to approximate the potential energy with a regression model:

$$\{\boldsymbol{R}\} \overset{\text{Regression}}{\longrightarrow} E = E\left(\{\boldsymbol{R}\}\right), \tag{6.1}$$

where $\{\boldsymbol{R}\}$ is the set of nuclear coordinates of our system. Most machine learning methods operate over a set of one-dimensional so-called *feature vectors*, where every vector element represents a feature of the data set. For example the amount of precipation in a given area at a given time is a function of features such as humidity, cloud cover, air pressure etc. This is a vector of some length D, while the nuclear coordinates represent a point in $3N$-dimensional phase space. This difference in representation requires some way of mapping the nuclear coordinates to features which can be employed by a machine learning method. The naive approach would be to simply feed in the nuclear coordinates as a 1D vector, and then perform a regression on the dataset in order to obtain the potential energy. However, our physical intuition imposes some constraints on the potential energy. In particular, the potential energy of a microscopic system should be translationally, rotationally and permutationally invariant.

Translational invariance implies that the addition of any three-dimensional vector to every coordinate in the system should not in any way alter the potential energy of the system. This should not be the case with a naive mapping, as for a given set of weights (or equivalent) smaller/larger coordinates values would be mapped to smaller/larger activations, and therefore alter the final output. Rotational invariance implies that the potential energy of the system should not change as the system is rotated about an axis. This also should not be the case in the context of a naive mapping, as any change to any of the inputs would be mapped in a non-linear way to produce a different output. Finally, permutation invariance implies that swapping the coordinates of any two atoms of the same chemical species would produce

the same potential energy. This should also not be the case, for the same reasons as we just discussed. These constraints together heavily restrict the functional form that any mapping to the potential energy could have, which means a more careful analysis should be considered.

In order for the mapping to be applicable to systems of varying size, a decomposition into atomic energy contributions is performed:

$$E(\{\boldsymbol{R}\}) = \sum_{i=1}^{N} E_{\text{atom}}(\{\boldsymbol{R}\}). \tag{6.2}$$

The individual energy contributions E_{atom} are then approximated by performing a regression analysis. The atomic energy contributions are usually limited to its local environment through the introduction of a cutoff radius R_c:

$$E_{\text{atom}}(\{\boldsymbol{R}\}) \approx E_{\text{atom}}(\boldsymbol{R}_i, \{\boldsymbol{R}\}_j \,|\, |\boldsymbol{R}_{ij}| < R_c) \tag{6.3}$$

meaning that interactions are only treated if the interatomic distance is smaller than the cutoff. This is a good approximation for a sensible choice of cutoff radius if no electrostatic interactions are involved. Long-range interactions can also be treated through the introduction of methods such as Ewald summation, but this will not be discussed here. A mapping that satisfies the above constraints we will refer to as a *descriptor*, and is used as input to the regression method:

$$\{\boldsymbol{R}\} \rightarrow \boldsymbol{G}(\{\boldsymbol{R}\}) \overset{\text{regression}}{\longrightarrow} E_{\text{atom}} = E_{\text{atom}}(\boldsymbol{G}(\{\boldsymbol{R}\})). \tag{6.4}$$

Once we have a descriptor and a regression model the dynamics can be readily obtained by taking derivatives. The force on atom i is calculated as:

$$\begin{aligned} \boldsymbol{F}_i &= -\nabla_i E \\ &= -\nabla_i \sum_{i}^{\text{local}} E_{\text{atom}}(\boldsymbol{G}(\{\boldsymbol{R}\})) \\ &= -\sum_{i}^{\text{local}} \sum_{j} \frac{\partial E_{\text{atom}}}{\partial G_j} \frac{\partial G_j}{\partial \boldsymbol{R}_i}, \end{aligned} \tag{6.5}$$

where we have applied the chain rule to break the gradient into derivatives with respect to the network inputs (obtained through backpropagation) and derivatives of the network inputs with respect to the coordinates of atom i. Once we have obtained the forces the system can be propagated through time classically using for example the Velocity-Verlet equations.

6.0.1 Gaussian descriptors

In their paper on neural-network representations of potential energy surfaces [11, Behler, Jörg and Parrinello, Michele] suggested the decomposition of the mapping $\boldsymbol{G}_i$ of atom i into two subvectors $\boldsymbol{G}_i^I$ and $\boldsymbol{G}_i^{II}$ representing pairwise and three-body interactions respectively. The components of $\boldsymbol{G}_i^I$ are comprised of sums of gaussian functions of the pairwise distance R_{ij}:

$$f_i^I = \sum_{j \neq i}^{\text{local}} \exp\left(-\eta(R_{ij} - R_s)^2 / R_c^2\right) f_c(R_{ij}), \tag{6.6}$$

with the sum over the local environment of atom i. The parameters η and R_s represent the width and center of the gaussian functions respectively. The term f_c is a cutoff function which decays smoothly to zero at the cutoff radius. Behler and Parrinello proposed the following cutoff:

$$f_c(R) = \begin{cases} \frac{1}{2}\left(1 + \cos\left(\pi R / R_c\right)\right) & R < R_c \\ 0 & R \geq R_c, \end{cases} \tag{6.7}$$

however other functional forms are possible. The only requirements we pose is that the function be continuous with a continuous first derivative in $r \in [0, \infty)$, approach one as $R \to 0$ and zero as $R \to R_c$. The AMP authors propose an alternative polynomial cutoff function:

$$f_c(R) = \begin{cases} 1 + \gamma \cdot (r/R_c)^{\gamma+1} - (\gamma+1)(r/R_c)^{\gamma} & R < R_c \\ 0 & R \geq R_c, \end{cases} \tag{6.8}$$

where γ is a user-specified parameters that controls the rate of decay. For values of $\gamma < 2$ this cutoff reproduces the cosine cutoff, while for higher values the polynomial has much larger values within the cutoff radius. Figure 6.1 shows the cutoff functions plotted together for different values of γ. Figure 6.2 shows radial and angular parts of the symmetry functions with different parameter values, where the radial functions are centered at $R_s = 0$.

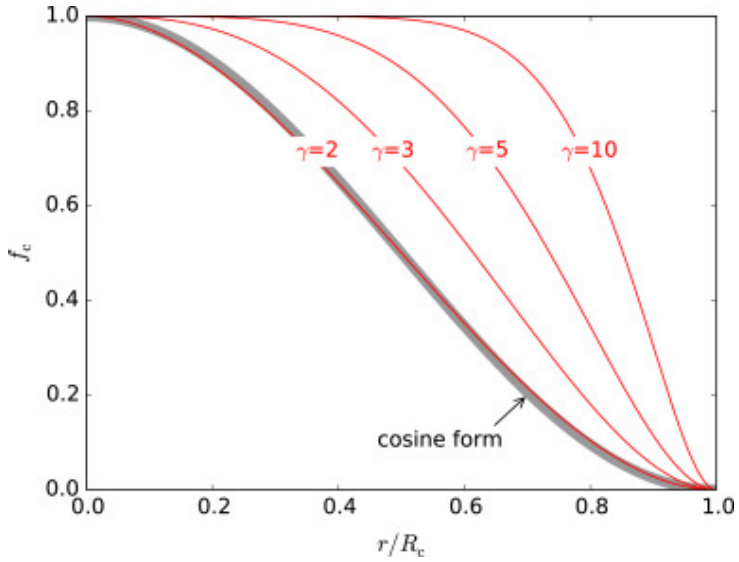


Figure 6.1: Cosine and polynomial cutoff functions plotted within the cutoff radius. Reprinted from [9, Khorshidi, Alireza and Peterson, Andrew A.].

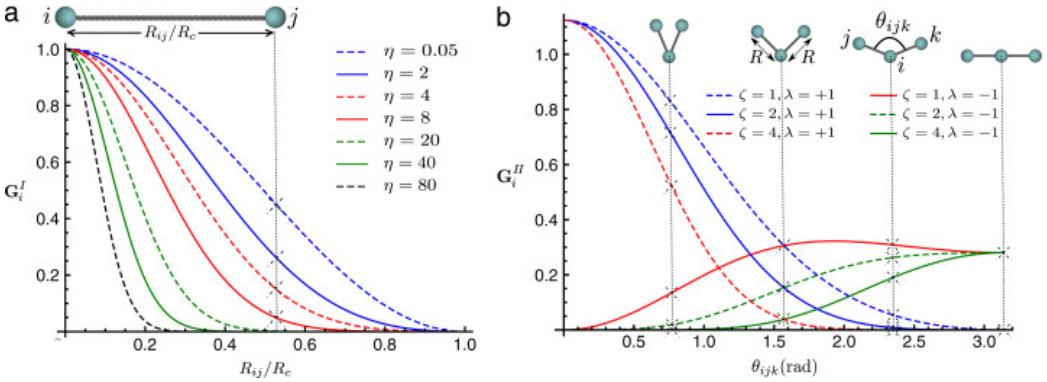


Figure 6.2: Radial and angular symmetry functions for different values of η, ζ, λ. The radial symmetry functions are centered at $R_s = 0$. Reprinted from [9, Khorshidi, Alireza and Peterson, Andrew A.].

The components of the three-body subvector are defined incorporating

the angles θ_{ijk} between every triplet of atoms:

$$f_i^{II} = 2^{1-\zeta} \sum_{j,k\neq i}^{\text{local}} (1 + \lambda \cos\theta_{ijk})^{\zeta} \exp\left(-\eta \left(R_{ij}^2 + R_{ik}^2 + R_{jk}^2\right)/R_c^2\right) \times f_c(R_{ij}) f_c(R_{ik}) f_c(R_{jk}). \tag{6.9}$$

The components for each subvector is calculated by varying the different parameters $\eta, R_s, \zeta, \lambda$. The choice of neither symmetry functions, cutoff, nor the parameters employed by Behler and Parrinello are unique. The guiding wisdom is that atomic environments with different potential energies should give differing energies, while remaining invariant under translation, rotation and permutation. Finally we note that the descriptors and the neural network models are not interchangeable, as each neural network is trained for a specific set of input vectors, and must be retrained if the way inputs are composed changes.

6.0.2 Zernike and bispectrum descriptors

Two closely related examples of atom-centered descriptors are the Zernike and Bispectrum descriptors. They are described by [9, Khorshidi, Alireza and Peterson, Andrew A.] in their paper on the Atomistic Machine-learning Package (AMP). Bispectrum descriptors are also discussed by [7, Behler, Jörg] in his perspective on machine learning potentials. Zernike descriptors represent a tensor product between spherical harmonics and Zernike polynomials. The local atomic environment of atom i is represented by an atomic density function $\rho_i(\boldsymbol{r}_i)$:

$$\rho_i(\boldsymbol{r}_i) = \sum_{j\neq i}^{\text{local}} \eta_j \delta\left(\boldsymbol{r}_i - \boldsymbol{r}_{ij}\right) f_c\left(||\boldsymbol{r}_{ij}||\right), \tag{6.10}$$

with $f_c(r)$ a cutoff function as described in the previous section. The 3-D Zernike basis set Z is formed by a tensor product between the Zernike polynomials basis set R and the spherical harmonics Y. The Zernike basis set is defined inside and on the surface of the S^2 unit sphere as:

$$Z_{nl}^m(r,\theta,\phi) = R_n^l(r) Y_l^m(\theta,\phi), \tag{6.11}$$

where $n \geq 0$ is an integer, l is restricted to even $n - l \geq 0$ and m is an integer such that $|m| \leq l$. Functions defined inside and on the S^2 sphere can be represented by the 3-D Zernike basis as:

$$f(r, \theta, \phi) = \sum_{n=0}^{\infty} \sum_{l} \sum_{m=-l}^{l} c_{nl}^{m} Z_{nl}^{m}(r, \theta, \phi), \tag{6.12}$$

with coefficients $c_{nl}^{m} = \langle Z_{nl}^{m}, f \rangle$ computed as projections of f onto the Zernike basis. These projections form the basis of the Zernike fingerprint, $\boldsymbol{G}_i$. Centering the atomic density preserves translation invariance of the atomic environment, while the projections onto the Zernike basis set preserves rotational invariance. Permutation invariance is maintained by keeping the constant values η_j the same within each chemical species. The Zernike descriptors are able to incorporate quadruple atomic interactions involving for example dihedral angles, while the gaussian descriptors are truncated at threebody interactions. The Zernike coefficients can also often be represented in terms of monomials, making them computionally cheaper than the bispectrum descriptors.

Bispectrum descriptors are computed much in the same way as the Zernike descriptors. The 4-D spherical harmonics form a complete, orthogonal basis set for the S^3 4-D unit sphere, and the components of the Bispectrum fingerprints $\boldsymbol{G}_i$ can be computed by projecting the atomic density function onto them. For more information the reader is encouraged to check out the [9, AMP paper].

6.0.3 Deep Potential Molecular Dynamics

Deep Potential Molecular Dynamics (DPMD) is a method proposed by [14, Zhang et al.] in response to the successes of methods such as Behler-Parrinello, Gaussian Approximation Potentials (GAP[37]) and Gradient-Domain Machine Learning (GDML[38]). These methods all involve some amount of handcrafting the inputs, and building these inputs for larger, more complex systems is not always straightforward. The Deep Potential method assigns a local environment and reference frame to each atom. The total potential energy is a sum of atomic contributions as before:

$$E = \sum_{i} E_{\text{atom}}\left(\{\boldsymbol{R}\}\right), \tag{6.13}$$

with the atomic energy determined by its nearest neighbors within a cutoff radius R_c:

$$E_{\text{atom},i} = E\left(\boldsymbol{R}_i, \{\boldsymbol{R}\}_j : |\boldsymbol{R}_{ij}| < R_c\right). \tag{6.14}$$

The position of each neighbor of atom i is described by the relative position $\boldsymbol{R}_{ij} = \boldsymbol{R}_j - \boldsymbol{R}_i$, which preserves translational symmetry. Rotational symmetry is conserved by constructing a local frame for each atom. Two neighboring atoms a and b are picked by a user-specified rule (default: two closest). The environment of atom i is then described by three unit vectors:

$$\begin{aligned} \boldsymbol{e}_{i1} &= \boldsymbol{e}(\boldsymbol{R}_{ia}), \\ \boldsymbol{e}_{i2} &= \boldsymbol{e}(\boldsymbol{R}_{ib} - (\boldsymbol{R}_{ib} \cdot \boldsymbol{e}_{i1})\boldsymbol{e}_{i1}), \\ \boldsymbol{e}_{i3} &= \boldsymbol{e}_{i1} \times \boldsymbol{e}_{i2}, \end{aligned} \tag{6.15}$$

where $\boldsymbol{e}(\boldsymbol{R})$ denotes the normalized vector $\boldsymbol{e}(\boldsymbol{R}) = \boldsymbol{R}/|\boldsymbol{R}|$. Together these vectors form an orthonormal basis for the reference frame of atom i. The local coordinates $\boldsymbol{R}_{ij}$ can then be obtained from the global coordinates $\boldsymbol{R}^0_{ij}$ through the transformation:

$$\boldsymbol{R}_{ij} = \boldsymbol{R}^0_{ij} \cdot \mathcal{R}, \tag{6.16}$$

where $\mathcal{R} = [\boldsymbol{e}_{i1}\ \boldsymbol{e}_{i2}\ \boldsymbol{e}_{i3}]$ is a rotation matrix with columns given by the local basis vectors. The neural network input vector for every atom-to-atom interaction $\boldsymbol{D}_{ij}$ can be given with radial-only or full radial-angular information:

$$\boldsymbol{D}^{\alpha}_{ij} = \begin{cases} \left(\dfrac{1}{R_{ij}}, \dfrac{\boldsymbol{R}_{ij}}{|\boldsymbol{R}_{ij}|}\right) & \text{full information,} \\ \left(\dfrac{1}{R_{ij}}\right) & \text{radial-only information,} \end{cases} \tag{6.17}$$

with $\alpha = 0$ when only radial information is specified and $\alpha = 0, 1, 2, 3$ when full information is provided. Radial information is typically sufficient for long-range interactions such as van-der-Waals forces, while covalent bonding can be modeled by including only the closest atoms. We therefore specify two separate cutoff shells, one for the radial information R_c and one for treating angular interactions R_a.

In order to preserve permutation symmetry the input vectors $\boldsymbol{D}_{ij}$ are sorted first according to chemical species, and then within each chemical species according to their inverse distance $1/R_{ij}$. The vector of subvectors $\boldsymbol{D}_i$ is then fed through a neural network to produce the atomic energy contribution. The network input size is fixed according to the maximum number of neighbors in the system which is being studied, with some entries $\boldsymbol{D}_{ij} = \mathbf{0}$ if there are fewer neighbors within the radial cutoff. Figure 6.3 shows how the neural network input vector is computed for the environment of a hydrogen atom.

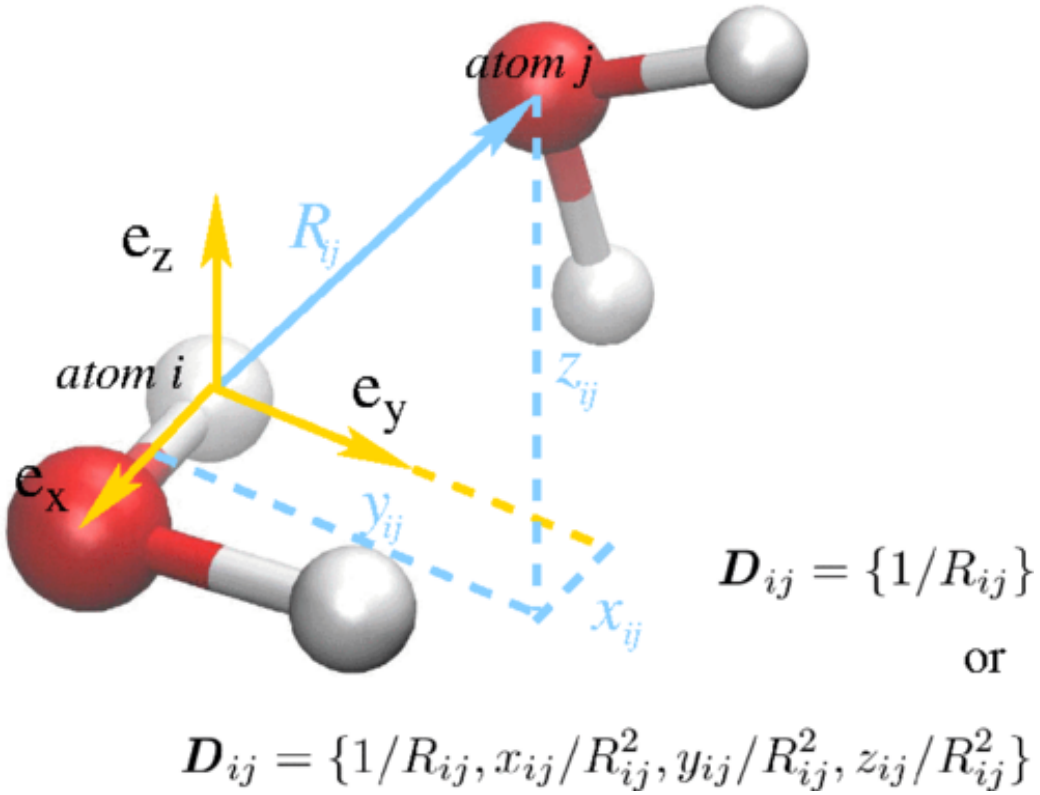


Figure 6.3: Computation of the neural network input vector for atom i. The coordinates of atom j are computed in the reference system of atom i. The vector $\boldsymbol{e}_x$ is along the OH bond, $\boldsymbol{e}_y$ is perpendicular to the plane of the water molecule and $\boldsymbol{e}_z$ is the cross product. Reprinted from [14, Zhang et al.].

