Constructing high-dimensional neural network potentials for molecular dynamics

by

John-Anders Stende

Thesis

for the degree of

Master of Science

Faculty of Mathematics and Natural Sciences
University of Oslo

May 2017

Abstract

This is an abstract text.

Acknowledgements

I acknowledge my acknowledgements.

Contents

1 Introduction **1**
1.1 Goals 1

I Theory **3**

2 Molecular dynamics **7**
2.1 Potential energy surfaces 7
2.1.1 From quantum mechanics to classical potentials 8
2.1.2 Constructing potential energy surfaces 8
2.2 Common empirical potentials 10
2.2.1 Lennard-Jones 10
2.2.2 Stillinger-Weber 13
2.2.3 Vashishta 13
2.3 Time integration 14
2.4 Force calculations and cutoff radius 15

3 Machine learning **17**
3.1 Artificial neurons 18
3.2 Neural network types 20
3.2.1 Feed-forward neural networks 21
3.2.2 Recurrent neural networks 22
3.2.3 Other types of networks 22
3.3 Multilayer perceptron 23
3.3.1 Why multilayer perceptrons? 23
3.3.2 Mathematical model 24
3.4 Activation functions 28
3.5 Training 30
3.5.1 Cost functions 31
3.6 Optimization 32
3.6.1 Gradient descent variants 32
3.6.2 Optimization algorithms 33
3.6.3 Backpropagation 36

4 Neural networks in molecular dynamics **43**
4.1 High-dimensional NNPs 44
4.1.1 Symmetry functions 46
4.1.2 Symmetry functions and forces 51

II Implementation and validation **55**

5 LAMMPS **57**
5.0.1 Installing LAMMPS 57
5.0.2 LAMMPS input script 58
5.0.3 LAMMPS structure 62
5.0.4 Extending LAMMPS 62

6 TensorFlow **65**
6.1 Installing TensorFlow 66
6.2 TensorFlow basic usage 66
6.2.1 Hello world 67
6.2.2 Creating a neural network 67
6.2.3 Visualizing the graph 69
6.2.4 Training a NN with TensorFlow 72

7 Training procedure **77**
7.1 Selecting the training data 77
7.1.1 Iterative molecular dynamics sampling 78
7.1.2 Sampling algorithms 79
7.2 Constructing the symmetry function sets 86

8 Validation **89**
8.1 Time usage 89
8.2 Training Lennard-Jones potential 91
8.2.1 Many-neighbour Lennard-Jones 93

III Results and discussion **99**

9 NN potential for Si **101**

10 NN potential for SiO2 **103**

Appendices **105**

A Symmetry functions derivatives **107**

Chapter 1

Introduction

Motivate the reader, outline structure of report and what we have done

1.1 Goals

The main goal of this thesis is to use artificial neural networks to construct many-body potential energy surfaces (PES) to be used in molecular dyanmics (MD) simulations. This goal can be split into the following intermediate objecives:

- Train an one-dimensional ANN with the machine learning library TensorFlow (TF) to reproduce the shifted Lennard-Jones (LJ) potential. As a first test case, we will train the NN on random data and investigate how different NN architectures and activation functions influence the quality of the potential fit.
- Train a many-body NN to reproduce the Stillinger-Weber (SW) potential for Si. Training data (configurations of particles and energies) will be sampled from SW LAMMPS simulations. The resulting NNP will then be used to simulate Si with LAMMPS. We therefor need to extend the LAMMPS library with our own neural network potential.
- Construct a many-body NNP that will be able to simulate a system consisting of several atom types. The method will be tested on the Vashishta potential for SiO2.

Part I

Theory

Theory needed to understand the results and implementations

Chapter 2

Molecular dynamics

Molecular dynamics (MD) is a method to simulate the physical movements of atoms and molecules in gases, liquids and solids. It is thus a type of N-body simulation. The atoms are modelled as point-like particles with interactions described by force fields. Their time-evolution is governed by Newton's equations of motion. MD allows us to study the microscopic movement of thousands or millions of atoms, enabling the sampling of macroscopic properties such at temperature, pressure, diffusion, heat capacity and so on.

The dynamics of an ensemble of particles is governed by their interactions. In MD, the interactions are described by a force field $\mathbf{F}$, which is defined as the negative gradient of a potential energy surface (PES) $V(\mathbf{r})$,

$$\mathbf{F} = -\nabla V(\mathbf{r}) \tag{2.1}$$

The PES describes the potential energy of a system for any configuration of atoms $\mathbf{r} = (\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_N)$, and thereby the forces, thus controlling the behaviour of the system. In other words, the PES is clearly the most important factor for a MD simulation to be physically plausible.

In this chapter, we will start out with a discussion on different ways to construct an adequate PES, followed by details about time-integration and functions that are commonly used to define a PES.

2.1 Potential energy surfaces

We know that atomic interactions are governed by the laws of quantum mechanics (QM). In principle, we can solve the time-dependent Schrödinger equation to obtain the trajectories of an ensemble of atoms. This is however a very computationally expensive task, and is not feasible except for small systems and time scales. Numerical methods for this purpose scales non-linearly with system size, while MD simulations scale linearly, if implemented in a certain way.

An alternative is to solve the time-*independent* Schrödinger equation for each time step in a simulation. The resulting PES is differentiated to obtain the forces on all atoms, and the new positions and velocities are calculated by employing Newton's laws. This method is called *ab inito* molecular dynamics. Although this is the most accurate approach to MD, it also suffers from bad scalability when the system size grows.

In *classical* MD, the PES is represented by a predefined, mathematical function, i.e. we do not have to update the PES during the simulation, resulting in better performance. The cost is a loss of accuracy, but for many systems it is fully possible to construct a predefined PES that yields the correct thermodynamics properties. In this thesis, we present a method to construct potential energy surfaces for classical MD.

2.1.1 From quantum mechanics to classical potentials

It it not obvious how an ensemble of atoms, which is a quantum mechanical system, can be simulated using laws from classical mechanics. To be able to do this, we must make the assumption that the motion of atomic nuclei and electrons in a molecule can be seperated, known as the Born-Oppenheimer approximation. Under this assumption, we can solve the electronic Schrödinger equation with the nucleonic degrees of freedom as parameters. The resulting wavefunction serves as a potential in a *nucleonic* Schrödinger equation, yielding a PES for the nuclear motion. In this way, the positions and velocities of the electrons are baked into the PES, allowing us to model the atomic nuclei as point particles that follow Newtonian dynamics.

2.1.2 Constructing potential energy surfaces

As mentioned above, we represent the PES by a predefined functional form in classical MD. We assume that this function can be constructed as a sum of n-body terms V_n,

$$V(\mathbf{r}) \approx \sum_{i}^{N} V_1(\mathbf{r}_i) + \sum_{i,j}^{N} V_2(\mathbf{r}_i, \mathbf{r}_j) + \sum_{i,j,k}^{N} V_3(\mathbf{r}_i, \mathbf{r}_j, \mathbf{r}_k) + \dots \tag{2.2}$$

where the sums are over all single atoms i, all pairs (i, j), all triplets (i, j, k) and so on. Each term is determined by empirical knowledge of the system, by quantum mechanical calculations or a combination of both. The resulting PES is called an empirical, quantum mechanical or semi-empirical potential respectively.

When constructing empirical potentials, we typically start out with a functional form that captures some of the known, basic properties of the system under investigation. The function has a set of parameters that are fitted to experimental data, hence the name. Empirical potentials are often quite simple and only

have a few parameters, making their computatioal performance superior. However, their accuracy is limited by the choice of functional form; there is always a risk that not all the physics of a system is captured. Further, the parameters need to be fitted specifically for a given system and are typically not transferable from one system to another. Also, the fitting process can be difficult and time-consuming.

For QM potentials, electronic structure calculations are performed for a number of atomic configurations, yielding a data set of configurations and corresponding potential energies. A very flexible functional form, often containing many parameters, are then fitted to this data set. Potentials of this type can in principle be fitted to arbitrary accuracy, and may thus be treated as *ab inito* potentials. QM potentials are purely mathematical, in the sense that no physics can be extracted from their functional form. Therefore, the construction of the potential has to be done with great care, i.e. the data set must contain all relevant configurations for the intended application of the PES. However, the unbiased nature of QM potentials make them more general than their empirical counterparts, and the methodology can be applied to any system.

Semi-empirical potentials are constructed as a combination of the two schemes above. Their general functional form are based on methods from computational QM, but many approximations are made and some parameters are obtained from empirical data.

Truncation and configuration space

Determining how many terms in (2.2) that is sufficient to describe a system adequately is not trivial. Noble gases like argon have weak interactions and may be well described by a two-body potential V_2, while molecules with strong angular dependencies on their bond, like silicon (Si), will need more terms. We note that the one-body term V_1 is only included if the system is experiencing an external force.

Another challenge is to identify the configuration space of the system. For three-body potentials and more the set of probable configurations is large, and running electronic structure calculations for each configuration is expensive. The problem of identifying the configuration space is described in section 7.1.

Fitting procedure

In this thesis we present a method to construct many-body QM potentials for MD simulations. An important step in this process is the fitting of the data set of atomic configurations and energies by a suitable functional form. Several methods have been proposed, like different types of splines, interpolating moving least squares [28] and modified Sheppard interpolation [19]. We will use *artificial neural networks* (ANNs) for this purpose. The basic theory of ANNs is presented

in chapter 2, while chapter 3 deales with physical symmetries that has to be considered. Finally, in the implementation part, the details of constructing and applying an AAN potential in MD simulations are described.

2.2 Common empirical potentials

Although the method described in this thesis is for the construction of QM potentials, empirical potentials are useful as test cases and validations of the method. We can define two types of empirical potentials based on how they relate to groups of atoms. *Bonded* potentials compute the energy of each (covalent) bond for a predefined set of atoms or molecules, while *non-bonded* potentials depend on the spatial configuration of *each possible set* of particles of a specific kind, e.g. all pairs of Ar atoms or all triplets of Si atoms. The total potential is the sum of the contributions from these potentials,

$$V_{\text{tot}} = V_{\text{bonded}} + V_{\text{non-bonded}} \tag{2.3}$$

We will only work with non-bonded potentials in this thesis. In the following we will discuss three such potentials that are common in the literature, and which we have employed to validate our PES construction method.

2.2.1 Lennard-Jones

The Lennard-Jones (LJ) potential [29] is one of the simplest and most widely used potentials. It has a simple functional form which is inexpensive to compute, and is able to reproduce weakly interacting systems like noble gases well. It is a two-body potential that only depends on the distance r_{ij} between pairs of atoms i and j. The general form is (Figure 2.1)

$$V_{LJ}(r_{ij}) = 4\epsilon \left[\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right] \tag{2.4}$$

where the two parameters ϵ and σ are the depth of the potential well and the zero-point of the potential respectively. The Lennard-Jones potential captures a basic, phyiscal property of many atomic systems: atoms repulse each other at short distances and are attracted to each other at long distances. The first term describes the Pauli repulsion due to overlapping electron orbitals, while the second term is the attraction due to the so-called van der Waals force. The functional form of the repulsive term has, in contrast to the attractive one, no theoretical justification other than that it approximates the Pauli repulsion well.

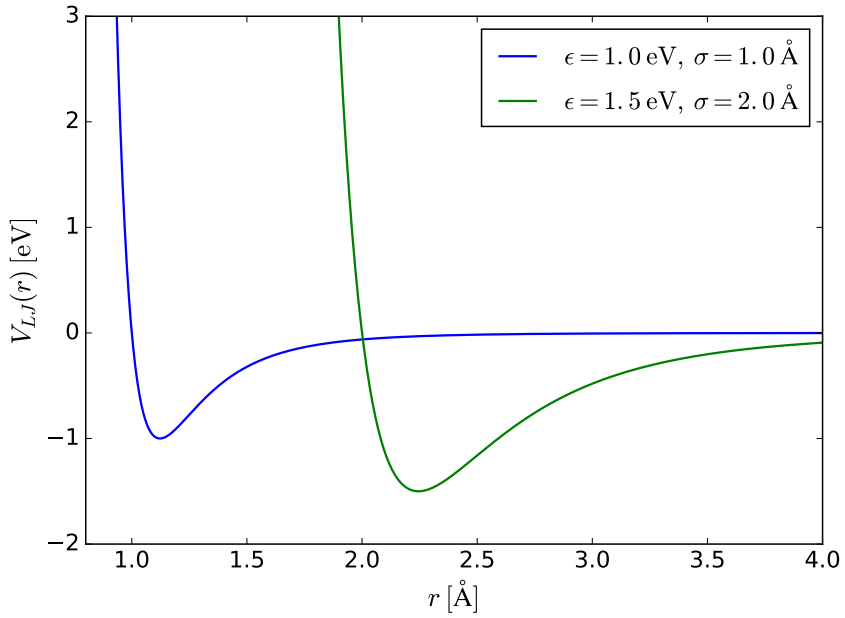


Figure 2.1: The Lennard-Jones potential (2.4) as a function of inter-atomic distance for two different parameter sets. The depth of the potential well (potential strength) is given by ϵ, while σ is the inter-atomic distance for which the potential is zero.

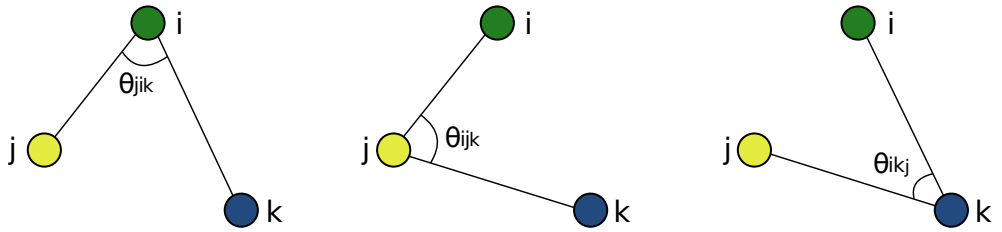


Figure 2.2

Calculating total potential energy

Before we move on to three-body potentials, we will briefly explain how the total potential energy of a system is calculated. For a two-body potential like LJ, each unique pair of atoms is only counted once, so that

$$V_{\text{tot}} = \sum_{i<j} V_2(r_{ij}) = \frac{1}{2}\sum_{i\neq j} V_2(r_{ij}) \tag{2.5}$$

In the second expression we count all pairs twice, thus a factor 1/2 is required. For the three-body case, the counting is a bit more complicated. An atomic triplet is defined by the three variables $(r_{ij}, r_{ik}, \theta_{jik})$, where r_{ij} and r_{ik} are the two inter-atomic distances to the central atom i, and θ_{jik} is the angle between the lines from i to j and from i to k. Note that there are only two *independent* variables, since

$$\theta_{jik} = \frac{\mathbf{r}_{ij}\cdot\mathbf{r}_{ik}}{r_{ij}^2 r_{ik}^2} \tag{2.6}$$

Figure 2.2 illustrates that three atoms (i, j, k) define three different triplets depending on which atom that is the central one. All three triplets have different energies, and must therefore be tallied individually. If we assume that the potential form is such that it is symmetric with respect to exchange of the j and k indicies (which is not always the case for multielemental systems), we have

$$\sum_{i<j<k}\left[V_3^{(1)}(r_{ij}, r_{ik}, \theta_{jik}) + V_3^{(2)}(r_{ij}, r_{jk}, \theta_{ijk}) + V_3^{(3)}(r_{ik}, r_{jk}, \theta_{ikj})\right] \tag{2.7}$$

where $V_3^{(i)}$ is one of the triplets in Figure 2.2. Alternatively, we can calculate the energy of one triplet at a time and instead sum over all j,

$$\sum_i \sum_{j\neq i} \sum_{k>j} V_3(r_{ij}, r_{ik}, \theta_{jik}) \tag{2.8}$$

This is the strategy we are going to adopt in the following. The total energy is now written,

$$V_{\text{tot}} = \sum_i \sum_{j>i} V_2(r_{ij}) + \sum_i \sum_{j\neq i} \sum_{k>j} V_3(r_{ij}, r_{ik}, \theta_{jik}) \tag{2.9}$$

2.2.2 Stillinger-Weber

The Stillinger-Weber (SW) potential was proposed in 1985 [30] for simulating Si atoms. It has a two-body and a three-body term, and is thus written on the form (2.9). The two-body part consists of a Lennard-Jones factor with variable exponents, together with an exponential decay factor that ensures zeroing out of the potential at a *cutoff* $a\sigma$,

$$V_2(r_{ij}) = A\epsilon\left[B\left(\frac{\sigma}{r_{ij}}\right)^p - \left(\frac{\sigma}{r_{ij}}\right)^q\right]\exp\left(\frac{\sigma}{r_{ij}-a\sigma}\right) \tag{2.10}$$

Silicon forms 4-coordinated tetrahedral bonded structures, and the purpose of the three-body component is to enforce the tetrahedral bond angle $\theta_0 \approx 109°$,

$$V_3(r_{ij}, r_{ik}, \theta_{jik}) = \lambda\epsilon[\cos\theta_{jik} - \cos\theta_0]^2 \exp\left(\frac{\gamma\sigma}{r_{ij}-a\sigma}\right)\exp\left(\frac{\gamma\sigma}{r_{ik}-a\sigma}\right) \tag{2.11}$$

We observe that for $\theta_{jik} = \theta_0$, the three-body term is zeroed out, making this configuration energetically favourable. The expression also contains two exponential decay factors to ensure that the potential goes to zero once r_{ij} and/or r_{ik} approach the cutoff.

SW simulations of solid and liquid silicon agree reasonably well with experimental data [30] The potential has also been used on other systems than silicon, for instance in monatomic water models [31].

2.2.3 Vashishta

In 1990, Vashishta et al. [32] suggested a three-body interaction potential for SiO_2. The three-body term is of similar form to that of Stillinger-Weber,

$$\begin{aligned} V_3(r_{ij}, r_{ik}, \theta_{jik}) \; &= B_{jik}\frac{[\cos\theta_{jik} - \cos\theta_{0jik}]^2}{1 + C[\cos\theta_{jik} - \cos\theta_{0jik}]^2} \cdot \\ &\exp\left(\frac{\gamma_{ij}}{r_{ij} - r_{0ij}}\right)\exp\left(\frac{\gamma_{ik}}{r_{ik} - r_{0ik}}\right) \end{aligned} \tag{2.12}$$

We note that indicies are included for the parameters, as they are generally different for each pair and triplet combination of elements; the parameter set must be fitted to e.g. Si-Si-Si and Si-O-O seperately. The angle θ_{0jik} has the same purpose as for SW, while r_{0ij} is the cutoff distance for each pair of elements. The two-body part is more complex,

$$V_2(r_{ij}) = \frac{H_{ij}}{r^{\eta_{ij}}} - \frac{W_{ij}}{r^6} + \frac{Z_i Z_j}{r}\exp(-r/\lambda_{1,ij}) - \frac{D_{ij}}{r^4}\exp(-r/\lambda_{4,ij}) \tag{2.13}$$

where we recognize the two first terms as typical LJ repulsive and attractive components. The other two describe electrostatic interactions, the third being a

so-called screened Coulomb potential for charges Z_i and Z_j. We will not expand upon the physical justification of these terms here.

The Vashishta potential has also been succesfully employed on other multi-component systems like SiC [33].

2.3 Time integration

To evolve the system forward in time, we need a numerical method to integrate Newton's equations of motion,

$$\mathbf{v}(t) = \mathbf{v}(0) + \int_0^t \mathbf{a}(t)dt \tag{2.14}$$

$$\mathbf{r}(t) = \mathbf{r}(0) + \int_0^t \mathbf{v}(t)dt \tag{2.15}$$

Various finite different schemes can be used for this purpose, and the choice of algorithm is of vital importance to the quality of the simulation. The most important property of an integrator for use in MD, is long-term energy conservation. MD simulations are often run for millions of time steps, and a significant energy drift may cause useless results. Further, all integration schemes approximate derivatives as finite differences, introducing truncation errors. Finally, numerical stability is important to prevent unphysical behaviour on large time scales.

A good choice is the Velocity-Verlet algorithm, which is a *symplectic* integrator. Put simply, this means that it preserves the form of Hamilton's equations, thereby conserving energy. [1] Employing Velocity-Verlet allows us to sample the phase space of the microcanonical (NVE) ensemble, and collect thermodynamic properties like temperature and pressure. The steps of the algorithm are,

$$\mathbf{v}(t + \Delta t/2) = \mathbf{v}(t) + \frac{\mathbf{F}(t)}{m}\frac{\Delta t}{2} \tag{2.16}$$

$$\mathbf{r}(t + \Delta t) = \mathbf{r}(t) + \mathbf{v}(t + \Delta t/2)\Delta t \tag{2.17}$$

$$\mathbf{v}(t + \Delta t) = \mathbf{v}(t + \Delta t) + \frac{\mathbf{F}(t + \Delta t)}{m}\frac{\Delta t}{2} \tag{2.18}$$

First, the velocity at half the time step forward $\mathbf{v}(t + \Delta t/2)$ is computed. This half-velocity is used to obtain the new positions $\mathbf{r}(t + \Delta t)$. Then we calculate the new accelerations $\mathbf{F}(t + \Delta t)/m$ based on the new positions, which is used to compute the velocities at a whole time step forward $\mathbf{v}(t + \Delta t)$.

In addition to being symplectic, Velocity-Verlet is time-reversible [34], which it should be considering that we are solving time-reversible equations (2.14) (2.15).

[1] Strictly speaking, Velocity-Verlet only conserves a pseudo-Hamiltonian approaching the true Hamiltonian in the limit of infinitesimal time steps. [34]

It is also computationally inexpensive; the only additional computation compared to the primitive Euler method is the half-velocity. The global error in position and velocity is $O(\Delta t^2)$, which may not seem very impressive. However, this is good enough for collecting statistical properties, our goal is not to reproduce the exact trajectories of the particles.

Velocity-Verlet is the integrator of choice in LAMMPS (chapter 5), and thus the only scheme used in this work.

2.4 Force calculations and cutoff radius

We need the force on each atom to compute the acceleration that is integrated in the equations of motion (2.14)(2.15). Using the Lennard-Jones potential as an example, the force on atom i from atom j is (2.1) (2.4),

$$F_{\mathrm{LJ}}(r_{ij}) = -\nabla V_{\mathrm{LJ}}(r_{ij}) = -24\epsilon \left[2\left(\frac{\sigma^{12}}{r_{ij}^{13}}\right) - \left(\frac{\sigma^{6}}{r_{ij}^{7}}\right)\right] \tag{2.19}$$

The force is of the same shape as the potential in Figure 2.1, which means it rapidly goes to zero. If we choose units so that $\sigma = \epsilon = 1.0$, the force at a distance $r_{ij} = 3\sigma$ is

$$F_{\mathrm{LJ}}(r_{ij} = 3.0\sigma) \approx 0.01 \tag{2.20}$$

This is a general property of systems where only van der Waal-interactions are present: atoms that are far apart do not effectively interact. This can be exploited in MD simulations by introducing a cutoff radius r_{cut}, and only compute forces for atoms displaced by a distance smaller than this cutoff. In principle, we have to sum over all pairs of atoms in the system to compute the energy and forces, which scales like $O(N^2)$. With a cutoff, this is reduced to $O(N)$, and the force is now written,

$$F_{\mathrm{LJ}}(r_{ij}) = \begin{cases} -24\epsilon \left[2\left(\frac{\sigma^{12}}{r_{ij}^{13}}\right) - \left(\frac{\sigma^{6}}{r_{ij}^{7}}\right)\right], & r_{ij} \leq r_{\mathrm{cut}} \\ 0, & r_{ij} > r_{\mathrm{cut}} \end{cases} \tag{2.21}$$

Chapter 3

Machine learning

Machine learning is the science of giving computers the ability to learn without being explicitly programmed. The idea is that there exist generic algorithms which can be used to find patterns in a broad class of data sets without having to write code specifically for each problem. The algorithm will build its own logic based on the data.

Machine learning is a subfield of computer science, and is closely related to computational statistics. It evolved from the study of pattern recognition in artificial intelligence research (AI), and has made contributions to AI tasks like computer vision, natural language processing and speech recognition. It has also, especially in later years, found applications in a wide variety of other areas, including bioinformatics, economy, physics, finance and marketing. (SHOULD I HAVE REFERENCES HERE??)

The approaches to machine learning are many, but are often split into two main categories. In *supervised learning* we know the answer to a problem, and let the computer deduce the logic behind it. On the other hand, *unsupervised learning* is a method for finding patterns and relationship in data sets withouth any prior knowledge of the system. Some authours also operate with a third category, namely *reinforcement learning*. This is a paradigm of learning inspired by behavioural psychology, where learning is achieved by trial-and-error, solely from rewards and punishment.

Another way to categorize machine learning tasks is to consider the desired output of a system. Some of the most common tasks are:

- Classification: Outputs are divided into two or more classes. The goal is to produce a model that assigns inputs into one of these classes. An example is to identify digits based on pictures of hand-written ones. Classification is typically supervised learning.

- Regression: Finding a functional relationship between an input data set and a reference data set. The goal is to construct a function that maps input data to continous output values.

