Freie Universität Berlin

Fachbereich Mathematik und Informatik

Masterarbeit

im Studiengang Mathematik

Thema: Determining the Potential Energy Surface of Atomic Clusters using Machine Learning

eingereicht von: Tony Schwedek
Matrikelnummer 4558730

eingereicht am: 23. November 2017

Erstgutachter: Herr Prof. Dr. Raúl Rojas

Selbstständigkeitserklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Zeichnungen oder Abbildungen sind von mir selbst erstellt worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Tony Schwedek (Berlin, den 23. November 2017)

Abstract

Applying machine learning techniques and especially artificial neural networks to problems related with quantum mechanical calculations has become more popular recently [8, 17].
Meanwhile, the development of software frameworks for efficient parallelized computations, like Numpy [10] and SciPy [11] for the CPU as well as Theano [12], TensorFlow [15] or Keras [13] for a GPU made enormous progress in the last few years.
The main part of this work is devoted to translating the techniques developed in the above mentioned works on applying artificial neural networks to approximate the Potential Energy Surface of an atomic cluster to modern computing frameworks, making use of the possibility to parallelize and hence speed up the computations, using a GPU. First, we test the software developed within this thesis and obtain results comparable to those in [17] for the approximation of the Potential Energy Surface. However, we will see that the software developed within this thesis is much faster, the factor of speed improvement is about 30 to 70, depending on the given problem and, of course, on the size of the data set.
The next step is to approximate the forces acting on the nuclei of the atoms in the atomic cluster, which was done before by differentiating the Potential Energy Surface with respect to the coordinate axes [17].
In this thesis, we will take a different approach, not differentiating the Potential Energy Surface but rather directly approximating these forces with an artificial neural network. We will see that this takes more time than approximating the Potential Energy, but because of the speed improvements gained by using the modern and optimized software frameworks like Keras, and because of the usage of a GPU, this approach is still feasible in terms of computational time needed.
The features for the training of an artificial neural network that approximates the Potential Energy Surface have been obtained using so called *symmetry functions* that have been applied to atoms lying within a certain range of a sphere, yielding features that are invariant under translation and rotation, just as the target output, the Potential Energy, is as well. However, when approximating the forces, the output is not invariant under rotation. But still, rotating the atomic cluster will rotate the forces in the same way. For this reason, we introduce a modified approach, replacing the aforementioned spheres by *ellipsoids*, more precisely spheres stretched along one of the coordinate axes, to ensure the same behaviour of features and target outputs when rotating the input, namely the atomic cluster. Introducing this change in feature calculation did not improve the result in terms of error measurement by a meaningful amount, but the idea is very new and there is definitely a lot of room for improvement.
It must be said that this is only a first try of directly approximating the forces through the use of an artificial neural network, but it yields promising results and asks for optimization in this direction.

Note of thanks

At this point I would like to express my thanks to Mr. Dr. Xim Bokhimi, for greatly supporting me during the work on this topic. It has always been a pleasure to work with him.