The authors also propose a scheme of force learning, wherein the force

root mean square error (RMSE) is incorporated into the Cost function:

$$\mathcal{C} = \frac{p_\epsilon}{N} \sum_i \left(\Delta E_i^2\right) + \frac{p_f}{3N} \sum_i \left|\Delta \boldsymbol{F}_i\right|^2, \tag{6.18}$$

with p_ϵ and p_f energy and force pre-factors which are adjusted throughout the learning process. The term ΔE_i denotes the error between the network outputs and the correct potential energy, while the term $\Delta \boldsymbol{F}_i$ denotes the error in the force output. The pre-factors are adjusted based on the learning rate:

$$p(t) = p^{\text{limit}} \left[1 - \frac{r_l(t)}{r_l^0}\right] + p^{\text{start}} \left[\frac{r_l(t)}{r_l^0}\right], \tag{6.19}$$

with $r_l(t)$ and r_l^0 the learning rate at time step t and time step 0 respectively. The learning rate decays exponentially as:

$$r_l(t) = r_l^0 \cdot d_r^{(t/d_s)}, \tag{6.20}$$

with d_r the decay rate and d_s the decay steps. The decay rate should be less than 1. The force error is often a magnitude or two larger than the energy error, and it is believed that incorporating the force into the loss should improve the learning rate for physics-based applications which incorporate forces. The virial information can also be treated in this manner, but we will not discuss this here.

Part II

Implementation

Chapter 7

Atomic Simulation Environment

The Atomic Simulation environment (ASE[1]) [39, Larsen et al.] is a software package written in Python for the purpose of setting up, steering and analyzing atomistic simulations. Python is an interpreted, high-level general purpose language, with a powerful, consise syntax which allows one to perform very complex tasks with few lines of code. Python can also easily be extended and interfaced with fast and mature libraries. The modular interface of Python makes ASE easily extensible; in particular the calculator interface for evaluating energies, forces and much more has been implemented for open- and closed-source software packages such as LAMMPS, VASP, Quantum Espresso and many more. The Atomic Simulation Environment is intended to be:

- Easy to use
- Flexible
- Customizable
- Pythonic
- Open to participation

[1] https://wiki.fysik.dtu.dk/ase/index.html

The real drawback of Python is that it is an interpreted language, which results in slow execution. It can also be quite memory-intensive, which makes pure Python unsuitable for large scale computations and simulations. It is therefore common to write the computationally demanding tasks in a lower level compiled language, and build a Python interface for calling functions and classes.

7.0.1 Installation

ASE requires an installation of

- Python 2.7, 3.4-3.6
- Numpy 1.9 or newer
- Scipy 0.14 or newer

This can be easily obtained through the Anaconda or Miniconda packages[2], or follow the instructions on the Python website[3]. Once you have the prerequisites ASE can be installed using pip:

```
pip install ase
```

7.0.2 Molecular Dynamics

Here we will demonstrate how to setup a simple Argon crystal, set the velocities and integrate the system using the Velocity Verlet equations. First we import some prerequisites and define the system:

```
1 from ase.lattice.cubic import FaceCenteredCubic
2 from ase import units
3 from ase.md.velocitydistribution import MaxwellBoltzmannDistribution
4 from ase.md.verlet import VelocityVerlet
5
6 symbol = "Ar"
7 size = (3, 3, 3)
```

[2] https://anaconda.org/
[3] https://www.python.org/

```
atoms = FaceCenteredCubic(symbol=symbol, size=size, pbc=True)
MaxwellBoltzmannDistribution(atoms, 300 * units.kB)
```

This defines a face-centered-cubic (FCC) crystal unit cell with 4 atoms, and a system size of $3 \times 3 \times 3$ unit cells for a total of $4 \cdot 3^3 = 108$ atoms with periodic boundary conditions. We thereafter give the atoms velocities according to the Maxwell-Boltzmann distribution such that the system temperature is approximately 300 Kelvin.

Next we give the atoms a Lennard-Jones *calculator* for calculating dynamics and create a Velocity Verlet *integrator* for evolving the atoms according to their forces. Calculators will be discussed in the next section.

```
calc = LennardJones(sigma=3.405, epsilon=1.0318e-2)
atoms.set_calculator(calc)
dyn = VelocityVerlet(atoms, 2 * units.fs)
```

The parameters σ and ϵ define the well known length and energy scales for the Lennard-Jones potential. The Velocity Verlet integrator is initialized with a timestep of 2 femtoseconds, which strikes a balance between accuracy and the timescale of the simulation. Finally we run the system for 1000 steps:

```
for i in range(100):
    dyn.run(10)
```

Every 10 timesteps the system quantities can be printed, written to file or put through some other form of analysis or post-processing.

7.0.3 Calculators

The calculator interface gives ASE an easy to use, flexible and customizable way of computing dynamics, and makes ASE viable for a wide range of electronic structure calculations. For ASE a calculator is a black box that receives positions, atomic numbers and so on and outputs energies, forces, stresses; all that is required to perform atomistic simulations. The basic interface is defined in the ASE source code as:

```python
import numpy as np

class Calculator:
    def get_potential_energy(self, atoms=None,
                             force_consistent=False):
        return 0.0

    def get_forces(self, atoms):
        return np.zeros((len(atoms), 3))

    def get_stress(self, atoms):
        return np.zeros(6)

    def calculation_required(self, atoms, quantities):
        return False
```

There is also an interface for DFT calculators, which also requires the implementation of spins, occupation numbers, the fermi level and so on.

In the previous section we used the Lennard-Jones calculator as an example, however this calculator is not suited for general use. Apart from Lennard-Jones being a toy potential unsuited for real systems, the ASE implementation is also written in Python, which makes it very, very slow. Usually the calculator interface is used as a wrapper for much faster compiled code or software packages; for example the ASAP[4] calculator is a Python wrapper for the Effective Medium Theory potential, which is implemented in the Fortran programming language and is orders of magnitudes faster. A Python molecular dynamics code is justified since the bulk of computation time is spent on computing forces, however for large scale molecular dynamics simulations one would be advised to use fast compiled code such as LAMMPS[5] or GROMACS[6]. ASE really shines when it comes to small-scale simulations, experimentation and testing out new methods, and this is why it has been chosen as a basis for this thesis. Fortunately, ASE has an extensive code base of calculators for Molecular Dynamics and Density Functional Theory codes

[4] https://wiki.fysik.dtu.dk/asap
[5] https://lammps.sandia.gov/
[6] http://www.gromacs.org/

such as ASAP, CP2K[7], LAMMPS, GROMACS which are all implemented in lower-level languages.

[7] https://www.cp2k.org/

Chapter 8

Atomistic Machine-learning Package

The Atomistic Machine-learning Package (AMP) [9, Khorshidi, Alizera and Peterson, Andrew A.] is a software package written in Python with the intent of bringing machine learning to electronic structure calculations. The software is intended to interface with ASE and the OpenKIM API[1] for usage in LAMMPS. The interface to AMP is written purely in Python while computationally intensive tasks are outsourced to Fortran modules and supports neural networks through both Python code and through a Tensorflow backend. The Python interface makes AMP flexible and easily extended, and this makes the package ideal for prototyping and testing both newer and more established machine learning methods.

A suggested workflow is as follows[2]:

- Use AMP for training, testing and validation of novel descriptors and systems
- Use the AMP calculator for smaller scale simulation in the ASE environment
- Export the network compliant with the OpenKIM API for usage in more mature, large-scale electronic structure calculation software such as LAMMPS

[1] https://openkim.org/

[2] https://bitbucket.org/andrewpeterson/amp/issues/79/parallelize-fingerprint-derivatives

Unfortunately, the software is not currently fully compliant with the latest version of the OpenKIM API, which introduces artifacts in the simulation involving periodic images. However, the authors have recently received a large grant from the U.S. Department of Energy[3], which should facilitate further development.

8.0.1 Theory

AMP is intended to interface extensively with ASE, and the primary interface to AMP is the AMP calculator. The AMP calculator is an ASE compliant calculator that accepts cartesian coordinates and outputs energies and forces, just as a classical molecular dynamics calculator. The primary difference between the AMP calculator and ASE calculators such as the Lennard-Jones calculator is the *train* method, which accepts a set of *images*, i.e. a set of snapshots of atomic configurations labeled with the potential energy and forces. The calculator is fitted to the images and can subsequently be used to predict atomic energies and forces from never before seen atomic configurations.

[3] https://www.brown.edu/news/2018-09-20/simulations

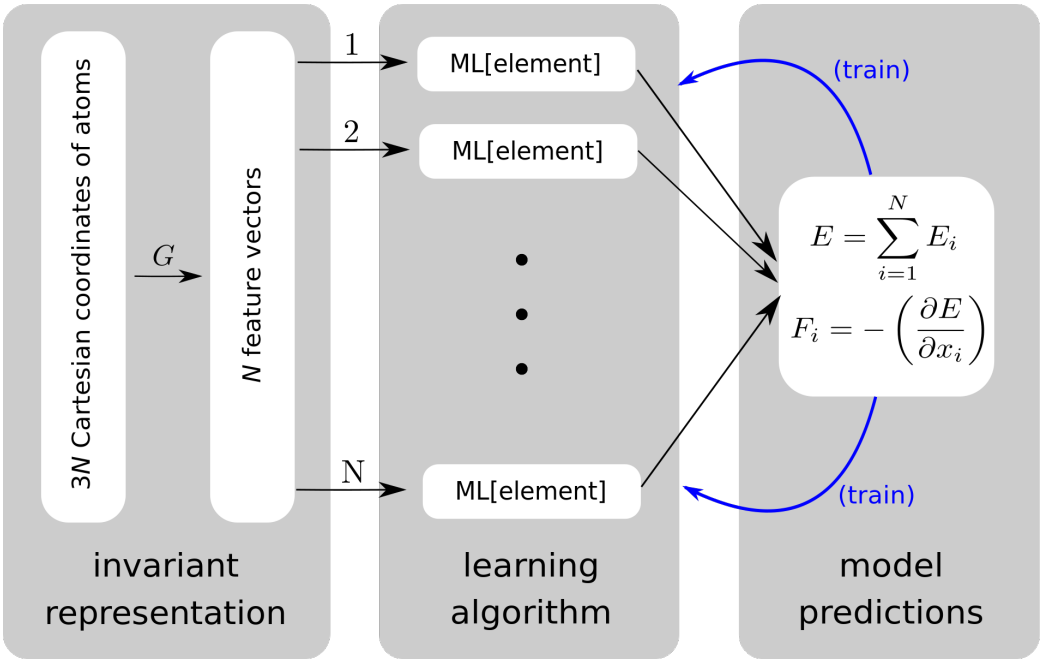


Figure 8.1: Schematics of how AMP works in atom-centered mode. Reprinted from [9, AMP paper].

AMP provides interfaces for both atom-centered descriptors, as we discussed in chapter 6, and image-centered descriptors, which are formed from the complete set of cartesian coordinates. Figure 8.1 shows the schematics of how the potential energy and forces is computed in atom-centered mode. A set of feature vectors are formed from an image, which are fed through a neural network to produce the potential energy. The neural network including backpropagation is written by the authors, though there exists a Tensorflow (0.12) interface only compatibable with Python 2. The inputs can be backpropagated through the network to produce both weight updates and derivatives with respect to the inputs. For the loss function, AMP provides support for custom implementations, but the default is the sum over atom-normalized residuals:

$$\mathcal{C} = \frac{1}{2}\sum_i \left\{ \left(\frac{E_i}{N} - \frac{E_{i,\mathrm{NNP}}}{N} \right)^2 + \frac{\alpha}{3N} \sum_j \left(F_{ij} - F_{ij,\mathrm{NNP}} \right)^2 \right\}, \tag{8.1}$$

where E_{NNP} and F_{NNP} represents the energies and forces predicted by the

neural network and α is a parameter that determines the relative weighting of the energies and forces in the loss function. Generally, an atomic configuration with N atoms will have 1 potential energy and $3N$ forces, which means incorporating the forces into the loss function can provide for much richer information into the potential energy surface. It is not known to us whether the energy normalization over the number of atoms represents an improvement over the simple sum of residuals squared.

8.0.2 Installation

AMP requires an installation of

- Python 2.7, 3.4-3.6, 3.6 is recommended
- Numpy 1.9 or newer
- Scipy 0.14 or newer
- ASE

Python installations can be easily obtained through the Anaconda or Miniconda packages[4], or follow the instructions on the Python website[5]. ASE installation is described in the chapter on ASE. Once you have the prerequisites AMP can be installed using pip:

```
pip install amp-atomistics
```

8.0.3 Training example

In the previous chapter we showed how to run a molecular dynamics simulation using ASE. Here we will show how an AMP calculator can be fitted to molecular dynamics data with only minor modifications to the code. First we import the prerequisites and define the system:

[4] https://anaconda.org/
[5] https://www.python.org/

```
import ase.io
from ase.lattice.cubic import FaceCenteredCubic
from ase import units
from ase.md.velocitydistribution import MaxwellBoltzmannDistribution
from ase.md.verlet import VelocityVerlet

from amp import AMP
from amp.descriptor.gaussian import Gaussian
from amp.model.neuralnetwork import NeuralNetwork
from amp.model import LossFunction

symbol = "Ar"
size = (3, 3, 3)
atoms = FaceCenteredCubic(symbol=symbol, size=size, pbc=True)
MaxwellBoltzmannDistribution(atoms, 300 * units.kB)
```

The class AMP is the primary object of the AMP package, and is implemented through the ASE interface. We will use a neural network machine learning model and Gaussian descriptors, which are an implementation of the Behler-Parrinello method. To implement force training we require access to the LossFunction class. We will also be using the ASE Input/Output (ase.io) module to generate an ASE *Trajectory*, which is an object storing the time-evolution of a simulation and usually interpreted as a time series of atoms.

```
traj = ase.io.Trajectory("training.traj", "w")
calc = LennardJones(sigma=3.405, epsilon=1.0318e-2)
atoms.set_calculator(calc)
atoms.get_potential_energy()
atoms.get_forces()
traj.write(atoms)
dyn = VelocityVerlet(atoms, 2 * units.fs)
```

In order to write the potential energy and forces to file they must first be calculated using the ASE Atoms methods, which require a calculator and otherwise raise an error. We can then evolve the system forward in time and save the atomic configuration every 10 time steps:

```
for i in range(100):
    dyn.run(10)
    atoms.get_potential_energy()
    atoms.get_forces()
    traj.write(atoms)
```

After the data has been generated we can train the AMP calculator using the Lennard-Jones calculations as input. We use a neural network model with three hidden layers, each containing 10 neurons. The training procedure will terminate once the energy and force root mean squared errors (RMSE) have reached values of 10^{-2} or less:

```
calc = Amp(descriptor=Gaussian(),
           model=NeuralNetwork(hiddenlayers=(10, 10, 10)))
calc.model.lossfunction = LossFunction(
           convergence={"energy_rmse": 1E-2,
                        "force_rmse": 1E-2})
calc.train(images="training.traj")
```

Once the calculator has been trained, the parameters are stored in a file called "amp.amp", and can be loaded using the command:

```
calc = Amp.load("amp.amp")
```

From there the calculator can be used as any other ASE calculator, in electronic structure calculations on Atoms objects.

8.0.4 Descriptors and models

Currently AMP provides support for three different descriptors. Gaussian descriptors implement the Behler-Parrinello method of radial and angular symmetry functions. For this descriptor the derivatives are also available, which means it can be used for computing dynamics. Some physical intuition and chemical knowledge is necessary to choose various parameters for the symmetry functions - unless you wish to do it by trial and error - but AMP provides defaults for multiple chemical species. The Zernike and Bispectrum

descriptors are also implemented, but without derivatives, which means the trained calculator cannot be used to for example perform molecular dynamics simulation. Currently the neural network and kernel ridge regression[6] models are the only models implemented. However, the modular nature of AMP should enable more additions in time, such as derivatives for the Zernike and Bispectrum descriptors or other regression models which have derivatives available.

[6] See for example: https://scikit-learn.org/stable/modules/kernel_ridge.html

Chapter 9

Fitting to the Lennard-Jones potential

In order to demonstrate the construction of a neural network potential, this chapter will demonstrate the use of Tensorflow to reconstruct the Lennard-Jones potential. The Lennard-Jones potential is the simplest realistic molecular dynamics potential, and since it is a function of only radial distance, symmetry functions are not required. We can thus use neural networks to perform a simple regression on a function accepting one input and producing one output. This serves to illustrate some of the intuitions and problems one runs into when using more complex methods such as atom-centered descriptors and gives a nice introduction into modern Tensorflow. We will be using the Tensorflow 2.0 beta version recently released, since it introduces a wide array of changes which will likely prove influential to the long-term direction of the Tensorflow project.