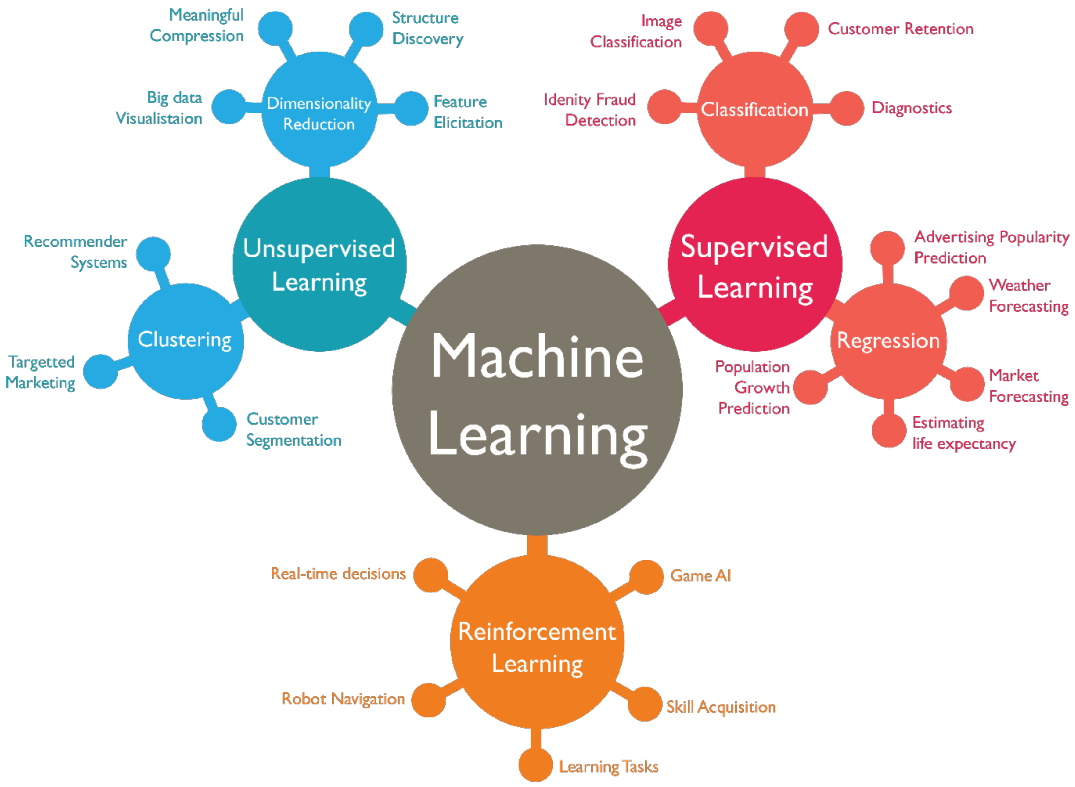


Figure 3.1: Machine learning, taken from from http://www.isaziconsulting.co.za/machinelearning.html.

- Clustering: Data are divided into groups with certain common traits, without knowing the different groups beforehand. It is thus a form of unsupervised learning.

Figure 3.1 gives an (incomplete) overview of machine learning methods with some selected applications. We will not go into further detail on all these approaches to machine learning, as this work only concerns one specific area of ML: artificial neural networks. The rest of this chapter concerns the ideas behind this machine learning approach.

3.1 Artificial neurons

The field of artificial neural networks has a long history of development, and is closely connected with the advancement of computer science and computers in general. A model of artificial neurons was first developed by McCulloch and Pitts in 1943 [23] to study signal processing in the brain and has later been refined by others. The general idea is to the mimic neural networks in the human brain,

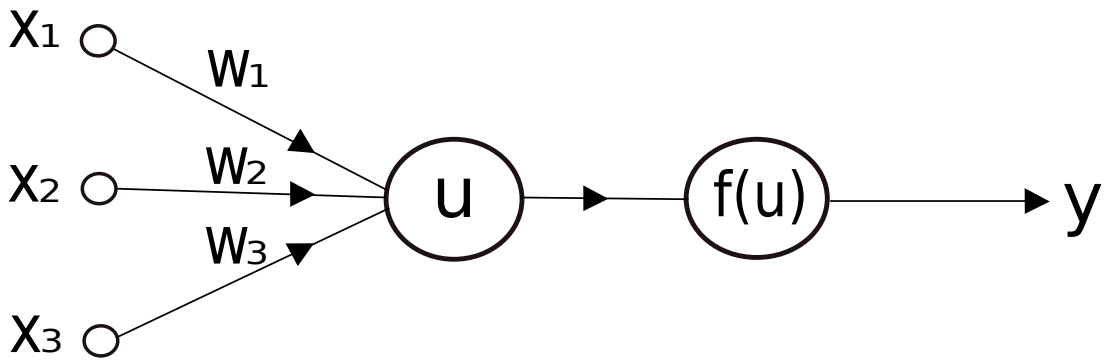


Figure 3.2: Mathematical model of an artificial neuron. The neuron receives input signals $x_i, \ldots, x_n$ from n other neurons. Each signal x_i is associated with a weight w_i, and the neuron accumulates all input signals as a weighted sum u. This sum is then used as input to its activation function f, which serves as the neuron's output signal.

which is composed of billions of neurons that communicate with each other by sending electrical signals. Each neuron accumulates its incoming signals, which must exceed an activation threshold to yield an output of one. If the threshold is not overcome, the neuron remains inactive, i.e. has zero output.

This behaviour has inspired a simple mathematical model for an artificial neuron [5],

$$y = f\left(\sum_{i=1}^{n} w_i x_i\right) = f(u) \tag{3.1}$$

Here, the output y of the neuron is the value of its activation function, which have as input a weighted sum of signals $x_i, \ldots, x_n$ received by n other neurons. The model is illustrated in Figure 3.2. We will now take a closer look at if and how this mathematical model is biologically plausible.

Biological model

Artificial neurons are designed to mimic certain aspects of their biological counterparts [24]. A schematic model of a biological nervous cell is depicted in Figure 3.3. The *dendrites* in a neuron acts as the input vector, and allow the cell to receive signals from a large number of neighbouring neurons. As in the mathematical model, each dendrite is associated with the multiplication of a "weight value", which is achieved by an increase or decrease of chemical neurotransmitters that amplifies or weakens the input signals. The cell can also transmit signal inhibitors (oppositely charged ions) to accomplish "negative" weight values.

The positive and negative ions arriving from the dendrites are mixed together in the solution inside the *soma*. which corresponds to the summation in (3.1).

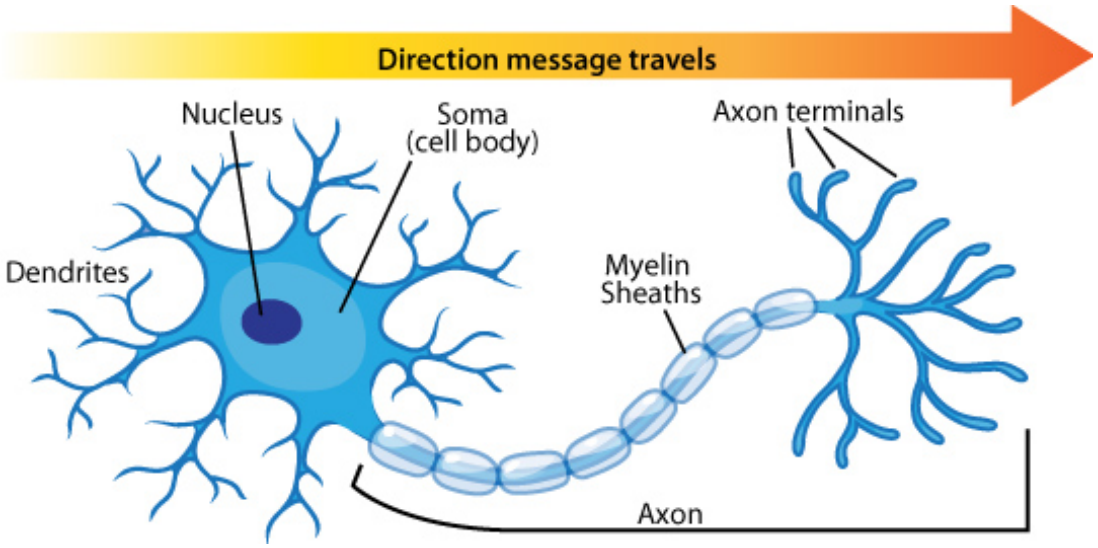


Figure 3.3: Biological model of a neuron. Source: https://askabiologist.asu.edu/neuron-anatomy

Finally, the *axon* serves as the activation function by sampling the electrical potential of this solution. If the potential reaches a certain strength, the axon transmits a signal pulse down its own length. The axon is connected to another neuron's dendrites (or other parts of the cell), enabling the neurons to communicate.

This discussion of nervous cells helps, to a certain degree, to validate the mathematical form (3.1) of artifical neurons. However, we note that the above description is a highly simplified picture of the great complexity of biological neurons, which is not yet fully understood. Our mathematical neurons are thus only caricatures of nature. Next, we will look at how networks of these mathematical objects can be formed to emulate the neural networks in the brain.

3.2 Neural network types

An artificial neural network (ANN), is a computational model that consists of layers of connected neurons, or *nodes*. It is supposed to mimic a biological nervous system by letting each neuron interact with other neurons by sending signals in the form of mathematical functions between layers. Each node is modelled according to (3.1). A wide variety of different ANNs have been developed, but most of them consist of an input layer, an output layer and eventual layers in-between, called *hidden layers*. All layers can contain an arbitrary number of nodes, and each connection between two nodes is associated with a weight variable.

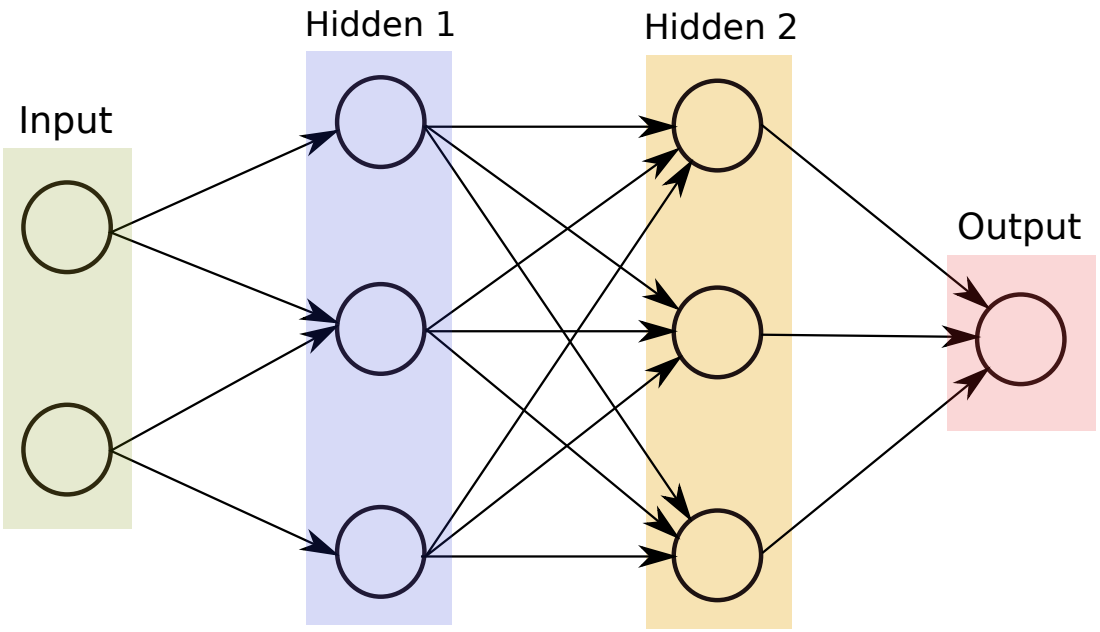


Figure 3.4: Example of a fully-connected feed-forward neural network with an input layer, two hidden layers and an output layer. Each node in a layer is connected to *all* nodes in the subsequent layer, and information only flows forward through the layers, hence the name.

The main factor that seperates the different types are how the neurons are *connected*. This section contains a short presentation of some of the most common types of ANNs, before we move on to a more detailed description of the ANN architecture used in this thesis. In the following we will refer to artificial neural networks simply as neural networks (NNs), i.e. not referring to its biological counterpart.

3.2.1 Feed-forward neural networks

The feed-forward neural network (FFNN) was the first and simplest type of NN devised. In this network, the information moves in only one direction - forward through the layers. An example FFNN is shown in Figure 3.4, which consists of an input layer, two hidden layers and an output layer. Nodes are represented by circles, while the arrows display the connections between the nodes, including the direction of information flow. Additionaly, each arrow corresponds to a weight variable, not displayed here. We observe that each node in a layer is connected to *all* nodes in the subsequent layer, making this a so-called *fully-connected* FFNN. This is the type of NN that is used in the present work, and will be further investigated in chapter (REF TO CHAPTER).

A different variant of FFNNs are *convolutional neural networks* (CNNs) [25],

which have a connectivity pattern inspired by the animal visual cortex. Individual neurons in the visual cortex only respond to stimuli from small sub-regions of the visual field, called a receptive field. This makes the neurons well-suited to exploit the strong spatially local correlation present in natural images. The response of each neuron can be approximated mathematically as a convolution operation.

CNNs emulate the behaviour of neurons in the visual cortex by enforcing a *local* connectivity pattern between nodes of adjacent layers: Each node in a convolutional layer is connected to a subset of the nodes in the previous layer, in contrast to the fully-connected FFNN in Figure 3.4. Often, CNNs consist of several convolutional layers that learn local features of the input, with a fully-connected layer at the end, which gathers all the local data and produces the outputs. They have wide applications in image and video recognition.

3.2.2 Recurrent neural networks

So far we have only mentioned NNs where information flows in one direction: forward. *Recurrent neural networks* on the other hand, have connections between nodes that form directed *cycles*. This creates a form of internal memory which are able to capture information on what has been calculated before; the output is dependent on the previous computations. Recurrent NNs make use of sequential information by performing the same task for every element in a sequence, where each element depends on previous elements. An example of such information is sentences, making recurrent NNs especially well-suited for handwriting and speech recognition.

3.2.3 Other types of networks

There are many other kinds of NNs that have been developed. One type that is specifically designed for interpolation in multidimensional space is the radial basis function (RBF) network. RBFs are typically made up of three layers: an input layer, a hidden layer with non-linear radial symmetric activation functions and a linear output layer ("linear" here means that each node in the output layer has a linear activation function). The layers are normally fully-connected and there are no cycles, thus RBFs can be viewed as a type of fully-connected FFNN. They are however usually treated as a seperate type of NN due the unusual activation functions (common activation functions are presented in chapter (REF TO CHAPTER)) and a different training algorithm, that we will not expand upon here.

Other types of NNs could also be mentioned, but are outside the scope of this work. Now we will move on to a detailed description of how a fully-connected FFNN functions, and how it can be trained to interpolate data.

3.3 Multilayer perceptron

In this thesis we use fully-connected feed-forward neural networks with three or more layers (an input layer, one or more hidden layers and an output layer), mainly consisting of neurons that have non-linear activation functions. Such networks are often called *multilayer perceptrons* (MLPs). [1] The *perceptron* was first introduced by Rosenblatt in 1958 [26], and was a FFNN made up of an input layer and an ouput layer only. It served as a *binary classifier*, i.e. a function that maps a real-valued vector $\mathbf{x}$ to a single binary value $f(\mathbf{x})$, employing a linear activation function. The addition of hidden layers and non-linear activation functions led to the MLP, which could produce an arbitrary number of continous output values.

3.3.1 Why multilayer perceptrons?

We have chosen to use MLPs to interpolate data sets for the construction of inter-atomic potentials. Other NN types could also have been used, but our choice is well justified. According to the *Universal approximation theorem* [4], a feed-forward neural network with just a single hidden layer containing a finite number of neurons can approximate a continous multidimensional function to arbitrary accuracy, assuming the activation function for the hidden layer is a "non-constant, bounded and monotonically-increasing continous function". Note that the requirements on the activation function only applies to the hidden layer, the output nodes are always assumed to be linear, so as to not restrict the range of output values. The only requirement on the NN information flow is that it must be feed-forward, thus recurrent NNs are out of the picture. Also, we might as well use the simplest multilayer FFNN available, the MLP, as adding complicating factors like local connectivity is unnecessary for our purposes.

We note that this theorem is only applicable to a NN with *one* hidden layer. Therefore, we can easily construct a NN which employs activation functions that do not satisfy the above requirements, as long as we have at least one layer with activation functions that *do*. Furthermore, although the universal approximation theorem lays the theoretical foundation for regression with neural networks, it does not say anything about how things work in practice: a NN can still be able to approximate a given function reasonably well without having the flexibility to fit *all other* functions.

[1] The terms "feed-forward neural network" and "multilayer perceptron" are used interchangeably in the literature, although the MLP is just one type of FFNN, namely a fully-connected one with mainly non-linear neurons.

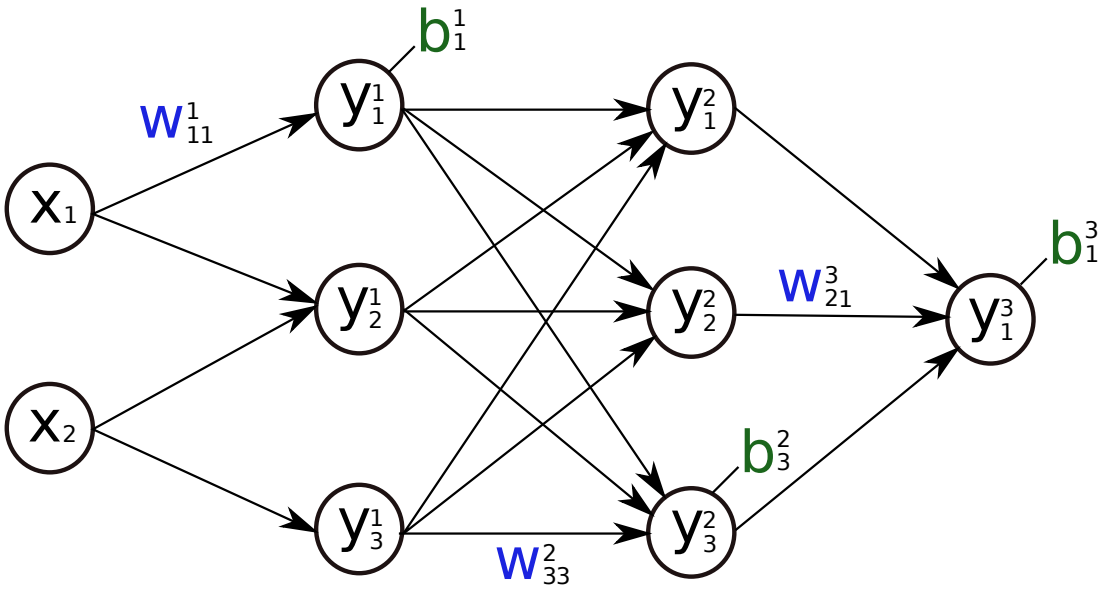


Figure 3.5: Example of a fully-connected feed-forward neural network with an input layer, two hidden layers and an output layer. Nodes are depicted as circles, while the arrows shows the connections between neurons, including their directions. Each connection has a weight w (blue), where the notation explained in the text is applied. Every node is marked with its output y and its associated bias b (green), while all input nodes are labeled with an x. Only a few weights and biases are listed.

3.3.2 Mathematical model

In an MLP, each node (neuron) is modelled according to (3.1). To increase the flexibility of the NN, we also add a *bias* to this model, resulting in

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b_i\right) = f(u) \tag{3.2}$$

In a FFNN of such neurons, the *inputs* x_i are the *outputs* of the neurons in the preceding layer. Furthermore, a MLP is fully-connected, which means that each neuron receives a weighted sum of the outputs of *all* neurons in the previous layer.

Figure 3.5 displays the same NN as in Figure 3.4, now with all nodes labeled by their output y, except for the input nodes, which are labeled with x. The input nodes are strictly speaking not artifical neurons, as they have no activation function. Their only purpose is to forward the NN inputs to the first hidden layer. A few weights w (blue) and biases b (green) are also included, next to the connections and nodes which they belong to respectively. The following notation is introduced: y_i^l is the output of the i-th node in layer l, where $l = 0$ is the input

layer, $l = 1$ is the first hidden layer and so on. The same notation applies to the biases. For the weights, we have that w_{ij}^l is the weight connecting node j in layer $l-1$ with node i in layer l. [2] All weights and biases are real-valued numbers.

The MLP in Figure 3.5 maps the inputs x_i to the output y_1^3 via two hidden layers with an (arbitrary) number of hidden nodes. The hidden neurons have no physical meaning, but have the purpose of defining the functional form of the NN. In the following we describe how to calculate the output of this MLP. The description is also valid for a NN with several outputs; the result applies to each ouput node individually.

First, for each node i in the first hidden layer, we calculate a weighted sum u_i^1 of the input coordinates x_m,

$$u_i^1 = \sum_{j=1}^{2} w_{ij}^1 x_j + b_i^1 \tag{3.3}$$

This value is the argument to the activation function f^1 of each neuron, producing the output y_i^1 of all neurons in layer 1,

$$y_i^1 = f^1(u_i^1) = f^1\left(\sum_{j=1}^{2} w_{ij}^1 x_j + b_i^1\right) \tag{3.4}$$

where we assume that all nodes in the same layer have identical activation functions, hence the notation f^l. For an arbitrary node i in layer l this generalizes to

$$y_i^l = f^l(u_i^l) = f^l\left(\sum_{j=1}^{N_{l-1}} w_{ij}^l y_j^{l-1} + b_i^l\right) \tag{3.5}$$

where N_l is the number of nodes in layer l. When the output of all the nodes in the first hidden layer are computed, the values of the subsequent layer can be calculated and so forth until the output is obtained. The output of neuron i in layer 2 is thus,

$$y_i^2 = f^2\left(\sum_{j=1}^{3} w_{ij}^2 y_j^1 + b_i^2\right) \tag{3.6}$$

$$= f^2\left[\sum_{j=1}^{3} w_{ij}^2 f^1\left(\sum_{k=1}^{2} w_{jk}^1 x_k + b_j^1\right) + b_i^2\right] \tag{3.7}$$

[2] This order of indicies i and j may seem illogical, but will become useful when converting the equations below to matrix equations.

where we have substituted y_m^1 with (3.4). Finally, the NN output yields,

$$y_1^3 = f^3\left(\sum_{j=1}^{3} w_{1m}^3 y_j^2 + b_1^3\right) \tag{3.8}$$

$$= f^3\left[\sum_{j=1}^{3} w_{1j}^3 f^2\left(\sum_{k=1}^{3} w_{jk}^2 f^1\left(\sum_{m=1}^{2} w_{km}^1 x_m + b_k^1\right) + b_j^2\right) + b_1^3\right] \tag{3.9}$$

We can generalize this expression to a MLP with l hidden layers. The complete functional form is,

$$y_1^{l+1} = f^{l+1}\left[\sum_{j=1}^{N_l} w_{1j}^3 f^l\left(\sum_{k=1}^{N_{l-1}} w_{jk}^2 f^{l-1}\left(\cdots f^1\left(\sum_{n=1}^{N_0} w_{mn}^1 x_n + b_m^1\right)\cdots\right) + b_k^2\right) + b_1^3\right] \tag{3.10}$$

which illustrates a basic property of MLPs: the only independent variables are the input values x_j. This confirms that a MLP, despite its quite convoluted mathematical form, is nothing more than an analytic function, specifically a mapping of real-valued vectors $\mathbf{x} \in \mathbb{R}^n \to \mathbf{y} \in \mathbb{R}^m$. In our example (Figure 3.5), $n = 2$ and $m = 1$. Consequentially, the number of input and output values of the function we want to fit must be equal to the number of inputs and outputs of our MLP.

Furthermore, the flexibility and universality of a MLP can be illustrated by realizing that the expression (3.10) is essentially a nested sum of scaled activation functions of the form

$$h(x) = c_1 f(c_2 x + c_3) + c_4 \tag{3.11}$$

where the parameters c_i are weights and biases. By adjusting these parameters, the activation functions can be shifted up and down or left and right, change slope or be rescaled (Figure 3.6), which is the key to the flexibility of a NN.

Activation function of output neuron

We stated in subsection 3.3.1 that the output neuron always have a linear activation function for regression to avoid restricting the output values. This is a requirement for a MLP to operate as a mapping $\mathbf{x} \in \mathbb{R}^n \to \mathbf{y} \in \mathbb{R}^m$. In this thesis, we will exclusively use a specific linear activation function for the output neuron o, namely the identity function $f(x) = x$,

$$f_o = f(u_o) = u_o \tag{3.12}$$

Matrix representation

We can introduce a more convenient notation for the activations in a NN. If we look at the structure of the NN in Figure 3.5, we realize that all the weights

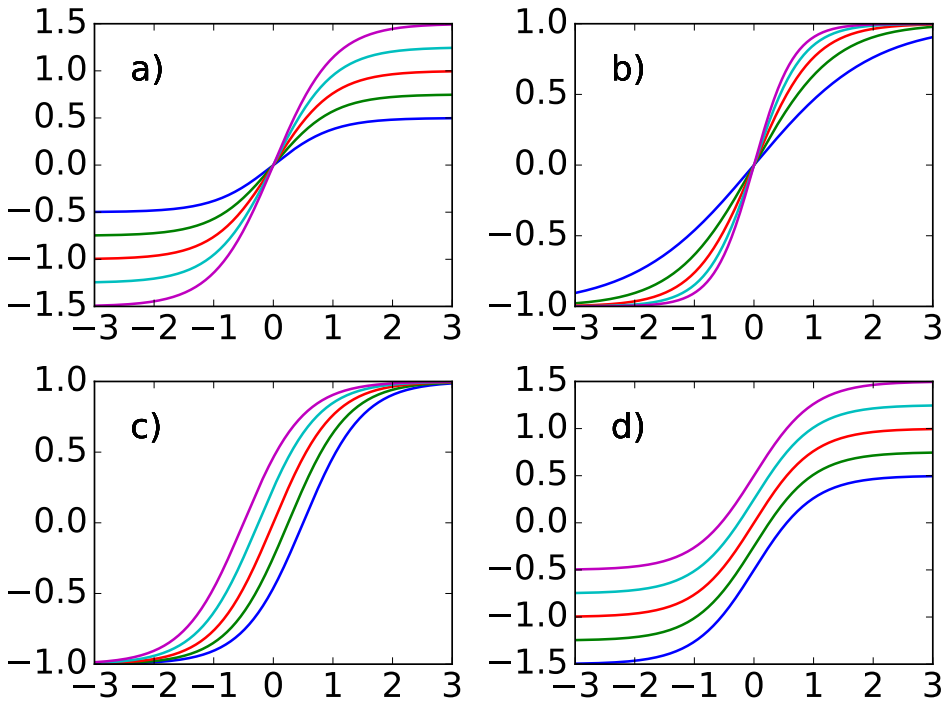


Figure 3.6: The mathematical expression for a MLP consists of nested terms of the form $h(x) = c_1 f(c_2 x + c_3) + c_4$, where f is the activation function and c_i are NN parameters. The flexibility of the MLP is shown by adjusting c_1, c_2, c_3 and c_4 such that $h(x)$ a) is scaled, b) has a change of slope, c) is shifted left and right, d) is shifted up and down, respectively.

connecting two adjacent layers can be represented as a matrix W, where $w_{ij} = w_{ij}^l$ is the weight connecting connecting node j in layer $l-1$ with node i in layer l. Additionaly, we can represent the biases and activations as layer-wise column vectors $\mathbf{b}$ and $\mathbf{y}$, so that $b_i = b_i^l$ is the bias of node i in layer l. We have that W_l is a $N_{l-1} \times N_l$ matrix, while $\mathbf{b}_l$ and $\mathbf{y}_l$ are $N_l \times 1$ column vectors.