Contents

1 Introduction

The terms *Machine Learning* and *Artificial Neural Network* are used more and more frequently within the last years. The "classical applications" of artificial neural networks are image recognition as well as speech recognition. In this work, we will apply artificial neural networks to a completely different problem setting. We will construct an artificial neural network to approximate the *Potential Energy Surface* of an atomic cluster. More precisely, given an atomic system, an artificial neural network is used to predict the Potential Energy of the system. The forces acting on the nuclei of the atomic cluster could then be obtained by differentiating the Potential Energy Surface with respect to the coordinate axes. This is not the first time that artificial neural networks have been applied in this quantum mechanical context, see for example [8, 17].
At the beginning of this work, we will talk about the basics of *machine learning*. We will introduce the concept of an *artificial neuron* and how to build an *artificial neural network* out of multiple artificial neurons. Then, we will discuss how an artificial neural network is enabled to *learn*. An artificial neural network *learns* or *is fitted* with respect to a certain task when a certain measurement of error decreases over time during a *training* process. Afterwards, we will give a very brief overview of *molecular dynamics*. More precisely, we will make clear why it is beneficial to have an artificial neural network that is capable of predicting the Potential Energy of an atomic cluster.
The second part of the work is dealing with the implementation of the aforementioned ideas. We start with an implementation of an artificial neural network approximating the Potential Energy Surface of an atomic cluster, and we will present some results, where the error is comparable to other state of the art software, while the time spent on doing the training of the artificial neural network is dramatically decreased by using modern computing frameworks that allow for doing the calculations highly parallelized using a GPU.
Finally, we will try to construct artificial neural networks that are capable of approximating the forces acting on the nuclei of the atoms. We will also discuss in more depth how the *features*, that are used as input for the artificial neural networks, are generated. We will as well present some results in this case. While they are not as good as the results obtained from the network that approximates the Potential Energy, it must be said that this is only a first approach of directly approximating the forces acting on the nuclei using an artificial neural network. There is still a lot of room for improvement in future works on this topic. Taking this into account, the results look promising.
At the very end we will also take a short look at *Deep Learning* and so called *Convolutional Neural Networks*. The technique of convolutional layers, originally designed mainly for improving the results of artificial neural networks for image and speech recognition, might very well improve the performance of artificial neural networks approximating the forces acting on the nuclei of an atomic cluster as well. Unfortunately implementing and testing all of these ideas to our specific problem would go beyond the scope of this thesis.
The computer system that was used to run the software consists (among others) of the following parts: A *Gigabyte GA-H97-D3H Intel H97* mainboard, an *Intel Xeon*

E3-1231v3 4x3.40GHz CPU, a *MSI Nvidia GForce GTX 1070 Gaming 8G* (8192MB memory) GPU and *16GB (4x4) Crucial Ballistix Sport DDR3-1600 DIMM CL9* RAM. The used operating system is *Ubuntu 16.04.3 LTS*.
The software versions are 1.12.1 for Numpy, 0.19.1 for SciPy, 0.9.0 for Theano, 2.0.2 for Keras and 2.0.2 also for Matplotlib. The Python version is 3.5.2.

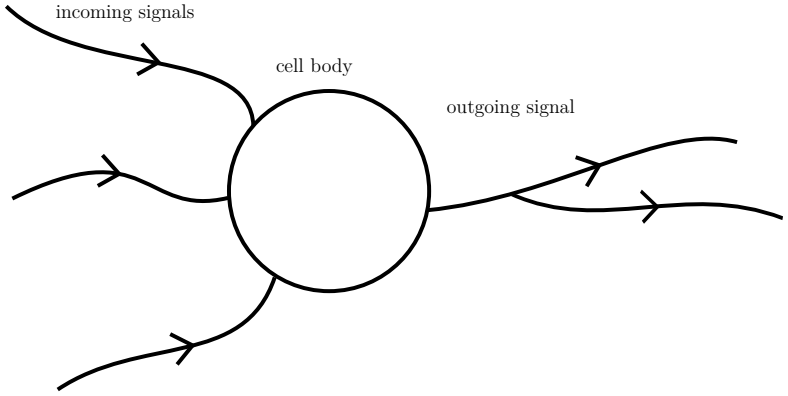


Figure 1: A simple scheme of a neuron, a cell in the human brain passing on electric signals from the left to the right.

2 Introduction to machine learning

Machine learning is a somewhat vague term used in a variety of different areas. Generally speaking, machine learning can be described as a *"subfield of artificial intelligence, developing self-learning algorithms to gain knowledge from data in order to make predictions"*. [1]
This thesis deals with *supervised learning*, a subarea of machine learning. In this case, a large data set of *inputs* and corresponding *target outputs* is given. The goal is to fit a model to them such that new, previously unseen input data gets mapped approximately to the corresponding target output, where the error should be as small as possible.
There are many different machine learning algorithms, sometimes also called models, for supervised learning such as *logistic regression*, *support vector machines* or *decision trees*. In this thesis, we will choose so called *artificial neural networks*.
This section will describe the terminology necessary to understand what neural networks are and how they work.