9.0.1 Tensorflow implementation

The code used to produce this implementation is a modification of the Tensorflow 2.0 for experts guide which is available online[1]. We start with some necessary imports from the Numpy and Tensorflow libraries. Tensorflow from 2.0 onwards is now primarily accessed through its Keras interface, which is a general high-level API for constructing and training neural networks. We also use a utility function from the scikit-learn library for splitting the

[1] https://www.tensorflow.org/beta/tutorials/quickstart/advanced

dataset into train and test data, but this is merely for convenience as it is very simple to implement from scratch. Finally we use the matplotlib and Seaborn libraries for producing the plots.

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import tensorflow as tf
from tensorflow.keras import Model
from tensorflow.keras.layers import Dense
from sklearn.model_selection import train_test_split
sns.set()
```

Our neural network is a simple feed-forward artificial neural network which inherits from the tensorflow.keras base model. It contains 2 layers with 50 and 10 neurons respectively using the hyperbolic tangent as the activation function. The neural network produces a single output which represents the potential energy and is computed as a linear function aka a dot product of the activations in the final hidden layer and the weights in the output layer. The gradient of the neural network with respect to the input can be computed using the tensorflow.GradientTape object. The GradientTape monitors the computation from the input to the output and is then able to backpropagate the output in order to obtain the gradient.

```
class MyModel(Model):
    def __init__(self):
        super(MyModel, self).__init__()
        self.d1 = Dense(50, activation="tanh")
        self.d2 = Dense(10, activation="tanh")
        self.d3 = Dense(1, activation="linear")

    def call(self, x):
        x = self.d1(x)
        x = self.d2(x)
        x = self.d3(x)

        return x
```

```
22
23      def derivative(self, x):
24          with tf.GradientTape() as t:
25              t.watch(x)
26              y = self.call(x)
27
28          return t.gradient(y, x)
```

The data is produced using the Lennard-Jones potential, which is a simple function of radial distance. The inputs need to be in the shape of matrices in order to be fed through the neural network in batches. We split the data into train and test using a test fraction of 0.2, with the remainder used for training.

```
29  def lennard_jones_data():
30      lj = lambda r: 4 * ((1.0 / r) ** (12) - (1.0 / r) ** 6)
31
32      x = np.linspace(0.90, 3.0, 10000).reshape(-1, 1)
33      y = lj(x)
34
35      x_train, x_test, y_train, y_test = train_test_split(
36          x, y, test_size=0.2
37      )
38
39      return (x_train, y_train), (x_test, y_test)
```

The data which is in the form of numpy arrays can easily be converted into tensors which can then be processed by Tensorflow. In order to perform regression we will use the mean squared error for evaluating the loss. We will use the Adam optimizer, which is currently the most popular optimizer for training neural networks, and generally outperforms the other less popular optimizers.

```
40  (x_train, y_train), (x_test, y_test) = lennard_jones_data()
41
42  train_ds = tf.data.Dataset.from_tensor_slices(
43          (x_train, y_train)
```

```
).shuffle(1000).batch(32)
test_ds = tf.data.Dataset.from_tensor_slices(
        (x_test, y_test)
).batch(32)

model = MyModel()
loss_object = tf.keras.losses.MeanSquaredError()
optimizer = tf.keras.optimizers.Adam()

train_loss = tf.keras.metrics.Mean(name="train_loss")
test_loss = tf.keras.metrics.Mean(name="test_loss")
```

The train and test steps are wrapped as Tensorflow functions so that they can be efficiently called and applied to tensors. For every train step the loss is computed and then backpropagated so that the optimizer can apply updates to the weights and biases. We subsequently train the neural network for 100 epochs, computing the train and test loss for every epoch.

```
@tf.function
def train_step(images, labels):
    with tf.GradientTape() as tape:
        predictions = model(images)
        loss = loss_object(labels, predictions)
    gradients = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))

    train_loss(loss)

@tf.function
def test_step(images, labels):
    predictions = model(images)
    t_loss = loss_object(labels, predictions)

    test_loss(t_loss)

epochs = 100

```

```
for epoch in range(epochs):
    for images, labels in train_ds:
        train_step(images, labels)

    for test_images, test_labels in test_ds:
        test_step(test_images, test_labels)