With this notation, the sum in (3.7) becomes a matrix-vector multiplication, and we can write the equation for the activations of hidden layer 2 in Figure 3.5 as

$$\mathbf{y}_2 = f(W_2\mathbf{y}_1 + \mathbf{b}_2) = f\left(\begin{bmatrix} w_{11}^2 & w_{12}^2 & w_{13}^2 \\ w_{21}^2 & w_{22}^2 & w_{23}^2 \\ w_{31}^2 & w_{32}^2 & w_{33}^2 \end{bmatrix} \cdot \begin{bmatrix} y_1^1 \\ y_2^1 \\ y_3^1 \end{bmatrix} + \begin{bmatrix} b_1^2 \\ b_2^2 \\ b_3^2 \end{bmatrix}\right) \tag{3.13}$$

and we see that the activation of node i in layer 2 is

$$y_i^2 = w_{i1}^2 y_1^1 + w_{i2}^2 y_2^1 + w_{i3}^2 y_3^1 + b_i^2 = \sum_{j=1}^{3} w_{ij}^2 y_j^1 + b_i^2 \tag{3.14}$$

which is in accordance with (3.7). This is not just a nice and compact notation, but also a useful and intuitive way to think about MLPs: the output is calculated by a series of matrix-vector multiplications and vector additions that are used as input to the activation functions. For each operation $W^l\mathbf{y}^{l-1}$ we move forward one layer. Furthermore, it is the easiest way to implement NNs with TensorFlow, which is discussed in subsection 6.2.2.

3.4 Activation functions

A property that characterizes a NN, other than its connectivity, is the choice of activation function(s). As described in subsection 3.3.1, the following restrictions are imposed on an activation function for a FFNN to fulfill the universal approximation theorem [4]:

1. Non-constant
2. Bounded
3. Monotonically-increasing
4. Continous

We realize that the second requirement excludes all linear functions. Furthermore, in a MLP with only linear activation functions, each layer simply performs a linear transformation of its inputs. Consequentially, regardless of the number of layers, the output of the NN will be nothing but a linear function of the inputs.

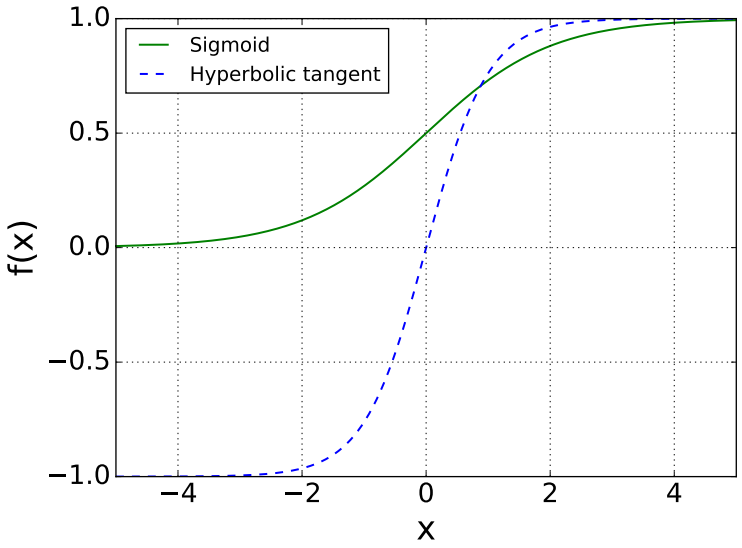


Figure 3.7: Two of the most common activation functions for neural networks. Both meet the requirements of the universal approximation theorem.

Thus we need to introduce some kind of non-linearity to the NN to be able to fit non-linear functions.

A group of non-linear functions that meet the above requirements are the *sigmoid* functions. The two sigmoid functions that are most commonly used as activation functions [5] in neural networks are the logistic function

$$f(x) = \frac{1}{1 + e^{-x}} \tag{3.15}$$

and the hyperbolic tangent

$$f(x) = \tanh(x) \tag{3.16}$$

The logistic functon is often referred to as "the sigmoid", which is the naming convention we will use in the following. Both functions are depicted in Figure 3.7. We observe that they are bounded above by 1, while the lower bound is -1 and 0 for the hyperbolic tangent and sigmoid respectively. This property keeps the activations from blowing up. The sigmoid are more bilogically plausible because the output of inactive neurons are zero. Such activation function are called *one-sided*. However, it has been shown [6] that the hyperbolic tangent performs better than the sigmoid for training MLPs. This will be further discussed in section (REF TO SECTION). Nevertheless, one should assess the problem at hand when deciding what activation function to use; the performance can vary from problem to problem.

In later years, the rectifier function (Figure 3.8)

$$f(x) = \max(0, x) \tag{3.17}$$

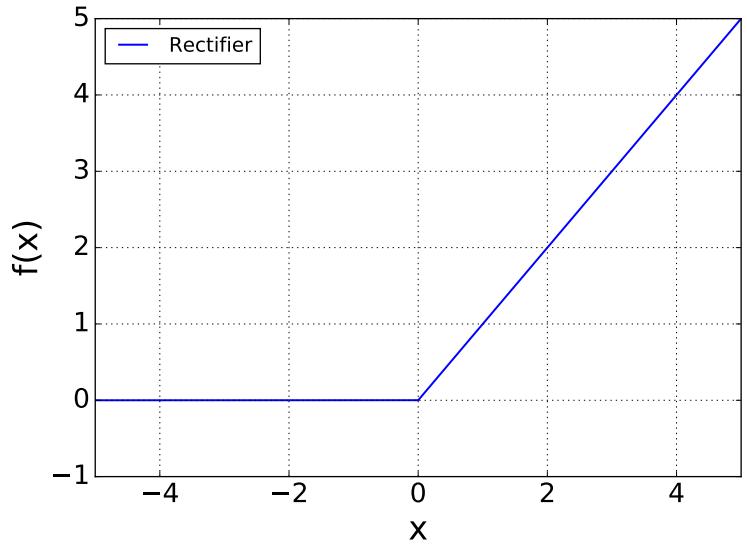


Figure 3.8: Two of the most common activation functions for neural networks. Both meet the requirements of the universal approximation theorem.

has become the most popular [7] for *deep neural networks*, i.e. NNs with a large number of hidden layers. It has been argued to be even more biologically plausible than the sigmoid and also perform better than the hyperbolic tangent, especially for deep, convolutional NNs [8]. Even though the rectifier is a piecewise *linear* function, it effectively behaves as a non-linearity. However, it does not have an upper bound, and therefore does not satisfy the universal approximator requirements. Further, the function values can potentially blow up. These problems are solved in practice by employing various teqhniques, illustrating that the universal approximation theorem should not be used as an absolute guideline for the construction of activation functions.

3.5 Training

In the beginning of this chapter we defined machine learning as the "science of giving computers the ability to learn without being explicitly programmed." In the neural network case, this learning is achieved by iteratively adjusting the parameters, i.e. the weights and biases (in the following, "weights" will often be equivalent to all the NN parameters, including the biases). The procedure depends on what learning paradigm we work with (unsupervised or supervised learning), and the type of NN we use.

In this thesis we employ NNs for regression: we want to interpolate data sets og atomic coordinates and energies. The goal is to adjust the weights of a

MLP so that it accurately represents a function that maps atomic coordinates to the corresponding energies. Interpolation with NNs is essentialy an optimization problem: the weights are typically initialized as random numbers, and then modified iteratively to minimize the error to a set of expected output values, in our case energies. This procedure is called *training*.

The input values to the function that we want to fit are called the *training set*, while the output values are called the *reference set* or the *target values*. The training set X and reference set Y are represented as matrices, where the i-th row of X and Y, denoted X_{i*} and Y_{i*} respectively, together form a *training example*. The combined data set consists of N such training examples. Note that the number of columns of X and Y are equal to the number of inputs and outputs of our MLP respectively. This is a restatement of the observation made in subsection 3.3.2: the number of inputs and outputs of our MLP must be equal to the number of input and output values of the function we are trying to fit.

3.5.1 Cost functions

To minimize the error, we need a way to *define* it. As in mathematical optimization, the error is represented by an objective function, also called *loss* function or *cost* function. Training a NN therefore amounts to the minimization of this function, which can be written in the following general way,

$$\Gamma = \Gamma(W, B, X, Y) \tag{3.18}$$

where W and B are the NN's weights and biases respectively. The value of this function is a measure of how well the NN is able to map $X \to Y$. By adjusting W and B, we try to minimize the value of this function.

The standard cost function used in regression with NNs is the mean-square error,

$$\Gamma = \frac{1}{2N} \sum_{i=1}^{N} (Y_i - y_i)^2 \tag{3.19}$$

where $y_i = y_i(W, B, X_{i*})$ is the value predicted by the NN for traing example X_{i*}. For a MLP with more than one output, Y_i and y_i are vectors. The constant $1/2$ is included to cancel out the exponent when this function is differentiated in subsection 3.6.3.

A variety of other cost functions are used for other machine learning tasks. In classificiation the output is often binary, hence the mean-square error function is inadequate. We will not go into further detail on other cost functions here.

3.6 Optimization

There are a large number of algorithms that can be used to determine the set of weights minimizing the cost function (3.19). Different kinds of gradient descent-based methods are widely used, while higher order methods like the conjugate gradient algorithm or Newton's method are rarely seen in the literature because they are too computationally expensive for large NNs and data sets. Therefore, only first-order methods will be discussed here.

The idea behind gradient descent methods is to minimize a function by iteratively taking steps in the direction of steepest descent towards a minima in parameter space. This direction is defined as the negative gradient of the function w.r.t. all its parameters. If we define $\boldsymbol{\theta} \in \mathbb{R}^d$ as a vector containing all the weights and biases of a MLP, we get the following iterative scheme,

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \gamma \nabla_{\boldsymbol{\theta}_k} \Gamma(\boldsymbol{\theta}) \tag{3.20}$$

where γ is a step size, called the *learning rate* in the context of machine learning. The process is started by initializing the parameters as random numbers. The value of γ is of great importance for the algorithm to converge, and is allowed to change at every iteration. Note that convergence to a minimum is not guaranteed withouth certain assumptions on the function Γ and the particular choices of γ. Also, the obatained minimum is generally local, not global. Different ways to update γ is discussed below.

3.6.1 Gradient descent variants

There are three versions of gradient descent [27], which differ in the number of training examples we present to the NN before updating the parameters. According to neural network terminology, the process of adjusting the parameters based on all the training examples, either in batches or all at once, is called an *epoch*.

In *batch* gradient descent, we compute the gradient of the cost function for the *whole* training set before updating, also called *offline learning*. This approach can be very slow and is intractable for datasets that do not fit in memory. Furthermore, there is a risk of performing redundant computations if many similar examples are present in the data set.

In contrast, *stochastic* gradient descent (SDG) performs a parameter update for *each* training example. SDG avoids redundant gradient calculations and is therefore faster than batch gradient descent. However, the accuracy of each update is lower, which can lead to quite large error oscillations. The concept of having more than one parameter update per epoch is called *online learning*.

Finally, we have *mini-batch* gradient descent, which is a mix of the two other approaches. The training set is divided into n equally sized mini-batches and a

parameter update is performed for each mini-batch. This is usually the algorithm of choice, as it has the optimal trade-off between speed and accuracy. Normal mini-batch sizes range between 50 and 256, but should in some degree be tailored to each problem.

We realize that SDG can be seen as a variant of mini-batch gradient descent with a mini-batch size of 1. In the literature, these two methods are often collectively referred to as SDG. We will adopt this naming convention for the rest of this thesis.

3.6.2 Optimization algorithms

In the following we will outline different optimization algorithms that are widely used in neural network research. They are all variations on the update rule (3.20). The main focus of the methods is to find a proper learning rate. A learning rate that is too small may lead to very slow convergence, while a learning rate that is too large can cause the loss function to fluctuate around the minimum or even diverge.

Momentum

Near local minima, the surface area of the cost function (in parameter space) often curve much more steeply in one direction than in another, forming ravines. SDG will have slow converge in such regions, as it will oscillate across the slopes while having slow progress along the bottom of the ravine. The Momentum method [15] avoids this problem by accelerating SDG in the downwards direction, while damping the oscillations. The update rule is

$$\begin{aligned} \mathbf{v}_k &= \eta \mathbf{v}_{k-1} + \gamma \nabla_k \Gamma \\ \boldsymbol{\theta}_{k+1} &= \boldsymbol{\theta}_k - \mathbf{v}_k \end{aligned} \tag{3.21}$$

where η is the momentum term, which is usually set to 0.9 or a similar value. The new update vector $\mathbf{v}_k$ is formed by adding a fraction η of the previous update vector $\mathbf{v}_{k-1}$ to the gradient. In this way, the magnitude of the update decreases for parameters (dimensions) whose gradients change direction, while it increases for parameters whose gradients points in the same direction as in the previous step. This helps to push SDG downhill towards the minima.

Adagrad

In SDG and Momentum, we have one learning rate γ that is applied to all the parameters. Ideally, the learning rate should be adapted to each individual parameter to perform larger or smaller updates depending on their importance. Adagrad [16] does just that. For brevity, we set

$$g_{k,i} = \nabla_{\theta_i} \Gamma(\boldsymbol{\theta}) \tag{3.22}$$

to be the gradient of the loss function w.r.t. parameter θ_i at step k. Adagrad adjust the general learning rate γ at each step k for every parameter θ_i based on the past gradients for that parameter,

$$\theta_{k+1,i} = \theta_{k,i} - \frac{\gamma}{\sqrt{G_{k,ii} + \epsilon}} \cdot g_{k,i} \tag{3.23}$$

where $G_k \in \mathbb{R}^{d \times d}$ is a diagonal matrix where each diagonal element $G_{k,ii}$ is the sum of squares of the gradient w.r.t. θ_i up to step k. The smoothing constant $\epsilon \sim 10^{-8}$ is present to avoid division by zero. This vectorized version of this equation yields,

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \frac{\gamma}{\sqrt{\text{diag}(G_k) + \epsilon}} \odot \mathbf{g}_k \tag{3.24}$$

where $\odot$ stands for element-wise multiplication. In other words, we do not have to manually tune the learning rate, Adagrad does that for us. A weakness of this method is that the learning rates inevitably shrink for each step, resulting in arbitrary small values, at which point the learning stops.

For the rest of subsection 3.6.2, we assume for brevity that all vector operations are element-wise, i.e. $\mathbf{g}\,\mathbf{h} = \mathbf{g} \odot \mathbf{h}$ and $\mathbf{g}^2 = \mathbf{g} \odot \mathbf{g}$.

Adadelta

Adadelta [17] is an extension of Adagrad that reduces the rate at which the learning rate decreases. The sum of all past squared gradients is replaced by a *exponentially decaying average* of all the previous squared gradients. We introduce the vector $\text{E}[\mathbf{g}^2]_k$ containing the decaying averages of the gradient w.r.t. all the parameters at step k. This vector is defined recursively as a weighted average of the previous averages $\text{E}[\mathbf{g}^2]_{k-1}$ and the current gradient $\mathbf{g}_k$,

$$\text{E}[\mathbf{g}^2]_k = \eta \text{E}[\mathbf{g}^2]_{k-1} + (1 - \eta)\mathbf{g}_k^2 \tag{3.25}$$

where η is a decay constant similar to that in the Momentum method, and is usually set to the same value (0.9). We also define a new parameter update vector $\Delta\boldsymbol{\theta}_k = -\mathbf{v}_k$ so that

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \Delta\boldsymbol{\theta}_k \tag{3.26}$$

Replacing the vector $\text{diag}(G_k)$ in (3.24) with the decaying average vector (3.25) yields the following parameter update vector,

$$\Delta\boldsymbol{\theta}_k = -\frac{\gamma}{\sqrt{\text{E}[\mathbf{g}^2]_k + \epsilon}}\mathbf{g}_k \tag{3.27}$$

Additionally, the learning rate γ is replaced by a decaying average of previous squared parameter updates up to step $k-1$, yielding the Adadelta update rule,

$$\mathbf{v}_k = \frac{\sqrt{\text{E}[\Delta\boldsymbol{\theta}^2]_{k-1} + \epsilon}}{\sqrt{\text{E}[\mathbf{g}^2]_k + \epsilon}}\mathbf{g}_k \tag{3.28}$$

The final replacement is done to obtain correct units for the update vector and to eliminate the learning rate from the equation. Thus, we do not even need to set a default learning rate with Adadelta.

Adam

The final algorithm we are going to discuss is Adapte Moment Estimation (Adam) [18]. This method computes adaptive learning rates for each parameter by storing exponentially decaying averages of both the gradients *and* the squared gradients (3.25),

$$\begin{aligned} \mathrm{E}[\mathbf{g}]_k &= \beta_1 \mathrm{E}[\mathbf{g}]_{k-1} + (1-\beta_1)\mathbf{g}_k \\ \mathrm{E}[\mathbf{g}^2]_k &= \beta_2 \mathrm{E}[\mathbf{g}^2]_{k-1} + (1-\beta_2)\mathbf{g}_k^2 \end{aligned} \tag{3.29}$$

We set $\mathbf{a}_k = \mathrm{E}[\mathbf{g}]_k$ and $\mathbf{b}_k = \mathrm{E}[\mathbf{g}^2]_k$ so that

$$\begin{aligned} \mathbf{a}_k &= \beta_1 \mathbf{a}_{k-1} + (1-\beta_1)\mathbf{g}_k \\ \mathbf{b}_k &= \beta_2 \mathbf{b}_{k-1} + (1-\beta_2)\mathbf{g}_k^2 \end{aligned} \tag{3.30}$$

where $\mathbf{a}_k$ and $\mathbf{b}_k$ are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively, hence the name of the method. The elements of these two vectors are initialized as zeros, which make them biased towards zero, especially during the first time steps. This bias is counteracted by computing bias-corrected version of the vectors (derived in [18],

$$\begin{aligned} \hat{\mathbf{a}}_k &= \frac{\mathbf{a}_k}{1-\beta_1^k} \\ \hat{\mathbf{b}}_k &= \frac{\mathbf{b}_k}{1-\beta_2^k} \end{aligned} \tag{3.31}$$

The parameter update vector for Adam is obtained by replacing the gradient $\mathbf{g}_k$ with $\mathbf{a}_k$ in (3.27), yielding the update rule,

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \frac{\gamma}{\sqrt{\hat{\mathbf{b}}_k} + \epsilon}\hat{\mathbf{a}}_k \tag{3.32}$$

The default values of the hyperparameters are $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$.

Which optimizer to use?

There is no general answer to the question stated above. The adaptive learning rate methods like Adagrad, Adadelta and Adam are generally more robust because they do not rely on manual fine-tuning of the learning rate. Also, an adaptive learning rate generally handles *sparse* data better, i.e. data where some learning features are poorly represented.

Adam has been proven to perform favourably over the other optimization methods for certain applications like classification with MLPs and convolutional

NNs [18]. However, the performance of various optimizers should be tested on the problem at hand to determine which method that works best.

3.6.3 Backpropagation

From: https://www.willamette.edu/ gorr/classes/cs449/backprop.html and http://neuralnetworksanddeeplearning.com/chap2.html. Mainly from the first one, but argument for error of neuron from second.

In the preceding sections we have discussed different optimizers that implements various update rules for the parameters of a loss function. In our case, these parameters are the weights and biases of a neural network and the cost function Γ is the mean-square-error (3.19)

$$\Gamma = \frac{1}{2}(Y - y)^2 \tag{3.33}$$

here written as a function of only *one* training example. What we have not discussed yet, is how we calculate the gradient of the cost function, i.e. how we obtain the partial derivatives g_{ij} and h_i w.r.t. all the weights and biases respectively,

$$\begin{aligned} g_{ij} &= \frac{\partial \Gamma}{\partial w_{ij}} \\ h_i &= \frac{\partial \Gamma}{\partial b_i} \end{aligned} \tag{3.34}$$

where we have dropped layer indicies for clarity. A common method to obtain these derivatives is *backpropagation* [9]. In backpropagation, a training example is propagated forward through the NN to produce an output. This output is compared to the desired output (target values), and the error is then propagated *backwards* through the layers to obtain the amount of which each parameter should be adjusted, hence the name. The method is essentially an implementation of the chain rule, and will allow us to calculate the partial derivatives of the cost with respect to all the parameters, thereby obtaining the gradient of the network.

1. Forward propagation

The first stage of the algorithm is forward propagation, producing an output as described in subsection 3.3.2. [3] In the following, we will use the same notation as in subsection 3.3.2. We also define

$$A_i = \{j : w_{ij}\} \tag{3.35}$$

[3] We assume that the MLP only has one output. As before, the results are valid for each individual output in a many-output MLP.

as the set $\{j\}$ of nodes anterior to node i and connected to node i with weights w_{ij}, in addition to

$$P_j = \{i : w_{ij}\} \tag{3.36}$$

as the set $\{i\}$ of nodes posterior to node j and connected to node j with weights w_{ij}.

2. Backward propagation

The second step is to backpropagate the error of the NN output. The weight derivative (3.34) can be expanded into two factors by use of the chain rule,

$$g_{ij} = \frac{\partial \Gamma}{\partial u_i} \frac{\partial u_i}{\partial w_{ij}} \tag{3.37}$$

Now we move in the opposite direction of the feed-forward stage: First we differentiate the cost w.r.t the input of neuron i, then we differentiate the input w.r.t. weight w_{ij} connecting neurons j (in the preceding layer) and i. The first term on the r.h.s. is defined as the the error δ_i of node i,

$$\delta_i = \frac{\partial \Gamma}{\partial u_i} \tag{3.38}$$

This definition can be justified by the following argument. To change the value of a cost function (3.18), we need to change the outputs of the neurons in the network. Changing the input u_j to neuron j by a small amount Δu_j results in the output

$$y_j = f_j(u_j + \Delta u_j) \tag{3.39}$$

This change will propagate through later layers in the network, finally causing the overall cost to change by an amount $\frac{\partial \Gamma}{\partial u_j} \Delta u_j$. If $\partial \Gamma / \partial u_j$ is close to zero, we are not able improve the cost much by perturbing the weighted input u_j; the neuron is already quite near the optimal value. This is a heuristic argument for $\partial \Gamma / \partial x_j$ to be a measure of the error of the neuron.
The second term on the r.h.s. of (3.37) is

$$\frac{\partial u_i}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \sum_{m \in A_i} w_{im} y_m = y_j \tag{3.40}$$

Combining the the two terms (3.38) and (3.40) yields

$$g_{ij} = \delta_i y_j \tag{3.41}$$

To compute this quantity, we thus need to know the outputs and the errors of all nodes in the network. The outputs are generated during the feed-forward stage (3.5),

$$y_i = f_i(u_i) = f_i \left(\sum_{j \in A_i} w_{ij} y_j + b_i \right) \tag{3.42}$$

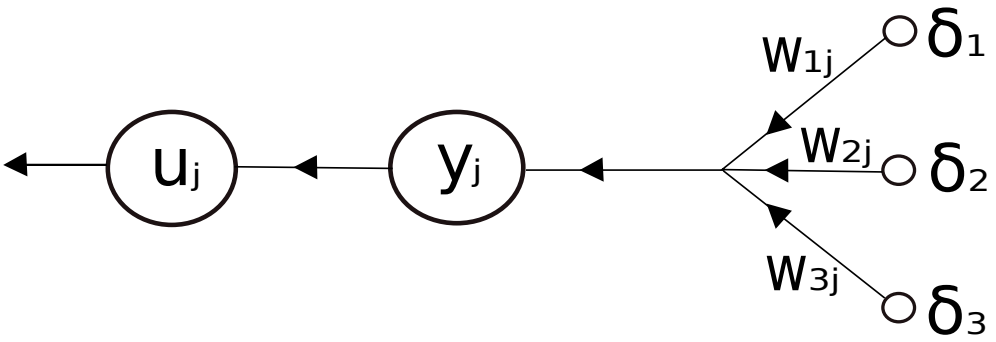


Figure 3.9: Illustration of the backpropagation algorithm. The direction of information flow is opposite of Figure 3.2. A hidden neuron j recieves a weighted sum of errors of all nodes in the posterior layer. Then, we differentiate in the backwards direction: the activation function (output) y_j of node j is differentiated w.r.t to its net input u_j.

and need to be stored for all nodes. The errors are obtained by backpropagating the error of the output neuron o,

$$\delta_o = \frac{\partial \Gamma}{\partial u_o} = \frac{\partial \Gamma}{\partial y_o}\frac{\partial y_o}{\partial u_o} = (Y - y_o)\frac{\partial y_o}{\partial u_o} \tag{3.43}$$

We remember from subsection 3.3.2 that the output neuron has the identity activation function $y_o = f_o(u_o) = u_o$, which reduces the error to

$$\delta_o = Y - y_o \tag{3.44}$$

This error is then propagated backwards through the network, layer by layer. The error of each neuron in layer l thus depends on the errors of all nerons in layer $l+1$. Consequentially, the error of an arbitrary hidden neuron can be written as a recursive equation (Figure 3.9),

$$\delta_j = \frac{\partial \Gamma}{\partial u_j} = \sum_{i \in P_j} \frac{\partial \Gamma}{\partial u_i}\frac{\partial u_i}{\partial y_j}\frac{\partial y_j}{\partial u_j} \tag{3.45}$$

We observe that also the inputs u_i to all nodes need to be stored during forward propagation for this scheme to work. The first term on the r.h.s. of (3.45) is the error δ_i of node i, which is assumed to be known. As long as there are no cycles in the network, there is an ordering of nodes from the output back to the input that respects this condition. It is therefore valid only for feed-forward NNs.

