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

3 Machine learning **11**
3.1 Artificial Neural Networks 12
3.1.1 Convolution 15
3.2 Training 15
3.2.1 Backpropagation 15

4 Neural networks in molecular dynamics **21**
4.1 High-dimensional NNPs 22
4.1.1 Symmetry functions 24
4.1.2 Symmetry functions and forces 29

II Implementation and validation **33**

5 LAMMPS **35**
5.0.1 Installing LAMMPS 35
5.0.2 LAMMPS input script 36
5.0.3 LAMMPS structure 40
5.0.4 Extending LAMMPS 40

6 TensorFlow **43**
6.1 Installing TensorFlow 44
6.2 TensorFlow basic usage 44
6.2.1 Hello world 45

6.2.2 Creating a neural network 45
6.2.3 Visualizing the graph 47

7 Validation 51
7.1 Time usage 51
7.2 Training Lennard-Jones potential 53
7.2.1 Many-neighbour Lennard-Jones 55

III Results and discussion 61
7.3 NN potential for Si 63
7.4 NN potential for SiO2 63

Appendices 65

A Symmetry functions derivatives 67

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 classical force fields. Their time-evolution is governed by Newton's equations of motion. MD allows one 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 classical MD, the interactions are described by a classical force field **F**, which is defined as the negative gradient of a potential energy surface (PES) E,

$$\mathbf{F} = -\nabla E \tag{2.1}$$

The PES is represented by a mathematical function, often split into a sum over two- and three-body interactions. Instead of using an analytical functional form to represent the energy, one can calculate the energies and forces for each time step in the MD simulation using an ab inito method. Different quantum mechanical methods like Hartree-Fock (HF) and Density Functional Theory (DFT) are used for this purpose. These approahces to MD are called quantum-classical molecular dynamics (QCMD).

2.1 Potential energy surfaces

The dynamics of an ensemble of particles is governed by their interactions. In classical MD, the interactions are described by a classical force field **F**, which is defined as the negative gradient of a potential energy surface (PES) E,

$$\mathbf{F} = -\nabla E \tag{2.2}$$

The expression for $V(r)$ may be obtained in different ways:

- Empirical potentials: the functional form are chosen and the parameters fitted from empirical knowledge about the system
- Semi-empirical potentials: based on methods from computational QM, but many approximations are made and some parameters are obatained from empirical data
- Quantum mechanical potentials: a computational QM method is used to obtain the energy for different configurations, and then fitted by a functional form

Instead of using a pre-defined functional form to represent the energy, one can calculate the energies and forces for each time step in the MD simulation using an ab inito method. Different quantum mechanical methods like Hartree-Fock (HF) and Density Functional Theory (DFT) are used for this purpose. These approahces to MD are called quantum-classical molecular dynamics (QCMD).

The PES represents the potential energy $V(\mathbf{r})$ of a system as a mathematical function of the configuration of the atomic positions $\mathbf{r} = (\mathbf{r}_1, \mathbf{r}_2, \dots \mathbf{r}_N)$. To reduce the N-body problem from a fully quantum description to a classical potential we must make the Born-Oppenheimer approximation, which is the assumption that the motion of atomic nuclei and electrons in a molecule can be seperated. This enables us to freeze the degrees of freedom of the nuclei and solve the electronic Schrdinger equation with the positions of the nuclei as parameters. Varying the positions of the nuclei in small steps and repeatedly soving the Schrdinger equation, the quantum PES is made. Classically, we can now approximate the atomic nuclei as point-particles that follow Newtonian dynamics, while the positions and velocities of the electrons are baked into the PES, usually representing the ground state. (HA ET EGET AVNSITT OM BORN-OPPENHEIMER I KVANTETEORIDELEN KANSKJE?)

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

$$V(\mathbf{r}) \approx \sum_{i=1}^{N} V_1(\mathbf{r}_i) + \sum_{i<j}^{N} V_2(\mathbf{r}_i, \mathbf{r}_j) + \sum_{i<j<k}^{N} V_3(\mathbf{r}_i, \mathbf{r}_j, \mathbf{r}_k) + \dots \tag{2.3}$$

Each term can be determined by performing an electronic structure calculation for a number of configurations, and fitted by a suitable functional form. Determining how many terms that is sufficient to describe a system adequately is however 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 at least three-body potentials V_3. 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 an *ab inito* simulation for each configuration is expensive. This problem will be discussed below.

An important step is how to fit the above potential terms obtained from QM calculations to a functional form. There exist a lot of different interpolation teqhniques; in this theses we will use ANNs to fit the data. (SKRIVE OM ANDRE INTERPOLASJONSTEKNIKKER?)

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.

There are two main machine learning methods, supervised learning and unsupervised learning. For the former method, you know the answer to a problem and let the computer deduce the logic behind it. When doing the latter, you do not know the answer and are just trying to find patterns and relationship in the data.

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. Typically supervised.

- Regression: Find a functional relationship between variables, the outputs are thus continous.

- Clustering: Inputs are divided into groups without knowing the different groups, i.e. a form of unsupervised classification.

In this thesis we will use machine learning to do regression, i.e. to find the PES of interacting molecules. One approach to machine learning regression is artificial neural networks.

3.1 Artificial Neural Networks

An Artificial Neural Network (ANN) consists of layers of connected nodes, or neurons. It is supposed to mimic a biological nervous system by letting each neuron interact with other neurons by sending signals between layers. The accumulated signals received by a neuron is used as input to its *activation function*, which is used to to send new signals to following layers. This function is supposed to mimic that a biological neuron is activated if the input is above a certain threshold. We will here follow Behler [1] and Dragly [3].

There are many types of ANNs made for different purposes. The most widely used are feed-forward and recurrent ANNs. The feedforward network was the first and arguably the most simple type of ANN. In this network the information or signals only move forward through the layers - from the input layer, through each successive hidden layer and finally to the output layer. On the other hand, recurrent networks have cycles and loops, i.e. bi-directional data flow. Feed-forward neural networks, also called Multilayer Perceptrons (MLP) were shown to be universal function approximators by K. Hornik et al. [4] in 1989. We will use these networks to approximate the PES in thesis.