    template = "Epoch {}, Loss: {}, Test Loss: {}"
    print(template.format(
            epoch + 1, train_loss.result(), test_loss.result()
        )
    )
```

9.0.2 Comparison and absolute error

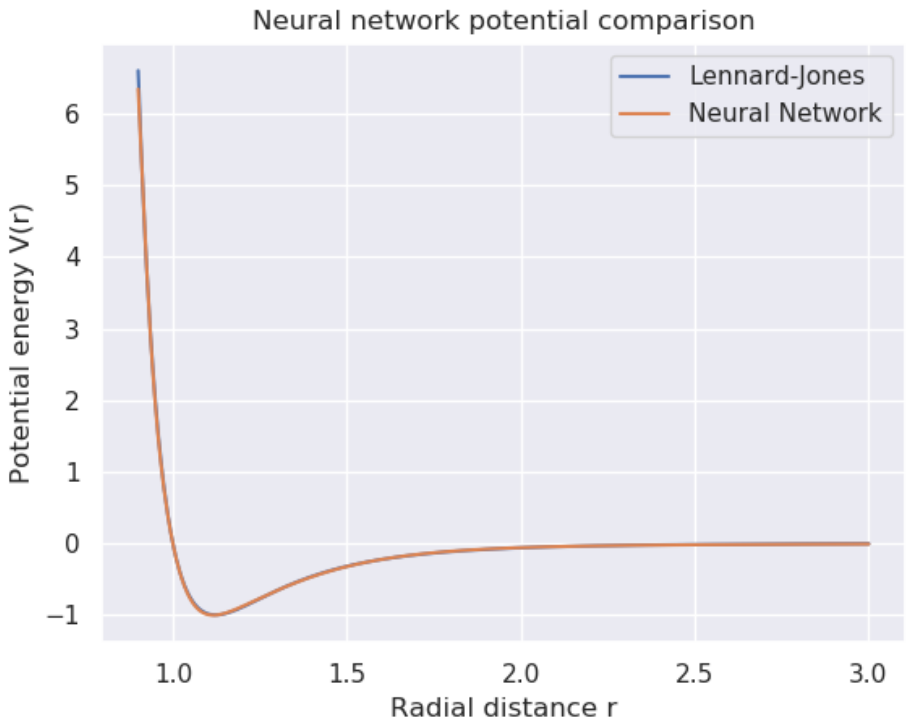


Figure 9.1: Neural network potential compared to Lennard-Jones.

After training the network we can evaluate its performance on test data. For the potential energy we obtain a root mean squared error of 0.56 (energy units). However, the root mean squared error does not tell the complete story, and the simplest way to evaluate the fit is to plot the potential. In figure 9.1 the neural network potential is plotted with the Lennard-Jones potential. We see that this potential is not difficult to reproduce, as the number of datapoints is relatively small, and we used a small number of epochs, amounting to only a few seconds of training. The potentials agree very well on most of the input space, and we can only spot a small divergence as the potential rises sharply around a radial distance of one.

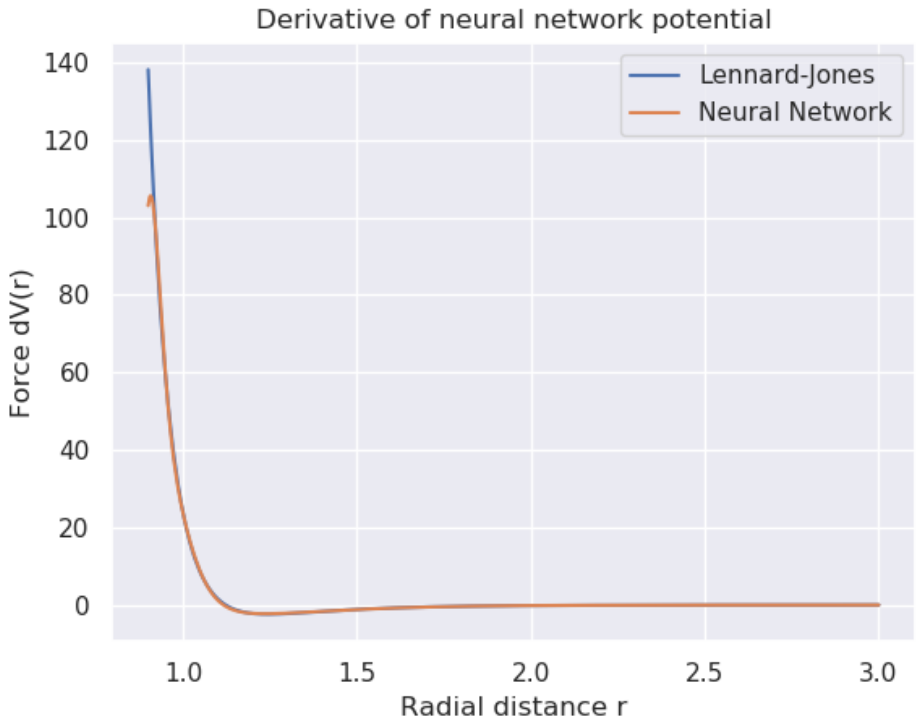


Figure 9.2: Neural network derivative compared to Lennard-Jones.

For the error in the derivative we obtain a force root mean squared error of 79.1 (energy units per length units), which is approximately two orders of magnitude larger. The derivatives are plotted in figure 9.2. The difference in derivatives is more pronounced, as the difference is much larger as the radial distance decreases from around one, and we can see that the neural network does not have the same asymptotic behaviour. This poses a potential problem for the dynamics, as the forces aka the derivatives keep the atoms from moving too closely and causing massive increases in the energy of the system. This implies that we should train the potential on a representative set of points in order to reproduce the correct asymptotic behaviour, especially at the edges of configuration space. It may also be beneficial to include force terms in the loss function, as this is expected to decrease the error in the derivatives.

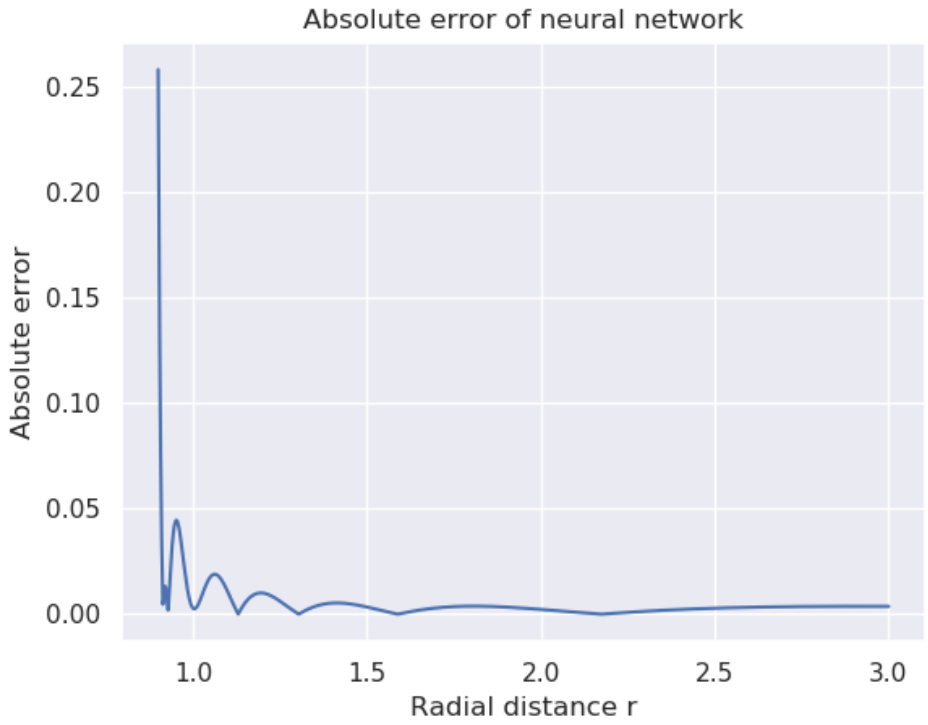


Figure 9.3: Absolute error of the neural network potential compared to Lennard-Jones.

In figure 9.3 we have plotted the absolute error of the neural network potential, defined as $|E_p - E_{\mathrm{NN}}|$ where E_{NN} is the potential energy from the neural network. From this we can see that the error in the fit is much larger at one edge of configuration space, as shown in the above plots. The neural network is able to reproduce small values at larger distances, but not the asymptotic behaviour as the interatomic distance approaches zero. Additionally, we can observe that the absolute error in the derivative is two orders of magnitude larger than the error in the potential energy fit. However, since this behaviour occurs only at the edge of the input space, dynamics could be preserved by training on a representative set of input data. Small distances between atoms in the system are highly unlikly configurations which should never be observed in molecular dynamics simulations, and they would cause problems regardless of the accuracy of the interatomic potential. This is occasionally a problem in conventional molecular dynamics, as for large

timesteps some atoms may "tunnel" too close, causing large spikes in the energy of the system. If the force accuracy is poor for some of the atoms and they move too close, we may also observe this using the neural network potential.

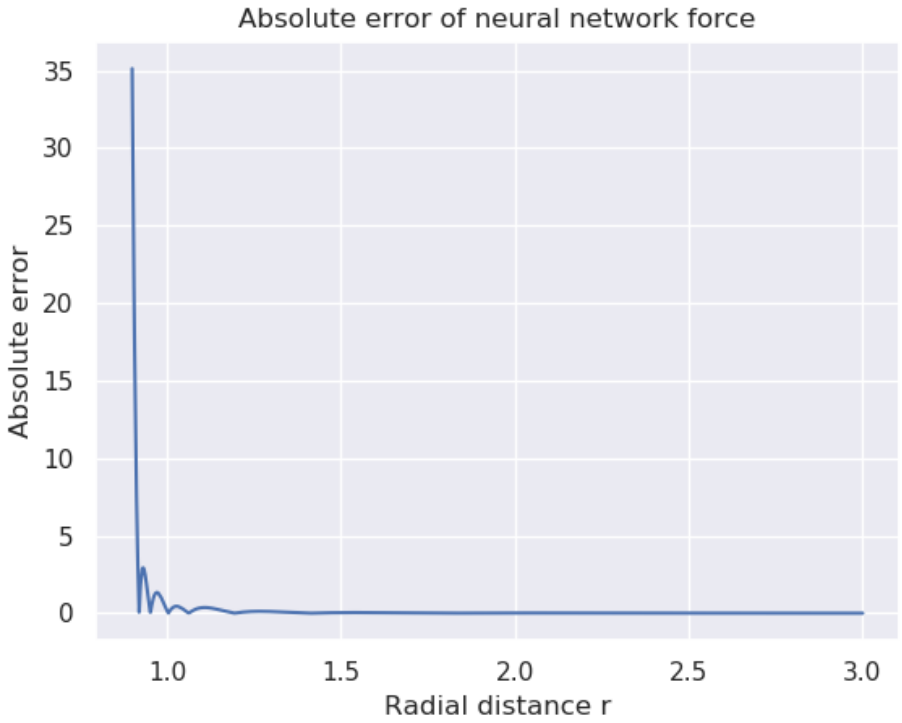


Figure 9.4: Absolute error of neural network derivative compared to Lennard-Jones.

Part III

Results

Chapter 10

Parameter search

In order to obtain optimal performance from the neural network given a set of atomistic configurations we need a careful choice of parameters and neural network architecture. The parameters can be classified as either *training parameters* - such as learning rate and force loss coefficient - or *architectural parameters* - such as the number of neurons and hidden layers, or the choice of interaction cutoff radius. The former are important in the training of the neural network, i.e. adjusting weights and biases while the latter influence both the training process and the final deployment of the neural networks on unfamiliar data. In this section we will be employing a grid search over a set of parameters, training multiple neural networks on the same data set and subsequently testing energy and force Root Mean Squared Errors on a smaller test data set. Generally we will only be training the neural networks on the energies, since training with forces is much more CPU- and memory-intensive. However, training with forces would likely affect the final result and improve the force RMSEs, as we will observe in the following section.

The parameters are unless otherwise specified the defaults listed in table 10.1. In their paper on Random Search, [40, Bergstra and Bengio] demonstrate that Random Search outperforms Grid Search for dimensions of search space larger than 3 or 4. However, since we have tested a small number of parameters (since training and testing is costly) and we have a general idea of what parameters are appropriate, we have chosen to employ Grid Search. AMP has a built in simulated annealing (see for example [41]) module, which performs simulated annealing on the weights and biases of the network, in order to find values for which the loss is lowest. The annealing is run for every trained neural network for 2000 steps at the beginning of the training procedure, with

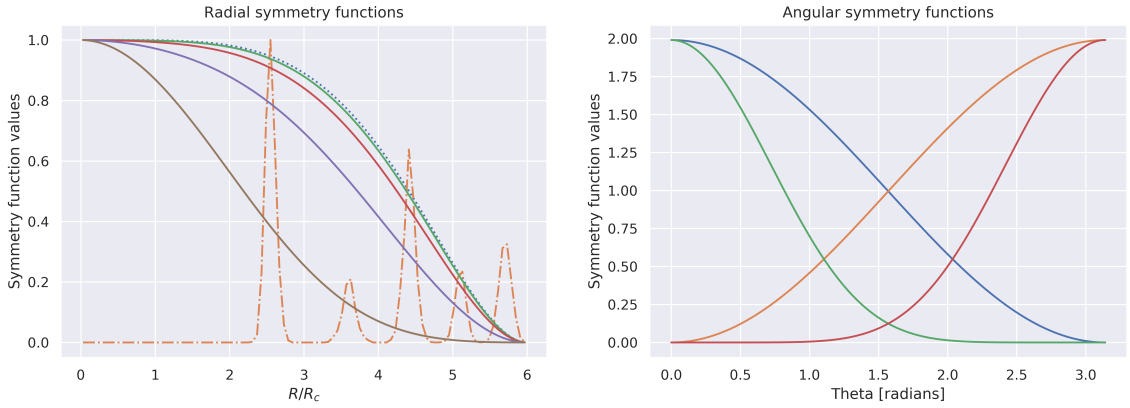
temperatures starting at $T_{max} = 20$ and ending with $T_{min} = 1$. Simulated annealing is a suitable algorithm for finding minima in a complicated energy landscape, and it reduces somewhat the randomness of initialization so that the results are comparable without having to perform many runs. We will also be using the BFGS Quasi-Newton (see for example [36]) method implemented through the *scipy.optimize* library. This is the optimizer interface currently provided by AMP and the BFGS optimizer generally performs the best. Finally we mention that AMP initializes the weights and biases of the neural network using Xavier initialization[35], as we discussed briefly in chapter 5.0.5.

Hyperparameters		
Architecture	Symmetry functions	12 radial, 20 angular
	Hidden layers	(10, 10)
	Activation	Hyperbolic tangent
	Cutoff function	Polynomial, $R_c = 6.0, \gamma = 5.0$
Training	Epochs	2000
	Energy coefficient	1.0
	Force coefficient	None
	Regularization	$\lambda = 10^{-8}$
	Optimizer	BFGS

Table 10.1: Training default values.

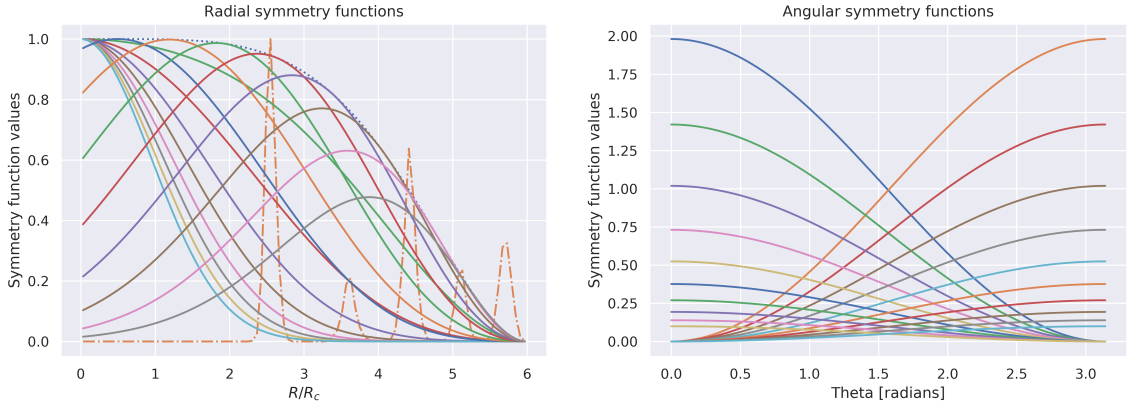
The AMP package provides a set of defaults for most elements, and these symmetry functions are plotted in figure 10.1. However, we did not feel that these covered the radial and angular space sufficiently so we constructed our own instead which is what we will use for training. AMP only provides centered radial functions, while we use a mix of centered and shifted radial functions since these have demonstrated higher performance, as is discussed in the section on symmetry functions. For the symmetry functions we have chosen a set of 6 uncentered radial, 6 centered radial and 20 angular (G4) symmetry functions. The uncentered radial functions have η's spaced evenly from 1 to 20 and are centered at 0. The centered radial functions are centered evenly from 0.5 to $R_c - 0.5$ with $\eta = 5.0$. The angular functions have η's spaced evenly from 0.01 to 3 with $\zeta = 1$ and $\gamma = \pm 1$. These symmetry functions are displayed in figure 10.2. We have also chosen to employ the

polynomial cutoff introduced in AMP with $\gamma = 5.0$, since this has a larger set of values inside the cutoff boundary. The numerical values of the parameters are available in A.



(a) Radial symmetry functions. (b) Angular symmetry functions.

Figure 10.1: AMP default symmetry function set. Radial functions plotted with radial distribution function.



(a) Radial symmetry functions. (b) Angular symmetry functions.

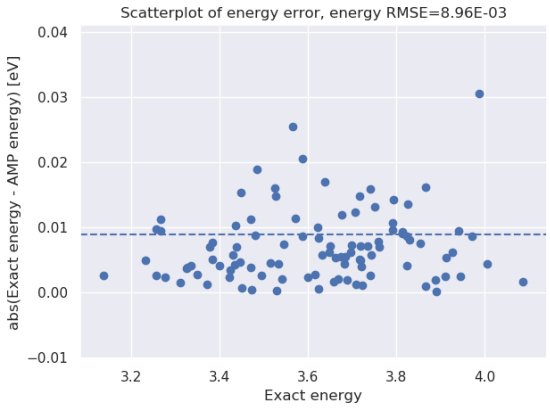
Figure 10.2: Selected symmetry function set used for parameter search. Radial functions are plotted with radial distribution function.

10.0.1 Force training

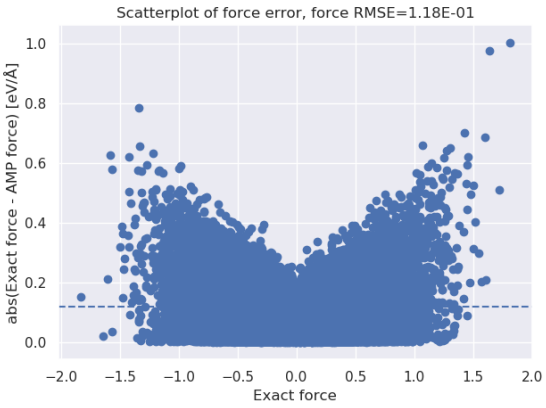
When training the neural networks we have the choice of whether to incorporate the forces into our loss function, or only fit the neural network to the potential energy. By default, unless we have access to a per-atom energy every configuration is labeled with only a single number for a potentially large number of atoms, which limits the improvement in loss metrics for every epoch, and the final result. If instead we incorporate the forces into the loss we have potentially $3N + 1$ labels for every epoch, which provides a lot more information for weight updates. In the previous chapter we also showed how adding derivatives to the loss function could significantly improve the accuracy of the derivatives. Since the forces determine the trajectories generated from molecular dynamics we would expect much better accuracy and numerical stability if we could improve the fit of the derivatives. The real drawback is the calculation of the derivatives in the input layer - aka the *fingerprintprimes*, of which there are a lot for every coordinate and input symmetry function - as they consume a lot of disk space, memory and CPU time.

In order to test the performance of neural networks we trained with and without forces on the same set of training images. A system of copper atoms is generated in the face-centered cubic (FCC) configuration with 4 atoms in the unit cell and $3 \times 3 \times 3$ unit cells for a total of $4 \cdot 3^3 = 108$ atoms. The atoms are given velocities from the Maxwell-Boltzmann distribution corresponding to a temperature of 500 Kelvin. The potential we will be using is the Effective Medium Theory (EMT), which has a very fast Fortran implementation in the ASE software package[1]. The training trajectory is ran for $5 \cdot 10^4$ steps with a timestep of $\Delta t = 5$ femtoseconds and written to file every 100 steps for a total of 500 atomic configurations. The test trajectory is integrated for $1 \cdot 10^4$ steps for a total of 100 atomic configurations.

[1] https://wiki.fysik.dtu.dk/asap/asap

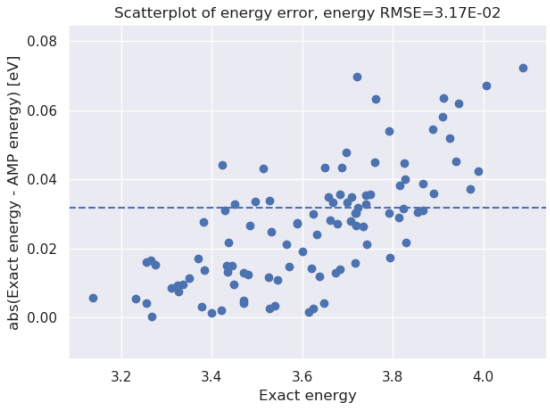


(a) Energy error.

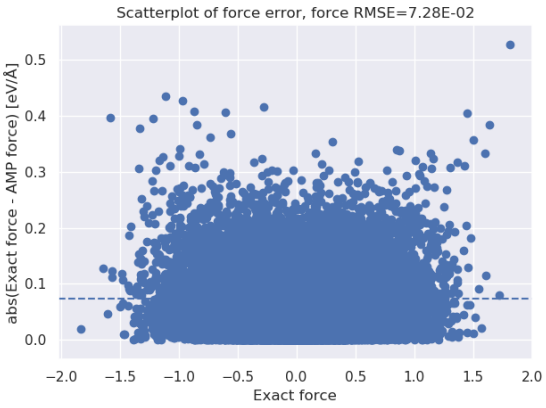


(b) Force component error.

Figure 10.3: Energy and force component absolute errors and root mean squared errors without force training.



(a) Energy error.



(b) Force error.

Figure 10.4: Energy and force component absolute errors and root mean squred errors with force training.

In figure 10.3 we have created a scatterplot of the energies and force components vs the absolute error after training without forces, and labeled the plots with the energy and force RMSEs. We observe that the potential energies can achieve fairly low error values of approximately 0.01 to 0.03 eV, and the result does not depend very much on the value of the exact potential

energy. The force error however is approximately an order of magnitude larger, which echoes some of the concerns raised in the previous chapter with fitting derivatives. The force errors produced by the neural also appear to be increasing as we move away from zero.

In figure 10.4 we have a scatterplot of energies and force components generated from a force-trained neural network with a force loss coefficient of 0.1. While the error in the potential energy is now approximately 3.5 times higher, the error in the forces is now about 60% smaller. This is arguably a good tradeoff, since the force error matters much more for long molecular dynamics trajectories, and will make any simulation requiring large amounts of data more stable. However, force training is as mentioned much costlier in terms of memory and CPU hours. Instead of training with forces on the full set of images, we instead suggest training on a large amount of configurations with only the energies (and arguably a small amount of atoms, since the ratio of fingerprints to labels is higher), and subsequently training with forces on a smaller set of images in order to improve the force fit, at some cost to the energy fit.

10.0.2 Activation, hidden layers

In order to test different network architectures we test different choices of the activation functions, and the number of hidden layers and nodes. At our chosen range of values, we do not expect a large variation in the performance. Since we do not have a large set of input parameters, we do not expect a large amount of nodes and hidden layers is required, as this would only increase the training time, and larger neural networks with a large amount of hidden layers might be expected to overfit the training data and generalize poorly. In the paper Efficient Backprop by [42, Lecun et al.] the authors suggest that the hyperbolic tangent outperforms the sigmoid, since the derivatives are larger, which provides more efficient training. The hyperbolic tangent is somewhat more computationally costly, however this is not a big consideration for our neural networks overall, since they are quite small and can be evaluated relatively fast. Inputs should generally be either standardized or scaled, as discussed in chapter 5.0.5, as these activation functions cannot sufficiently distinguish large inputs ($\tanh(1000) \approx \tanh(10000)$). AMP rescales inputs to the range $[-1, 1]$, which should be appropriate for both activation functions. To test different architectures a copper system of $4 \cdot 2^3 = 32$ atoms is generated and integrated for $8 \cdot 10^4$ steps with a timestep of $\Delta t = 5.0$ femtoseconds