The second term is the derivative of the net input of all posterior nodes $\{i\}$ w.r.t. the output of node j,

$$\frac{\partial u_i}{\partial y_j} = \frac{\partial}{y_j}\left(\sum_{m \in A_i} w_{im} y_m + b_i\right) = w_{ij} \tag{3.46}$$

while the third is the derivative of node j's activation function w.r.t. its net input:

$$\frac{\partial y_j}{\partial u_j} = \frac{\partial f_j(u_j)}{\partial u_j} \equiv y_j' \tag{3.47}$$

Putting all the pieces together we obtain

$$\delta_j = y_j' \sum_{i \in P_j} w_{ij} \delta_i \tag{3.48}$$

This expression is illustrated in Figure 3.9. Each hidden neuron j recieves a weighted sum of the errors of all nodes in the posterior layer. Then we differentiate in the backwards direction (compare with Figure 3.2): the output y_j of node j is differentiated w.r.t. its input u_j.

The errors are propagated backwards through the whole NN until we reach the weights connecting the input layer and the first hidden layer. By propagating the error of only one ouput neuron, we thus obtain the errors of all the neurons at once. This is the main strength of the backpropagation algorithm, and the reason for its popularity in neural network research.
For the biases, we have (3.34),

$$h_j = \frac{\partial \Gamma}{\partial b_j} = \sum_{i \in P_j} \frac{\partial \Gamma}{\partial u_i} \frac{\partial u_i}{\partial y_j} \frac{\partial y_j}{\partial u_j} \frac{\partial u_j}{\partial b_j} \tag{3.49}$$

The only new term her is the last one:

$$\frac{\partial u_j}{\partial b_j} = \frac{\partial}{\partial b_j} \left(\sum_{m \in A_j} w_{jm} y_m + b_j \right) = 1 \tag{3.50}$$

Consequentially, the bias gradient is simply the error of each neuron:

$$h_j = \delta_j \tag{3.51}$$

Matrix notation

In subsection 3.3.2 we rewrote the forward propagation framework as matrix-vector equations. This is easily done also for the backpropagation case. From subsection 3.3.2 we have the set of vectors $\mathbf{b}^l$ and $\mathbf{y}^l$ for $l = 1, \ldots, L+1$, where L is the number of hidden layers. We now extend this set with the vectors $\boldsymbol{\delta}^l$ and $\mathbf{u}^l$, i.e. the errors and inputs (or *preactivations*) of layer l respectively. These are all column vectors of size $N_l \times 1$.
Comparing the sums in (3.42) and (3.48), we realize that since

$$\sum_j w_{ij} y_j \;\Rightarrow\; W\mathbf{y} \tag{3.52}$$

we have

$$\sum_i w_{ij}\delta_i \;\Rightarrow\; W^T\boldsymbol{\delta} \tag{3.53}$$

i.e. the weight matrices used in backpropagation are the transpose of the matrices used in forward activation. Thus, for nodes $\{j\}$ in layer l and nodes $\{i\}$ in layer $l+1$, the vectorized equation for the error of each hidden neuron is

$$\delta_j = y'_j \sum_{i\in P_j} w_{ij}\delta_i \;\Rightarrow\; \boldsymbol{\delta}_l = \mathbf{y}'_l \odot (W_{l+1}\boldsymbol{\delta}_{l+1}) \tag{3.54}$$

where $\odot$ signifies element-wise multiplication. The expression for the weight gradients (3.41) is converted into an outer product of the errors of layer $l+1$ and the outputs of layer l,

$$g_{ij} = \delta_i y_j \Rightarrow G_l = \boldsymbol{\delta}_{l+1}\mathbf{y}_l^T \tag{3.55}$$

where G_l is a matrix containing all gradients for the weights connecting layer l and $l+1$. The corresponding matrix for the biases is a $N_l \times 1$ column vector $\mathbf{H}_l$,

$$\mathbf{H}_l = \boldsymbol{\delta}_l \tag{3.56}$$

The set $\{G_l, \mathbf{H}_l\}$ for $l = 1, \ldots, L+1$ thus make up the total gradient of the cost function (3.19).

Training algorithm

We are now ready to express the complete training algorithm for a MLP with backpropagation using matrix-vector notation. The below process is performed for each training example $\mathbf{x} = X_{i*}$ in the whole training set.

1. Initialize the input layer:
$$\mathbf{y}_0 = \mathbf{x} \tag{3.57}$$
2. Propagate the activity forward: for $l = 1, \ldots, L+1$:
$$\mathbf{u}_l = W_l\mathbf{y}_{l-1} + \mathbf{b}_l \tag{3.58}$$
$$\mathbf{y}_l = f_l(\mathbf{u}_l) \tag{3.59}$$
Store all vectors $\mathbf{u}_l$ and $\mathbf{y}_l$.
3. Calculate the error in the output layer:
$$\boldsymbol{\delta}_{L+1} = \mathbf{Y} - \mathbf{y}_{L+1} \tag{3.60}$$
4. Backpropagate the error: for $l = L, L-1, \ldots, 1$:
$$\boldsymbol{\delta}_l = \mathbf{y}'_l \odot (W_{l+1}^T\boldsymbol{\delta}_{l+1})\cdot \tag{3.61}$$
Store all errors $\boldsymbol{\delta}_l$.

5. Compute weight and bias gradients:

$$G_l = \boldsymbol{\delta}_l \mathbf{y}_{l-1}^T, \quad \mathbf{H}_l = \boldsymbol{\delta}_l \tag{3.62}$$

6. Update weights and biases using an update rule of choice

Chapter 4

Neural networks in molecular dynamics

Following [10] and [2]. All the dynamics of a molecular system is determined by the multidimensional potential-energy surface (PES), which in general is a real-valued function depending on all atomic coordinates in the system. Using the Born-Oppenheimer approximation (FORKLARE ELLER REFERERE), various quantum mechanical methods like Hartree-Fock and density functional theory (DFT) are available to directly calculate the PES and forces for a given configuration. Calculating the energies for all relevant configurations of a system is demanding and time-consuming, and only a limited number of energies can be computed and stored. Consequentially, during a *ab inito* molecular dynamics (MD) simulations, the energies and forces for all configurations visited are not available beforehand. In *ab inito* MD, energies and forces are therefore calculated on-the-fly, typically using DFT. This is however a very inefficient approach. A more efficient method is to construct an analytical PES to use in simulations, but this can only be done for very simple systems.

A solution to this problem is the introduction of approximate PESs, and there are two fundamental approaches. The most widely used and convential method is to replace the solution of the Schrdinger equation by a simplified energy expression based on physical considerations and reasonable approximations. These functions are not obtained by ab inito methods, but can in many cases be sufficiently accurate to be used in simulations (BR HA EN EGEN SEKSJON OM DE VANLIGSTE POTENSIALENE).

An alternative approach is to employ machine learning (ML) potentials that have no direct physical meaning. The aim of these purely mathematical functions is to fit an analytic expression to a set of reference data obtained by quantum mechanical calulations. There are various methods (REFERENCES). ML potentials are particularly useful

- if long MD simulations are required

- if many MD trajectories are needed
- if the systems are too large for the application of QM methods

Physical and ML potentials thus enables us to extend the time and length scales of MD simulations beyond the realm of ab inito methods.

Feed-forward neural networks (FFNN) have been demonstrated to be useful for the construction of PESs because they are universal function approximators. There is no restriction in the accuracy that can be achieved when constructing neural network potentials (NNP). Unlike physical potentials, they are not restricted by any *ad hoc* functional form. NNPs offer a number of advantages for the construction of PESs:

- Energies can be fitted to very high accuracy, leaving only the underlying error of the reference data
- Evaluation of NNPs require much less CPU time than QM methods
- The NNP expression is unbiased and generally applicable to all types of bonding

Still, there are disadvantages one should be aware of:

- The evaluation of NNPs are notably slower compared to the use of classical force fields
- NNP expressions have no physical interpretation, and have very limited extrapolation capabilities
- NNPs are, for the time being, only applicable to systems containing only a few different chemical elements (but many atoms)

4.1 High-dimensional NNPs

The use of a single FFNN to represent complicated systems containing many atoms of different types is not possible for several reasons. First of all, the deegrees of freedom (the number of weight parameters) will be very many, making the training and evaluation of the network slow. Secondly, the NN has a symmetry problem. It does not take into account that exchanging two or more atoms can lead to an energetically equivalent configuration. For example, exchanging the positions of both hydrogen atoms in a water molecule will not alter the configuration energy. However, noting that all weights have numerically different values, changing the order of the input coordinates to the NN will result in a different energy value. This problem can be solved by a different choice of input coordinates, discussed below. The third problem in using a single FFNN is that

it is only applicable to the system size that has been used during the training. If the NN has been trained with 5 inputs, it can not be used for a system containing any other number of atoms.

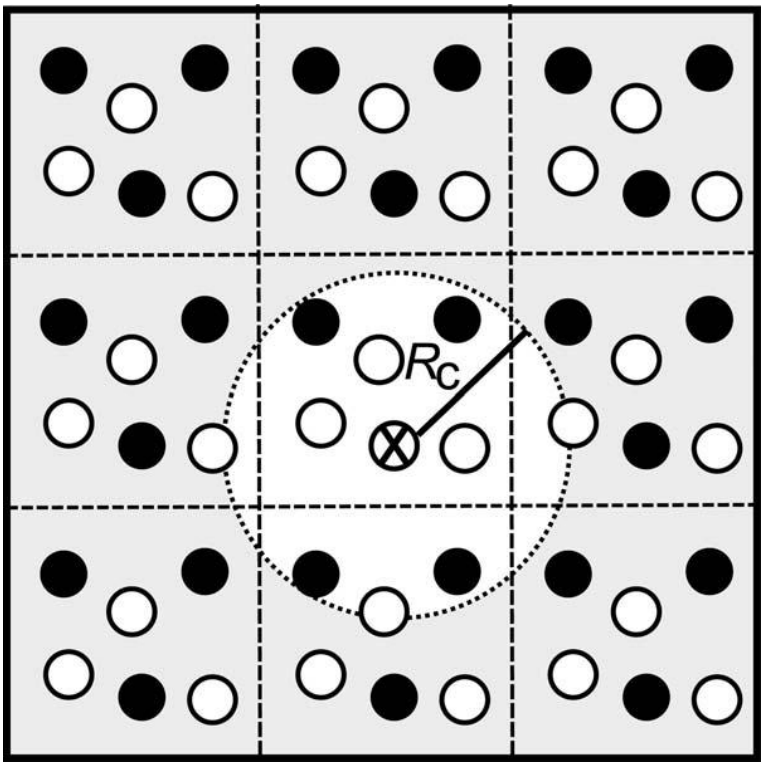


Figure 4.1

A solution to all these problems is to construct the system energy E_s as a sum of N atomic energy contributions E_i, which are provided by a set of individual atomic NNs,

$$E_s = \sum_{i=1}^{N} E_i \tag{4.1}$$

The atomic energies E_i depend on the local chemical environment up to a cutoff radius R_c as shown in figure 4.1 (STJLET FRA PAPER, M LAGE EGEN?). This is analogous to MD simulations with neighbour lists, thus making it possible to obtain the energies and forces of an atom and all its neighbours simultaneously. The positions of the neighbouring atoms within the cutoff sphere are described by a set of many-body symmetry functions discussed below. As the order of the summation does not change the total energy E_s, the symmetry problem above is solved. Each chemical element have seperate types of NNs with their own architecture and weights, but for a given element all atomic NNs are equal. Such

a system of NNs are also applicable to different system sizes: To add an atom of a given type, we simply extend the set of NNs by another network identical to the other NNs of the same type, and if an atom is removed, we delete the respective NN.

- The introduction of the cutoff R_c reduces the effective dimensionality of the problem, which allows to use NNs of tractable size
- The total energy is invariant with respect to the order of the atoms
- The NNP can be used for systems of different sizes
- We

We know that the energy of a molcule does not change under a translation or a rotation. Thus our symmetry functions should be invariant with respect to these operations.

4.1.1 Symmetry functions

To construct adequate symmetry functions, we need to have a cutoff function that defines the atomic environments. One such function is [10],

$$f_c(R_{ij}) = \begin{cases} 0.5 \cdot \left[\cos\left(\frac{\pi R_{ij}}{R_c}\right) + 1\right], & R_{ij} \leq R_c \\ 0, & R_{ij} > R_c \end{cases} \tag{4.2}$$

which is the monotonically decreasing part of a cosine function on $R_{ij} \in [0, R_c]$, seen in Figure 4.2.

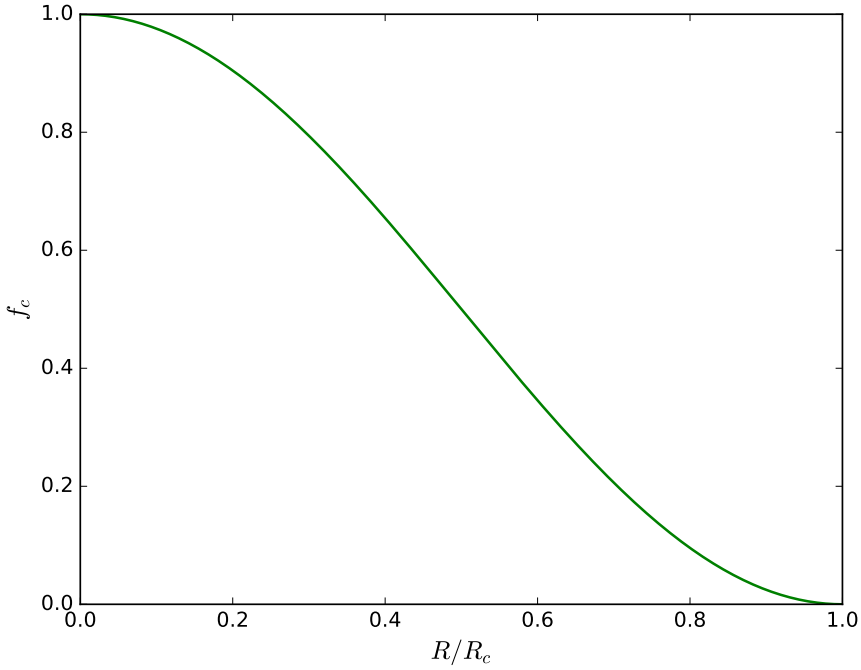


Figure 4.2: Plot of the cutoff function (4.2) applied in this thesis. This function is used to define a chemical environment around a central atom: only the atoms within the cutoff radius R_c contribute to its energy. These are called neighbouring atoms. The closer a neighbouring atom is, the larger the energy contribution, as is the case for most physical systems.

This function have the desirable property that it decreases with increasing distance R_{ij} between the central atom i and its neighbour j. At the cutoff radius R_c, is has zero value and slope, which is important to avoid discontinuities when computing energies and forces. Atoms beyond the cutoff radius are not a part of the central atom's chemical environment and therfore do not contribute to its energy.

Several types of many-body symmetry functions can be constructed based on these cutoff functions. They can be divided into two classes: *radial* symmetry functions, describing the radial distribution of neighbours up to the cutoff radius, and *angular* symmetry functions, specifying their angular arrangement. All symmetry functions depend on the positions of all the atoms inside the cutoff spheres. They make it possible to obtain a constant number of function values independent of the number of neighbours, which can change during MD simulations.

The most basic radial symmetry function is simply the sum of the cutoff

functions for all the neighbours j to atom i,

$$G_i^1 = \sum_{j=1}^{N} f_c(R_{ij}) \tag{4.3}$$

We need a set of such functions with different cutoff radii to describe the radial arrangement of the neighbouring atoms. A better alternative is to use a sum of products of Gaussians and the cutoff function,

$$G_i^2 = \sum_{j=1}^{N} \exp[-\eta(R_{ij} - R_s)^2] \cdot f_c(R_{ij}) \tag{4.4}$$

We now have two parameters that can be adjusted to probe different radii. The width parameter η determines the radial extension of the symmetry functions, while the shifting parameter R_s displaces the Gaussians to improve the sensitivity of the symmetry functions at specific radii. A third option is

$$G_i^3 = \sum_{j=1}^{N} \cos(\kappa R_{ij}) \cdot f_c(R_{ij}) \tag{4.5}$$

which are damped cosine functions with a period length adjusted by parameter κ. We will however not use this function because of the existence of negative function values which can lead to atoms canceling each other's contribution to the sum. In Figure 4.3 we see the radial symmetry functions for several different parameters. It is clear that a set of such functions have a large flexibility when we tune the parameters in different ways. This is what makes them able to represent the radial distribution of neighbours around a central atom.

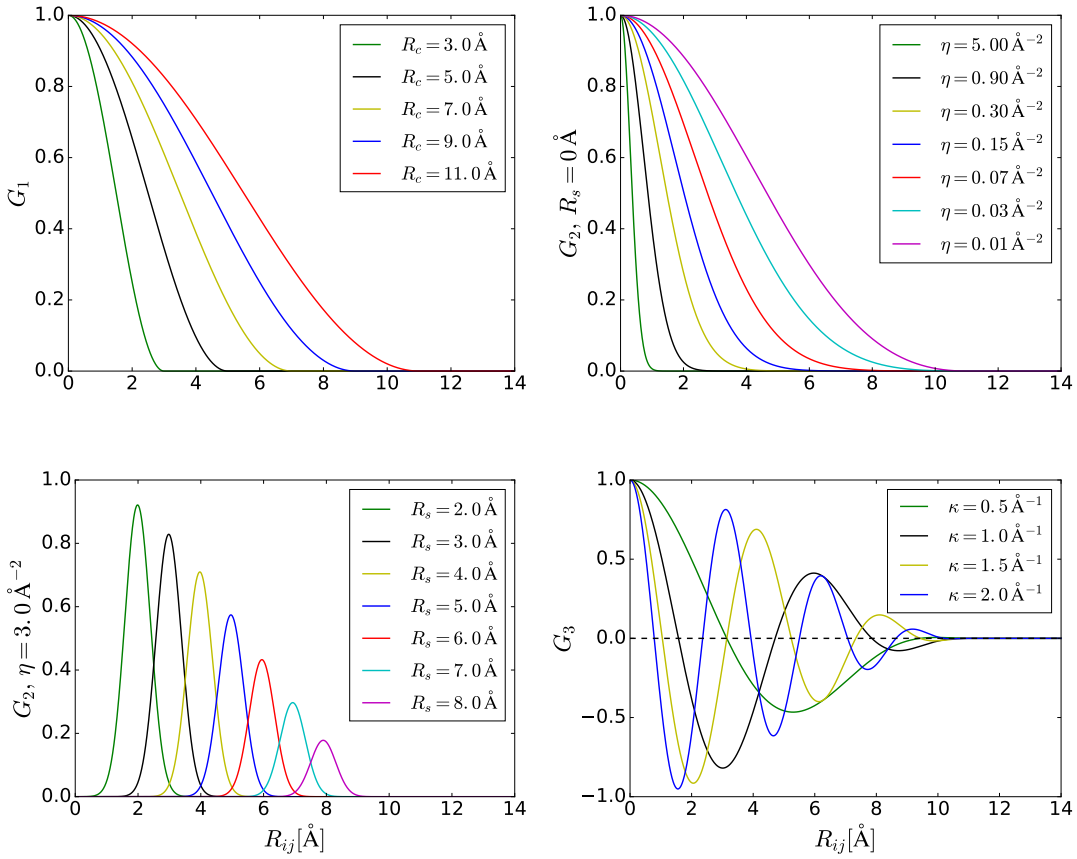


Figure 4.3: Radial symmetry functions G^1, G^2 and G^3 for an atom with one neighbour only. A set of such functions represents the radial distrubution of neighbours around a central atom placed at the origin. For G^2 and G^3 a cutoff $R_c = 11.0\,\text{Å}$ has been used.

To obtain a suitable structural fingerprint of the atomic environments, we also need the angular distribution of neighbouring atoms. This can be achieved by using functions depending on θ_{ijk}, which is the angle formed by the central atom i and the two interatomic distances R_{ij} and R_{ik}. The potential is periodic with respect to this angle, so we can use the cosine of θ_{ijk} instead. We thus define an angular symmetry function as a sum over all cosines with respect to any possible pair of neighbours j and k, multiplied by Gaussians of the three interatomic distances and the respective cutoff functions,

$$G_i^4 = 2^{1-\zeta} \sum_{j \neq i} \sum_{k>j} [(1 + \lambda \cos\theta_{ijk})^\zeta \cdot \exp(-\eta(R_{ij}^2 + R_{ik}^2 + R_{jk}^2)) \cdot \tag{4.6}$$

$$f_c(R_{ij}) f_c(R_{ik}) f_c(R_{jk})] \tag{4.7}$$

This function becomes zero if any of the interatomic distances is larger than R_c.

The parameter η takes here into account that the angular contribution depends on the atomic seperations. The angular arrangement can be investigated by using different values for ζ while the normalization factor $2^{1-\zeta}$ ensures that the range of values is independent of the choice of ζ. The parameter $\lambda \in -1, 1$ can be used to invert the shape of the cosine function: for $\lambda = +1$ the maxima of the cosine terms are at $\theta_{ijk} = 0°$, while for $\lambda = -1$ they are located at $\theta_{ijk} = 180°$. The cutoff function R_{jk} is included to ensure that only triplets where all three interatomic distances are within the cutoff radius. Another function can be defined that has no constraint on R_{jk},

$$G_i^5 = 2^{1-\zeta} \sum_{j \neq i} \sum_{k>j} [(1 + \lambda \cos\theta_{ijk})^\zeta \cdot \exp(-\eta(R_{ij}^2 + R_{ik}^2)) \cdot \tag{4.8}$$

$$f_c(R_{ij}) f_c(R_{ik})] \tag{4.9}$$

(4.9) will generally have larger function values than (4.7) because the lack of constraint on R_{jk} results in a larger number of non-zero terms in the summation. The angular part of G^4 and G^5 is identical, shown in Figure 4.4 for different values of ζ.

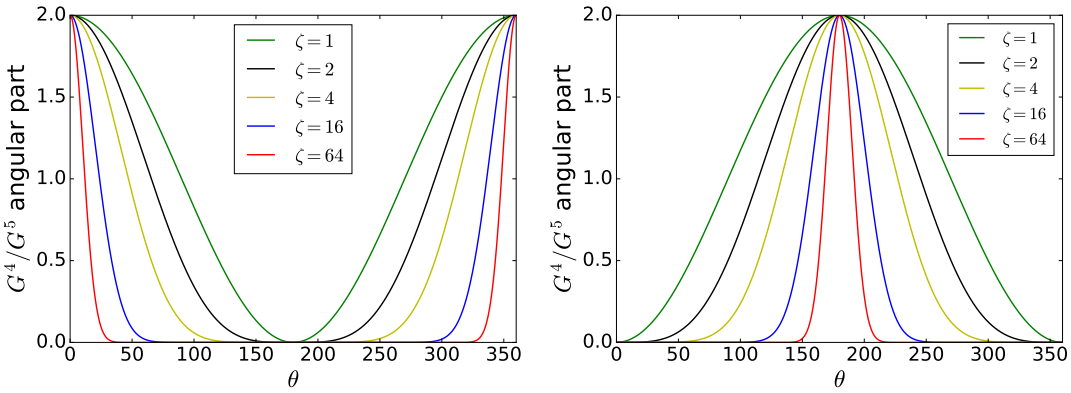


Figure 4.4: Angular part of symmetry functions G^4 and G^5 for an atom with one neighbour only. A set of such functions represents the angular distrubution of neighbours around a central atom placed at the origin. $\lambda = +1$ for Figure 4.4a, $\lambda = -1$ for Figure 4.4b.

The parameter values R_c, η, R_s, ζ and λ are not automatically optimized, like the weights. Once a set of symmetry functions has been determined, they remain fixed during the training of the NN. They are therefore an integral part of the NN, and need to be evaluated together with the NN itself when applied in simulations. The symmetry function set have to be customized for different systems, but in our experience the quality of the fit is not very sensitive to the choice of

parameters. An adequate set will shorten the convergence time, but it will not necesseraly result in a lower RMSE. However, it is important to use at least as many symmetry functions as there are degrees of freedom in the system. If this is not the case, the NN will recieve insufficient information and will not be able to fit the data properly.

4.1.2 Symmetry functions and forces

To integrate Newton's laws in MD simulations we need the forces on all the atoms. The force field associated with a PES was introduced in section 2.1 and is restated here for reference:

$$\mathbf{F} = -\nabla E \tag{4.10}$$

In the present case, the total energy E of the system is defined as a sum over all the atomic energies (4.1). Further, we have applied a cutoff so that each atom only gets energy contributions from its atomic environment up to a certain distance R_c. The force component $F_{k,\alpha}$, $\alpha = (x, y, z)$ acting on atom k with respect to coordinate $R_{k,\alpha}$, can therefore be written [2]

$$F_{k,\alpha} = -\frac{\partial E}{\partial R_{k,\alpha}} = -\sum_{i=1}^{N} \frac{\partial E_i}{\partial R_{k,\alpha}} = -\sum_{i=1}^{N_k+1} \frac{\partial E_i}{\partial R_{k,\alpha}} \tag{4.11}$$

N is as before the total number of atoms, while N_k is the number of neighbours of atom k. Note that E_k is included in the above sum - atom k itself contributes to the total energy of its atomic environment. We therefore have $N_k + 1$ contributions.