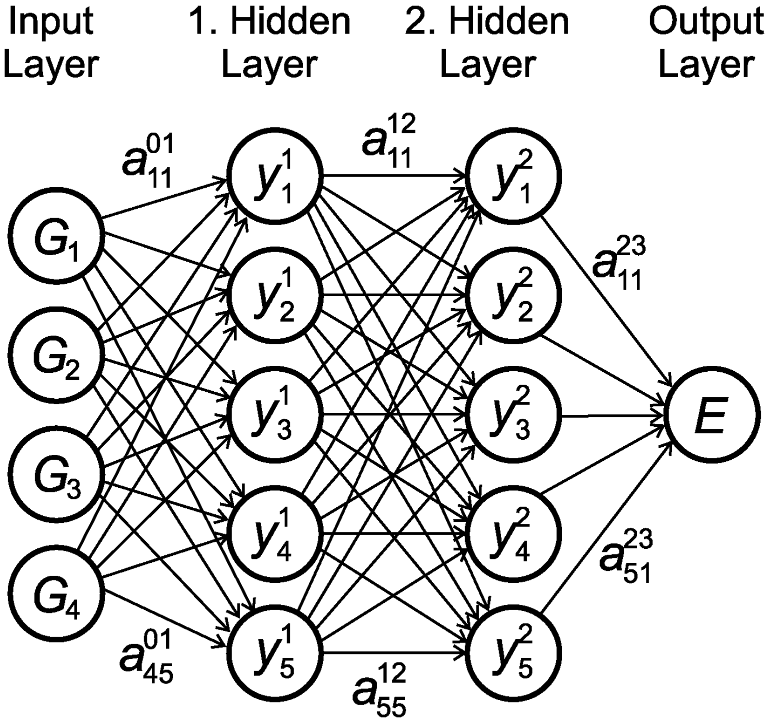


Figure 3.1: Neural network

An MLP networks is shown schematically in Figure 3.1 (STOLEN FROM [1], EDIT TO ONE INPUT LATER).

This network maps input coordinates G_i to a potential energy in the output layer 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. All nodes in each layer are connected to the nodes in the adjacent layer by *weights*, the fitting parameters of the NN. These are called fully connected layers. We use the notation a_{ij}^{kl} for the weight connecting node number i in layer k with node number k in layer $l = k+1$. Also, each node in the hidden layers and the output node have a *bias* b_i^j, where i indicates node number and j layer. All weights and biases are real-valued numbers.

The output of this MLP is calculated in the following way. The configuration are specifiec by the input nodes. Then, a weighted sum x_m^1 of the input coordinates G_i is calculated for each node m in the first hidden layer 1

$$x_m^i = \sum_{i=1}^{4} G_i a_{i,m}^{01} + b_m^i \tag{3.1}$$

where we also add the bias b_m^i for each node. This is a linear combination of the input coordinates using the connection weights as coordinates, which is used as input to the activation function f_m^1 of each neuron, thus producing the output y_m^1 of all neurons in layer 1:

$$y_m^1 = f_m^1(x_m^1) \tag{3.2}$$

For an arbitrary node m in a hidden layer n this generalizes to

$$y_m^n = f_m^n(x_m^n) = f_m^n\left(\sum_{i=1}^{N_{n-1}} y_i^{n-1} a_{i,m}^{n-1,n}\right) \tag{3.3}$$

where N_n is the number of nodes in layer n. 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 complete functional form of an MPL with n hidden layers is thus

$$E = f_1^{n+1}\Bigg(\sum_{m}^{N_n} f_m^n\Bigg(\sum_{l}^{N_{n-1}} f_l^{n-1}\Bigg(\cdots\sum_{j}^{N_1} f_j^1\Bigg(\sum_{i=1} a_{i,j}^{0,1} G_i + b_j^1\Bigg)\cdots\Bigg)\Bigg)\Bigg) \tag{3.4}$$

i.e. a nested sum of activation functions. A desired property of activation functions is to converge asymptotically to constant output values for large and small arguments, with a non-linear region in-between. The non-linearity enables the ANN to fit arbitrary functions [4]. Common activation functions [5] for function approximation are the sigmoid function

$$f(x) = \frac{1}{1+e^{-x}} \tag{3.5}$$

and the hyperbolic tangent

$$f(x) = \tanh(x) \tag{3.6}$$

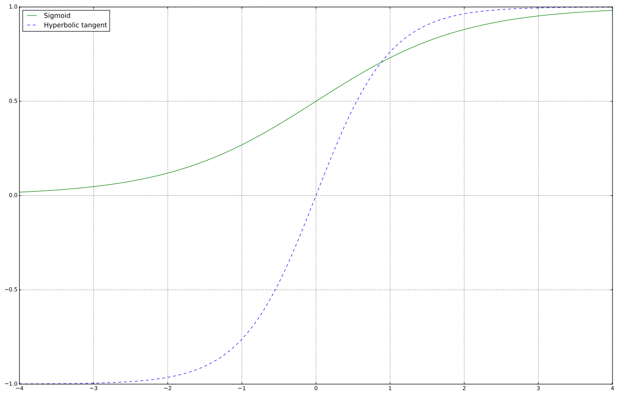


Figure 3.2: Activation functions

Both these functions (Figure 3.2) have the desired properties mentioned above: They are constant (1 or 0) for all values except a small interval around zero, where they have non-linear behaviour. This keeps the outputs of the neurons 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 multi-layer networks. Nevertheless, one should assess the problem at hand when deciding what activation function to use; the performance can vary from problem to problem. (VENTE MED DISKUTERE DETTE TIL JEG HAR SNAKKET OM BACKPROPAGATION OSV.) SNAKKE OM SATURATION OSV.

In later years, the rectifier function (Figure **??**)

$$f(x) = \max(0, x) \tag{3.7}$$

has become the most popular for deep neural networks [7]. PLOT DENNE FUNKSJONEN. The two former functions are symmetrical around zero osv.... It have been argued to be even more biologically plausible than the sigmoid and also perform better than the hyperbolic tangent for deep NNs. [8].

A linear activation function $f(x) = x$ is often used in the output layer to avoid any constraint in the range of output values. For regression, the most widely used output activation is the unity function $f(x) = 1$, i.e. no activation (HVORFOR?).