and written to file every 100 steps for a total of 800 training images. By the same procedure we also obtain 200 test images. The velocities are generated according to a Maxwell-Boltzmann distribution with a temperature of 500 Kelvin. The results are shown in table 10.2. The network is trained only on energies, while for the test set we calculate the energy and force root mean squared errors (RMSEs).

Activation/Hidden layers	Energy RMSE	Force RMSE
tanh-[10]	1.78E-03	2.67E-01
tanh-[20]	1.83E-03	1.80E-01
tanh-[30]	1.71E-03	5.71E-01
tanh-[40]	1.81E-03	3.17E-01
tanh-[10, 10]	1.73E-03	7.71E-02
tanh-[20, 10]	1.65E-03	3.95E-01
tanh-[30, 10]	1.83E-03	3.68E-01
tanh-[40, 40]	1.87E-03	3.42E-01
sigmoid-[10]	4.58E-03	8.83E-02
sigmoid-[20]	3.52E-03	1.25E-01
sigmoid-[30]	4.72E-03	7.27E-02
sigmoid-[40]	5.05E-03	6.19E-02
sigmoid-[10, 10]	3.07E-03	2.38E-01
sigmoid-[20, 10]	4.89E-03	9.93E-02
sigmoid-[30, 10]	4.74E-03	9.22E-02
sigmoid-[40, 40]	4.09E-03	7.37E-02

Table 10.2: Training defaults

From the results we can see that the number of hidden layers and nodes of the network does not significantly influence the result. The hyperbolic tangent clearly outperforms the sigmoid for all sizes in the case of the energy, while the sigmoid appears to outperform when it comes to the forces. This is likely because we have not trained the network with force losses, which means the tangent is better able to fit the energy, and this degrades somewhat the performance on the forces. For the hyperbolic tangent with $[10, 10]$ we are able to achieve a low energy and force error, and we have seen this result consistently. This implies that this size and activation strikes an appropriate balance between the training epochs required and overfitting, and we have

settled on this architecture for the remainder of the thesis. However, it remains to be seen if a larger network trained for longer could outperform the smaller models.

10.0.3 Cutoff radius

The cutoff radius defines the boundary outside of which no interactions between atoms take place, and the magnitude of the interactions which take place within. We therefore expect it to have a reasonable effect on the final result. Ideally we might like all atoms to interact, as they plausibly do in the real world. However, since the cutoff radius defines a sphere with a volume $V \propto R_c^3$, the average number of neighbors and therefore calculations increases substantially with the cutoff radius. This means there is a tradeoff between the accuracy and speed when calculating fingerprints and fingerprint derivatives, with substantial CPU and memory cost the larger the cutoff radius. Accuracy may also be impacted if any atom interacts with an atom which in actuality does not have a substantial energy and force contribution. In figure 11.3 we have plotted the radial distribution function of copper atoms governed by the EMT potential, together with the polynomial cutoff function with $\gamma = 5.0$. The radial distribution function is scaled to 1 in order to compare with the cutoff function. The radial distribution function is highly peaked at $R \approx 2.5, 4.5$, with smaller peaks as the radial distance grows. This may justify a fairly small cutoff radius, but it is not clear how large the cutoff sphere should be until we have examined the results on the test set. It is nevertheless reasonable to assume a cutoff between 4 and 8 Angstrom would perform well, without sacrificing too much accuracy.

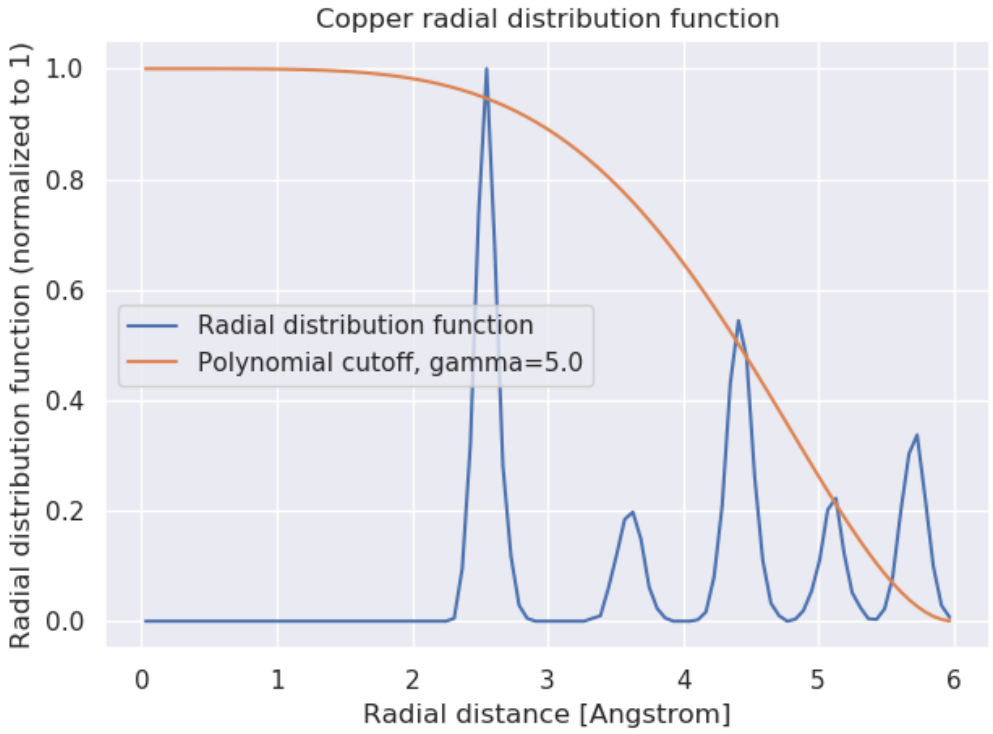


Figure 10.5: Radial distribution function of copper atoms governed by the EMT potential.

We will be comparing the cosine and polynomial cutoffs at different cutoff radii to test whether one outperforms the other. As in the previous section we produce a system with 32 atoms, with 800 training configurations and 200 test configurations. The network is only trained on the energy, since this is substantially cheaper, and then the network is evaluated on the energy and force RMSE on the test set. The results are presented in table 10.3.

Cutoff	Energy RMSE	Force RMSE
Cosine-2.0	2.18E-01	4.57E-01
Polynomial-2.0	2.18E-01	4.57E-01
Cosine-3.0	5.08E-03	3.96E-01
Polynomial-3.0	4.85E-03	4.12E-01
Cosine-4.0	1.34E-03	1.57E-01
Polynomial-4.0	1.21E-03	1.42E-01
Cosine-5.0	7.39E-04	2.05E-01
Polynomial-5.0	9.34E-04	1.91E-01
Cosine-6.0	1.99E-03	2.68E-01
Polynomial-6.0	2.42E-03	3.84E-01
Cosine-7.0	4.46E-03	2.04E-01
Polynomial-7.0	7.71E-03	1.18E-01
Cosine-8.0	6.40E-03	2.63E-01
Polynomial-8.0	1.05E-02	3.41E-01

Table 10.3: Cutoffs

From the energy RMSEs we can see that the intermediate values of the cutoff produce the lowest values. This may hint at some kind of overfitting on the training set, such that better values are obtained by considering fewer atoms more strongly. The intermediate values also seem to produce somewhat lower values of the force RMSE, of which the lowest is the polynomial with a cutoff of 7 Ångstrøm. We do not find a substantial difference between the polynomial and cosine cutoff functions, and these results indicate that not much is to be gained by choosing one over the other. We do note that there is a substantial stochastic element in these tests, and also that longer training periods may have produced more substantial differences, but training is costly in terms of CPU time. For the remainder of this thesis we settled on the polynomial with a cutoff of 5 Ångstrøm, while the cosine may have performed just as well.

10.0.4 Symmetry functions

The symmetry functions determine the local environment of every atom, which is then fed to and backpropagated through a neural network to produce the potential energy and forces, and should in theory have substantial impact

on the final results. The authors of AMP suggest through experience that the most important factors are the quality of data, the cutoff radius and the symmetry functions[2]. The symmetry functions should cover both the radial and angular environment of the atoms in the system we are trying to learn from. They should also be distinguishable in order for the neural network to be able to separate different chemical environments. Jörg Behler writes in his article on neural network potential energy surfaces [43, Behler] that the symmetry functions should describe the structures uniquely, and not be too highly correlated. In their article on Weighted Atom-Centered Symmetry Functions [44, Gastegger et al.] suggest that uncentered aka shifted radial symmetry functions perform better than symmetry functions centered at $R_s = 0$, since these offer better coverage of the radial space. They also suggest that this is not the case for angular symmetry functions, since they are products of multiple gaussians which narrows their range of values. Furthermore they suggest that for angular symmetry functions the parameter ζ should be limited to values of 1 or 2, and they demonstrate that increasing this parameter narrows the functions to a smaller range of angles close to the maxima (i.e. 0 or 90 degrees). We follow some of their suggestions and use a combined set of centered and uncentered radial functions. In the introduction to this chapter we defined a method for constructing the set of symmetry functions, with the parameters being the number of radial functions, angular functions, zetas and the type of angular function. To find the optimal set we perform tests of the energy and force RMSE while varying the number of radial and angular symmetry functions, and testing whether the G4 or G5 type of angular functions performs noticeably better than the other type. The results are presented in table 10.4. The first parameter is the number of radial η's, the second is the number of angular η's and the third is the number of ζ values.

[2] AMP mailing list

Symmetry function	Energy RMSE	Force RMSE
Default	8.07E-02	6.99E+00
Gs-4-8-1-G4	2.55E-03	8.54E-02
Gs-4-8-1-G5	3.11E-03	5.26E-02
Gs-5-9-1-G4	2.08E-03	2.42E-01
Gs-5-9-1-G5	2.65E-03	1.36E-01
Gs-6-10-1-G4	1.88E-03	6.60E-02
Gs-6-10-1-G5	2.14E-03	1.99E-01
Gs-7-11-1-G4	1.68E-03	2.98E-01
Gs-7-11-1-G5	1.96E-03	4.54E-02
Gs-8-12-1-G4	1.75E-03	1.38E-01
Gs-8-12-1-G5	1.66E-03	8.30E-02
Gs-9-13-1-G4	1.70E-03	1.00E-01
Gs-9-13-1-G5	2.58E-03	2.08E-01
Gs-10-14-1-G4	1.52E-03	4.30E-02
Gs-10-14-1-G5	2.58E-03	1.70E-01
Gs-10-14-2-G4	1.18E-03	1.71E-01
Gs-10-14-2-G5	1.92E-03	2.04E-01

Table 10.4: Symmetry functions

From these results we observe that the energy errors are marginally improved by increasing the number of symmetry functions, though the errors may be increasing at the upper end. This may indicate that an intermediate mix of symmetry functions offers appropriate coverage of the radial and angular space, while too many symmetry functions makes environments difficult to distinguish. The type of angular function seems to matter, but is dependent on the mix of symmetry functions, as one type outperforms the other dependent on the composition. We speculate that the force errors generally improve as we increase the number of symmetry functions, as forces represent the change in energy from a small perturbation of the position of a single atom, and this may be better represented by having a larger number of functions sensitive to this change. Our method of generating symmetry function markedly outperforms the AMP default symmetry functions, which suggest that we exercise some care when selecting the symmetry function set. As for the number of zetas, we only test two values with $\zeta = 1, 2$, thus doubling the number of angular symmetry functions, and this does not seem

to significantly improve the error rate, though we may have found larger differences if this was tested more.

10.0.5 Overfitting and regularization

Overfitting describes the phenomenom whereby statistical learning methods that perform well on a training set perform poorly on the test set, and indicates poor generalization. In order to guard against overfitting we try to ensure that the neural network architecture is not too complicated, and typically we add a term to the loss function proportional to the size of the weights and biases, as this ensures that no weight can be adjusted to arbitrary size to fit the training dataset. In order to test whether the neural network is overfitting the training data we generate the same configurations as before and vary the value of the regularization proportionality term λ. Once the neural networks have been trained we evaluate their performance on the test set. The results are shown in table 10.5.

Regularization	Energy RMSE	Force RMSE
1.00e+00	9.36E-02	3.92E-01
1.00e-01	9.33E-02	3.92E-01
1.00e-02	9.34E-02	3.92E-01
1.00e-03	9.34E-02	3.92E-01
1.00e-04	3.35E-02	2.97E-01
1.00e-05	9.86E-03	1.63E-01
1.00e-06	5.14E-03	1.20E-01
1.00e-07	2.29E-03	1.11E-01
1.00e-08	1.52E-03	3.02E-01
1.00e-09	1.32E-03	2.35E-01

Table 10.5: Regularization parameter search.

From these results we see that for values of the regularization parameter that are too large, the regularization is too punishing, and the training is not able to find any appropriate minima. The errors seem to improve as the regularization decreases, while the force errors rise slightly as the regularization continues to decrease. Since we have such a small neural network, large values of the regularization are too punishing, while small values are equivalent to

no regularization, and overfitting is not prevented. From these results we do not find much evidence of overfitting, but they may suggest some small regularization is appropriate for optimal generalization performance.

10.0.6 Sampling and scaling

In order for the neural network to learn patterns from data the data needs to be of high quality, and atomic configurations should be sufficiently distinguishable while being representative of the underlying distribution. Neural networks typically perform poorly or unexpectedly on unseen data, in addition to being data-hungry requiring fairly large amounts of data in order to train well over many epochs. One approach to learning atomic configurations might be to generate random configurations (not too closely spaced) and calculate energy and forces from these using a suitable calculator. However, this phase space of configurations would be enormous using only a few particles, and we would encounter many highly unlikely configurations. Since we want to train the neural network to perform molecular dynamics, the obvious solution is to sample configurations from molecular dynamics trajectories. However, these configurations are (mostly) in equilibrium, with relatively small forces, and thus the network might struggle if it encounters smaller distances between atoms. In their paper on fitting potential energy surfaces using neural networks, [45, Pukrittayakamee et al.] suggest using a variable time interval dependent on the maximum acceleration:

$$\tau = \begin{cases} \text{trunc}[\alpha/a_{max}]\Delta t & \text{trunc}[\alpha/a_{max}] > 0, \\ \Delta t & \text{trunc}[\alpha/a_{max}] = 0, \end{cases} \tag{10.1}$$

where Δt is a minimum time interval and α is a parameter which must be determined empirically. The rationale behind this is that we will sample more from areas where the forces are large, while skipping areas where the forces are small. In our experiments using this algorithm (slightly modified) we find that appropriate choices of Δt and α can broaden the distribution of force somewhat. However it does not add a substantial amount of larger forces, since these are simply not present in equilibrium. Other sampling algorithms have been suggested, for example iteratively pruning a large data set of images by removing images where the errors are not very large, and then retraining on the pruned datasets until we are satisfied with the result. One can also check for images where new values of the fingerprints are encountered