NNs have as we have seen well-defined functional forms, and analytical derivatives are therefore readily available. However, when symmetry transformations is applied, there is only an indirect relation between the energy and the cartesian coordinates of the atoms. To calculate the force (4.11) we therefore need to apply the chain rule,

$$F_{k,\alpha} = -\sum_{i=1}^{N_k+1} \frac{\partial E_i}{\partial R_{k,\alpha}} = -\sum_{i=1}^{N_k+1} \sum_{s=1}^{M_i} \frac{\partial E_i}{\partial G_{i,s}} \frac{\partial G_{i,s}}{\partial R_{k,\alpha}} \tag{4.12}$$

where M_i is the number of symmetry functions of atom i. In other words, we need to sum over all the symmetry functions of all the neighbours i of atom k and k itself (IS THIS CORRECT???) Looking at Fiure (MAKE A FIGURE ILLUSTRATING THE SYMMETRY TRANSFORMATION OF THE CARTESIAN COORDINATES THAT IS SENT TO THE NN), we realize that the first term $\partial E_i \partial G_{i,s}$ is the derivative of the output of the atomic NN of atom i with respect to the inputs, i.e. the gradient of the NN. This is an analytical gradient which is given by the architecture of the NN, including the values of the weights

and biases. We recall from 3.6.3 that the backpropagation algorithm calculates the derivatives of all neurons in a NN to obtain the amount of which each weight and bias should be adjusted during training. We can thus use a slightly modified version of this algorithm to compute the gradient of a NN,

1. Instead of backpropagating the *error* of the output neuron, we backpropagate the *derivative* of the output neuron's activation function.

2. The derivative is propagated all the way back to the input nodes. During training, the propagation stops at the weights connecting the input layer and the first hidden layer.

This procedure clarifies how a fully-connected feed-forward NN is built: To get the derivative of the output neuron w.r.t. the input neurons, we have to also compute the derivative of all the neurons in-between because all nodes in each layer are connected to all nodes in the following layer.

The second term $\partial G_{i,s}/\partial R_{k,\alpha}$ in (4.12) is given by the definition of the employed symmetry functions. The derivatives of the Behler symmetry functions given in subsection 4.1.1 are listed in A. WRITE ABOUT DIFFERENTIATION WITH SYMPY AND TENSORFLOW ETC.

In an MD simulation we only work with relative or inter-atomic coordinates and potentials. The training of neural networks is also based on an atom-centered approach; the cartesian coordinates that is given as input to the symmetry functions are always atom-centered. This means that the atom in question is placed at the origin and the positions of all neighbouring atoms are measured relative to the origin. Effectively, we are making a change of coordinates to the neighbours $\{j\}$ of the central atom i,

$$\mathbf{r}_j \to \mathbf{r}_j - \mathbf{r}_i \tag{4.13}$$

This is in accordance with the Behler symmetry functions, which only operate with relative coordinates. It is however not trivial how this affects the above equations for the forces, which are not written in an atom-centered way. The process of obtaining the forces can be illustrated with an example system consisting of three atoms of the same element with positions $\mathbf{r}_i$ where $i = 0, 1, 2$, shown in **??**. We want to find the force on atom 1 in the x-direction. Atom 0 is outside the cutoff sphere of atom 1, so according to (4.11) the force is,

$$F_{1,x} = \frac{\partial E_1}{\partial x_1} + \frac{\partial E_2}{\partial x_1} \tag{4.14}$$

For simplicity, we assume that each atom only have two different symmetry functions describing their atomic environment; one $G^2_{i,0}$ function describing the radial arrangement of neighbours and one $G^4_{i,1}$ dealing with the angular distribution. Each atomic NN thus have two inputs and one output. Since the three atoms are all of the same type, they will have identical NNs and symmetry vectors. We

will therefore denote the the two symmetry functions simply as G^2 and G^4. In the following we apply the notation $r_{ij} = |\mathbf{r}_j - \mathbf{r}_i|$. The expressions for E_1 and E_2 then are,

$$E_1 = \mathrm{NN}\big[G^2(r_{10}) + G^4(r_{10}, 0, 0)\big] = \mathrm{NN}\big[G^2(r_{10})\big] \tag{4.15}$$

where NN denotes the atomic neural network of the atoms. G^4 is zero because atom 1 only has one neighbour. Further,

$$E_2 = \mathrm{NN}\big[G^2(r_{20}, r_{21}) + G^4(r_{20}, r_{21}, \theta_{021})\big] \tag{4.16}$$

We note that atom 2 receives an energy contribution from both atom 0 and 1. To compute the derivatives of these energies with respect to the x-coordinate of atom 1, we must take into account the change of coordinates (4.13). We have that

$$\frac{\partial G^n(x_{ij})}{\partial x_j} = \frac{\partial G^n(x_{ij})}{\partial x_{ij}}\frac{\partial x_{ij}}{\partial x_j} = \frac{\partial G^n(x_{ij})}{\partial x_{ij}}\frac{\partial (x_j - x_i)}{\partial x_j} = \frac{\partial G^n(x_{ij})}{\partial x_{ij}} \tag{4.17}$$

and

$$\frac{\partial G^n(x_{ij})}{\partial x_i} = \frac{\partial G^n(x_{ij})}{\partial x_{ij}}\frac{\partial (x_j - x_i)}{\partial x_i} = -\frac{\partial G^n(x_{ij})}{\partial x_{ij}} \tag{4.18}$$

This symmetry enables us to calculate the derivatives above. Applying the chain rule we get,

$$\frac{\partial E_1}{\partial x_1} = \frac{\partial \mathrm{NN}}{\partial G^2(r_{10})}\frac{\partial G^2(r_{10})}{\partial x_1} = -\frac{\partial \mathrm{NN}}{\partial G^2(r_{10})}\frac{\partial G^2(r_{10})}{\partial x_{10}} \tag{4.19}$$

where we have applied the above change of coordinates (4.17). Using (4.18), the derivative of E_2 can be calculated correspondingly,

$$\frac{\partial E_2}{\partial x_1} = \frac{\partial \mathrm{NN}}{\partial G^2(r_{20}, r_{21})}\frac{\partial G^2(r_{20}, r_{21})}{\partial x_1} + \frac{\partial \mathrm{NN}}{\partial G^4(r_{20}, r_{21}, \theta_{021})}\frac{G^4(r_{20}, r_{21}, \theta_{021})}{\partial x_1} \tag{4.20}$$

$$= \frac{\partial \mathrm{NN}}{\partial G^2(r_{20}, r_{21})}\frac{\partial G^2(r_{20}, r_{21})}{\partial x_{21}} + \frac{\partial \mathrm{NN}}{\partial G^4(r_{20}, r_{21}, \theta_{021})}\frac{G^4(r_{20}, r_{21}, \theta_{021})}{\partial x_{21}} \tag{4.21}$$

Based on these expressions we can do an observation that at first might seem strange. From the definition of G_2 (4.4) we see that the sum contains two terms in the case of atom 2, one for each pair interaction between atom 2 and its two neighbours. The interaction term between atom 0 and 2 will disappear when differentiated with respect to atom 1's coordinates, and is thus not included in the force on atom 1. The sum in G_4 (4.7) however, consists of only one term corresponding to the only unique triplet with atom 2 as the central atom. This term is directly dependent on the coordinates of atom 0, which means that an atom which is outside the cutoff sphere of atom 1 still contributes to the force.

This is an artefact of many-body potentials like the NN potentials employed in this thesis: The force acting on an atom i depends on the positions of the atoms as far as $2R_c$ away because neighbours k of the neighbours j of i are included in the energies E_j that is differentiated.

This large effective range of atomic interactions is however not of great importance taking into account that the interaction strength falls rapidly with distance for most systems where Coulomb forces are not included.

Part II

Implementation and validation

Chapter 5

LAMMPS

REFERENCE TO MANUAL - FOOTNOTE OF REFERNCE? LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator) is a classical molecular dynamics simulation package developed at Sandia National Laboratories [12]. It is written in highly portable and extendable objected-oriented c++. The package's functionality encompasses a wide variety of potentials, atom types, integrators, thermostats and ensembles and are able to simulate a large number of different systems, including atoms, molecules, coarse-grained particles, polymers, graunular materials and metals. LAMMPS can be run on a single processor or in parallell. There exist several other molecular dynamics packages, like GROMACS, OpenMD, Aber and NAMD that could also have served the purpose of this thesis. We have chosen to work with LAMMPS because it is well documented, [1] easy to expand upon [2] and because the Computational physics group at UiO has a lot of experience with using it for molecular dynamics simulations. In this section we will outline the basic usage of LAMMPS, including a walk-through of an example input script. We also describe its class hierarchy and how we can add our own functionality to the package.

5.0.1 Installing LAMMPS

We have chosen to install LAMMPS by cloning the Github repository and compile the source code by running `make` in the source directory. To compile a serial version of the software, we simply run the command

```
make serial
```

while the corresponding command for the parallel MPI version is

```
make mpi
```

[1] http://lammps.sandia.gov/doc/Manual.html
[2] http://lammps.sandia.gov/doc/Developer.pdf

New updates can be downloaded by setting an upstream git remote to point to the LAMMPS GitHub repository and doing a `git fetch upstream`. LAMMPS also have several additional packages that can be installed. This can be done by running

```
make yes-<package name>
```

e.g. `manybody` to install many-body potentials like Stillinger-Weber and Vashishta.

5.0.2 LAMMPS input script

LAMMPS is run by providing an input script as argument to the excecutable. This input script is read line by line and has its own syntax. A good way to show the basics an input file is to look at a simple example of a script to measure diffusion in liquid argon,

```
# initialization
units           metal
dimension       3
boundary        p p p
atom_style      atomic

# create geometry and atoms
lattice         fcc 1.08506
variable        Nc equal 10
region          myRegion block 0 ${Nc} 0 ${Nc} 0 ${Nc}
create_box      1 myRegion
create_atoms    1 box

# set mass and initial temperature
mass            1 1.0
variable        temp equal 300
velocity        all create ${temp} 87287 mom yes

# compute diffusion
compute         disp all displace/atom

# potential
pair_style      lj/cut 2.5
pair_coeff      1 1 1.0 1.0 2.5
neighbor        0.5 bin
neigh_modify    every 20 delay 0 check no

# integration
timestep        0.01
run_style       verlet
fix             integration all nve
```

```
# output
thermo          50
thermo_style    custom step temp density press ke pe etotal
thermo_modify   norm yes

# run simulation
run             5000
dump            diff all custom 100 diff*.dat c_disp[4]
run             10000
```

We will in the following briefly explain what processes that are evoked in LAMMPS when the above commands are read. We start with the initialization of three basic properties of the simulation: units, boundary conditions and particle types.

```
units       metal
```

This command defines the units that are used in the simulation and the output. LAMMPS have eight different unit sets. The `metal` set measures distance in Ångstrøm, energy in eV and temperature in Kelvin. The choice of units depends on the system that is investigated and the scale we are looking at.

```
boundary    p p p
```

We want to measure diffusion in a bulk Argon liquid, thus we want to have periodic boundary conditions in all three dimensions, annotated by a `p`. LAMMPS can also handle stiff `f` and adaptive `s` non-periodic boundaries. Adaptive means that the position of the face is set so as to encompass the atoms in that dimension.

```
atom_style atomic
```

Different systems need different information to be stored for each atom. For style `atomic`, only the default attributes are associated with each atom, namely coordinates, velocities, atom IDs and types. This is sufficient for pair-interacting, non-bonded systems like Argon.
Next, we create the geometry of the simulation domain.

```
lattice     fcc 5.720
```

This defines the initial configuration of the atoms. Here, a ffc lattice with a lattice constant of 5.720 is used. Other types of lattices based on cubic or square unit cells are also available.

```
variable    Nc equal 10
region      simBox block 0 ${Nc} 0 ${Nc} 0 ${Nc}
```

The system's geometry is defined with the `region` command. The `block` style is simply a 3-dimensional straight-faced box with a size of $N_c = 10$ unit cells in each dimension. We have labelled the region `simBox`. The number of unit cells `N_c` is defined as a LAMMPS `variable`. Variables can be referenced elsewhere in the script by writing `${variable_name}` to become part of a new input command like above. LAMMPS enables many styles of variables to be defined.

```
create_box      1 simBox
create_atoms    1 box
```

The command `create_box` creates a simulation box based on the specified region, in our case the `block` region defined above. The argument specifies the number of atom types that will be used in the simulation. Next, `create_atoms` with the argument `box` fills the simulation domain with atoms of type 1 on the lattice. LAMMPS also lets you create a random collection of atoms or single atoms at specified coordinates.

```
mass            1 28
variable        temp equal 300
velocity        all create ${temp} 87287
```

We need to assign mass to the atoms. For metal units, mass is measured in grams/mole. The atoms are also given an initial velocity that corresponds to the given initial temperature.

```
pair_style      lj/cut 2.5
pair_coeff      1 1 1.0 1.0 2.5
```

The choice of potential is made with the `pair_style` command. We want to simulate interactions using the cutoff Lennard-Jones potential with a global cutoff of 2.5 . The LJ parameters are set with `pair_coeff`, the way this is done depends on the potential being used. For our choice of potential the arguments is as follows: `atom-type-1 atom-type-2 sigma epsilon cutoff`. The parameters and cutoff can thus be set seperately for each pair of atom types.

```
neighbor        0.5 bin
neigh_modify    every 20 check yes
```

These commands sets parameters that affect the building of neighbor lists. The first argument to `neighbor` is the skin size, while the second selects what algorithm is used to build the lists. The `bin` style creates the lists by binning, which in most cases (including ours) is the fastest method. Further, we can control how often the lists are built with `neigh_modify`. The above arguments specifies that new neighbour lists are built every 20 steps, but every step LAMMPS checks if

any atom has moved more than half the skin distance. If that is the case, new lists are built.

```
timestep        0.01
run_style       verlet
fix             integration all nve
```

LAMMPS integrates Newtons' equations of motion with the velocity-Verlet algorithm by defualt. This is the integrator of choice for most MD applications due to its simplicity and symplectic nature (REF TO SECTION). The rRESPA integrator [11] scheme is also available. LAMMPS does however not integrate and update the positions and velocities of the particles if not explicitly told so. This is done with a `fix`, which is any operation that is applied to the system during timestepping. The above fix tells LAMMPS to integrate all atoms in the system so that they follow trajectories consistent with the microcanonical ensemble.

```
thermo          50
thermo_style    custom step temp press ke pe etotal
thermo_modify   norm yes
```

We can control what thermodynamic properties to calculate and output with `thermo_style`, while `thermo` decides how often they should be computed. We want to output the time step, temperature, pressure, kinetic energy, potential energy and total energy. Also, we want to normalize the extensive quantities (the energies) by the number of atoms.

```
compute         disp all displace/atom
dump            diff all custom 100 diff*.dat c_disp[4]
```

To measure diffusion in the Argon liquid, we need to calculate the net displacement of all atoms. This can be done with a `compute`, which defines a computation that is performed on a group of atoms, in this case all atoms. The `displace/atom` compute calculates the current displacement of each atom from its original coordinates, including all effects due to atoms passing through periodic boundaries. For the compute to actually be performed, it needs to be evoked by other LAMMPS commands, like `dump`, which writes a snapshot of atom quantities to one or more files every N timesteps. Computes are referenced via the following notation `c_ID`, where ID is the ID of the compute.

```
run     5000
```

Lastly, we run the simulation for 5000 timesteps. LAMMPS allowes several run commands to be issued after one another. This comes in handy if we want

to thermalize the system before measuring diffusion. This can be achieved by writing the following commands:

```
run     5000
dump    diff all custom 100 diff*.dat c_disp[4]
run     10000
```

When parsing the second run command, LAMMPS will continue the dynamics while computing eventual new fixes, computes, dumps etc. defined since the last run command.

This section has illuminated some of the basic funcionality of LAMMPS through several comments on a simple input script. Now we move on to how class hierarchy of LAMMPS is structured.

5.0.3 LAMMPS structure

LAMMPS is written in `C++` in an object-oriented fashion. The class hierarchy is shown schematically in figure (MAKE MY OWN FIGURE IN INKSCAPE). The blue classes are the core classes, these are visible anywhere in LAMMPS. Most of these have subclasses, of which the red ones are called style classes. We recognize many of the input script commands among the style classes; a rule of thumb is that every input script command has a corresponding class and a corresponding file name in the source directory. For instance, the cutoff Lennard-Jones potential used in our input script is a subclass of Pair, and is evoked by the command `pair_style lj/cut` and the source file is named `pair_lj_cut`.

We will not go into detail about the functionality and communication of these classes. The one that is the most relevant for this work is the `Pair` class. This is the parent class of non-bonded or pair-wise potentials, which in LAMMPS includes many-body potentials like Stillinger-Weber and the Tersoff 3-body potential. The NN potential in this thesis falls under this category, and thus inherits from `Pair`. In the following section we will describe how LAMMPS can be extended with such a new `pair_style`.

5.0.4 Extending LAMMPS

To extend LAMMPS with a new component, we simply add a new `*.cpp` and `*.h` file to the source directory and re-compile. The new class will (in theory) work with all other LAMMPS classes without breaking any functionality. A good strategy is to start with an existing potential source file that is somewhat similar to the one we want to make, instead of programming from scratch. We aim to construct a many-body non-bonded potential, and have chosen to base our implementation on the Stillinger-Weber potential file, `pair_sw.cpp`.

The new potential class is added to LAMMPS with the following function call in the header file,

```
PairStyle(nn/manybody,PairNNManyBody)
```

The first argument is the name the potential will have in the input script, while the second is the name of the class. To use our new NN potential in a simulation we simply write,

```
pair_style      nn/manybody
```

in the input script. This command evokes two functions that need to be implemnted: `settings` and `init_style`. The former assures that no additional arguments are provided to `pair_style` (I SHOULD MAYBE CHANGE THIS TO PROVIDE GLOBAL CUTOFF LATER???), while the latter decides what kind of neighbour lists that LAMMPS should operate with. Our NN potential needs full neighbour lists to be able to compute three-body terms, something which applies to all many-body potentials.

The parent class `Pair` has three *pure virtual member functions*, i.e. functions that is required to be implemented by a derived class: `compute`, `coeff` and the aforementioned `settings`. The `compute` method is where the forces and energies are calculated. This function is called once each time step by the Verlet algorithm and the resulting forces are used to integrate Newton's equations of motion. The coefficients of the employed potential are set in `coeff`, which in the case of a NN potential is the weights and biases. These are read from file, and based on information on the architecture of the NN and choice of activation function, the network is reconstructed as a set of Armadillo matrices and vectors. This is achieved with the input script command,

```
pair_coeff      * * <path-to-saved-NN> <cutoff>
```

where the wildcard asterisks signify that the coefficients should be given to all pairs of atom types. THIS IS ONLY VALID FOR ONE ATOM TYPE. WRITE ABOUT MORE THAN ONE TYPE LATER. ALSO WRITE ABOUT HOW THE ENERGIES AND FORCES ARE CALCULATED WHEN I HAVE TWO ATOM TYPES UP AND RUNNING.

Chapter 6

TensorFlow

TensorFlow [13] [14] is an open source software library developed by Google for numerical computation using data flow graphs (DFG). A DFG is a graphical representation of how data is advanced through a system, including what kind of information is input to and output from the system and where data is stored. The data flow is represented as a directed graph with a number of *nodes* that is connected by *edges*. In TensorFlow, nodes represent mathematical operations, while the edges represent the multidimensional data arrays (tensors) communicated between them.

A simple example of a DFG is displayed in Figure (LAGE EN FIGUR BASERT P EKSEMPLET http://web.cecs.pdx.edu/ mperkows/temp/JULY/data-flow-graph.pdf , which is a visualization of the flow of data for the (real) solutions

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \tag{6.1}$$

$$x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \tag{6.2}$$

of a quadratic equation

$$ax^2 + bx + c = 0 \tag{6.3}$$

We observe that each node (circles) contain the mathematical operations needed to compute (6.2), while the arrows (edges) show what data each node recieve and where the output is sent. The inputs and outputs of the computation is marked with rectangular boxes.

TensorFlow was originally developed by the Google Brain Team [1] with the purpose of conducting machine learning and deep neural networks research, but is applicable in other domains as well, e.g. for solving partial differential equations. The package incorporates a wide variety of machine learning algorithms, including convolutional neural networks and recurrent neural networks. A computation

[1] https://research.google.com/teams/brain/

expressed using TensorFlow can be executed on a wide variety of devices and systems, from mobile phones to large-scale distributed systems and GPUs.

In this chapter we will go through a simple example on how to build a computational graph with TensorFlow, and how to train a feed-forward neural network. But first, a few tips on how to install the software.

6.1 Installing TensorFlow

The easiest way to install TensorFlow on Ubuntu [2] (CHANGE LINK, IT IS WRONG BECAUSE OF UNDERSCORE...) is to use pip, [3] a package management system for software written in Python. Pip can be installed with the command

```
$ sudo apt-get install python-pip python-dev
```

TensorFlow can be installed with or without GPU support. The GPU version requires a system with CUDA-enabled NVIDIA GPUs [4]. We have used the CPU version for the development of our code, which is installed with the command

```
$ pip install tensorflow
```

A simple

```
$ python -c "import tensorflow"
```

will test if the installation was succesful.

6.2 TensorFlow basic usage

TensorFlow relies on highly-optimized `C++` for its computations, but offers APIs [5] in both `Python`, `C++` and a few other languages. The `Python` API is at present the most complete, the easiest to use and the best documented. We have therefore chosen to use `Python` to train neural networks. For a discussion on ways to import a trained NN to `C++` see subsection 5.0.4 and section 8.1.

[2] https://www.tensorflow.org/install/installlinux
[3] https://pip.pypa.io/en/stable/
[4] https://developer.nvidia.com/cuda-gpus
[5] https://www.tensorflow.org/api_docs/

6.2.1 Hello world

As already mentioned, a TensorFlow computation is described by a graph which consists of nodes that communicate with multi-dimensional arrays along interconnected edges. Each node has zero or more outputs, and represents the instantation of an *operation*. Each operation occurs within a so-called *session*. A `Hello world!` program can be written as

```
import tensorflow as tf

hello = tf.constant('Hello world!')
sess = tf.Session()
print sess.run(hello)
# --> Hello world!
```

This simple example outlines the general workflow for building a TensorFlow graph,

1. Declare nodes (operations)
2. Initiate a session
3. Run operations within session

The only operation in the above example is the built-in TF constant that holds the `Hello world!` string, and the corresponding DFG only consists of one node. In more involved examples, visualizing the DFG can be useful for finding out how the script works. This is done with the included TF visualization tool TensorBoard. We will now look at how to construct a neural network with TF and visualize the corresponding DFG.

6.2.2 Creating a neural network

In section 3.1 we saw how an artifical neural network could be written as a nested sum of activation functions (3.10) where the weights connecting layer $l-1$ and l were represented as a matrix W_l, while the biases of layer l were represented as a vector $\mathbf{b}_l$. The input vector $\mathbf{x}$ to the neurons in layer l, also called the *preactivation*, can then be written

$$\mathbf{x}_l = W_l \mathbf{y}_{l-1} + \mathbf{b}_l \tag{6.4}$$

where $\mathbf{y}$ is the output vector of layer $l-1$. The activations of layer l is thus

$$\mathbf{y}_l = f(\mathbf{x}) \tag{6.5}$$

where f is the chosen activation function. This demonstrates that a NN can be constructed as a set of matrix-vector multiplications and additions which are

input to a chosen activation function. We now want to construct a NN with 3 inputs neurons, one hidden layer with 10 neurons and 1 output neuron. We use the sigmoid (3.15) activation function for the hidden layer, while the output layer have the linear activation function (REF). This can be achieved with the following `python` script,

```
import tensorflow as tf

# NN parameters
inputs  = 3
nodes   = 10
outputs = 1

# declare input tensor
inputLayer = tf.placeholder( tf.float32, [None,inputs] )

# weights input layer -> hidden layer and biases hidden layer
W1 = tf.Variable( tf.random_normal([inputs,nodes]) )
b1 = tf.Variable( tf.constant(0.1, shape=[nodes]) )

# activation hidden layer
preAct1 = tf.add( tf.matmul(inputLayer, W1), b1 )
act1 = tf.nn.sigmoid(preAct1)

# weights hidden layer -> output layer and bias output layer
W2 = tf.Variable( tf.random_normal([nodes,outputs]) )
b2 = tf.Variable( tf.constant(0.1, shape=[outputs]) )

# activation output layer
preActOutput = tf.add( tf.matmul(act1, W2), b2 )
activationOutput = preActOutput
```

There are several aspects of the TensorFlow `python` API to comment on here. We observe that the input layer is declared as a TF `placeholder`. This is a special type of TF tensor that is not initialized with any values, but is rather fed with data on its execution. Specifying the first component of its shape as `None` signifies that we can supply an arbitrary number of data points at the same time. These properties enables the same NN to be evaluated for different data vectors of varying size on-the-fly during training, explained in more detail below.

The weights and biases are declared as TF `Variables`. These are the trainable parameters of the computational graph, and are normally constructed as random numbers. In the present case, the weights are normally distributed numbers while the biases are all set to the same value 0.1. We note that the shapes of the weight matrices and bias vectors depend on the number of nodes in each layer. If n_l stands for the number of neurons in layer l, we have in general that the weight matrix connecting layer $l-1$ and l is of shape $n_{l-1} \times n_l$, while the bias vector of layer l has n_l number of elements.

The activations (outputs) of each layer are calculated according to (6.4) and (6.5). Together with the input `placeholder`, these are of type `tf.Tensor`, i.e. nodes in the computational graph.