As shown in (3.4), the NN consists of several nested terms of the form

$$h(x) = c_1 f(c_2 x + c_3) + c4 \tag{3.8}$$

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, thus making the NN more flexible.

3.1.1 Convolution

ANNs are supposed to mimic biological systems. We know that neurons in the visual cortex of animals have localized receptive fields, i.e. they respond only to stimuli in a certain location of the visual field. These regions are overlapping and covers the entire visual field. This can be exploited in ANNs by making the hidden layers only connect to a small contigous region of the input layer. This also speeds up computations. These are called locally connected networks, unlike fully connected networks, where all the hidden layers are connected to the complete input layer.

Using this approach, we can aggregate statistics of these features at various locations. This is called pooling.

3.2 Training

The ability of the NN to represent a function accurately depends on the values of the weights. The weights are typically initialized as random numbers, and then optimized iteratively to minimize the error to a set of expected output values. The expected output values are in our case known energies of a set of atomic configurations, which have been obtained in quantum mechanical calculations of electronic structure. This optimization is called *training* and an iteration is called an *epoch*. The fitting process corresponds to the minimization of an error function, also known as the quadratic *cost* function

$$\Gamma = \frac{1}{2N}\sum_{i=1}^{N}(E_{i,ref} - E_{i,NN})^2 \tag{3.9}$$

$E_{i,ref}$ are the known target energies, while $E_{i,NN}$ are the energies produced by the network. The constant 1/2 are there to cancel out the exponent when we differentiate later. Other cost functions are used for other ML tasks (SKRIVE OM DISSE. KANSKJE HELE DENNE SEKSJONEN BR SKRIVES MER GENERELL OG IKKE KUN FOR TILFELLET PES?)

3.2.1 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.

There are a large number of algorithms that can be used to determine the set of weights minimizing the cost function (3.9). Different kinds of gradient descent methods are widely used, but to implement these we need the partial derivatives of the cost function w.r.t. to all weights and biases to determine how we should adjust each weight. The idea behind gradient descent methods is to minimize a function by iteratively taking steps in the direction of the negative of its gradient w.r.t. all its parameters, i.e. the direction of steepest descent towards a minima:

$$w_{ij}^{new} = w_{ij}^{old} - \gamma \frac{\partial \Gamma}{\partial w_{ij}^{old}} \tag{3.10}$$

where we have skipped the layer indicies for clarity. Also, w_{ij} now denotes the weight connecting node j in layer k with node i in the following layer. This change of order is introduced to make it easier to rewrite the below equations for backpropagation into matrix-vector equations. As seen from (3.10), we need to adjust each weight by an amount

$$\Delta w_{ij} = \frac{\partial \Gamma}{\partial w_{ij}} \tag{3.11}$$

The corresponding expression for the biases is

$$\Delta b_i = \frac{\partial \Gamma}{\partial b_i} \tag{3.12}$$

A common method to obtain these derivatives is backpropagation [9]. In backpropagation, the output of the NN is compared to the desired output, the error is then propagated backwards to adjust the weights. 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 weights, thereby obtaining the gradient of the network.

To change the value of the cost function (**??**), we need to change the outputs of the neurons in the network. Changing the input to neuron j by a small amount Δx_j results in the output

$$y_j = f_j(x_j + \Delta x_j) \tag{3.13}$$

This change will propagate through later layers in the network, finally causing the overall cost to change by an amount $\frac{\partial \Gamma}{\partial x_j}\Delta x_j$. If $\partial \Gamma / \partial x_j$ is close to zero, then we can't improve the cost much by perturbing the weighted input x_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:

$$\delta_j \equiv \frac{\partial \Gamma}{\partial x_j} \tag{3.14}$$

We also define

$$A_i = \{j : w_{ij}\} \tag{3.15}$$

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

$$P_j = \{i : w_{ij}\} \tag{3.16}$$

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

The weight change (3.11) can be expanded into two factors by use of the chain rule:

$$\Delta w_{ij} = \frac{\partial \Gamma}{\partial x_i} \frac{\partial x_i}{\partial w_{ij}} \tag{3.17}$$

Now we move in the opposite direction compared to the feed-forward function composition stage: First we differentiate the cost w.r.t neuron i's input, then we differentiate the input w.r.t. weight w_{ij} connecting neurons j and i. The first term on the right is the error δ_i of node i. The second is

$$\frac{\partial x_i}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \sum_{m \in A_i} w_{im} y_m = y_j \tag{3.18}$$

Putting the two together, we get

$$\Delta w_{ij} = \delta_i y_j \tag{3.19}$$

To compute this quantity, we thus need to know the output and the error for all nodes in the network. The outputs are as we have seen

$$y_i = f_i(x_i) = f_i \left(\sum_{j \in A_i} w_{ij} y_j + b_i \right) \tag{3.20}$$

The error for output neuron o is

$$\delta_o = \frac{\partial \Gamma}{\partial x_o} = \frac{\partial \Gamma}{\partial y_o} \frac{\partial y_o}{\partial x_o} = (d_o - f_o(x_o)) \frac{\partial f_o(x_o)}{\partial x_o} \tag{3.21}$$

When doing regression our output neurons will have the unity activation function, which means that the error is reduced to

$$\delta_o = d_o - y_o \tag{3.22}$$

This error is then propagated backwards through the network. Each hidden neuron will have an error

$$\delta_j = \frac{\partial \Gamma}{\partial x_j} = \sum_{i \in P_j} \frac{\partial \Gamma}{\partial x_i} \frac{\partial x_i}{\partial y_j} \frac{\partial y_j}{\partial x_j} \tag{3.23}$$

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

$$\frac{\partial x_i}{\partial y_j} = \frac{\partial}{y_j}\left(\sum_{m\in A_i} w_{im}y_m + b_i\right) = w_{ij} \tag{3.24}$$

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

$$\frac{\partial y_j}{\partial x_j} = \frac{\partial f_j(x_j)}{\partial x_j} \equiv f_j'(x_j) \tag{3.25}$$