and add these to the training set, or checking images for large overall values of the feature vector. Behler in his review[43] on neural network potentials suggests that sampling from equilibrium states may leave holes in the potential energy surface, and suggests a few ways of dealing with this sampling problem. However, we have not had time to test these different methods. Other methods such as metadynamics[3] have seen increasing application and there are now numerous implementations available such as PLUMED [4]. In general, though sampling has not been studied closely in this thesis, an increase in high quality data is an increase in neural network performance, both for electronic structure calculations and for other applications. Neural networks are known as data-hungry, and typically require a large amount of data to perform well and also scale well with the number of data points. In order to test the accuracy as we add data points we train a neural network on progressively larger sets of images generated from molecular dynamics, and evaluate them on a test set. The results are shown in table 10.6.

Number of images	Energy RMSE	Force RMSE
10	1.33E-01	6.84E-01
20	2.51E-02	3.10E-01
50	1.07E-02	1.32E-01
100	5.58E-03	3.03E-01
200	2.00E-03	1.81E-01
500	1.78E-03	9.79E-02
1000	1.82E-03	3.07E-01
2000	1.58E-03	1.06E-01
5000	1.36E-03	1.62E-01
10000	1.42E-03	9.61E-02

Table 10.6: Neural network accuracy as a function of the number of images.

From these results we see that the energy RMSE scales favorably as the number of images available increase, and the force RMSE decreases as well, though not as systematically, due to stochasticity and the fact that the networks have not been trained on force. However, the gain in performance tails off after approximately 500 data points, which suggests that there is

[3] https://parrinello.ethz.ch/research/metadynamics.html
[4] https://www.plumed.org/

not much more to be gained from adding more configurations. Instead of sampling more configurations from equilibrium, we may be better served by looking at alternative sampling algorithms, sampling at higher or varying temperatures to introduce larger forces into the trajectory or introducing force training.

Chapter 11

Empirical potentials

In order to test the validity of using neural networks trained on molecular dynamics trajectories to generate new trajectories we train neural networks on systems of copper and silicon atoms using the Effective Medium Theory and Stillinger-Weber potentials respectively. These potentials have efficient implementations through ASE and their ASAP interface, which makes it ideal for our purposes. Additionally these potentials have an intermediate complexity, with Stillinger-Weber explicitly including threebody interactions, which makes them ideal for testing whether the Behler-Parrinello method can replicate this. At temperatures which are not too large these potentials describe atoms in a crystalline structure in equilibrium, and we test whether the neural network can reproduce the correct potential energy, forces, radial distribution and mean squared displacement. In table 11.1 we have listed the parameters we have used in the training process. While we have used a large amount of training images for the energy training, since this is relatively inexpensive, 8000 training images was not a noticeable improvement over approximately 5000-1000 images, and only seemed to affect the speed of convergence. We finally used 1000 images to refine the force accuracy, as we discussed in the previous chapter. In table 11.2 we have listed the parameters used for testing the neural network. For sampling data we used a larger timestep and sampling interval than for applying the neural network, in order to obtain a larger diversity of configurations, though this did not appear to matter much in the final analysis.

Hyperparameter	Value
Hidden layers	$[10, 10]$
Activation	Hyperbolic tangent
Time (fs)	$9 \cdot 10^5$
Timestep (fs)	5
Sampling intervall (timesteps)	100
Max epochs	4000
Regularization	$\lambda = 10^{-7}$
Optimizer	BFGS
Energy coefficient	1.0
Force coefficient	0.1

Table 11.1: Hyperparameters used in fitting.

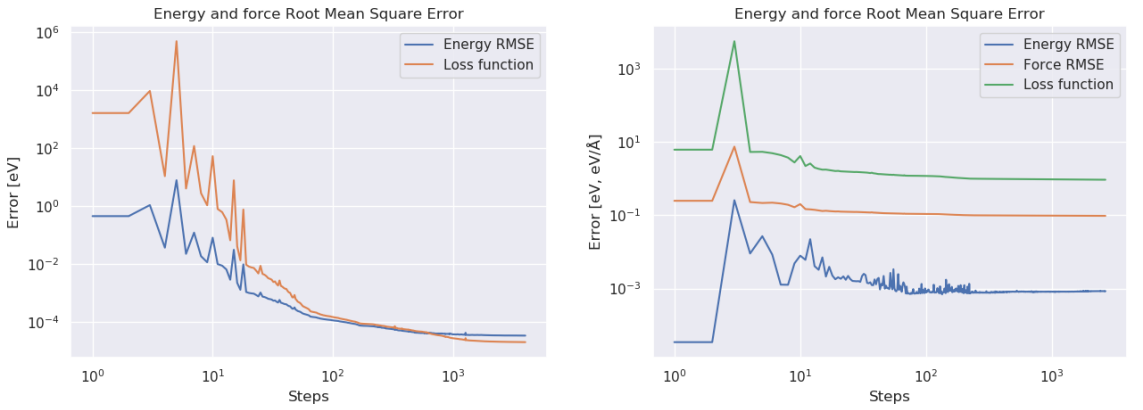
Hyperparameter	Value
Hidden layers	$[10, 10]$
Activation	Hyperbolic tangent
Time (fs)	$5 \cdot 10^3$
Timestep (fs)	1
Sampling intervall (timesteps)	10

Table 11.2: Hyperparameters used in testing.

11.0.1 Effective Medium Theory

The Effective Medium Theory (EMT) potential gives a good description of the late transition metals in a Face-Centered Cubic (FCC) crystal lattice, and has a very efficient implementation in ASE, which makes it ideal for producing large amounts of data. We will train on a rather small system of $4 \times 2^3 = 32$ atoms with a temperature of 500 Kelvin since this means a larger amount of labels available for atoms when we are only using the potential energy. We train with only the energy for $8 \cdot 10^5$ steps with a timestep of $\Delta t = 5.0$ fs writing to file every 100 steps and then subsequently train using both energy and forces for $1 \cdot 10^5$ steps for a total of 8000 and 1000 configurations. We train on both sets of images for 4000 steps, where the BFGS optimizer has generally stopped improving by much. The cutoff radius is set to 5 Angstrom, using 14 radial and 16 angular functions of the G4 type. After the calculator

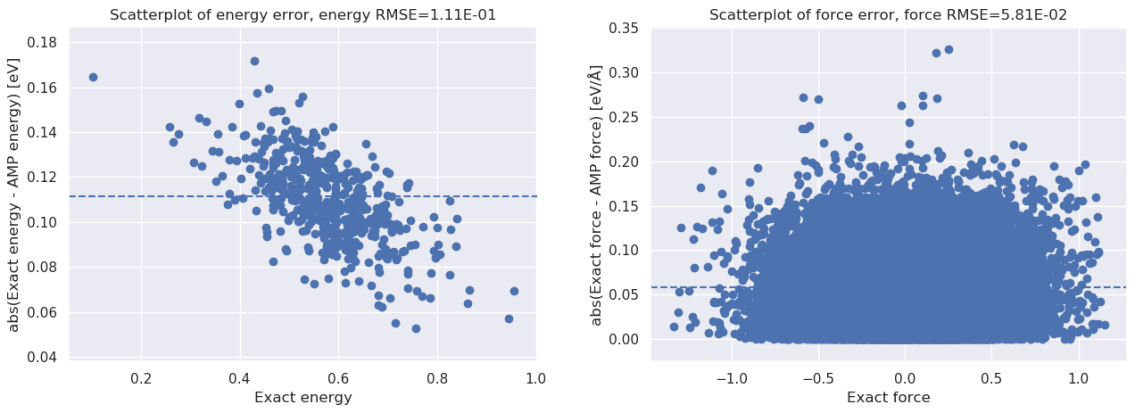
is trained we compare the performance of the neural network with the EMT potential on a system of 32 atoms with a temperature of 300 Kelvin for 5000 steps writing to file every 100 steps.



(a) Training loss and energy RMSE. (b) Training loss and energy and force RMSE.

Figure 11.1: Training loss, energy and force RMSE for the copper system (using logarithmic axes).

In figure 11.1 we have plotted the loss and root mean squared errors for the training process. After a few large oscillations in the beginning the losses generally settle down and begin to decrease more smoothly. After approximately 500-1000 steps the network seems to have converged, and the change in loss is much smaller than before. Subsequently we train with forces and we observe a large increase in the energy RMSE in exchange for a modest decrease in force RMSE, as discussed in the previous chapter. After training for a while both errors stop improving, and we consider the training converged. Note that the loss function is not equivalent to the energy or force root mean squared errors, as discussed in chapter 8.0.1.



(a) Energy error.

(b) Force component error.

Figure 11.2: Energy and force component errors on test trajectory.

In figure 11.2 we have plotted the energy and force component (i.e. in the x,y,z direction) absolute errors on the EMT test trajectory. We obtain an energy error of approximately 0.1 eV, with a max value of approximately 0.16 eV. For the force errors we obtain an force RMSE of 0.05 eV/Å, but some of the force errors considerably higher, up to and including values of 0.3 eV/Å, which may pose a problem for the long term stability of the system. These values are comparable to other works studying neural network potential energy surfaces, such as [12][13][9][14], at least for the forces. For the energy, the energy is higher than usual, and this is caused at least partially by training with force terms. Generally we observe that large force residuals cause an increase in energy and translational momentum over time. As we discussed in chapter 10.0.4 we believe this is because the sampling has left holes in the neural network.

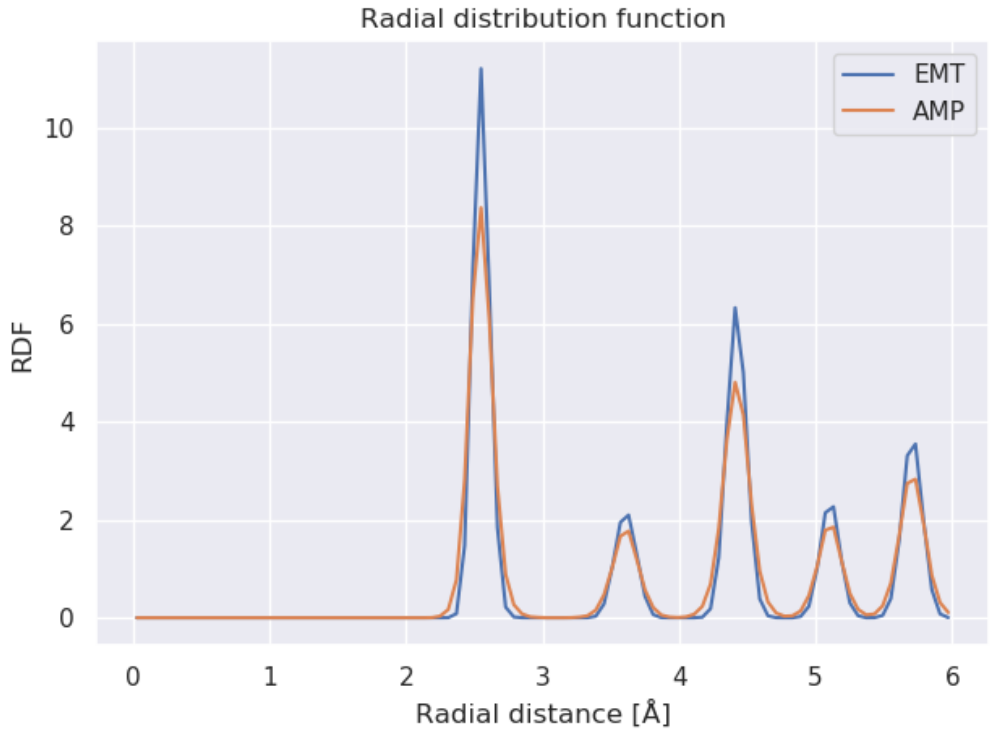


Figure 11.3: AMP radial distribution function plotted against EMT radial distribution.

In figure 11.3 we have plotted the AMP neural network radial distribution function compared to the EMT radial distribution. We see that the AMP potential can reproduce the copper crystal structure fairly well, though with smaller peaks. As we will soon discuss, the neural network appears to be able to reproduce the equilibrium crystal structure, though increases in kinetic energy and translational momentum over time makes the atoms more dispersed.

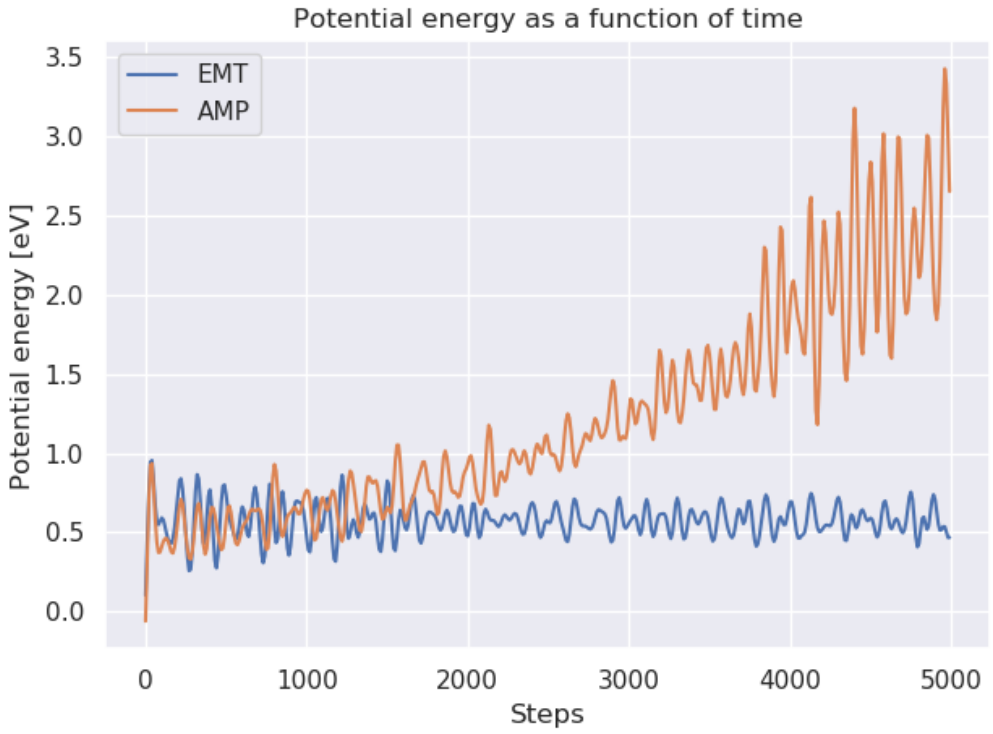


Figure 11.4: AMP and EMT potential energy as a function of time.

If we examine the potential energy as a function of time in figure 11.4 we see that the neural network follows the EMT potential energy fairly well for approximately 1500 steps, but then starts to significantly increase. As the atoms move away from the energy minimum due to an increase in kinetic energy, the potential energy starts to increase. This is most noticeable in figure 11.5, where we have plotted the total energy as a function of time. While the EMT energy is flat or oscillating around a mean value, the neural network potential exhibits increasing energy over time. At the beginning the increase in energy appears to be attributable to an increase in kinetic energy, which may be caused by errors in the interpolated forces from the neural network. This increase in kinetic energy also appears to lead to an increase in translational momentum.

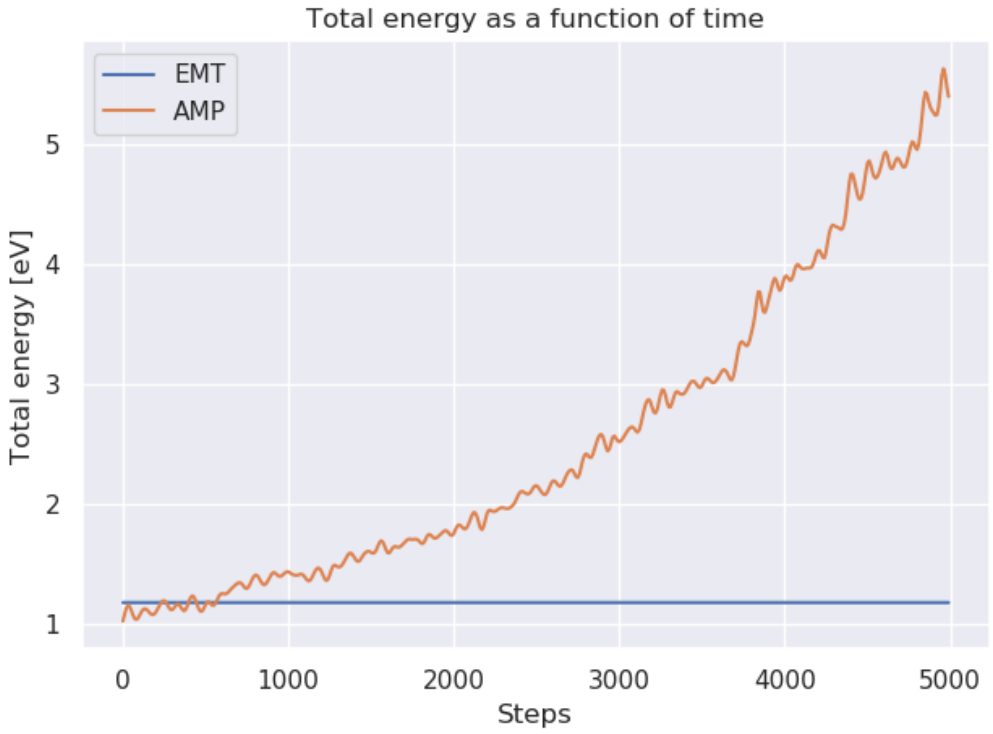


Figure 11.5: AMP and EMT total energy as a function of time.

In figure 11.6 we have plotted the mean squared displacement (MSD), which measures mean distance travelled averaged over all atoms in the system. We observe that the MSD for the neural network is significantly larger than for the EMT potential, and increasing non-linearly. In equilibrium we expect for a crystal lattice that the mean squared displacement be linear, as the atoms mostly oscillate in stable energy minimums. For the neural network potential the motion in the system appears to be increasing significantly, as the kinetic and total energy is increased over time.

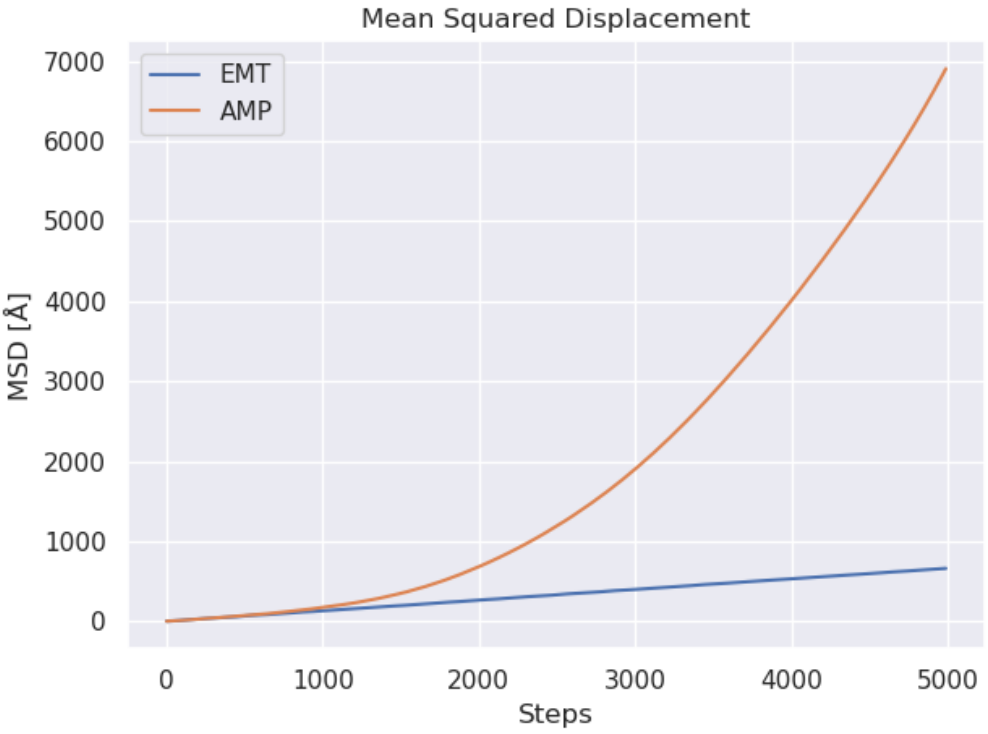
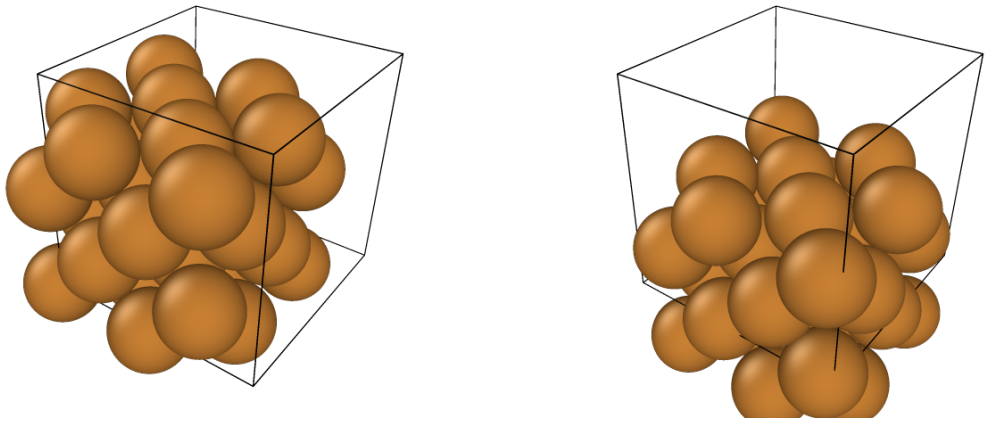


Figure 11.6: AMP and EMT mean squared displacements as a function of time.

If we examine the system trajectory in a program such as Ovito[1] we find that the crystal structure has mostly remained intact, while the system has picked up a certain amount of translational momentum. In figure 11.7 we see that the system has moved as a whole, although this is easier to see if you open up the trajectory file in Ovito yourself. This is in contrast to the EMT potential, in which the atoms vibrate in place, and the system remains more or less in place. Altogether, this suggests that while the neural network potential is able to reproduce the crystal structure, numerical errors propagate to a linear (possibly non-linear) increase in energy over time, which threatens the long-term numerical stability of the trajectory. In order to obtain better results we would likely require datasets containing more unlikely configurations and forces (i.e. slightly out of equilibrium). We also

[1] Open VIsualization TOol (OVITO)

generally find that the performance improves as we add more symmetry functions, particularly radial functions are belivied to improve the accuracy with this potential, as the potential contains no explicit treatment of angular interactions. However, more symmetry functions add significant CPU-time cost, and the set of symmetry functions would have to be pruned to remove significant correlations.



(a) Copper atoms after 10 steps.

(b) Copper atoms after 5000 steps.

Figure 11.7: The system of copper atoms after 10 and 5000 timesteps.

Finally, we tested the time-scaling of the neural network as the number of atoms increased. To test this we simply made a forces call on lattices of different sizes, which returns the force on every atom in the system. Ideally we would have taken averages over multiple force calls, however at these time scales we did not think it would significantly impact the results.

In figure 11.8 we see as expected that the trained neural network scales linearly with the number of atoms, though with a significant pre-factor. We see that it takes approximately 50 seconds to evaluate all the forces for a system of 50 atoms, while it takes 250 seconds for a system of 200 atoms. This pre-factor is dependent on the average number of neighbors of each atom, which is again dependent on the cutoff radius. Since the neural network has been trained with a set cutoff radius, this radius should be considered a part of the neural network architecture and cannot be significantly changed without impacting accuracy. This pre-factor is of course also dependent on the time it takes to evaluate the symmetry functions and derivatives, which is dependent on the number and type of symmetry functions, as the angular

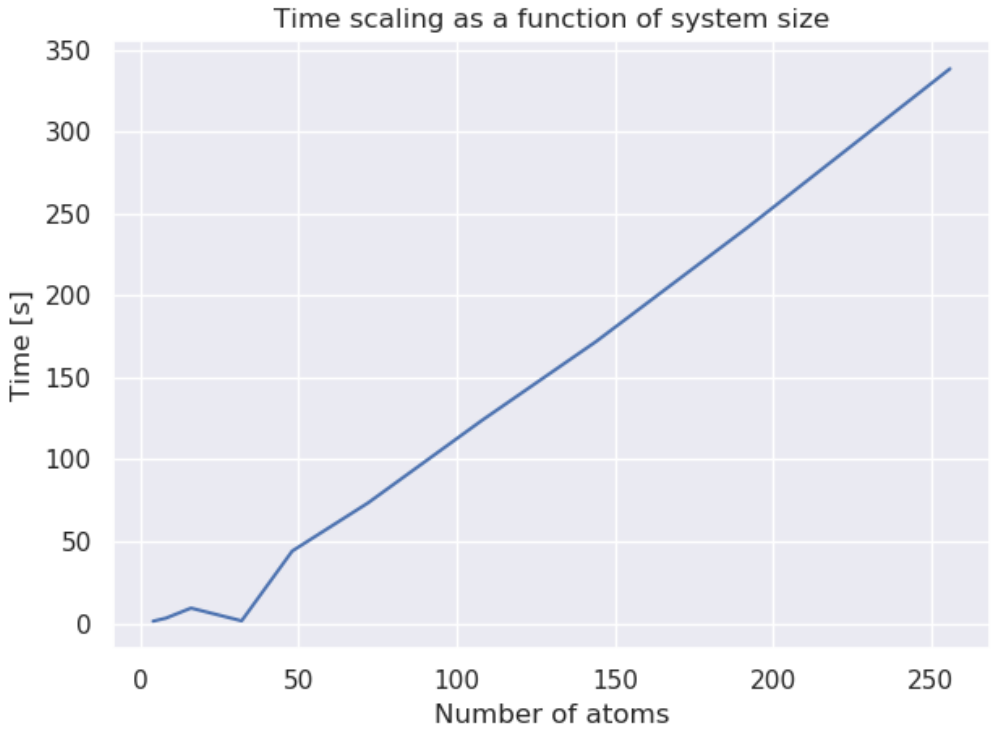


Figure 11.8: Time scaling of the neural network as the number of atoms increases.

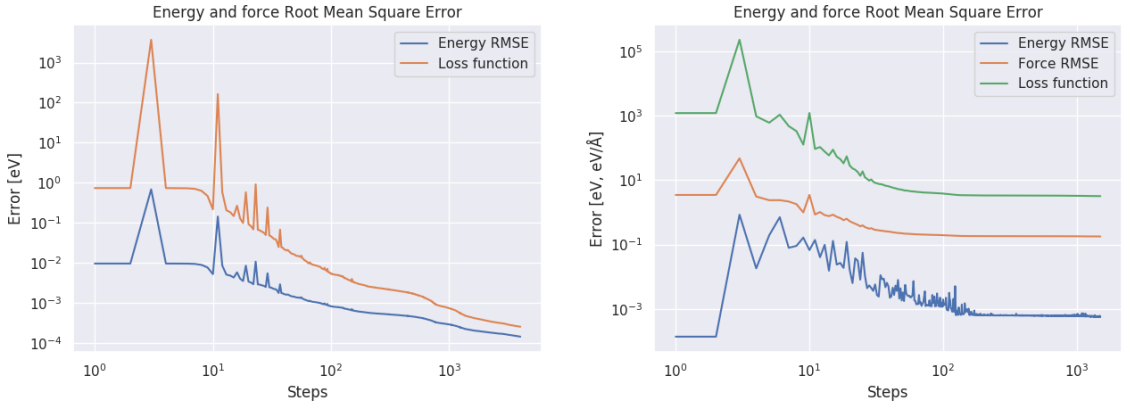
function derivatives are significantly more expensive to evaluate than the radial angular functions. For the system of 250 atoms the 5000 steps take approximately 2 weeks to integrate over, which cannot compete with classical potentials, often evaluated on the order of milliseconds. However, this is only on a single core, and parallelizing using neighbor list algorithms such as those found in the LAMMPS package could be a big improvement without too much overhead. While this would help deployment, training would still be too slow. As it stands now, the symmetry function derivatives have significant parts implemented in Python which could with some effort be moved entirely to Fortran, such as if-tests, dictionaries and neighbor lists. If these parts of the codebase were moved fully to a lower-level compiled language, this would help both training and deployment, and would enable training and testing of