6.2.3 Visualizing the graph

We can use TensorBoard to visualize the DFG of the above script. The DFG for the present example is quite simple, but for more involved models it quickly becomes difficult to keep track of all the different components. A convenient tool to order and group the DFG is *name scoping*. Both `Tensors` and `Variables` can be name scoped and TensorBoard uses this information to define a hierarchy of the nodes in the graph. An example of name scoping is

```
with tf.name_scope('layer1'):
   W1 = tf.Variable( tf.random_normal([inputs,nodes]),
       name='weights')
    b1 = tf.Variable( tf.constant(0.1, shape=[nodes]),
       name='biases')
```

which results in the following names,

- `layer1/weights`
- `layer1/biases`

Additionally, TensorBoard will group the two variables together under `layer1`. To save the graph to file we have to start a TF session. The following script deploys name scoping, initiates a session and writes the DFG to file,

```
import tensorflow as tf

# NN parameters
inputs = 3
nodes = 10
outputs = 1

with tf.name_scope('inputLayer'):
    inputData = tf.placeholder(tf.float32, [None,inputs],
       name='inputData')

with tf.name_scope('layer1'):

    W1 = tf.Variable( tf.random_normal([inputs,nodes]),
       name='weights' )
    b1 = tf.Variable( tf.constant(0.1, shape=[nodes]),
       name='biases' )
```

```
    preAct1 = tf.add( tf.matmul(inputData, W1), b1,
        name='preActivation' )
    act1 = tf.nn.sigmoid(preAct1, name='activation')

with tf.name_scope('outputLayer'):

    W2 = tf.Variable( tf.random_normal([nodes,outputs]),
        name='weights' )
    b2 = tf.Variable( tf.constant(0.1, shape=[outputs]),
        name='biases' )

    preActOutput = tf.add( tf.matmul(act1, W2), b2,
        name='preActivation' )
    actOutput = tf.identity(preActOutput, name='activation')

# initialization node
initOperation = tf.global_variables_initializer()

# start session
with tf.Session() as sess:

    # initialize all variables
    sess.run(initOperation)

    # write summaries
    tf.summary.FileWriter('Summaries', sess.graph)
```

There are two new TF function calls to comment on in this script. First, we have the line

```
initOperation = tf.global_variables_initializer()
```

Constants are initalized when we call `tf.constant`, and their value can never change. By contrast, variables need to be initialized using a seperate operation within a session. The function `tf.global_variables_initializer` adds such an operation to the graph, which initializes all global variables when run with `sess.run(initOperation)`.

Secondly, the graph is written to file for visualization with the function `tf.summary.FileWriter`. The first argument is the location of the files, while the second is the graph that is launched in the present session.

The resulting graph is displayed in Figure 6.1. By default, only the top of the hierarchy defined by name scoping is shown. In our case, this is the three NN layers. In this figure, the output layer is expanded to reveal all the nodes in this scope. The nodes are marked by ellipses, while the rectangles are namespaces defining another level in the hierarchy, each containing more nodes. The edges show how the nodes are connected. [6]

[6]Each TF variable and constant are actually made up of several nodes, including *assign* and

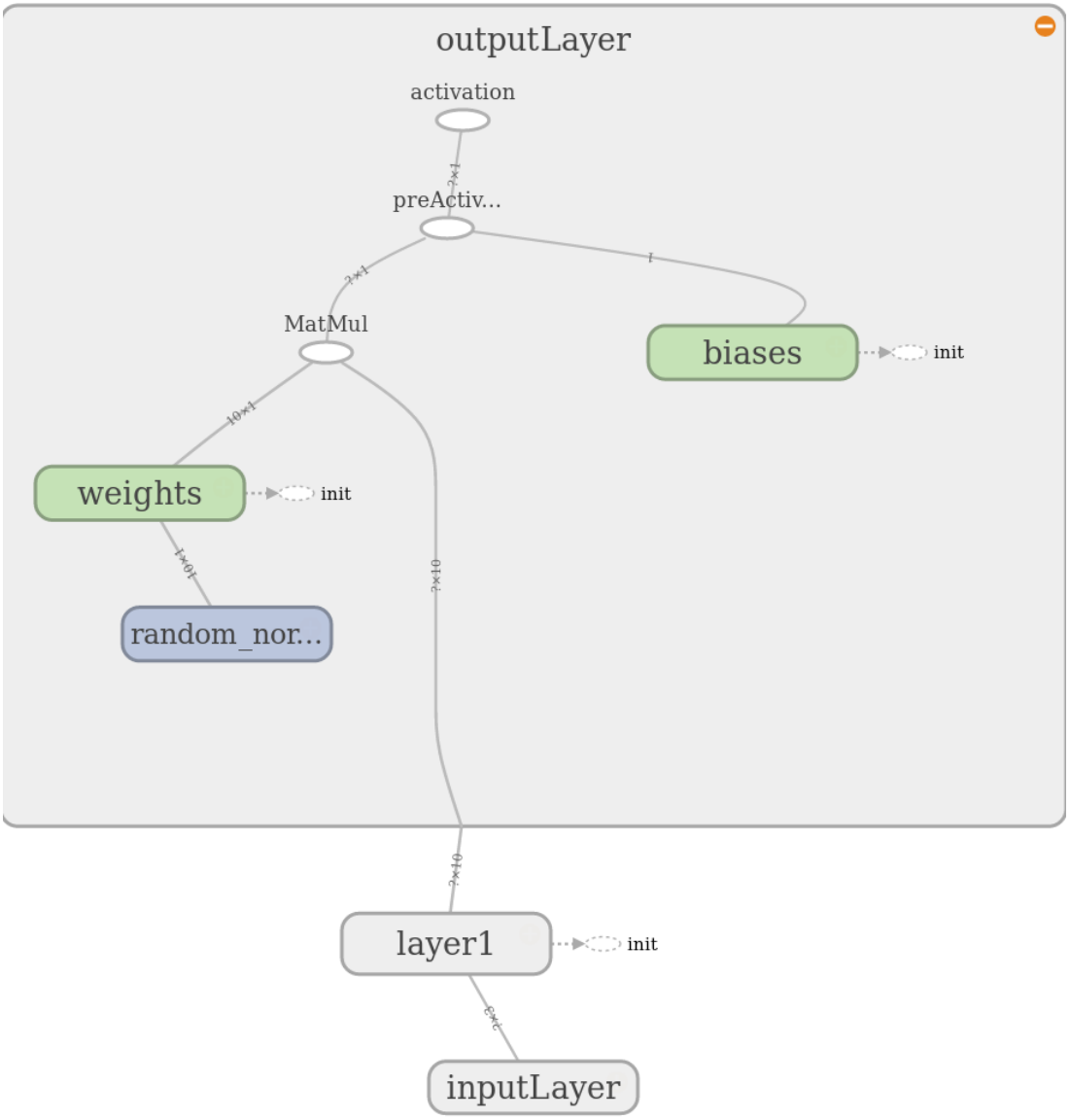


Figure 6.1: Example of a computational graph in TensorBoard of a NN consisting of an input layer, one hidden layer and an output layer. Nodes are marked by ellipses, the rectangles are namespaces containing several nodes.

Next, we will demonstrate how to train a NN with TensorFlow.

read operations. See TF API for more information.

6.2.4 Training a NN with TensorFlow

To train the NN described in the previous section, we need two additional components: a cost function and an optimizer. The cost function expresses the error of the NN fit, while the optimizer employs an algorithm to minimize this error. The most widely used cost function for regression with NNs is the L2 norm (**??**), which is available as a built-in function in TF, `tf.nn.l2_loss`. To compute the error, we need to supply known target values that is fed into a new `placeholder`,

```
outputData = tf.placeholder(tf.float32, [None,outputs],
    name='outputData')
```

The cost function is now defined as

```
trainCost = tf.nn.l2_loss( tf.subtract(actOutput,
    outputData), name='cost')
```

This adds an operation to the graph that calculates the L2 norm of the error between the NN prediction `actOutput` on the training data and the known target values `outputData`. In this work, the training data is configurations of atoms, while the target values are the atomic energies of these configurations.

A number of gradient descent-based optimizers are implemented in TensorFlow, including vanilla gradient descent, Momentum [15], Adagrad [16], Adadelta [17] and Adam [18]. To add an Adam optimizer operation to the graph, we simply write

```
optimizer = tf.train.AdamOptimizer()
trainStep = optimizer.minimize(trainCost)
```

To sum up, we have added the following to the script in the previous section (now with name scoping),

```
outputData = tf.placeholder(tf.float32, [None,outputs],
    name='outputData')

# cost function
with tf.name_scope('cost'):
    error = tf.subtract(actOutput, outputData,
        name='deviation')
    trainCost = tf.nn.l2_loss( error, name='L2norm')

# optimizer
with tf.name_scope('optimizer'):
    optimizer = tf.train.AdamOptimizer(learning_rate=0.001,
        name='Adam')
    trainStep = optimizer.minimize(trainCost,
        name='trainStep')
```

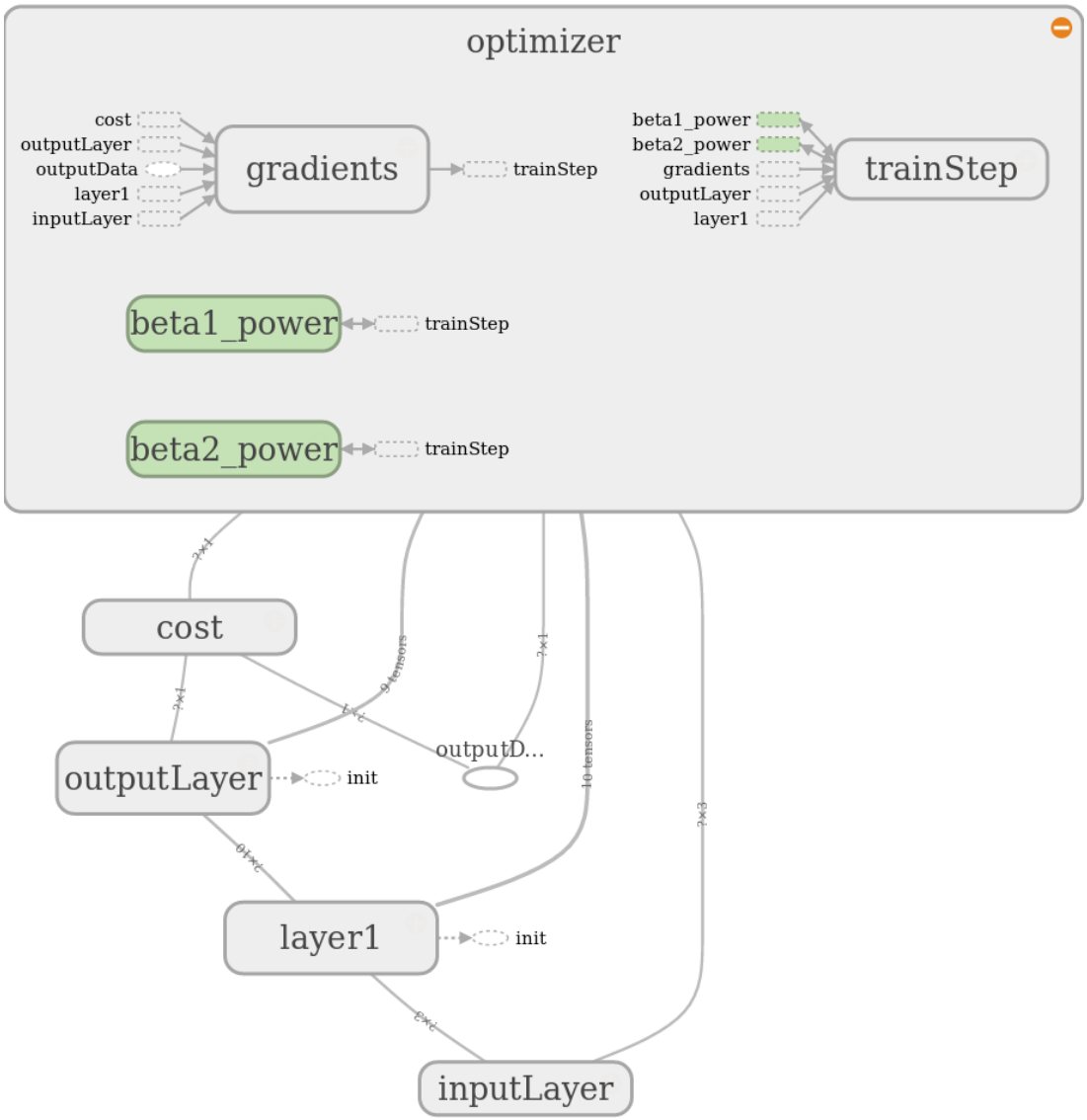


Figure 6.2: Example of a computational graph in TensorBoard for training a NN consisting of an input layer, one hidden layer and an output layer. Nodes are marked by ellipses, the rectangles are namespaces containing several nodes. A cost function describing the error of the NN fit and an optimizer that minimizes this error is included in the graph. The optimizer namespace is expanded to display its contents: `gradients` computes the gradient of all nodes in the graph, `trainStep` implements the algorithm for updating the NN parameters, while `beta1_power` and `beta2_power` are the parameters of the Adam optimizer algorithm.

The resultant graph can be seen in Figure 6.2. The optimizer namespace is expanded to display its contents. We observe that the optimizer is connected directly to all other parts of the graph, enabling it to compute the derivative of the cost function with respect to all the weights and biases. This task is performed by the `gradients` operation, that in turn sends the total gradient to `trainStep`, which implements the Adam algorithm for updating the NN parameters.

We are now ready to launch this graph in a training session.

Chapter 7

Training procedure

In this chapter we present the general workflow for constructing a high-dimensional NNP intended for a specific physical application. We describe different ways of assembling an adequate training set for the NN, which is the most crucial factor for the performance of the final potential. Some of the outlined procedures are primarily used to construct *ab inito* NNPs, but are included for reference.

Next, we discuss how to find a symmetry function set that fits the system under study. The specifics of different training teqhniques are found elsewhere. (WHERE?).

7.1 Selecting the training data

The number of possible configurations of atomic or molecular systems quickly grows very large. Consider for instance water H_2O, which is a three-body molecule. The configuration of the three atoms can be described by three internal coordinates, each with a certain range of values. If each range is divided into N equal segments, the total number of distint configurations is N^3. If we were to construct an *ab inito* potential that characterizes this system, we would have to compute the energies and forces of all these configurations. Such a brute-force approach may be feasible for a three-atom molecule, but for more complicated systems it quickly becomes practically impossible due to the computational cost.

In this work we do not use *ab inito* methods to compute energies and forces, but this problem is still highly relevant due to the high-dimensional nature of our approach. The average number of neighbours for the Stillinger-Weber Si simulations in this thesis is about $n = 6$, which is a small number compared to most other systems. This number is of great importance to the efficiency of the fitting process because it defines the dimensionality of the problem,

1. When increasing n, each symmetry function is composed of more terms. The number of terms N_{terms} scales like $N_{terms} \propto n$ for the radial symmetry

functions, but for the angular symmetry functions we have that $N_{terms} \propto n^2$.

2. A larger n leads in general to a larger configuration space to fit, thus additional symmetry functions and a larger data set is required.

3. The number of NN parameters must also be increased to obtain the same error as for a lower-dimensional problem

These observations demonstrate that performance is a critical factor whether we fit *ab inito* data or not. However, we know that only a small subset of the possible configurations of a system is physically realizable. Most of the configurations will not be visited by a system obeying Newton's laws, and are therefore not important to the dynamics. We thus need some kind of importance sampling method that adequately samples this subset. We realize that the obvious way to achieve this is to sample configurations from molecular dynamics simulations, which yields precisely the subset we are looking for.

7.1.1 Iterative molecular dynamics sampling

The method for constructing a NN PES with data sampled from MD simulations is as follows [1] [10],

1. Sample an initial data set from MD simulations.

2. Calculate the energies and optionally the forces of this initial data set with a chosen method.

3. Train a NN on the data from 1. with reference values from 2. to construct a preliminary NN potential.

4. Carry out simulations using the NNP to find new relevant structures.

5. Compute the energies (and forces) of the new structures with the method from 2. and add them to the data set

6. Train a NN on the extended data set to improve the NNP

7. Repeat 4.-6. until a stopping criteria is met

The idea is to improve the NN potential iteratively and in a self-consistent way by applying it in MD simulations to find "holes" in the data set, i.e. configurations on which the NNP predicts incorrect energies and forces. The process is different depending on whether we are constructing a NNP from *ab inito* electronic structure data or reproducing known, empirical MD potentials.

For the latter, we construct the initial data set by sampling configurations from MD simulations that is relevant for the intended applications of the NNP. Furthermore, we are extracting both coordinates and energies from the simulations, i.e. we get both the training data in step 1 and the reference data in step 2 simultaneously. To compute the energies in step 4, we simply run a pseudo-simulation on the new structures. This is easily done in LAMMPS by using the `rerun` command, which tells LAMMPS to compute the energies and forces of atomic configurations supplied from file without doing any integration.

When constructing an *ab inito* NNP however, we are immediately faced with a problem when trying to execute this process: to run MD simulations in step 1 we need the PES we are trying to develop. This is often termed as the "self-starting problem", and several procedures have been suggested for handling it.

One option is obviously to generate random configurations. This does however take us back to the original problem discussed in the beginning of this chapter: the large configuration space volume of high-dimensional systems enforces the use of importance sampling methods. It is highly unlikely that random sampling will yield the important regions of configuration space, thus vast amounts of data is needed. Nonetheless, random sampling can be applicable to small systems of 1-3 atoms where the complete configuration space can be represented without too much effort.

Another option is to use chemical intuition to select certain configurations of the system under investigation. For instance, if a specific chemical reaction is studied, one can use previous knowledge about this reaction to select configurations along the reaction path [19]. The disadvantage of this approach is that such information is not always available.

A more effective method has been presented by Raff et al. [20], where a semi-empirical potential has been deployed to initiate the sampling. If a reasonable potential is chosen and an adequate sampling method is devised (discussed below), several thousand physically relevant structures can be obtained that will constitue the initial data set. It has also been shown [20] that the final PES obtained by the iterative sampling of MD trajectories is not sensitive to the choice of the empirical surface employed in the initial step.

In this thesis we do not construct *ab inito* NNPs, thus we will not go into further details of the above methods.

7.1.2 Sampling algorithms

We have seen that MD simulations supply the data that is needed to construct a NN potential, but we also have to consider how this sampling should be performed to identify new relevant structures. NNs have a very flexible functional form which makes them able to accurately fit data sets, but they can also yield potentially large energy and force errors when evaluated for configurations that

are very different from the ones included in the training set. It is therefore of vital importance to include all structures that are relevant for the intended application of the NNP.

Sampling methods should in general be optimized to produce a reasonably uniform density of data points in those regions of configuration space that are relevant to the application of the fit [21]. If there are regions with very high point density compared to others, the NN will be trained to accurately fit these, at the cost of regions that are poorly represented. An exception to this rule is regions where the potential gradient, or forces, are large. These regions, typically configurations where two atoms are close to each other and experience strong repulsive forces, are the most difficult for the NN to fit (REFERENCE?), and should therefore have a higher density.

Initial sampling

The initial data set is supposed to function as a starting point for the construction of the NNP. In this work we use empirical MD potentials to do the initial sampling. Before the process is initiated, we must decide on the specific application for the NNP that we want to develop. This entails defining a range of temperatures, pressures etc. that the final NNP will be valid for, and sample configurations based on these.

The time interval between selecting configurations during the MD simulations must also be decided. One alternative is to simply sample points at constant time intervals. This approach will however result in a non-uniform density of data points considering that atoms spend most of their time in regions where the forces are small, at least in equilibrium MD.

To sample configuration space with something close to a uniform density, the time interval τ between sampling needs to be a function of the atomic accelerations (or forces). One such variable interval sampling algorithm, suggested by Pukrittayakamee et al. [22], is given as

$$\tau = \begin{cases} \text{floor}\left[\alpha/a_{\max}\right]\Delta t, & \text{if floor}\left[\alpha/a_{\max}\right] > 0 \\ \text{floor}\left[\alpha/a_{\max}\right]\Delta t + \Delta t, & \text{if floor}\left[\alpha/a_{\max}\right] = 0 \end{cases} \tag{7.1}$$

where a_i is the absolute value of the acceleration of atom i, Δt is the integration step size, $a_{\max}$ is the maximum acceleration of a chosen molecule and floor(x) yields the integer part of x. The constant α is system dependent and must be determined empirically. It is the value of α that determines the sampling frequency. We see from the second condition in (7.1) that configurations with an acceleration $a_i > \alpha$ yields $\tau = \Delta t$, i.e. the shortest possible sampling interval. On the other hand, if $a_i < \alpha$, the interval increases by a factor floor$(\alpha/a_{\max})$. This approach ensures that we sample all configurations where the gradient of

the PES is large, while skipping many of the low-gradient regions of configuration space.

We have chosen to use the forces $|\mathbf{F}_i| \propto a_i$ instead of the accelerations, simply because they are readily available in LAMMPS. When sampling systems with periodic boundary conditions, we have to consider the way forces are updated during a simulation. In LAMMPS, any forces attributed to ghost atoms are added back to the originals after *all* forces are computed. This is done in `verlet.cpp`,

```
if (pair_compute_flag) force->pair->compute(eflag,vflag);
// ...
if (force->newton) comm->reverse_comm();
```

The first line computes the forces on all atoms with the chosen `pair_style` potential, while the second line communicates the forces on ghost atoms back to the original atoms according to Newton's third law. This means that to employ the algorithm (7.1), we need to sample configurations *after* this reverse communication is executed, if not the forces will be incorrect. This can be achieved by extending LAMMPS with a new `compute` (described in subsection 5.0.4), as all `computes` and `fixes` are invoked after the force calculations.

We have made an additional modification to the algorithm (7.1). Pukrittayakamee et al. [22] sample `clusters` of atoms for subsequent DFT calculations of the energies of these clusters, and computes τ based on the maximum acceleration of an atom in a cluster. Our strategy is to select a few atoms, usually 1-10, and compute τ seperately for each of these atoms. When the sampling criterium is met, the coordinates of all neighbours relative to the central atom is written to file, as well as the atomic energy the central atom experiences in the presence of its neighbours. These coordinates and energies make up the training data and reference values, respectively, for a NN to interpolate. Our modified sampling algorithm is thus as follows (written in a less convoluted way than above),

$$\tau = \begin{cases} \text{floor}\left[\alpha/F_i\right], & F_i \leq \alpha \\ 1, & F_i > \alpha \\ \tau_{\max}, & \text{floor}\left[\alpha/F_i\right] > \tau_{\max} \end{cases} \tag{7.2}$$

where τ is measured in units of Δt and $\tau_{\max}$ is the maximum value of τ. F_i is the total force on atom i including ghost contributions. We have implemented this sampling algorithm as follows,

```
// ...
// loop through atoms that is to be sampled
for (auto ii : chosenAtoms) {
    i = ilist[ii];
    itype = type[i]-1;

    double F = sqrt(f[i][0]*f[i][0] + f[i][1]*f[i][1] + f[i][2]*f[i][2]);

    // update tau every maxDelay step
    if ( !(myStep % maxDelay) ) {

      // no delay if force is larger than alpha
      if (F > alpha[itype]) tau[atomCount] = 1;

      // calculate delay if force less than alpha
      else {
        int factor = floor( alpha[itype] / F );
        if (factor > maxDelay) tau[atomCount] = maxDelay;
        else                   tau[atomCount] = factor;
      }
    }

    // decide whether to sample or not based on current value of tau for atom i
    if ( (myStep % tau[atomCount] > 0) && useAlgo) continue;

    double xi = x[i][0];
    double yi = x[i][1];
    double zi = x[i][2];

    // loop through neighbours of atom i
    // write relative coordinates and atomic energy to file
    jlist = firstneigh[i];
    jnum = numneigh[i];
    for (jj = 0; jj < jnum; jj++) {
      j = jlist[jj];
      // ...
```

The vector `chosenAtoms` contains the indicies of the atoms that are to be sampled. These indicies are decided when the parameters of the employed potential is initiliazed with the input script command `pair_coeff`. The variable `maxDelay` decides how often τ is updated, and functions as a maximum value of τ. Its value does not change in the course of the simulation.

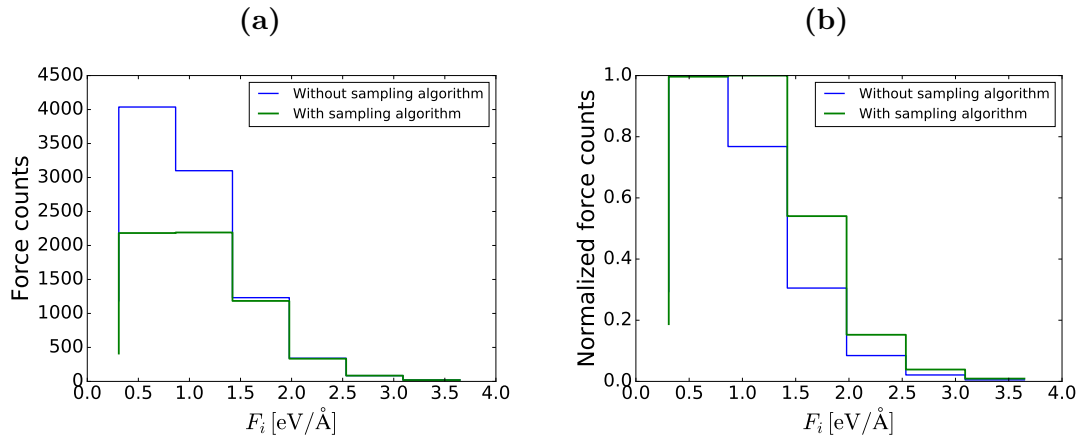


Figure 7.1: Distribution of forces for one Si atom in a quartz SiO_2 crystal with and withouth the use of the sampling algorithm (7.2). The crystal consists of 576 atoms, with an initial temperature $T = 1000\,\mathrm{K}$, run for $N = 10000$ time steps. The atoms are sampled from the microcanonical ensemble with periodic boundary conditions. Sampling algorithm parameters: $\alpha = 3.0$ and $\tau_{\max} = 10$.