Putting all the pieces together we obtain

$$\delta_j = f_j'(x_j)\sum_{i\in P_j}\delta_i w_{ij} \tag{3.26}$$

The above expression requires that we know the errors of all the posterior nodes of node j. 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 errors are propagated backwards through the whole NN until we reach the input nodes. By propagating the error of only one ouput neuron, we thus obtain the errors of all the neurons at once. This is the strength of the backpropagation algorithm. For the biases, we have

$$\Delta b_j = \frac{\partial \Gamma}{\partial b_j} = \sum_{i\in P_j}\frac{\partial \Gamma}{\partial x_i}\frac{\partial x_i}{\partial y_j}\frac{\partial y_j}{\partial x_j}\frac{\partial x_j}{\partial b_j} \tag{3.27}$$

The only new term her is the last one:

$$\frac{\partial x_j}{\partial b_j} = \frac{\partial}{\partial b_j}\left(\sum_{m\in A_j} w_{jm}y_m + b_j\right) = 1 \tag{3.28}$$

Consequentially, the update rule for the biases is simply the error of each neuron:

$$\Delta b_j = \delta_j \tag{3.29}$$

For fully-connected FFNNs we can easily rewrite these equations in matrix notation. In this notation, the biases, inputs, outputs and errors for all nodes in a layer are combined into vectors, while all weights from one layer to the next form a matrix W_l, where l denotes the layer number. The inputs to all the nodes in layer l can be written

$$\begin{pmatrix} x_1^l \\ x_2^l \\ x_3^l \end{pmatrix} = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{pmatrix}\begin{pmatrix} y_1^k \\ y_2^k \\ y_3^k \end{pmatrix} + \begin{pmatrix} b_1^l \\ b_2^l \\ b_3^l \end{pmatrix}$$

so that equation (3.20) takes the form

$$\mathbf{y}_l = f_l(W_l \mathbf{y}_{l-1} + \mathbf{b}_l) \tag{3.30}$$

Comparing equations (3.20) and (3.26), we observe that

$$\sum_{j \in A_i} w_{ij} = \left(\sum_{i \in A_j} w_{ij} \right)^T \tag{3.31}$$

i.e. the weight matrices used in backpropagation are the transpose of the matrices used in forward activation. The complete backpropagation algorithm including forward activation then looks as follows:

1. Initialize the input layer:
$$\mathbf{y}_0 = \mathbf{x}_0 \tag{3.32}$$
2. Propagate the activity forward: for $l = 1, 2, \ldots, L$:
$$\mathbf{y}_l = f_l(W_l \mathbf{y}_{l-1} + \mathbf{b}_l) \tag{3.33}$$
3. Calculate the error in the output layer:
$$\boldsymbol{\delta}_L = \mathbf{t} - \mathbf{y}_L \tag{3.34}$$
4. Calculate the error: for $l = L-1, L-2, \ldots, 1$:
$$\boldsymbol{\delta}_l = (W_{l+1}^T \boldsymbol{\delta}_{l+1}) \cdot f_l'(\mathbf{x}_l) \tag{3.35}$$
5. Update the weights and biases:
$$\Delta W_l = \boldsymbol{\delta}_l \mathbf{y}_{l-1}^T, \quad \Delta \mathbf{b}_l = \boldsymbol{\delta}_l \tag{3.36}$$

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.

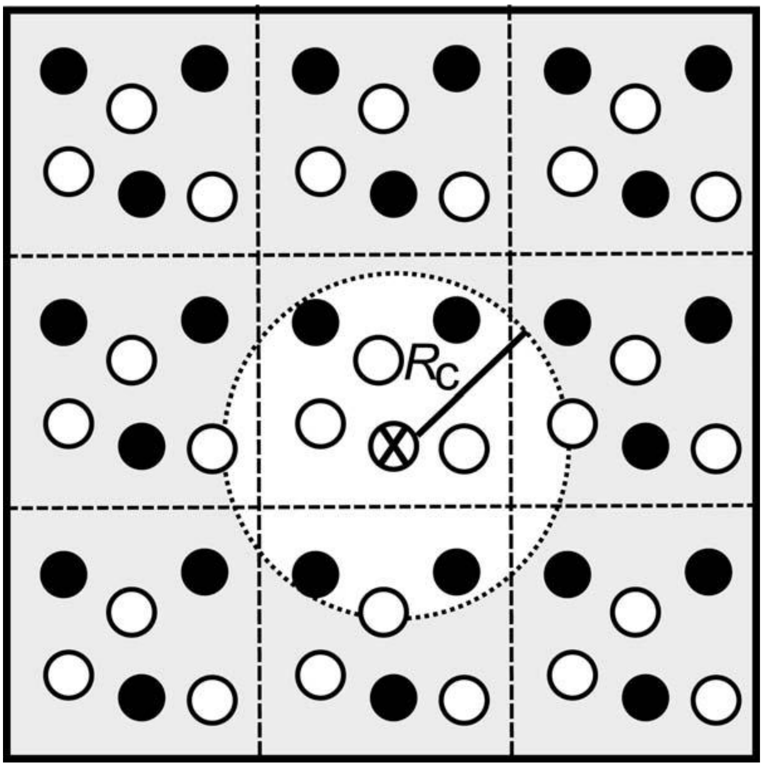


Figure 4.1

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.

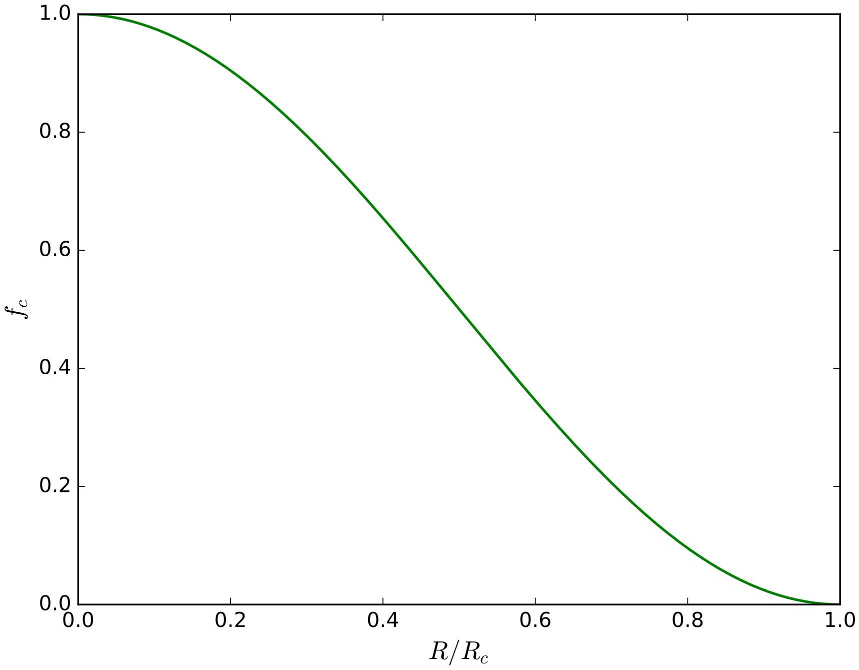


Figure 4.2: Plot of the cutoff function (4.2) applied in this thesis. This function is used to define a chemical environment around a central atom: only the atoms within the cutoff radius R_c contribute to its energy. These are called neighbouring atoms. The closer a neighbouring atom is, the larger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