larger and more complex systems[2].

11.0.2 Stillinger-Weber

The Stillinger-Weber is a potential which describes accurately Silicon atoms in the diamond lattice structure, and was one of the first potentials used to describe a realistic atomic-scale model of Silicon. It is also one of the most common examples of a potential with a threebody interaction, and its intermediate complexity makes it ideal for verification with for example quantum calculations or in our case machine learning methods. We initialize a system of 8 unit cells with 8 atoms in each unit cell for a total of $8 \times 2^3 = 64$ atoms with velocities corresponding to a temperature of 500 Kelvin. As in the previous section we integrate the system over $9 \cdot 10^5$ steps using a timestep of $\Delta t = 5.0$ fs (suitable for most metals in a crystalline structure) writing to file every 100 steps for a total of 8000 images for energy learning and 1000 images used to train forces. The neural network is trained for 4000 steps after using simulated annealing for 2000 steps in order to search for optimal initial weights. We then generate test sets using the Stillinger-Weber and trained neural network potentials integrated for 5000 steps and compare the results using the potential energy, radial distribution function, mean squared displacement and more. The cutoff radius is set to $R_c = 5$ Angstrom using 14 radial and 22 angular symmetry functions of the G2 and G4 types.

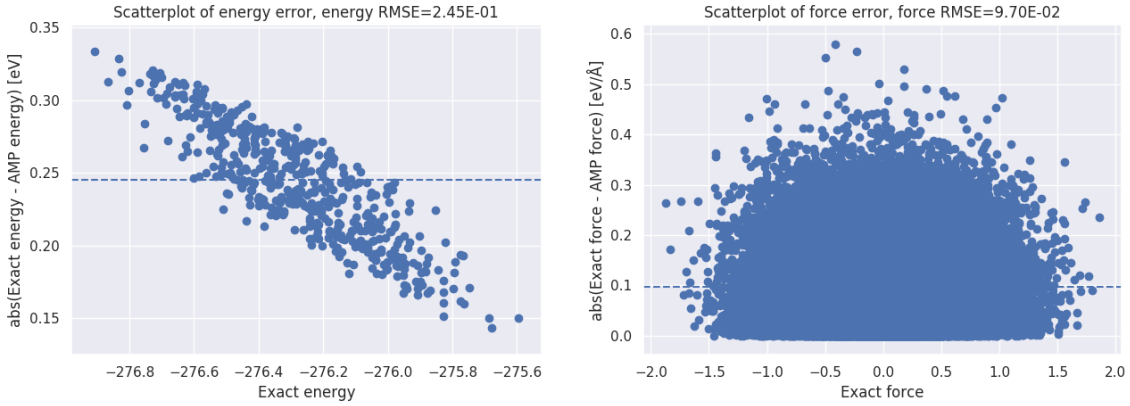
[2] See for example this exchange on the AMP mailing list.



(a) Training loss and energy RMSE. (b) Training loss and energy and force RMSE.

Figure 11.9: Training losses and energy and force root mean squared errors.

For both the energy training and the force training phases, the losses often rise sharply at the beginning, and then decay smoothly over time, reaching something of a convergence after approximately 500-1000 steps. These values are somewhat comparable to other works studying neural network potential energy surfaces, such as [12][13][9][14], at least in terms of the forces, though the energies are higher than usual. The losses are somewhat higher for the Stillinger-Weber than for the EMT potential, this may indicate more difficulty reproducing the distribution or overfitting for the EMT potential. This is also indicated in the test losses in figure 11.10, where both the energy and force RMSEs are slightly higher than for the EMT potential. This may also be an artifact of initialization, as finding good high-dimensional minima using gradient descent is a process which in some cases may require many restarts. However, if we examine the energies interpolated over time, it paints a better picture than for the EMT potential.



(a) Energy error.

(b) Force component error.

Figure 11.10: Energy and force component errors on test trajectory.

In figure 11.11 we have plotted the energy as a function of time. While the energy for the Stillinger-Weber potential is flat or oscillating around a mean value over time, the neural network exhibits as before a seemingly linear increase in energy over time. However, if we compare with the EMT potential the change over time is now significantly smaller.

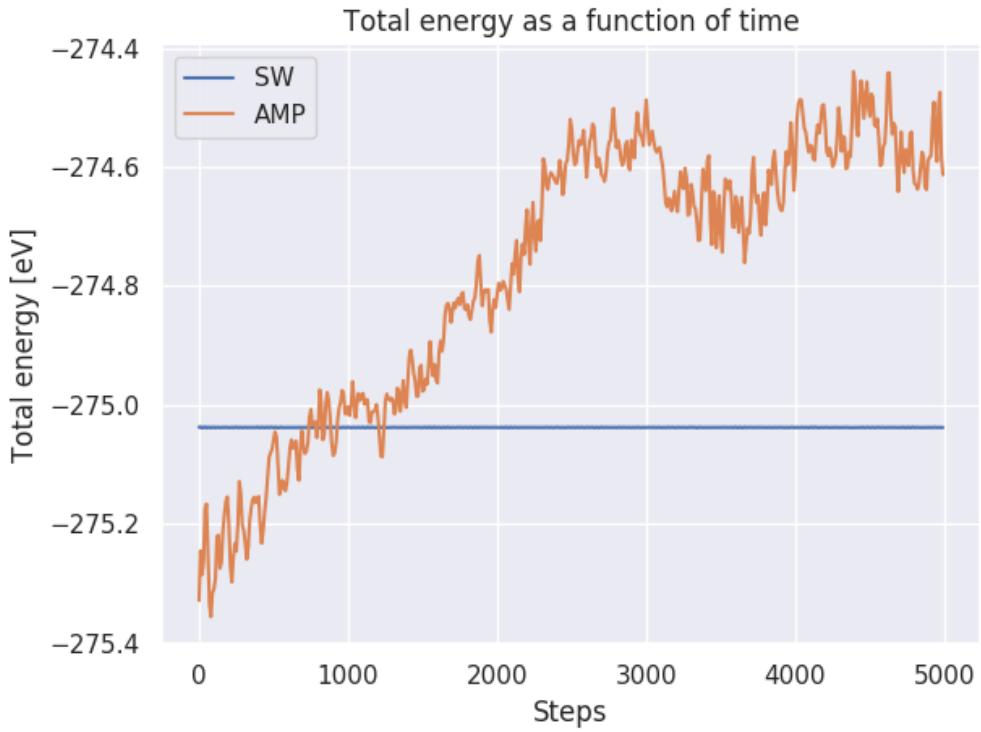


Figure 11.11: AMP and Stillinger-Weber total energy as a function of time.

This is also seen in the potential energy over time in figure 11.12, which is as before increasing, though the increase is smaller, and in general the potential energy fits better than before. We speculate on a few reasons for this. One reason may be that since angular interactions contribute significantly to the Stillinger-Weber potential, a mix of radial and angular symmetry functions provides a better fit than for the EMT potential, where only radial interactions are explicitly treated. Another reason may also be that the Silicon atoms are radially distributed more discretely, such that there are fewer neighbors for any given atom to consider. We expect in general that if the symmetry functions are not too correlated, adding more symmetry functions increase fit and numerical stability, but we are limited by the time it takes to evaluate their derivatives.

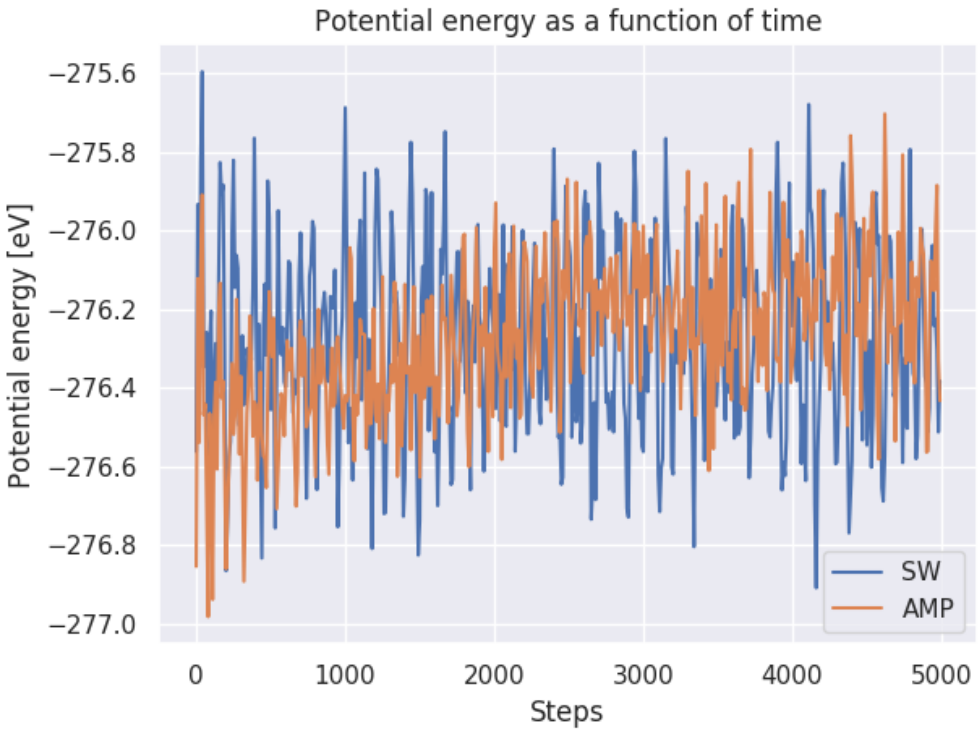


Figure 11.12: AMP and Stillinger-Weber potential energy as a function of time.

As before there are indications that the increase in energy is initially attributed to the kinetic energy, and as we see in the radial distribution functions in figure 11.13, the neural network system is slightly more dispersed than the Stillinger-Weber system. In addition in figure 11.14 we can see that the mean squared displacement is increasing in a non-linear fashion as the atoms pick up both kinetic energy and translational momentum.

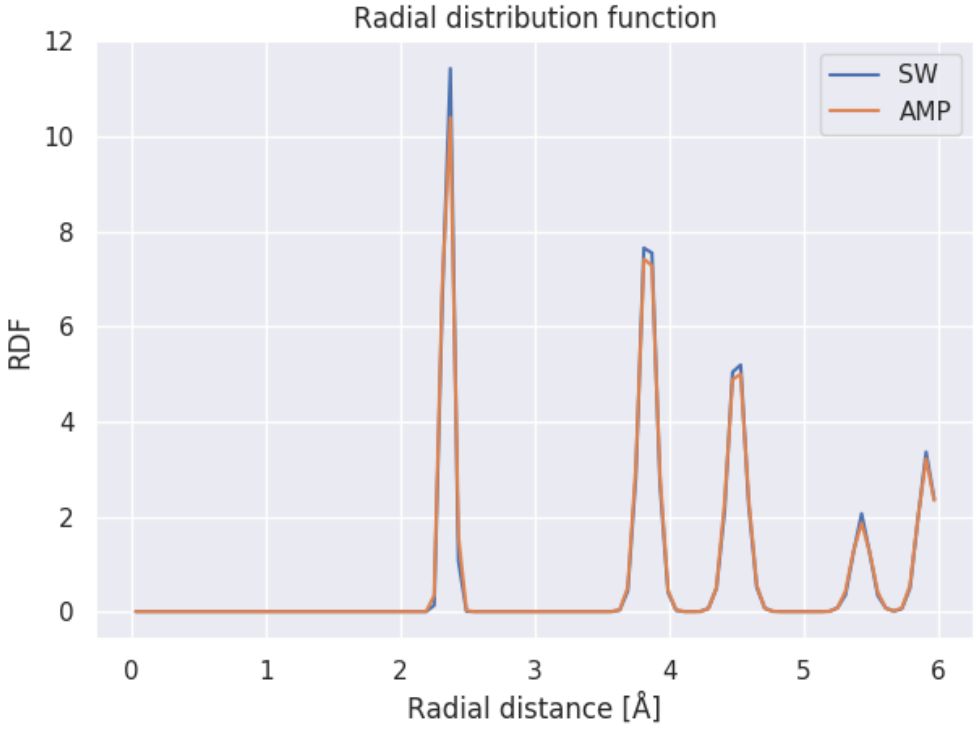


Figure 11.13: Radial distribution functions for the Stillinger-Weber and neural network potentials.

The increase in kinetic energy and translational momentum is illustrated both in the mean squared displacement and from snapshots of the system as in figure 11.15. This is even better illustrated in visualization software, where we can see that while the Stillinger-Weber system remains stationary over time, the neural network system picks up momentum and starts moving. We see that the system has moved slightly over time, while the crystal structure has remained mostly intact as evidenced by the radial distribution function. However, since the energy conservation is better for the neural network trained on the Stillinger-Weber potential, this motion is smaller than what we have observed for the EMT potential.

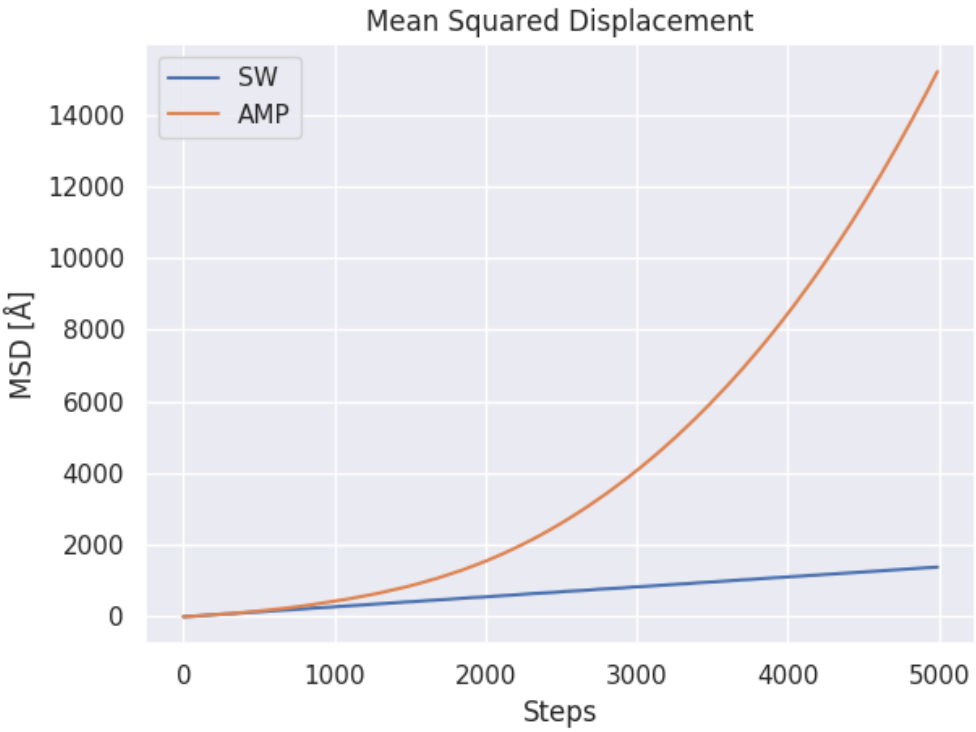
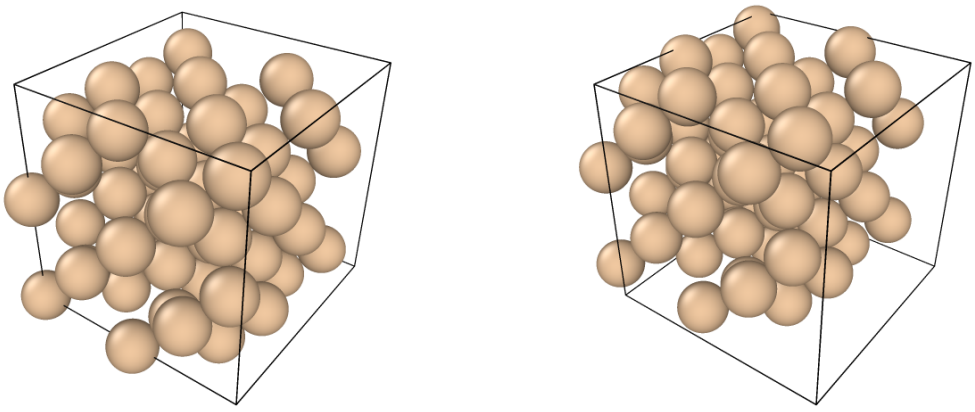


Figure 11.14: Mean squared displacement over time for the Stillinger-Weber and neural network potentials.



(a) Silicon atoms after 10 steps.

(b) Silicon atoms after 5000 steps.

Figure 11.15: The system of silicon atoms after 10 and 5000 steps.

Finally, we also examined the time scaling of the system as we did for the neural network trained on the EMT potential. As before we calculated the time it took for a single forces call on all the atoms in the system, as this is the limiting factor for integrating the atoms a single timestep using the Verlet algorithm. Ideally we would take averages of the time this takes, but except for the smaller systems this does not affect the results very much, due to the timescales involved. The results are shown in figure 11.16. From this plot we observe that integrating a system with 100 silicon atoms one step would take approximately 25-30 seconds, while a system of 300 atoms would take approximately 110 seconds. This means integrating a system of 100 atoms for 5000 steps would take about 35 hours, which is quite a large amount of time compared to typical empirical potentials. For the Stillinger-Weber potential this force call takes on the order of milliseconds, several orders of magnitude removed. However, since the neural network potential scales linearly, this may be competive with ab-initio methods which exhibit much poorer scaling as the system size increases.

Even though the neural network potentials have the same cutoff radius, the Stillinger-Weber potential takes roughly 2-4 times faster to evaluate the forces for an equivalent size. It is unclear why this is the case, as the trained potentials have a comparable number of radial symmetry functions, while the Stillinger-Weber potential has a larger amount of angular symmetry functions. This may be because the copper atoms have a larger average amount of neighbors, or because the potentials have been tested on different computers

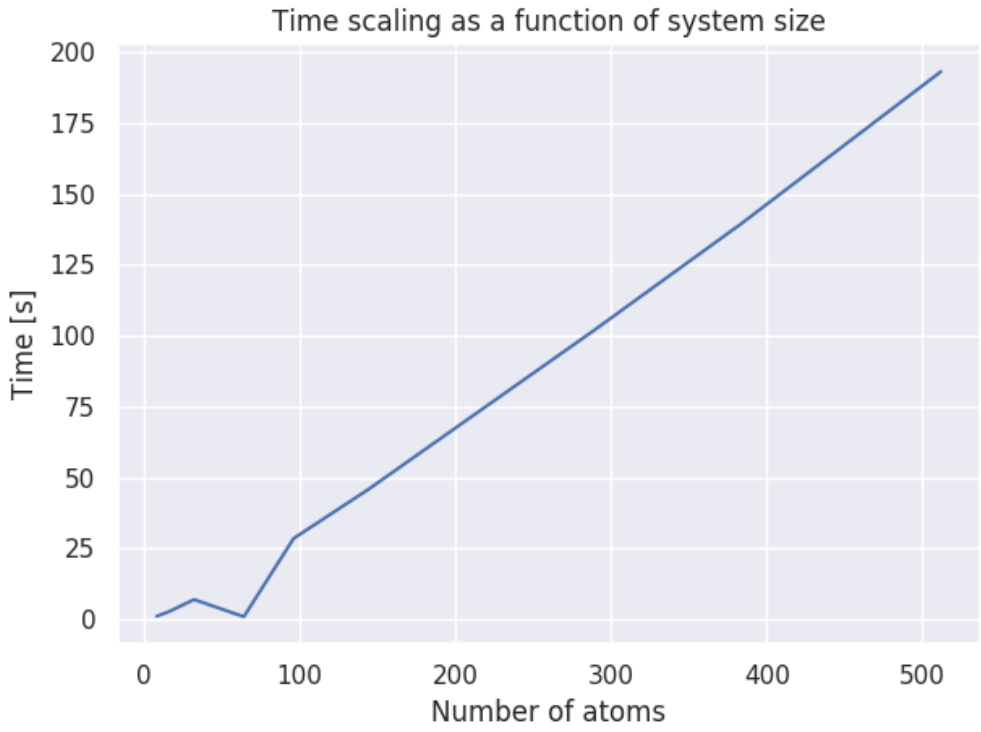


Figure 11.16: Time scaling of the neural network potential as a function of the size of the system.

(due to some difficulty in installing the Stillinger-Weber potential on one computer). Otherwise, this may be down to bugs in the code which stalls the network when calculating neighborlists, fingerprints or other quantities.

Chapter 12

Conclusions and future work

In this thesis we have trained neural networks to reproduce molecular dynamics potentials using the Behler-Parrinello method. These potentials are high-dimensional functions with many parameters to be determined, and are often developed through intimate knowledge of the physical and chemical properties of the systems they are designed for. In addition there are certain symmetries which must be respected when developing potentials, in particular translational, rotational and permutational symmetries, as well as conserving energy over time (in the NVE ensemble). Traditional ab-initio methods suffer from poor scaling as the size of the systems increases, while classical potentials derived from ab-initio calculations allows us to simulate realistic scales of up to millions of atoms depending on the computer resources and the complexity of the atoms in the system. Since these neural networks when trained offer linear scaling, they could serve both as supplements to ab-initio methods or be deployed for calculations using classical molecular dynamics. For example, one could save many CPU cycles by producing and training on a small trajectory calculated from DFT, and then use the neural networks to produce the remainder of the data, provided the neural networks are reasonably accurate. While our neural network potential scales linearly with system size, it has a rather large pre-factor dependent on the symmetry function set, cutoff radius and the average number of neighbors in the system. We believe this could be improved by moving more calculations to lower level compiled languages, such as calculations of neighbor lists, fingerprints and fingerprint derivatives for every step. Although we have only ran the neural networks on a single core, the potential could be parallelized over the atoms using algorithms such as LAMMPS neighbor lists without too much overhead.

Unfortunately we were not able to achieve energy conservation with our neural networks. This a central feature of any neural network potential, and without energy conservation, we cannot sample properties in equilibrium, as for example the radial distribution function or the diffusion constant is increasing over time and ill defined. Generally we find an increase in kinetic energy over time, corresponding to an increase in potential energy as the atoms move apart, and an increase in translational momentum. This indicates holes in the training data from sampling in equilibrium, as the network seems to perform poorly on unseen configurations with larger forces, and produce large force residuals. The results are notably different for the EMT and Stillinger-Weber potential, this is caused by among other things the symmetry function sets and the average number of neigbors in the system. The Stillinger-Weber explicitly includes threebody interactions, while the EMT potential is a function of interatomic distances, and our results suggest that the balance between the number of radial and symmetry functions should be decided by more careful analysis of their relative importance.

On the test trajectory we achieved reasonable values of the energy and force root mean squared errors, approximately 0.1 eV for the energy and 0.05-0.1 eV/Å for the force. These are mostly consistent with the results others have achieved such as [12][13][9][14], though the energy is often on the order of 1-10 meV, which is lower than what we have achieved. This is partly due to the fact that we have emphasized training with forces, which comes at the expense of the energy fit. Since the training RMSE is substantially lower than the test RMSE this may indicate overfitting, though we would not expect this to be a substantial problem with such a small network with applied regularization. We note that this has not been tested extensively, and more testing could produce different results.

12.0.1 Prospects and future work

In this thesis we barely scratched the surface of combining machine learning methods with molecular dynamics. There are therefore many, many paths to be explored for future theses or even articles to be published, which may be achieved working through ASE and/or AMP and making modifications or writing your own code from scratch. We will list some of the prospects considered while writing this thesis in semi-ordered order of importance, though there are likely many which have not been considered.

- Numerical optimization: In order to speed up the training and deployment of neural networks the AMP authors created efficient Fortran implementations of the Behler-Parrinello symmetry functions. This is an improvement over pure Python code, but not enough to be satisfied. Many parts of the fingerprinting, including neighborlists, dictionary data structures, IF tests and so on are currently implemented in Python code. This means we are limited in terms of CPU time and memory in how fast we can alter hyperparameters and train neural networks on new datasets, and how quickly the neural networks can be deployed and evaluated in molecular dynamics. First and foremost we would suggest moving the calculation of the neighborlists, fingerprints and fingerprint derivatives entirely to Fortran or some other efficient compiled language. This would reduce the speed of evaluations which are currently performed in Python, and reduce communication overhead between the Python APIs and the Fortran compiled functions. For more information see this post on the AMP mailing list. Once we have our input data and labels, training can be performed efficiently using software packages such as Tensorflow or Pytorch, and training moved to GPUs with little effort.