Figure 7.1 displays the distribution of forces for one Si atom in a MD simulation of quartz SiO_2, with and withouth the use of the sampling algorithm (7.2). We see from the figure a) that the sampling procedure cuts the number of low-gradient configurations significantly, while keeping all the environments corresponding to large forces. The resulting distribution is however not uniform, simply because the ratio of force counts in the range $F_i = 0\,\text{Å} - 2\,\text{Å}$ to the range $F_i = 2\,\text{Å} - 4\,\text{Å}$ is too high originally.

One way to smooth the distribution further is to manually delete configurations after the simulation is finished, but then we risk removing relevant structures. The gradient of the PES is not the only way to measure the quality of the sampled set, the time aspect is also important. By selecting samples on-the-fly, we are guaranteed that the they are distributed somewhat uniformly in the time domain. Deleting structures blindly subsequent to the simulation is therefore not a good idea (I AM NOT SURE IF THIS IS TRUE, HAVE TO BE TESTED).

Another option is to sample different types of systems where the forces are more evenly distributed. Examples are non-periodic systems or systems containing only a few atoms. Also, it can be a good idea to do short simulations where the atoms start out in non-equilibrium positions to obtain configurations that are not visited by systems with a regular lattice initial state. In general, it is not required that the data used to train the NNs are the results of physically plausible simulations, the only requirement is that it should cover all possible configurations that the NNP have the possibility of experiencing in the course of its application. WRITE MORE / DO TESTS ON THIS???

Iterative sampling

The initial data set is usually not sufficient to construct a stable neural network PES. It is difficult to identify all the relevant structures that the NN have to learn to perform well, and since there is always a small fitting error, the NNP will generate different trajectories than the PES that has been used to train it. This is solved by sampling new relevant structures from simulations using the preliminary NNP trained on the initial data set. In this way, new configurations will "naturally present themselves", and the NNP will iteratively improve in a self-consistent way.

The obvious way to identify new structures not already in the initial data set is to evalate both the NNP and the PES we are trying to fit for each structure in an NNP MD simulation and compare the energies (and eventually the forces). If the energy deviation is above a certain treshold, the structure is added to the training set. The simulations should be carried out under the conditions of the intended application of the NNP to ensure that the relevant parts of configuration space are sampled. After a certain amount of new data is added, the NN is trained on this extended data set and thus improved. Then, new simulations are carried out with the improved NNP and so on, according to the iterative process item 7 described in the beginning of this chapter.

This approach is easily feasible when the target PES is an empirical potential, but too computationally expensive in the case where electronic structure methods are employed. Raff et al. [20] have developed a sampling algorithm called *modified novelty sampling* that does not require any quantum mechanical calculations of the energies of new structures. They describe the database as a set of vectors $\{\mathbf{q}\}$, where vector $\mathbf{q}_i$ contains the internal coordinates of the i_{th} cluster of atoms. New configurations $\mathbf{q}_n$ are generated using the NNP trained on this data base and the 2-norm difference $\mathbf{d}$ between vectors $\mathbf{q}_n$ and $\mathbf{q}_i$ are computed,

$$\mathbf{d} = |\mathbf{q}_i - \mathbf{q}_n| = [(\mathbf{q}_i - \mathbf{q}_n) \cdot (\mathbf{q}_i - \mathbf{q}_n)]^{1/2} \tag{7.3}$$

for all clusters i. The difference vector is characterized by its minimum value $d_{\text{min}} = \min(\mathbf{d})$ and the mean seperation distance $d_{\text{mean}} = \langle \mathbf{d} \rangle$. The basic idea is to add configuration point $\mathbf{q}_n$ to the database with high probability if d_{min} or d_{mean} is large, but low probability if both d_{min} and d_{mean} are small. The energy output of the NNP are also included in the scheme, along with a more involved selection algorithm. As this thesis only deals with empirical target potentials, the details will not be further discussed here.

In this work we use the Behler symmetry functions to transform atomic cartesian coordinates to vectors of symmetry values. These functions act as descriptors of chemical environments, and should therefore be excellent candidates to identify relevant structures during MD simulations without computing additional target energies. Behler has suggested two methods [1] where these symmetry functions

are employed to improve the NNP. The first and simplest method identifies structures that fall outside of the range of input values spanned by the training data. This is achieved by comparing the symmetry function vector of each atom during the MD simulation with the minimum and maximum values of each symmetry function in the database for the respective element. When such configurations are encountered, a warning can be issued and the structure sampled.

The other method developed by Behler is called the *multiple-NN method* and are used to identify structures that are within the range of existing symmetry values, but located in parts of configuration space that are poorly represented in the training set. The idea is to use the high flexibility and reasonably fast training and evaluation of NNs to validate the potential. The procedure is as follows,

1. Fit several NNs with different architectures to the same training set. All the NNs should be trained to the same accuracy.

2. Generate a large number of trajectories and corresponding energies using one of these NNPs in MD simulations.

3. Recalculate the energies of the obtained trajectories using the other NNPs

4. Structures for which the energy disagreement between the different NNPs is significant should be added to the training set.

This method takes advantage of the poor extrapolation abilites of NNs: the energy predictions of the different NNPs will be more or less random on structures that are far away from the training points. Thus, missing structures can be identified by comparing the energy predictions of the different NNPs on the genereated trajectories. The procedure also provides a criterion for determining the convergence of the PES without requiring the repeated computation of dynamic properties of the system under study. When, after several iterations of the multiple-NN method, no problematic structures are found, the NNP should be reliable and ready for use.

Summary

In this section we have presented the workflow for constructing a NNP, including several procedures to assemble the training set for the NN, which depends on how the target energies are computed. In this work, we know the PES we are trying to fit, which are well-known empirical MD potentials. Therefore, the NNP is validated simply by comparing its predictions to the target PES. On the other hand, if an *ab inito* NNP is to be constructed, the underlying target PES is not known, and computing energies and forces of eventual new structures are expensive. Two methods circumventing this problem has been described, but they are not used in this thesis.

Still, there are several factors to consider when constructing the training set, irrelevant of how the target PES looks like. We have implemented and employed the sampling algorithm (7.2) in this thesis to obtain a more uniform distribution of forces in the data set. The thermodynamical properties of the MD simulations that we sample configurations from have also been carefully considered, described in more detail in chapter 9.

7.2 Constructing the symmetry function sets

The choice of symmetry functions is important to achieve a good fit to the reference data. They need to be able to sufficiently distinct inequivalent structures to avoid fitting contradictory data. If this distinction is not possible, different reference structures with different energies and forces give rise to the same symmetry function values, which results in poor fits.

Although a NN potential is general and unbiased because it has no pre-defined functional form, the construction of the symmetry function sets is to some extent empirical. The functional form of the NN is adjusted "automatically" when trained, while the symmetry function set has to be constructed manually. This observation does however not change the fact that the high-dimensional neural network approach to constructing potentials used in this work is general, and can in principle be applied to any system. The training data and symmetry function sets have to be customized to each system, but the overarching procedure is the same.

There are a number of analyses that can be carried out to investigate if we have an adequate symmetry function set,

- The symmetry functions should cover the configuration space evenly. A reasonable initial choice for the radial symmetry functions is to use an equidistant set to ensure that the whole range of distances in the reference set are probed. Also, it is a good idea to have some knowledge about the radial and angular distribution of atoms for the system we are looking at. This information can be extracted from MD runs with semi-empirical potentials or from experiments and will enable us to identify high-density regions of configuration space that should be covered by a larger number of symmetry functions than low-density regions. HMMMM, THIS IS VERY BIASED?????????? BEHLER SAYS "IN AND UNBIASED WAY".

- The range of values for each symmetry function should be analyzed. If the difference between the largest and smallest function value is too small, it will not be able to identify distinct structures properly. Symmetry functions which have a value of zero for all configurations should be avoided, as these will not contribute to the description of the chemical environment.

- SEARCH FOR CONTRADICTORY DATA? THEN I MUST HAVE ACCESS TO THE FORCES? BUT IS HE TALKING ABOUT DIFFERENT ATOM TYPES? MAYBE CHOOSE A NEIGHBOUR LIST (ATOM), FIND ΔG AND ΔF TO ALL OTHER LISTS AND PLOT?

- There is a trade-off between the ability of the symmetry function set to cover the configuration space and the efficiency of the calculation of the NN energy output. The number of symmetry functions can be kept as low as possible by investigating the correlations between the values of two different symmetry functions for the whole reference set. If there is a large correlation, the two symmetry functions are almost linearly dependent, and we have a redundancy of information. Consequentially, one of them should be removed, or the parameters adjusted to decrease the correlation. WRITE ABOUT CORRELATIONS AND TRAINING.

These tests provide a framework for constructing suitable symmetry function sets. We note that they all depend on the composition of the available data set, including its size. When the iterative approach to constructing data sets are used, the size will increase step by step, and we are required to add more symmetry functions as we go.

Chapter 8

Validation

8.1 Time usage

The workflow when training a NN and using it in MD is as follows:

1. Generate training data

2. Train a NN to fit the data

3. Use the analytical expression for the trained NN as a potential in MD

The main application programming interface (API) for TensorFlow (TF) is Python. The NN is therefore trained using Python scripts, while C++ is utilized to run the MD simulations for speed purposes. Thus we need a way to transfer the NN from Python to C++.

There are built-in functions in TF that automatically saves computational graphs (architecture of network plus nodes for training) and variables (weights and activation functions) as binary files. These files can then be read and the NN evaluated using the TF C++ API. This API is however quite under-developed and the documentation is very sparse. Another alternative is to manually save the weights and biases as a text file, along with information on which activation function(s) that has been used. This file file can then be used to reconstruct the NN in the same way that it is made in Python: By representing the connection weights between layers as matrices and the biases as vectors. The activation functions must be coded manually. A linear algebra library should be utilized for speed and simplicity; we have chosen to use Armadillo.

To find out which alternative is the most efficient for evaluating a NN, we compare the time usage in three cases: with TF in Python, with TF in C++ and with Armadillo. The results are shown in Figure 8.1.

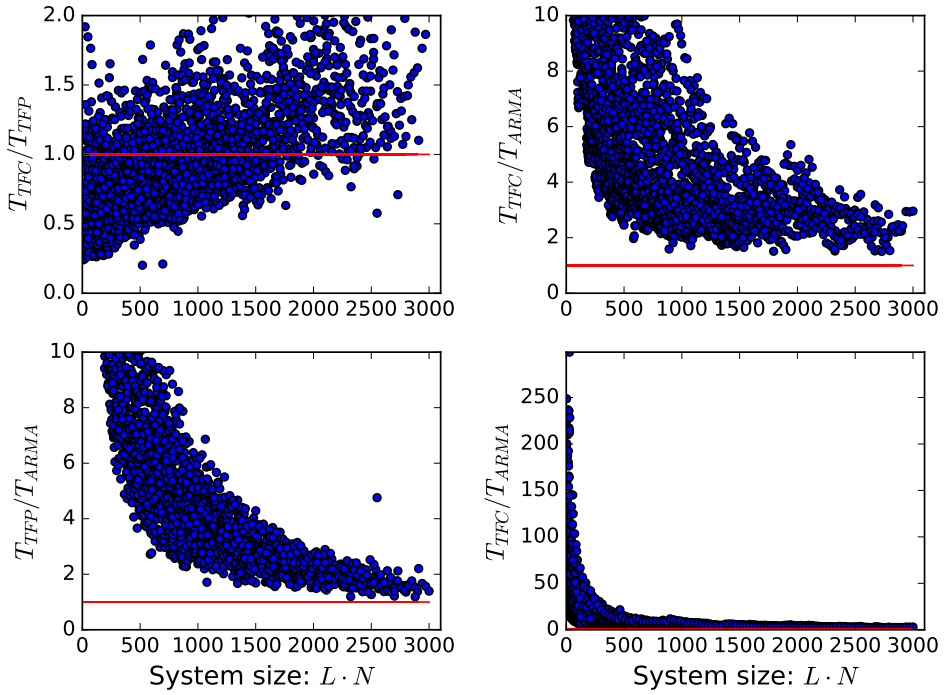


Figure 8.1: Scatter plot of CPU time when evaluating untrained NNs with random weights and sigmoid activation functions using the TF Python API (T_{TFP}), the TF C++ API (T_{TFC}) and Armadillo (T_{ARMA}). L is the number of layers, N is the number of nodes in each hidden layer. All the NNs have one input and one output. The time has been found by averaging over 50 evaluations for each NN architecture. T_{TFC}/T_{ARMA} is also shown in an uncut version to demonstrate how large the time difference is for very small NNs.

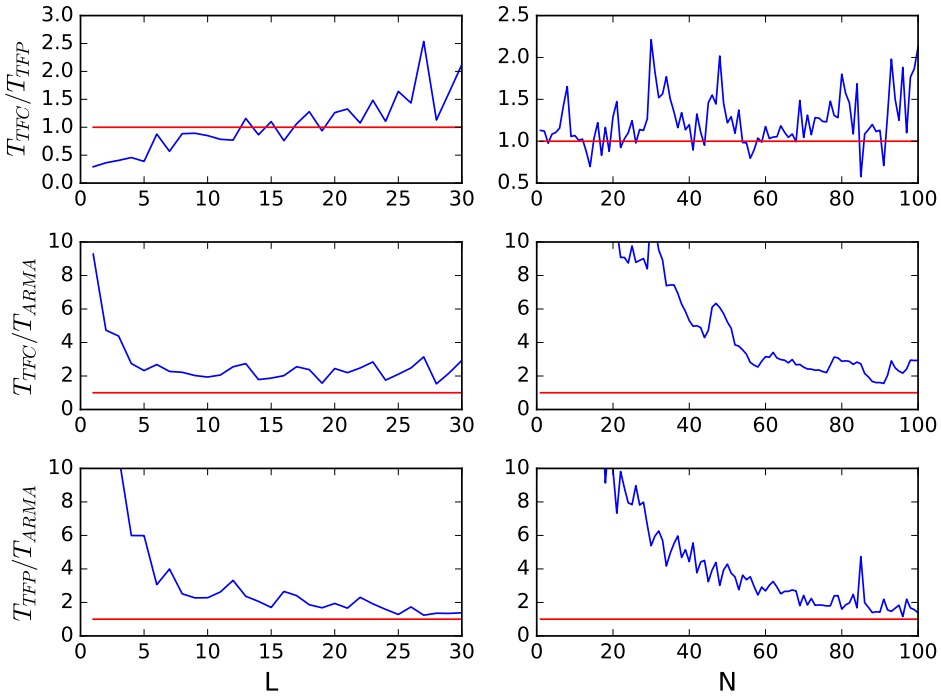


Figure 8.2: Left column: CPU time comparison for $N = 100$. Right column: CPU time comparison for $L = 30$.

As we can see, evaluation of an NN with the built-in TF methods are notably slower than with Armadillo for small NNs, but seems to converge to 1 for large networks. The convergence rate is however surpisingly low. One reason for this can be that the *Session* environment in TF has lot of overhead. However, the main computational strength of TF resides in GPUs, which will be tested out later on.

8.2 Training Lennard-Jones potential

To verify that the implementation of the NN and the backpropagation algorithm with Armadillo in our MD code is correct, we train a NN to reproduce the shifted Lennard Jones potential (REF TIL LIKNING). The error of the trained network on the training data interval is evaluated both in Python and C++. The two error plots are shown in Figure 8.3.

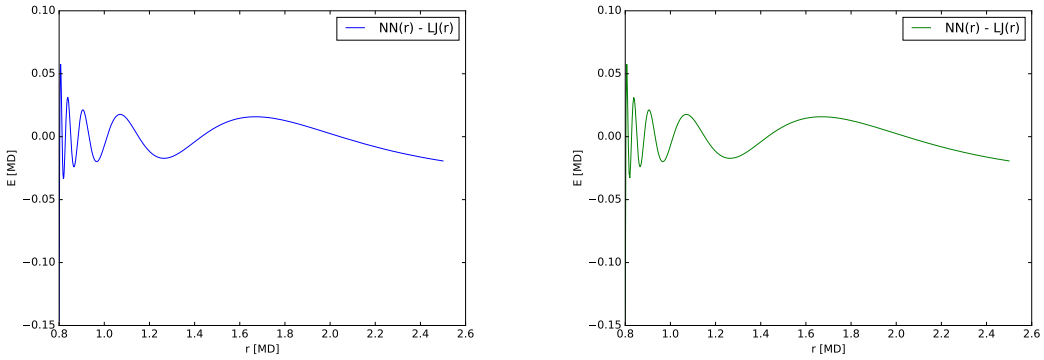


Figure 8.3: Error of a network trained to reproduce the shifted Lennard-Jones potential. The NN is trained in Python and the error on the training data interval is calculated, shown on the left. The NN is also written to file for evaluation in C++, shown on the right.

We observe that the error has excactly the same shape and value on the whole interval, we can therefore confirm that the NN is implemented correctly.

To test the implementation of the backpropagation algorithm, the same NN is differentiated in C++ and compared to the analytical derivative of the L-J potential. The gradient of a NN is defined as the derivative of one of the outputs with respect to the input(s). There are two differences when using backpropagation to find the gradient of a network compared to the use of backpropagation in training:

1. The derivative of the output neuron's activation function is backpropagated instead of the output error

2. We backpropagate all the way down to the input layer, during training we stop at the first hidden layer

As mentioned before, the backpropagation algorithm is simply an application of the chain rule: To obtain the derivative of the output node with respect to the input nodes we need to differentiate through all the nodes in-between because we have fully connected layers. The gradient error is plotted in Figure 8.4.

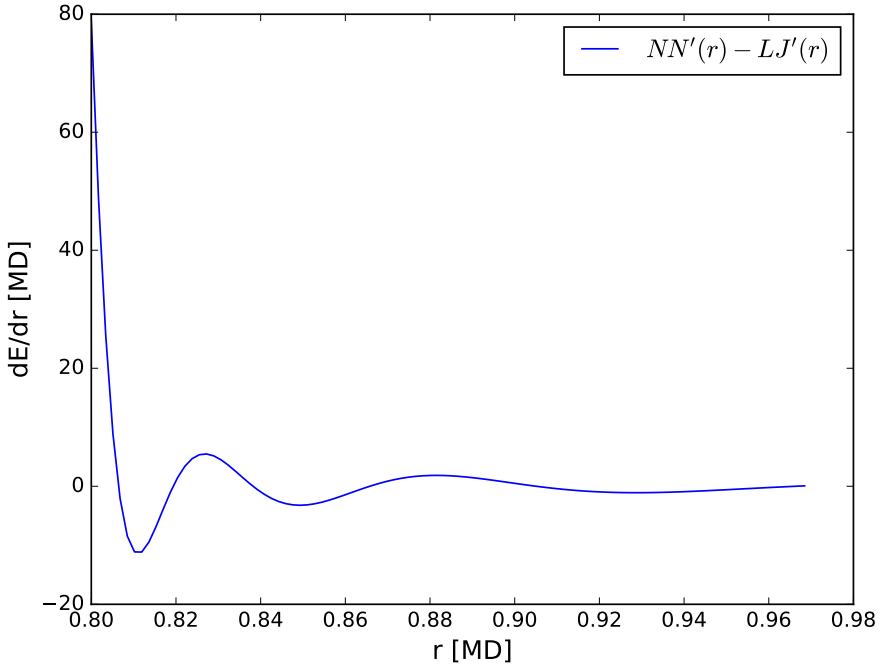
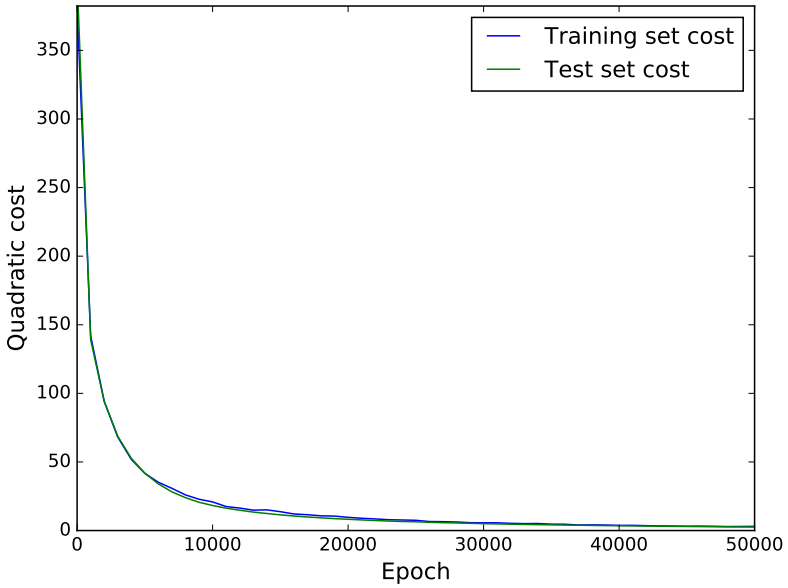


Figure 8.4: Error of the gradient of a NN trained to reproduce the shifted Lennard-Jones potential. The NN is trained in Python and evaluated and differentiated in C++ using the backpropagation algorithm. The result is compared to the analytical derivative of the LJ potential. Only a part of the training interval is shown, the graph is essentially flat after this point

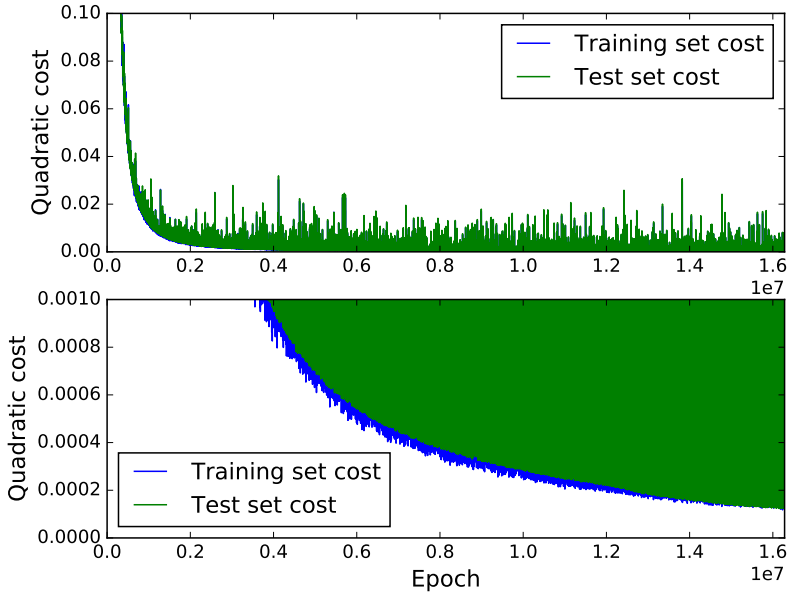
Again we get an oscillating error over the training interval. The oscillations near the left end of the interval is much larger than for the energy, but they get damped more quickly. The large error near the left end of the interval is a natural consequence of the shape of the LJ potential for these values: the $1/r^{13}$ term dominates with its very steep slope, the NN is not fed a sufficient amount of data in this region to excactly reproduce the rapid changing derivative. This tells us that it is important to train the NN on a larger data interval than needed in simulations or possibly feed the NN more data in critical regions during training.

8.2.1 Many-neighbour Lennard-Jones

A NN can also be trained to obtain the energy of several neighbouring atoms at once. The training procedure is excactly the same, but now we train the network so that the output is the sum of N LJ-potentials for N different distances r. The NN will have N inputs and 1 output. To get an error of the same order as for 1 neighbour, we need to have a lot more nodes. The result of a training session with $N = 20$ inputs is shown in figures Figure 8.5. (Training session 02.12-19.36.50)



(a) Zoomed out



(b) Zoomed in

Figure 8.5: Quadratic cost (**??**) of training set and test set for NN trained to yield the sum of the shifted LJ-potential of 20 neighbouring atoms. The NN has 20 inputs, one hidden layer with 200 neurons and one output node. All nodes in the hidden layer have sigmoid activation functions (3.15). Figure 8.5a shows the cost for the first 50 000 epochs, while Figure 8.5b displays the cost for all epochs, but on a smaller scale.

The cost rapidly decreases during the first 10 000 epochs, then it starts to flatten out. This behaviour seems to be general when doing regression with ANNs. There is no sign of overfitting: we do not observe an increase in the test cost relative to the training cost. This is (PROBABLY) due to the fact that our training set and test set are random numbers, i.e. there will be no specifics patterns or configurations present in the test set that is not covered by the training set. When we start doing training on data produced by MD trajectories or QM calculations, the situation will different.

The NN above outputs the total energy on an atom that is surrounded by N neighbouring atoms, but it does not provide the forces. These must be obtained by calculating the analytical derivative of the NN with backpropagation as described above for each time step. An alternative approach that may speed up the MD simulation is to have the NN also output the total force on the central atom. This can be done giving the cartesian coordinates and distance of N atoms as input to the NN so that each input vector looks like

$$(x_1, y_1, z_1, r_1, x_2, y_2, z_2, r_2, \ldots, x_N, y_N, z_N, r_N) \tag{8.1}$$

which yields an output vector

$$(F_x^{TOT}, F_y^{TOT}, F_z^{TOT}, E^{TOT}) \tag{8.2}$$

We have to make sure that the input coordinates are in the range $x, y, z \in [-r_{cut}, r_{cut}]$ to cover the whole configuration space. The distances however, are still in the range $r \in [a, r_{cut}]$ where a is the minimum distance that two atoms can have in a given MD simulation (see function *energyAndForceCoordinates* in *generateData*). As the NN now have 4 inputs per neighbour and 4 outputs, we need to have more nodes to get a sufficiently low error. We run a training session with 5 neighbours (20 inputs) (session 13.12-12.59.51). The cost has the same shape as before, but the error is much larger to begin with and also converges slower (SKAL JEG GIDDE HA ENDA ET ERROR PLOT HER?).