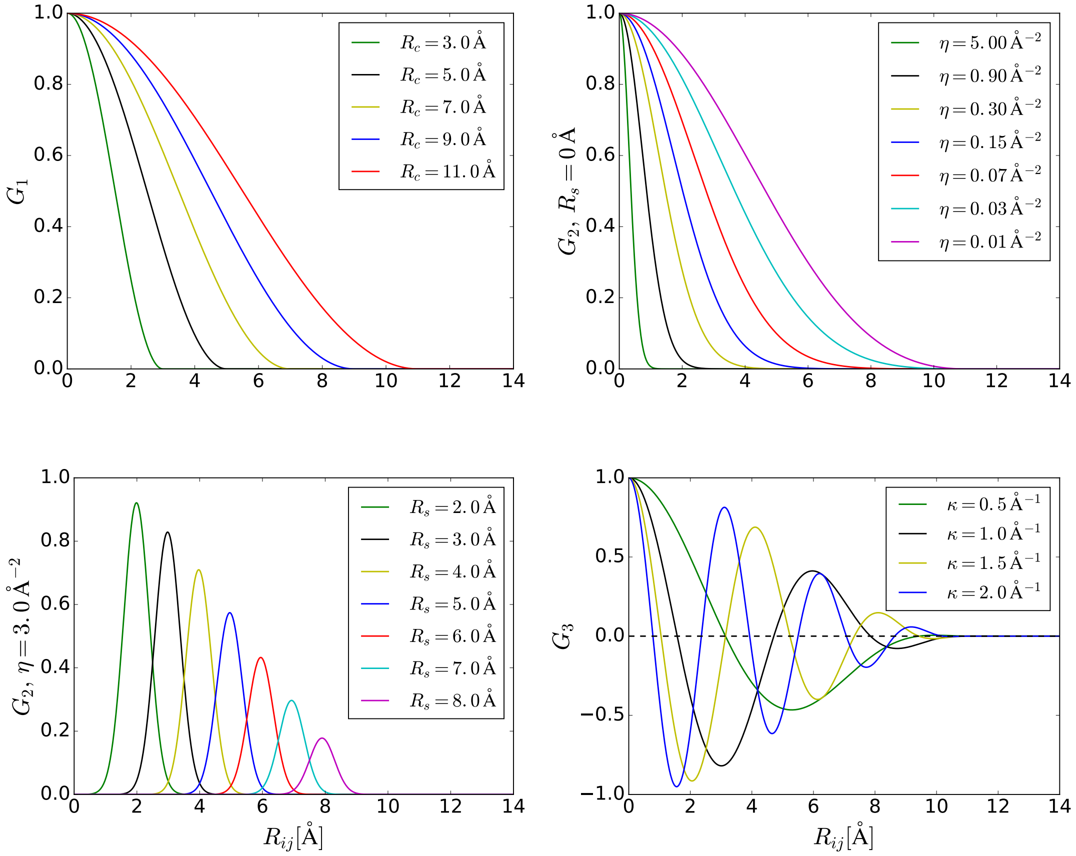


Figure 4.3: Radial symmetry functions G^1, G^2 and G^3 for an atom with one neighbour only. A set of such functions represents the radial distrubution of neighbours around a central atom placed at the origin. For G^2 and G^3 a cutoff $R_c = 11.0\,\text{Å}$ has been used.

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,

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

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

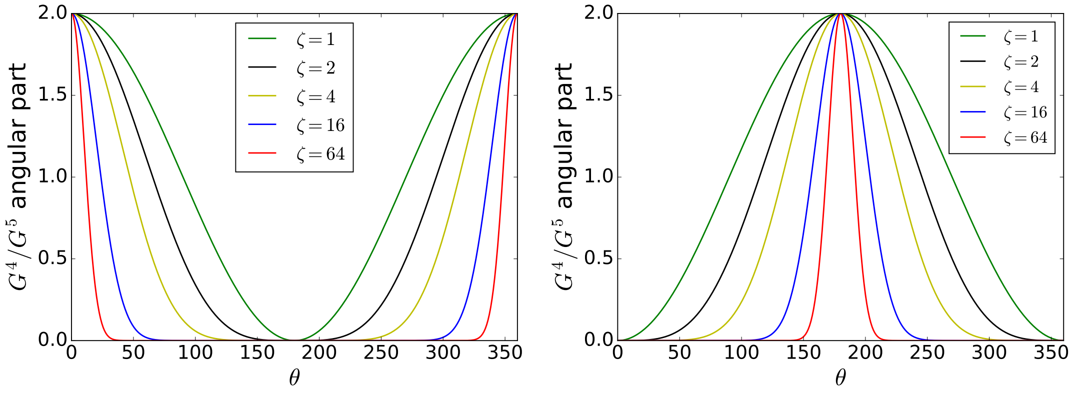
This function becomes zero if any of the interatomic distances is larger than R_c.

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.



(a) **(b)**

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 3.2.1 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,

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

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

---

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

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 7.1.