- Parallelization and LAMMPS: As we have discussed in the chapter on the Atomistic Machine-learning Package, AMP is planned to provide support for the OpenKIM API which would provide the means to export the neural networks to LAMMPS for deployment. If this is implemented (currently unknown) this could significantly speed up the deployment and testing of the neural networks, as LAMMPS is efficient compiled code, with efficient algorithms for parallelizing the force evaluations over multiple cores.

- Improved sampling method: We have observed that sampling from molecular dynamics trajectories limits us mostly to systems in equilibrium, which limits the range of energies and forces observed in the dataset. Neural networks perform unexpectedly when encountering unseen data, and the training data therefore restricts the generalization properties of the network. Improved sampling algorithms to sample a wider range of energies and forces out of equilibrium would likely improve the long timescale performance of the neural network and make the neural network potential more accurate on new configurations.

- Neural network implementation: Currently AMP provides a neural network implementation written by the authors, and a Tensorflow 0.11 module compatible only with Python 2.7. We would suggest writing a more modern Tensorflow interface, using for example the new Tensorflow 2.0 beta, in order to take full advantage of these mature neural network implementations. This could among other things improve speed, more efficient algorithms for initialization and training and a large set of helpful functions for training neural networks efficiently.

- Determining symmetry function sets: We have discussed some methods of determining symmetry functions, but not in great detail. Generally you want the symmetry functions to cover the radial and angular space, while no two symmetry functions should be highly correlated. However, we have observed that the same general mix of symmetry functions can produce very different results. It would be very beneficial if we had an automated approach or highly specified approach to determining the numbers of and parameters of the symmetry functions for a wide range of potential energy surfaces, which would significantly ease the difficulty in training and deploying neural networks using the Behler-Parrinello method.

- Finding minima: The AMP package finds minima in the loss function using an interface to the scipy.optimize library of functions. In this thesis we have only tested with the BFGS optimizer, which is the default in AMP, and the one generally favored by the authors and other users. However, finding minima in the cost function of neural networks is a complicated affair, which has been examined in great detail in the literature. First and foremost one could test other optimizer available through scipy, such as the basin hopping optimizer. If an interface to a more mature neural network software package such as Tensorflow were implemented it would be easy to test optimizers such as ADAM, SGD, Adagrad and many more.

- New descriptors: In this thesis we have restricted our interest to the tried and tested Behler-Parrinello symmetry functions as the mapping from coordinates to inputs. There have since been many suggestions on how to fingerprint atomic systems, such as DPMD, SOAP, Zernike and Bispectrum descriptors and so forth discussed in chapter 6, and if forces

can be calculated efficiently and accurately these should be evaluated for use in molecular dynamics.

- New machine-learning models: In this thesis we have only tested neural networks as the machine-learning method for regression, due to the explosion of interest and application in recent times. Neural networks are favorable due to their ability to scale well as the data available increases, but if data is limited other machine-learning algorithms may be competitive. Since we require the calculation of forces for usage in molecular dynamics we require the algorithm to have continuous derivatives, and a good example is Kernel Ridge Regression, which is currently supported in AMP.

- Multiple atom types: In this thesis we have limited our attention to single-atom systems, though AMP provides support for multiple atoms. The Behler-Parrinello method is limited in that a neural network has to be trained for every type-type interaction, for example an O-H interaction must be treated separately. This means that fewer potential energy labels are available for training per neural network, and creates a combinatorialproblem as the number of interactions in the system increases. Suggestions have been made to solve this problem such as weighted atom-centered symmetry functions (WACSFs[44]) and these could be implemented and evaluated against empirical potentials and neural networks trained using the standard approach.

- Long-range interactions: The Behler-Parrinello approach we have deployed is currently limited to short-range interactions within a cutoff sphere, and this is not sufficient for long-range interactions such as the coulomb interaction. Using methods such as Ewald summation[46], the Behler-Parrinello method could accommodate this, and this would facilitate training on systems containing for example water or biological molecules.

Appendices

Appendix A

Symmetry function parameters

Text.

Appendix B

Codebase

The software stack for this thesis involves primarily Python, using the Anaconda package on the Ubuntu operating system. We have employed a mix of the Atomic Simulation Environment, the Atomistic Machine-learning Package and parts of AMP implemented in Fortran. The code used to generate this thesis is available on Github:
user:aglgit, repository: master-thesis
The code used to generate the results are also available on Github:
user:aglgit, repository: python-md
user:aglgit, repository: amp

The AMP code is a fork of the AMP project with minor modifications, which is available on Bitbucket:
user:andrewpeterson, repository: amp

Appendix C

Bibliography

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[2] Md. Zahangir Alom et al. "The History Began from AlexNet: A Comprehensive Survey on Deep Learning Approaches". In: *CoRR* abs/1803.01164 (2018). arXiv: 1803.01164. URL: http://arxiv.org/abs/1803.01164.

[3] Nicholas M Ball and Robert J Brunner. "Data mining and machine learning in astronomy". In: *International Journal of Modern Physics D* 19.07 (2010), pp. 1049–1106.

[4] Juan Carrasquilla and Roger G Melko. "Machine learning phases of matter". In: *Nature Physics* 13.5 (2017), p. 431.

[5] Pierre Baldi, Peter Sadowski, and Daniel Whiteson. "Searching for exotic particles in high-energy physics with deep learning". In: *Nature communications* 5 (2014), p. 4308.

[6] Michael J Gillan, Dario Alfè, and Angelos Michaelides. "Perspective: How good is DFT for water?" In: *The Journal of chemical physics* 144.13 (2016), p. 130901.

[7] Jörg Behler. "Perspective: Machine learning potentials for atomistic simulations". In: *The Journal of chemical physics* 145.17 (2016), p. 170901.

[8] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural networks* 2.5 (1989), pp. 359–366.

[9] Alireza Khorshidi and Andrew A Peterson. "Amp: A modular approach to machine learning in atomistic simulations". In: *Computer Physics Communications* 207 (2016), pp. 310–324.

[10] David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, et al. "Learning representations by back-propagating errors". In: *Cognitive modeling* 5.3 (1988), p. 1.

[11] Jörg Behler and Michele Parrinello. "Generalized neural-network representation of high-dimensional potential-energy surfaces". In: *Physical review letters* 98.14 (2007), p. 146401.

[12] John-Anders Stende. "Constructing high-dimensional neural network potentials for molecular dynamics". MA thesis. 2017.

[13] Håkon Vikør Treider. "Speeding up ab-initio molecular dynamics with artificial neural networks". MA thesis. 2017.

[14] Linfeng Zhang et al. "Deep Potential Molecular Dynamics: A Scalable Model with the Accuracy of Quantum Mechanics". In: *Phys. Rev. Lett.* 120 (14 2018), p. 143001. DOI: 10.1103/PhysRevLett.120.143001. URL: https://link.aps.org/doi/10.1103/PhysRevLett.120.143001.

[15] Jun John Sakurai and Eugene D Commins. *Modern quantum mechanics, revised edition*. AAPT, 1995.

[16] Attila Szabo and Neil S Ostlund. *Modern Quantum Chemistry: Introduction to advanced electronic structure theory*. Dover Publications Inc, 1996.

[17] C. David Sherrill. *An Introduction to Hartree-Fock Molecular Orbital Theory*. http://vergil.chemistry.gatech.edu/notes/hf-intro/hf-intro.html. Online; accessed March 2019. 2000.

[18] Morten Hjorth-Jensen. *FYS4411/9411: Computational Physics 2 notes, Definitions of the many-body problem and Hartree-Fock theory*. https://compphysics.github.io/ComputationalPhysics2/doc/web/course. Online; accessed March 2019. 2019.

[19] Julien Toulouse. *Introduction to density-functional theory*. http://www.lct.jussieu.fr/pagesperso/toulouse/enseignement/introduction_dft.pdf. Online; accessed April 2019. 2017.

[20] Pierre Hohenberg and Walter Kohn. "Inhomogeneous electron gas". In: *Physical review* 136.3B (1964), B864.

[21] Mel Levy. "Universal variational functionals of electron densities, first-order density matrices, and natural spin-orbitals and solution of the v-representability problem". In: *Proceedings of the National Academy of Sciences* 76.12 (1979), pp. 6062–6065.

[22] John P Perdew et al. "Density-functional theory for fractional particle number: derivative discontinuities of the energy". In: *Physical Review Letters* 49.23 (1982), p. 1691.

[23] John P Perdew and Mel Levy. "Physical content of the exact Kohn-Sham orbital energies: band gaps and derivative discontinuities". In: *Physical Review Letters* 51.20 (1983), p. 1884.

[24] W. Kohn and L. J. Sham. "Self-Consistent Equations Including Exchange and Correlation Effects". In: *Phys. Rev.* 140 (4A 1965), A1133–A1138. DOI: 10.1103/PhysRev.140.A1133. URL: https://link.aps.org/doi/10.1103/PhysRev.140.A1133.

[25] Dominik Marx and Jörg Hutter. *Ab Initio Molecular Dynamics: Theory and Implementation*. Online; accessed April 2019. John von Neumann Institute for Computing, 2000.

[26] Daan Frenkel and Berend Smit. *Understanding molecular simulation: from algorithms to applications*. Vol. 1. Academic Press, 2001.

[27] M. Scott Shell. *Advanced molecular dynamics techniques*. https://sites.engineering.ucsb.edu/~shell/che210d/Advanced_molecular_dynamics.pdf. Online; accessed April 2019. 2012.

[28] Jostein Blyverket. "MOLECULAR DYNAMICS MODELING OF CLAY-FLUID INTERFACES". PhD thesis. June 2015.

[29] Cameron Abrams. *Molecular Simulations*. http://www.pages.drexel.edu/~cfa22/msim/node41.html. Online; accessed April 2019. 2013.

[30] KW Jacobsen, JK Norskov, and Martti J Puska. "Interatomic interactions in the effective-medium theory". In: *Physical Review B* 35.14 (1987), p. 7423.

[31] Karsten W Jacobsen, Per Stoltze, and JK Nørskov. "A semi-empirical effective medium theory for metals and alloys". In: *Surface Science* 366.2 (1996), pp. 394–402.

[32] Pankaj Mehta et al. "A high-bias, low-variance introduction to machine learning for physicists". In: (2019). Online; accessed March 2019.

[33] Poggio et al. *9.520: Statistical Learning Theory and Appplications*. https://www.mit.edu/~9.520/spring12/slides/class01/class01.pdf. Online; accessed September 2019. 2012.

[34] Sandra Vieira, Walter HL Pinaya, and Andrea Mechelli. "Using deep learning to investigate the neuroimaging correlates of psychiatric and neurological disorders: Methods and applications". In: *Neuroscience & Biobehavioral Reviews* 74 (2017), pp. 58–75.

[35] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010, pp. 249–256.

[36] Joseph-Frédéric Bonnans et al. *Numerical optimization: theoretical and practical aspects*. Springer Science & Business Media, 2006. Chap. 1–4.

[37] Albert P. Bartók et al. "Gaussian Approximation Potentials: The Accuracy of Quantum Mechanics, without the Electrons". In: *Phys. Rev. Lett.* 104 (13 2010), p. 136403. DOI: 10.1103/PhysRevLett.104.136403. URL: https://link.aps.org/doi/10.1103/PhysRevLett.104.136403.

[38] Stefan Chmiela et al. "Machine learning of accurate energy-conserving molecular force fields". In: *Science Advances* 3.5 (2017). DOI: 10.1126/sciadv.1603015. eprint: https://advances.sciencemag.org/content/3/5/e1603015.full.pdf. URL: https://advances.sciencemag.org/content/3/5/e1603015.

[39] Ask Hjorth Larsen et al. "The atomic simulation environment—a Python library for working with atoms". In: *Journal of Physics: Condensed Matter* 29.27 (2017), p. 273002.

[40] James Bergstra and Yoshua Bengio. "Random search for hyper-parameter optimization". In: *Journal of Machine Learning Research* 13.Feb (2012), pp. 281–305.

[41] Dimitris Bertsimas, John Tsitsiklis, et al. “Simulated annealing”. In: *Statistical science* 8.1 (1993), pp. 10–15.

[42] Yann A LeCun et al. “Efficient backprop”. In: *Neural networks: Tricks of the trade*. Springer, 2012, pp. 9–48.

[43] Jörg Behler. “Neural network potential-energy surfaces in chemistry: a tool for large-scale simulations”. In: *Physical Chemistry Chemical Physics* 13.40 (2011), pp. 17930–17955.

[44] Michael Gastegger et al. “wACSF—Weighted atom-centered symmetry functions as descriptors in machine learning potentials”. In: *The Journal of chemical physics* 148.24 (2018), p. 241709.

[45] A Pukrittayakamee et al. “Simultaneous fitting of a potential-energy surface and its corresponding force fields using feedforward neural networks”. In: *The Journal of chemical physics* 130.13 (2009), p. 134101.

[46] Abdulnour Y Toukmaji and John A Board Jr. “Ewald summation techniques in perspective: a survey”. In: *Computer physics communications* 95.2-3 (1996), pp. 73–92.