One way to test if this works is to move only one of the input atoms while freezing the others and see if we reproduce the LJ-potential form. We load the above training session and test with the command python approximateFunction2.py –load TrainingData/13.12-12.59.51/Checkpoints/ckpt-999 –test with this code:

```
if testFlag:

  # pick an input vector
  coordinates = xTrain[0]
  coordinates = coordinates.reshape([1,self.inputs])
  neighbours = self.inputs/4
  xNN = np.zeros(neighbours)
  yNN = np.zeros(neighbours)
  zNN = np.zeros(neighbours)
```

```
    rNN = np.zeros(neighbours)
    # extract coordinates and distances
    for i in range(neighbours):
        xNN[i] = coordinates[0,i*4]
        yNN[i] = coordinates[0,i*4 + 1]
        zNN[i] = coordinates[0,i*4 + 2]
        rNN[i] = coordinates[0,i*4 + 3]

    # vary coordinates of only one atom and see
    # if the resulting potential is similar to LJ
    N = 500
    r = np.linspace(0.8, 2.5, N)
    energyNN = []
    energyLJ = []
    forceNN = []
    forceLJ = []
    xyz = np.zeros(3)
    for i in range(N):
        r2 = r[i]**2
        xyz[0] = np.random.uniform(0, r2)
        xyz[1] = np.random.uniform(0, r2-xyz[0])
        xyz[2] = r2 - xyz[0] - xyz[1]
        #np.random.shuffle(xyz)
        x = np.sqrt(xyz[0])# * np.random.choice([-1,1])
        y = np.sqrt(xyz[1])# * np.random.choice([-1,1])
        z = np.sqrt(xyz[2])# * np.random.choice([-1,1])
        coordinates[0][0] = x; coordinates[0][1] = y;
            coordinates[0][2] = z
        coordinates[0][3] = r[i]
        energyAndForce = sess.run(prediction,
            feed_dict={self.x: coordinates})
        energyNN.append(energyAndForce[0][3])
        rNN[0] = r[i]
        energyLJ.append(np.sum(self.function(rNN)))
        forceNN.append(energyAndForce[0][0])
        xNN[0] = x
        forceLJ.append(np.sum(self.functionDerivative(rNN)*xNN/rNN))

    # convert to arrays
    energyNN = np.array(energyNN); energyLJ = np.array(energyLJ)
    forceNN = np.array(forceNN); forceLJ = np.array(forceLJ)

    # plot error
    plt.plot(r, energyNN - energyLJ)
    plt.show()
    plt.plot(r, forceNN - forceLJ)
    plt.show()
    print 'Cost: ', (np.sum((energyNN - energyLJ)**2 + (forceNN
        - forceLJ)**2))/N

    # see if the energy is zero when all neighbours is at
```

```
    cutoff distance
inputz = np.array([1.87, 1.32, 1.006,
    2.5]*neighbours).reshape([1,self.inputs])
r = np.array([2.5]*neighbours)
energyLJ = sum(self.function(r))
ef = sess.run(prediction, feed_dict={self.x: inputz})

print 'Approximate energy: ', ef[0,3]
print 'LJ energy: ', energyLJ

numberOfEpochs = 0
```

which produces the following output:

```
Model restored
Cost:  0.132622806039
NN energy at cutoff:  0.125104
LJ energy at cutoff:  0.0
Time elapsed: 1.2e-05
```

We see that the NN energy at cutoff is not zero, because of the cost (IS THIS CORRECT? THE COST AT CUTOFF IS PRACTICALLY ZERO). The error of the NN energy and forces over the training data interval is shown in

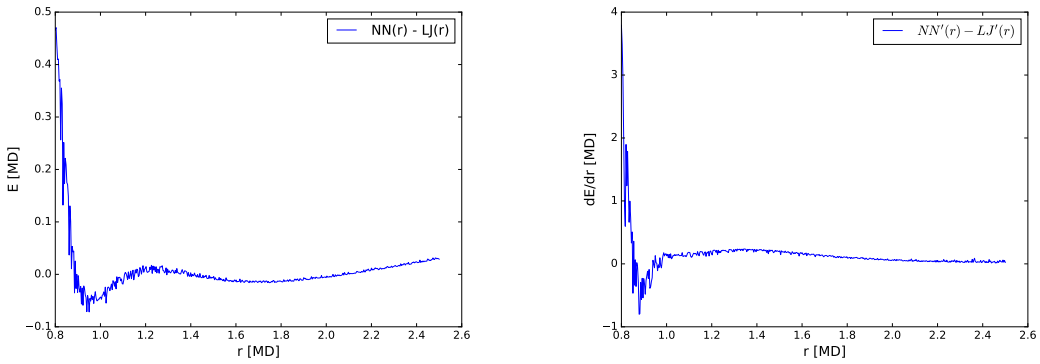


Figure 8.6: Error of a network trained to yield the total energy and force on a central atom from 5 neighbouring atoms. The energy contribution from each neighbour is a shifted LJ potential. Trained for 1e6 epochs, 100 nodes in 2 layers. The energy error is shown on the left, force error on the right.

The error oscillates in exactly the same way as for the 1-input NN above, suggesting that the training is done correctly. However, it is very important that the energies and forces in an MD simulation is consistent, and this is not ensured here, where the two are obtained in a somewhat independent way. Any incosistency between the energies and forces will magnify for each time step, resulting

in inaccurate statistical properties like temperature and pressure. Therefore it is a better idea to have the NN only output the total energy, and find the forces by analytically differentiating the NN for each time step. 4 Å

Part III

Results and discussion

Chapter 9

NN potential for Si

Construct an NN potential for Si that works in a certain temperature range (only solid, or liquid too?) Compute thermodynamic properties and things like radial distrubution function and compare both with experiments and with results with Stillinger-Weber.

Chapter 10

NN potential for SiO2

Construct an NN potential for SiO2 that works in a certain temperature range (only solid, or liquid too?). Can use various potentials as the black box, e.g. Vashishta or ReaxFF. ReaxFF would be great, because then we would maybe get a speed-up.

Appendices

Appendix A

Symmetry functions derivatives

We list here the derivatives of the symmetry functions used for training neural networks. These have to be known to compute forces for MD simulations. In some cases we only show the deriviative with respect to $R = \sqrt{x^2 + y^2 + z^2}$. Each component α can be calculated by

$$\frac{\partial R}{\partial \alpha} = \frac{\alpha}{R} \tag{A.1}$$

The following notation applies in the following,

$$R_{ij} = \left(x_{ij}^2 + y_{ij}^2 + z_{ij}^2\right)^{1/2} = \left((x_j - x_i)^2 + (y_j - y_i)^2 + (z_j - z_i)^2\right)^{1/2} \tag{A.2}$$

Cutoff function (4.2):

$$\frac{\partial f_c(R)}{\partial R} = -\frac{1}{2}\frac{\pi}{R_c}\sin\left(\frac{\pi R}{R_c}\right) = M(R) \tag{A.3}$$

This expression is valid for all combinations of indicies. The derivatives w.r.t. to individual coordinates are

$$\frac{\partial f_c(R_{ij})}{\partial x_{ij}} = M(R_{ij})\frac{x_{ij}}{R_{ij}} \tag{A.4}$$

and

$$\frac{\partial f_c(R_{ik})}{\partial x_{ik}} = M(R_{ik})\frac{x_{ik}}{R_{ik}} \tag{A.5}$$

while

$$\frac{\partial f_c(R_{ij})}{\partial x_{ik}} = \frac{\partial f_c(R_{ik})}{\partial x_{ij}} = 0 \tag{A.6}$$

because atom i is always in origo, which means we can do the substitutions $x_{ij} \to x_j$ and $x_{ik} \to x_k$. SHOULD I DO THIS CONSEQUENTLY BELOW????. Also,

$$\frac{\partial f_c(R_{jk})}{\partial x_{ij}} = -M(R_{jk})\frac{x_{jk}}{R_{jk}} \tag{A.7}$$

and

$$\frac{\partial f_c(R_{jk})}{\partial x_{ij}} = M(R_{jk})\frac{x_{jk}}{R_{jk}} \tag{A.8}$$

The derivative of each term in G_i^1 (4.3) is simply (A.3).
G_i^2 (4.4):

$$\frac{\partial G_i^2}{R_{ij}} = \exp(-\eta(R_{ij} - R_s)^2)\left[2\eta(R_s - R_{ij}) + \frac{\partial f_c}{\partial R_{ij}}\right] \tag{A.9}$$

G_i^3 has not been used in this theses.
G_i^4 (4.7):

$$G_i^4 = F_1(\theta)F_2(R_{ij}, R_{ik}, R_{jk})F_3(R_{ij}, R_{ij}, R_{jk}) \tag{A.10}$$

where

$$F_1(\theta) = 2^{1-\zeta}(1 + \lambda\cos\theta_{ijk})^\zeta \tag{A.11}$$
$$F_2(R_{ij}, R_{ik}, R_{jk}) = \exp[-\eta(R_{ij}^2 + R_{ik}^2 + R_{jk}^2)] \tag{A.12}$$
$$F_3(R_{ij}, R_{ij}, R_{jk}) = f_c(R_{ij})f_c(R_{ik})f_c(R_{jk}) \tag{A.13}$$

where

$$\cos\theta = \frac{x_{ij}x_{ik} + y_{ij}y_{ik} + z_{ij}z_{ik}}{R_{ij}R_{ik}} \tag{A.14}$$

Using the product rule:

$$\frac{\partial G_i^4}{\partial x_{ij}} = \frac{\partial F_1}{x_{ij}}F_2F_3 + F_1\left(\frac{\partial F_2}{\partial x_{ij}}F_3 + F_2\frac{\partial F_3}{\partial x_{ij}}\right) \tag{A.15}$$
$$\frac{\partial F_1}{x_{ij}}F_2F_3 + F_1\frac{\partial F_2}{\partial x_{ij}}F_3 + F_1F_2\frac{\partial F_3}{\partial x_{ij}} \tag{A.16}$$

We have

$$\frac{\partial F_1}{\partial x_{ij}} = \frac{\partial F_1}{\partial\cos\theta}\frac{\partial\cos\theta}{\partial x_{ij}} \tag{A.17}$$

where

$$\frac{\partial F_1}{\partial\cos\theta} = \lambda\zeta(1 + \cos\theta)^{\zeta-1} = K \tag{A.18}$$

and

$$\frac{\partial\cos\theta}{\partial x_{ij}} = \frac{x_{ik}}{r_{ij}r_{ik}} - \frac{x_{ij}\cos\theta}{r_{ij}^2} \tag{A.19}$$

To get the corresponding expression for x_{ik}, simply substitute $x_{ij} \leftrightarrow x_{ik}$ and $r_{ij} \leftrightarrow r_{ik}$. Further,

$$\frac{\partial F_2}{\partial x_{ij}} = \frac{\partial F_2}{\partial r_{ij}}\frac{\partial r_{ij}}{\partial x_{ij}} = (-2\eta r_{ij}\frac{x_{ij}}{r_{ij}} - 2\eta r_{jk}\frac{x_{jk}}{r_{jk}})\exp[-\eta(R_{ij}^2 + R_{ik}^2 + R_{jk}^2)] \tag{A.20}$$
$$= -2\eta F_2 x_{ij} - 2\eta F_2 x_{jk} \tag{A.21}$$

The corresponding expression for x_{ik} is

$$\frac{\partial F_2}{\partial x_{ik}} = -2\eta F_2 x_{ik} + 2\eta F_2 x_{jk} \tag{A.22}$$

Lastly,

$$\frac{\partial F_3}{\partial x_{ij}} = \frac{\partial f_c(r_{ij})}{\partial x_{ij}} f_c(r_{ik}) f_c(r_{jk}) + f_c(r_{ij}) f_c(r_{ik}) \frac{\partial f_c(r_{jk})}{\partial x_{ij}} \tag{A.23}$$

$$= f_c(r_{ik}) \left(M(r_{ij}) \frac{x_{ij}}{r_{ij}} f_c(r_{jk}) - f_c(r_{ij}) M(r_{jk}) \frac{x_{jk}}{r_{jk}} \right) \tag{A.24}$$

The corresponding expression for x_{ik} is

$$\frac{\partial F_3}{\partial x_{ik}} = f_c(r_{ij}) \left(\frac{\partial f_c(r_{ik})}{\partial x_{ik}} f_c(r_{jk}) + f_c(r_{ik}) \frac{\partial f_c(r_{jk})}{\partial x_{ik}} \right) \tag{A.25}$$

$$= f_c(r_{ij}) \left(M(r_{ik}) \frac{x_{ik}}{r_{ik}} f_c(r_{jk}) + f_c(r_{ik}) M(r_{jk}) \frac{x_{jk}}{r_{jk}} \right) \tag{A.26}$$

$$\tag{A.27}$$

Then, taking (A.16) into account:

$$\frac{\partial G_i^4}{\partial x_{ij}} = \left(\frac{x_{ik}}{r_{ij} r_{ik}} - \frac{x_{ij} \cos\theta}{r_{ij}^2} \right) K F_2 F_3 - \tag{A.28}$$

$$2\eta F_1 F_2 F_3 x_{jk} - 2\eta F_1 F_2 F_3 x_{ij} + \tag{A.29}$$

$$F_1 F_2 f_c(r_{ik}) \left(M(r_{ij}) \frac{x_{ij}}{r_{ij}} f_c(r_{jk}) + f_c(r_{ij}) M(r_{jk}) \frac{x_{jk}}{r_{jk}} \right) \tag{A.30}$$

and for x_{ik}:

$$\frac{\partial G_i^4}{\partial x_{ik}} = \left(\frac{x_{ij}}{r_{ij} r_{ik}} - \frac{x_{ik} \cos\theta}{r_{ik}^2} \right) K F_2 F_3 + \tag{A.31}$$

$$2\eta F_1 F_2 F_3 x_{jk} - 2\eta F_1 F_2 F_3 x_{ik} + \tag{A.32}$$

$$F_1 F_2 f_c(r_{ij}) \left(M(r_{ik}) \frac{x_{ik}}{r_{ik}} f_c(r_{jk}) - f_c(r_{ik}) M(r_{jk}) \frac{x_{jk}}{r_{jk}} \right) \tag{A.33}$$

or in terms of x_{ij} and x_{ik} and x_{jk}:

$$\frac{\partial G_i^4}{\partial x_{ij}} = x_{ij} \left(- \frac{\cos\theta}{r_{ij}^2} K F_2 F_3 - 2\eta F_1 F_2 F_3 + \right. \tag{A.34}$$

$$\left. F_1 F_2 M(r_{ij}) f_c(r_{ik}) f_c(r_{jk}) \frac{1}{r_{ij}} \right) + \tag{A.35}$$

$$x_{ik} \frac{K F_2 F_3}{r_{ij} r_{ik}} - x_{jk} \Big(F_1 F_2 M(r_{jk}) f_c(r_{ik}) f_c(r_{ij}) + 2\eta F_1 F_2 F_3 \Big) \frac{1}{r_{jk}} \tag{A.36}$$

and

$$\frac{\partial G_i^4}{\partial x_{ik}} = \frac{x_{ik}}{r_{ik}}\Bigg(- \frac{\cos\theta}{r_{ik}^2} K F_2 F_3 - 2\eta F_1 F_2 F_3 + \tag{A.37}$$

$$F_1 F_2 M(r_{ik}) f_c(r_{ij}) f_c(r_{jk}) \frac{1}{r_{ik}}\Bigg) + \tag{A.38}$$

$$x_{ij}\frac{K F_2 F_3}{r_{ij} r_{ik}} + x_{jk}\Big(F_1 F_2 M(r_{jk}) f_c(r_{ij}) f_c(r_{ik}) - 2\eta F_1 F_2 F_3\Big)\frac{1}{r_{jk}} \tag{A.39}$$

The derivative of G_i^5 (4.9) is found in a similar way,

$$G_i^5 = F_1(\theta) F_2(R_{ij}, R_{ik}) F_3(R_{ij}, R_{ij}) \tag{A.40}$$

where

$$F_1(\theta) = 2^{1-\zeta}(1 + \lambda\cos\theta_{ijk})^\zeta \tag{A.41}$$

$$F_2(R_{ij}, R_{ik}) = \exp[-\eta(R_{ij}^2 + R_{ik}^2)] \tag{A.42}$$

$$F_3(R_{ij}, R_{ij}) = f_c(R_{ij}) f_c(R_{ik}) \tag{A.43}$$

The derivative of F_1 is the same, while for F_2 we obtain

$$\frac{\partial F_2}{\partial x_{ij}} = -2\eta F_2 x_{ij} \tag{A.44}$$

and

$$\frac{\partial F_2}{\partial x_{ik}} = -2\eta F_2 x_{ik} \tag{A.45}$$

For F_3,

$$\frac{\partial F_3}{\partial x_{ij}} = \frac{\partial f_c(r_{ij})}{\partial x_{ij}}\frac{x_{ij}}{r_{ij}} f_c(r_{ik}) = M(r_{ij}) f_c(r_{ik})\frac{x_{ij}}{r_{ij}} \tag{A.46}$$

and

$$\frac{\partial F_3}{\partial x_{ik}} = f_c(r_{ij})\frac{\partial f_c(r_{ik})}{\partial x_{ij}}\frac{x_{ik}}{r_{ik}} = f_c(r_{ij}) M(r_{ik})\frac{x_{ik}}{r_{ik}} \tag{A.47}$$

so that

$$\frac{\partial G_i^5}{\partial x_{ij}} = \left(\frac{x_{ik}}{r_{ij} r_{ik}} - \frac{x_{ij}\cos\theta}{r_{ij}^2}\right) K F_2 F_3 - 2\eta F_1 F_2 F_3 x_{ij} + \tag{A.48}$$

$$F_1 F_2 M(r_{ij}) f_c(r_{ik})\frac{x_{ij}}{r_{ij}} \tag{A.49}$$

and

$$\frac{\partial G_i^5}{\partial x_{ik}} = \left(\frac{x_{ij}}{r_{ij} r_{ik}} - \frac{x_{ij}\cos\theta}{r_{ik}^2}\right) K F_2 F_3 - 2\eta F_1 F_2 F_3 x_{ik} + \tag{A.50}$$

$$F_1 F_2 M(r_{ik}) f_c(r_{ij})\frac{x_{ik}}{r_{ik}} \tag{A.51}$$

In terms of x_{ij} and x_{ik}:

$$\frac{\partial G_i^5}{\partial x_{ij}} = x_{ij}\left(-\frac{\cos\theta}{r_{ij}^2} K F_2 F_3 - 2\eta F_1 F_2 F_3 + F_1 F_2 M(r_{ij}) f_c(r_{ik}) \frac{1}{r_{ij}} \right) + \tag{A.52}$$

$$x_{ik} \frac{K F_2 F_3}{r_{ij} r_{ik}} \tag{A.53}$$

and

$$\frac{\partial G_i^5}{\partial x_{ik}} = x_{ik}\left(-\frac{\cos\theta}{r_{ik}^2} K F_2 F_3 - 2\eta F_1 F_2 F_3 + F_1 F_2 M(r_{ik}) f_c(r_{ij}) \frac{1}{r_{ik}} \right) + \tag{A.54}$$

$$x_{ij} \frac{K F_2 F_3}{r_{ij} r_{ik}} \tag{A.55}$$

Bibliography

[1] J. Behler. "Neural network potential-energy surfaces in chemistry: a tool for large-scale simulations". In: *Phys. Chem. Chem. Phys.* 13. (2011), 17930-17955. DOI: 10.1039/c1cp21668f.

[2] J. Behler. "Atom-centered symmetry functions for constructing high-dimensional neural network potentials". In: *The Journal of Chemical Physics* 134. (2011), 074106. DOI: 10.1063/1.3553717.

[3] S. A. Dragly. "Bridging quantum mechanics and molecular dynamics with artificial neural networks". MA thesis. University of Oslo (2014).
URL: https://www.duo.uio.no/handle/10852/41843.

[4] K. Hornik, M. Stinchcombe, and H. White. "Multilayer Feedforward Networks are Universal Approximators". In: *Neural Networks* 2. (1989), 359. DOI: 10.1016/0893-6080(89)90020-8

[5] R. Rojas. *Neural Networks: A Systematic Introduction*. Springer, 1996.

[6] B. Karlik and A. V. Olgac. "Performance Analysis of Various Activation Functions in Generalized MLP Architectures of Neural Networks". In: *IJAE* 1. (2011), 111-122.

[7] Y. LeCun, Y. Bengio, and G. Hinton. "Deep learning". In: *Nature* 521. (2015), 436-444. DOI: 10.1038/nature14539

[8] X. Glorot, A. Bordes, and Y. Bengio. "Deep Sparse Rectifier Neural Networks". In: *JMLR* 15. (2011), 315-323.

[9] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323. (1986), 533-536. DOI: 10.1038/323533a0

[10] J. Behler. "Constructing High-Dimensional Neural Network Potentials: A Tutorial Review". In: *International Journal of Quantum Chemistry* 115. (2015), 1032-1050. DOI: 10.1002/qua.24890

[11] Tuckerman, Berne, and Martyna . "Reversible multiscale molecular dynamics". In: *J. Chem. Phys.* 97. (1992), 1990.

[12] S. Plimpton. "Fast Parallel Algorithms for Short-Range Molecular Dynamics". In: *Journal of Computational Physics* 117. (1995) 1-19. DOI: 10.1006/jcph.1995.1039.

[13] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, R. Jozefowicz, Y. Jia, L. Kaiser, M. Kudlur, J. Levenberg, D. Man, M. Schuster, R. Monga, S. Moore, D. Murray, C. Olah, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Vigas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu and X. Zheng. "TensorFlow: Large-scale machine learning on heterogeneous systems". *Google Research* (2015). arXiv: 1603.04467

[14] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. "TensorFlow: A System for Large-Scale Machine Learning". *Google Brain* (2016). arXiv: 1605.08695

[15] N. Qian. "On the momentum term in gradient descent learning algorithms" In: *Neural Networks* 12(1). (1999) 145-151. DOI: 10.1016/S0893-6080(98)00116-6

[16] J. Duchi, E. Hazan, and Y. Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". In *The Journal of Machine Learning Research* 12. (2011) 2121-2159. URL: http://jmlr.org/papers/v12/duchi11a.html

[17] M. D. Zeiler. "Adadelta: an adaptive learning rate method". (2012). arXiv: 1212.5701

[18] D. P. Kingma and J. Ba. "Adam: A Method for Stochastic Optimization". (2014). arXiv: 1412.6980

[19] J. Ischtwan and M. A. Collins. "Molecular potential energy surfaces by interpolation". In: *The Journal of Chemical Physics* 100. (1994), 8080. DOI: 10.1063/1.466801

[20] L. M. Raff, M. Malshe, M. Hagan, D. I. Doughan, M. G. Rockley, and R. Komanduri. "Ab inito potential-energy surfaces for complex, multichannel systems using modified novelty sampling and feedforward neural networks". In: *The Journal of Chemical Physics* 122. (2005), 084104. DOI: 10.1063/1.1850458

[21] L. M. Raff, R. Komanduri, M. Hagan, and S. T. S. Bukkapatnam *Neural networks in chemical reaction dynamics*. Oxford University Press, 2012.

[22] A. Pukrittayakamee, M. Malshe, M. Hagan, L. M. Raff, and R. Narulkar, S. Bukkapatnum and R. Komanduri. "Simultaneous fitting of a potential-energy surface and its corresponding force fields using feedforward neural networks". In: *The Journal of Chemical Physics* 130. (2009) 134101. DOI: 10.1063/1.3095491

[23] W. S. McCulloch and W. Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *Bulletin of Mathematical Biophysics* 5. (1943), 115. DOI: 10.1007/BF02478259

[24] D. Kriesel. "A Breif Introduction to Neural Networks" Available at http://www.dkriesel.com

[25] Y. LeCun, P. Haffner, L. Bottou, and Y. Bengio. "Object recognition with Gradient-Based Learning". In: *Shape, Contour and Grouping in Computer Vision. Lecture Notes in Computer Science*. 1681. (1999) 319-345. DOI: 10.1007/3-540-46805-6_19

[26] F. Rosenblatt. "The Perceptron: A probabilistic model for information storage and organization in the brain." In: *Psychological Review* 65. (1958), 386-408.

[27] S. Ruder. "An overview of gradient descent optimization algorithms." (2016). arXiv: 1609.04747

[28] R. Dawes, D. L. Thompson, A. F. Wagner, and Michael Minkoff. "Interpolating moving least-squares methods for fitting potential energy surfaces: A strategy for efficient automatic data point placement in high dimensions". In: *The Journal of Chemical Physics* 128. (2008), 084107. DOI: 10.1063/1.2831790

[29] J. E. Jones. "On the Determination of Molecular Fields - II. From the Equation of State of a Gas." In: *Proceedings of the Royal Society of London. Series A* 106. (1924), 463. DOI: 10.1098/rspa.1924.0082

[30] F. H. Stillinger and T. A. Weber. "Computer simulation of local order in condensed phases of silicon." In: *Physical Review B* 31. (1985), 5262. DOI: 10.1103/PhysRevB.31.5262

[31] V. Molinero, E. B. Moore. "Water modeled as an intermediate element between carbon and silico." In: *J. Phys. Chem. B* 113 (13). (2008), 4008-4016. DOI: 10.1021/jp805227c

[32] P. Vashishta, R. K. Kalia, J. P. Rino, and I. Ebbsjö. "Interaction potential for SiO_2: A molecular-dynamics study of structural correlations." In: *Physical Review B* 41. (1990), 12197. DOI: 10.1103/PhysRevB.41.12197

[33] P. Vashishta, R. K. Kalia, A. Nakano, J. P. Rino. "Interaction potential for silicon carbide: A molecular dynamics study of elastic constants and vibrational density of states for crystalline and amorphous silicon carbide". In: *Journal of Applied Physics* 101. (2007), 103515.

[34] D. Frenkel and B. Smit. "Understanding molecular simulation: from algorithms to applications." Academic Press, 2001.