---

[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 (3.4) 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 (3.5) 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,

```python
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 as ellipses, while the rectangles are namespaces consisting of several nodes. [6]

---

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

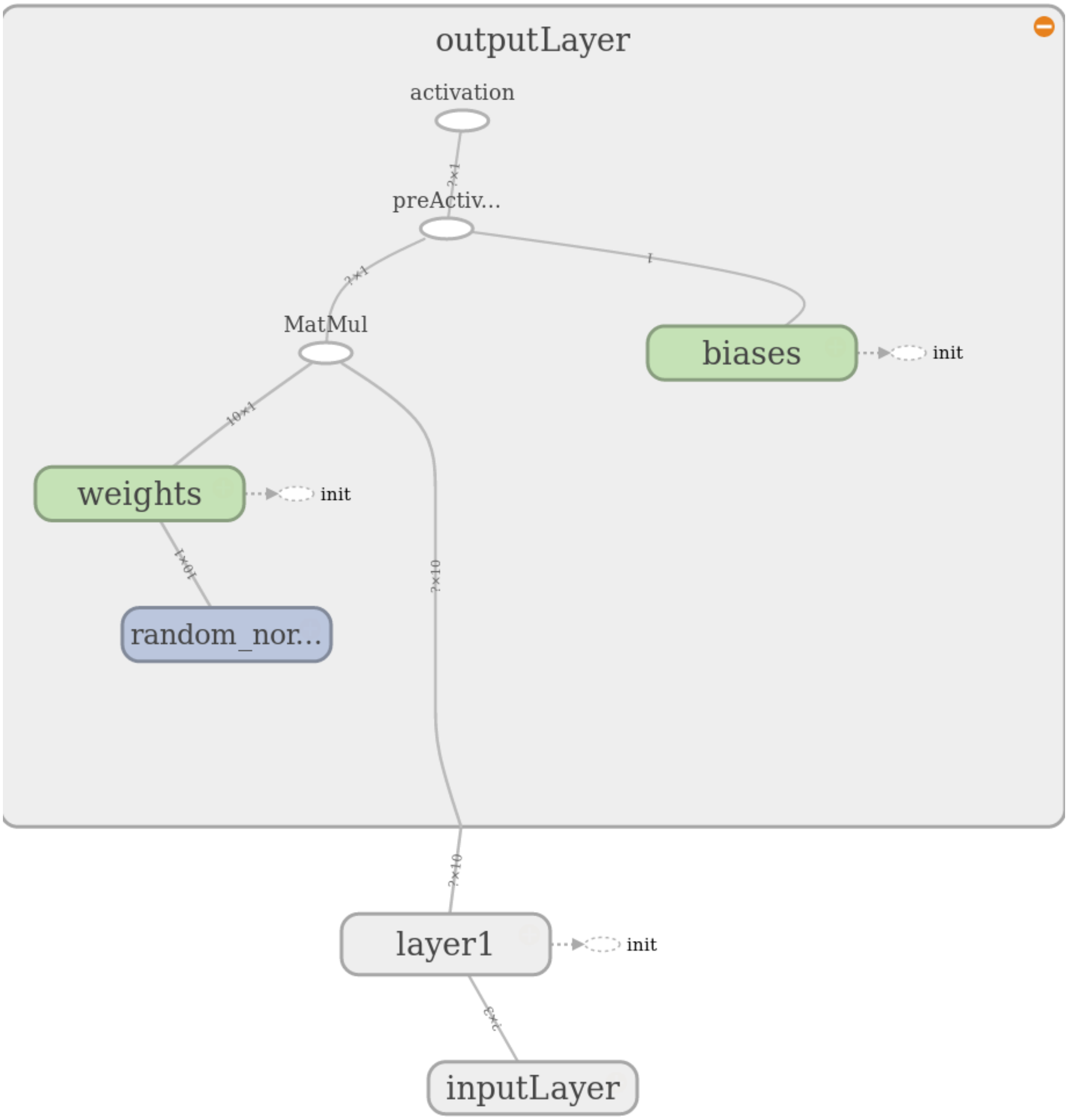

outputLayer
activation
?×1
preActiv...
?×1
1
MatMul
biases
init
10×1
weights
init
10×1
?×10
random_nor...
?×10
layer1
init
?×3
inputLayer

Figure 6.1

---

read operations. See TF API for more information.

Chapter 7

Validation

7.1 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 7.1.

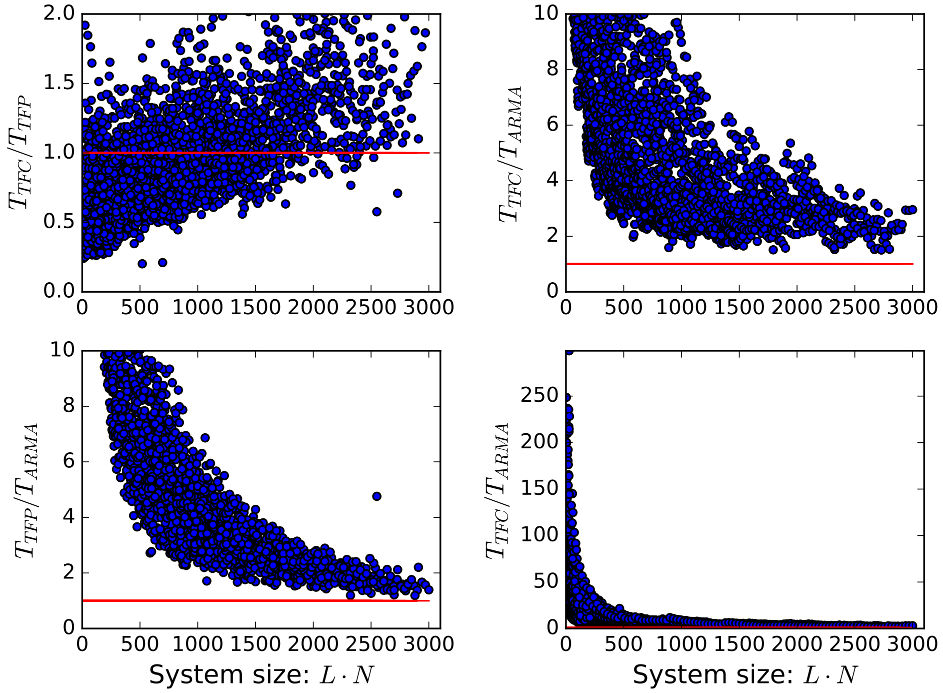


Figure 7.1: Scatter plot of CPU time when evaluating untrained NNs with random weights and sigmoid activation functions using the TF Python API (T_{TFP}), the TF C++ API (T_{TFC}) and Armadillo (T_{ARMA}). L is the number of layers, N is the number of nodes in each hidden layer. All the NNs have one 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.

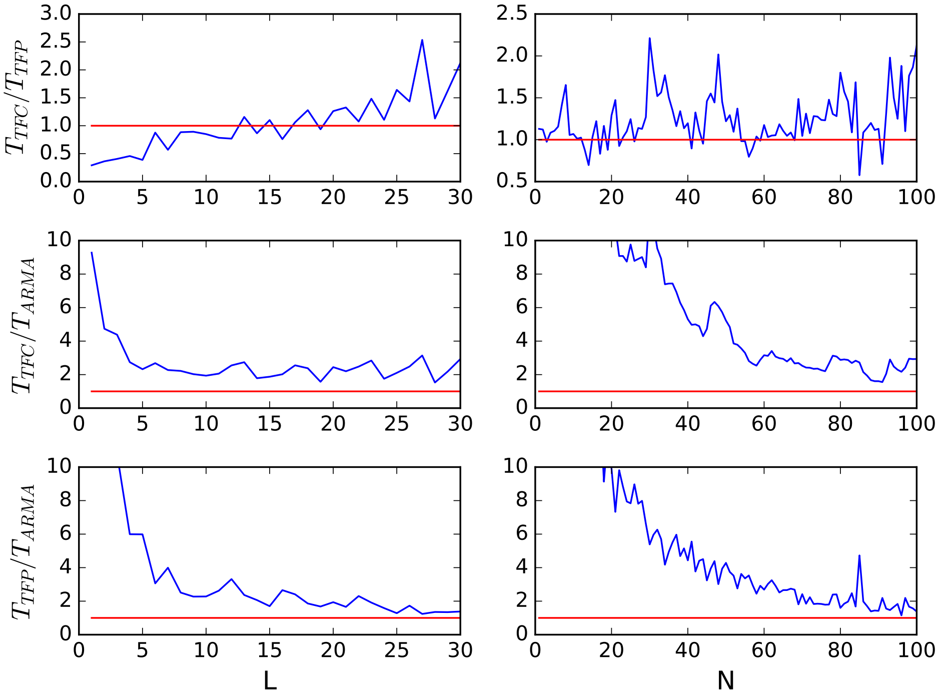


Figure 7.2: Left column: CPU time comparison for $N = 100$. Right column: CPU time comparison for $L = 30$.

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.

7.2 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 7.3.

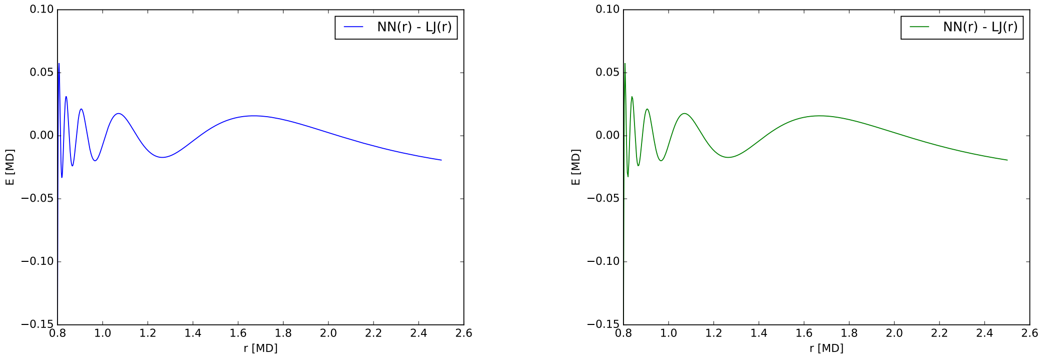


Figure 7.3: Error of a network trained to reproduce the shifted Lennard-Jones potential. The NN is trained in Python and the error on the training data interval is calculated, shown on the left. The NN is also written to file for evaluation in C++, shown on the right.

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 7.4.

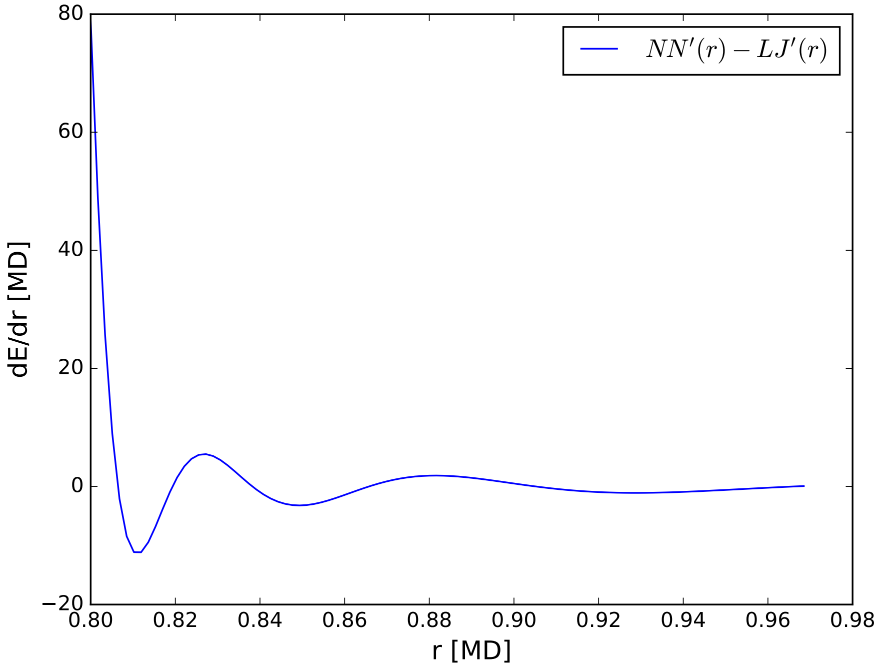
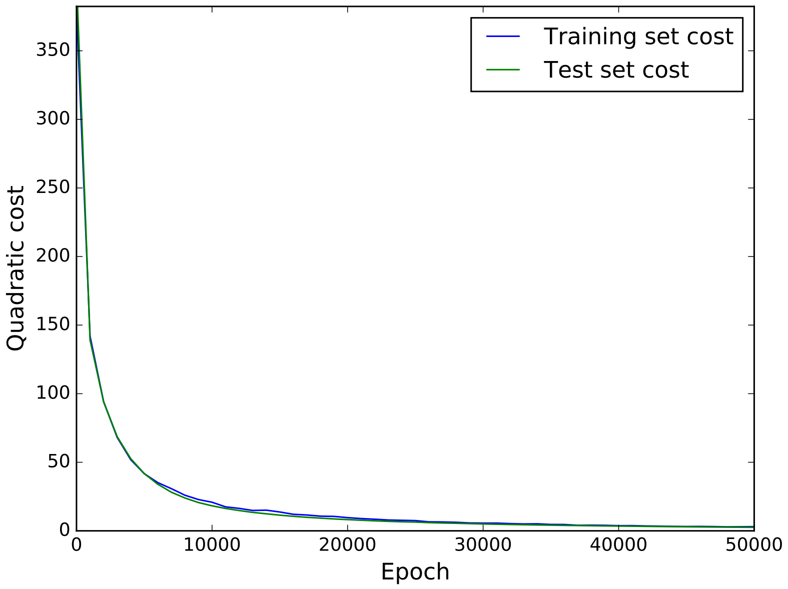


Figure 7.4: Error of the gradient of a NN trained to reproduce the shifted Lennard-Jones potential. The NN is trained in Python and evaluated and differentiated in C++ using the backpropagation algorithm. The result is compared to the analytical derivative of the LJ potential. Only a part of the training interval is shown, the graph is essentially flat after this point

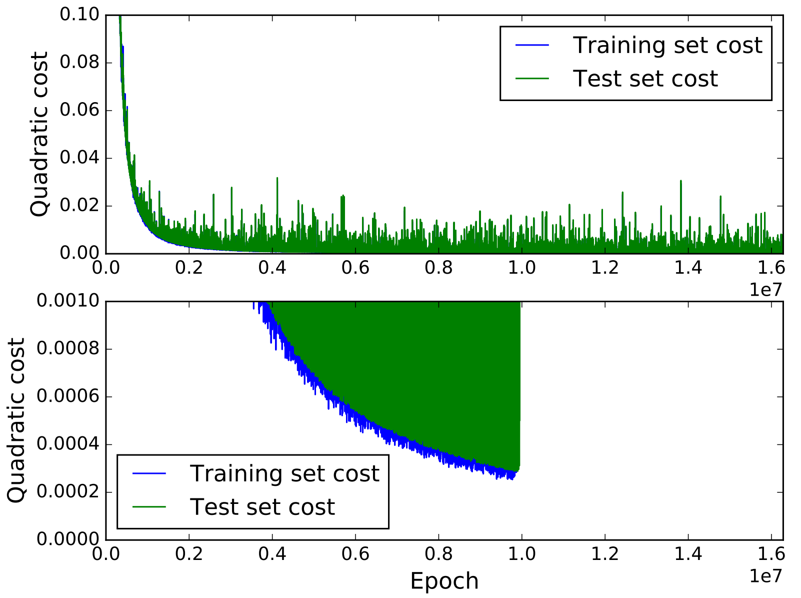
Again we get an oscillating error over the training interval. The oscillations near the left end of the interval is much larger than for the energy, but they get damped more quickly. The large error near the left end of the interval is a natural consequence of the shape of the LJ potential for these 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.

7.2.1 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 7.5. (Training session 02.12-19.36.50)



(a) Zoomed out



(b) Zoomed in

Figure 7.5: Quadratic cost (3.9) 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 (3.5). Figure 7.5a shows the cost for the first 50 000 epochs, while Figure 7.5b 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

$$(x_1, y_1, z_1, r_1, x_2, y_2, z_2, r_2, \dots, x_N, y_N, z_N, r_N) \tag{7.1}$$

which yields an output vector

$$(F_x^{TOT}, F_y^{TOT}, F_z^{TOT}, E^{TOT}) \tag{7.2}$$

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

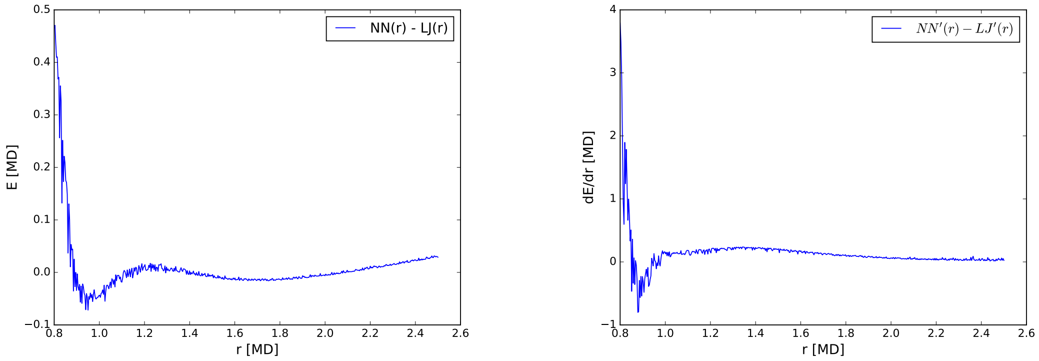


Figure 7.6: Error of a network trained to yield the total energy and force on a central atom from 5 neighbouring atoms. The energy contribution from each neighbour is a shifted LJ potential. Trained for 1e6 epochs, 100 nodes in 2 layers. The energy error is shown on the left, force error on the right.

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

7.3 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.

7.4 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,

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

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,

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

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

The corresponding expression for x_{ik} is

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

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

$$\tag{A.27}$$

Then, taking (A.16) into account:

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

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

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

and for x_{ik}:

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

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

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

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

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

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

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

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". Phys. Chem. Chem. Phys. 13, 17930-17955 (2011). DOI: 10.1039/c1cp21668f.

[2] J. Behler. "Atom-centered symmetry functions for constructing high-dimensional neural network potentials". The Journal of Chemical Physics 134, 074106 (2011). 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". Neural Networks 2, 359 (1989). 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". IJAE 1, 111-122 (2011).

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

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

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

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

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

[12] S. Plimpton. ”Fast Parallel Algorithms for Short-Range Molecular Dynamics”. Journal of Computational Physics 117, 1-19 (1995). 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” Preliminary White Paper (2015). Software available from tensorflow.org.