2.1 Artificial Neurons or Perceptrons

The idea of artificial neural networks is pretty much copied from nature, more precisely from the way human brains work. At first, we will take a look at Figure 1, depicting a very simplified scheme of a *neuron*, a cell in the brain of a human being which transmits electric signals. Incoming electric signals from other neurons on the left are added up and if they surpass a certain activation energy level, the neuron itself will fire an electric signal to the right, which is passed to other neurons. This schematic view of neurons was published by McCulloch and Pitts in [4].
The next step is to describe a simple mathematical model of this scheme. Figure 2 on the next page shows such a model. We say that an *artificial neuron* has inputs $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)^T$, representing the incoming electric signals from the left, and corresponding weights $\boldsymbol{w} = (w_1, w_2, \ldots, w_n)^T$, to determine the importance of the inputs.

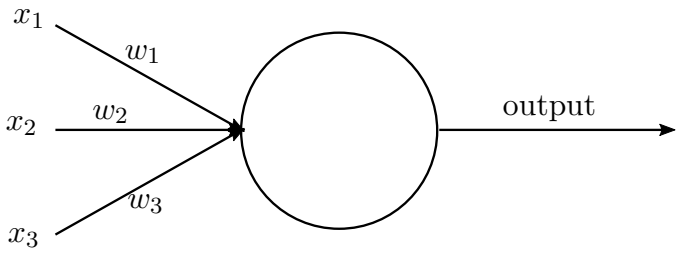


Figure 2: A scheme of an artifial neuron, a mathematical model to describe a neuron.

The *weighted input* is defined to be $\langle \boldsymbol{x}, \boldsymbol{w} \rangle = x_1 \cdot w_1 + x_2 \cdot w_2 + \ldots + x_n \cdot w_n$. The last step is to determine which signal is passed from one artificial neuron to the next one. According to nature, either the neuron fires, or not, so we could in principle set either 1 or 0 as output, but we will be a little more general from the very beginning on. Let $\phi : \mathbb{R} \to \mathbb{R}$ be the so called *activation function* of the artificial neuron. We define the *output* a (sometimes also called *activation*, hence the letter a) of an artificial neuron to be the result of its *activation function* ϕ applied to the weighted input, $a = \phi(\langle \boldsymbol{x}, \boldsymbol{w} \rangle)$. For simplification and to be even more general, we also allow an artificial neuron to have a *bias* b which shifts the activation function. The equation for the output becomes

$$a = \phi(\langle \boldsymbol{x}, \boldsymbol{w} \rangle + b) \quad . \tag{1}$$

It becomes clear that a depends on $\boldsymbol{x}$, $\boldsymbol{w}$ and b. Later, $\boldsymbol{x}$ will be the input, derived from the data, so it is fixed and won't change, hence the paramaters a is really depending on are the weights $\boldsymbol{w}$ and the bias b. These are the parameters we want to optimize during the *training process*.
For now, let's assume we choose

$$\phi(z) = \begin{cases} 1, z \geq 0 \\ 0, z < 0 \end{cases} \quad ,$$

following the analogy to nature. In this case, b would be the negative of the activation energy level, because

$$\langle \boldsymbol{x}, \boldsymbol{w} \rangle + b \geq 0 \iff \langle \boldsymbol{x}, \boldsymbol{w} \rangle \geq -b \quad ,$$

where $-b$ has to be the activation energy threshold. Figure 3 on the following page shows the step function for $-b = 1$. Later, we will see that it would be very helpful to have a differentiable activation function. In the next subsection we will analyse how to smoothe out the step function.

2.2 Continuous activation functions

The core idea of the machine learning algorithms used in this thesis is an iterative process, improving the parameters a little bit within each iteration. Hence, we need a continuous activation function, because this is the only way we can guarantee that small

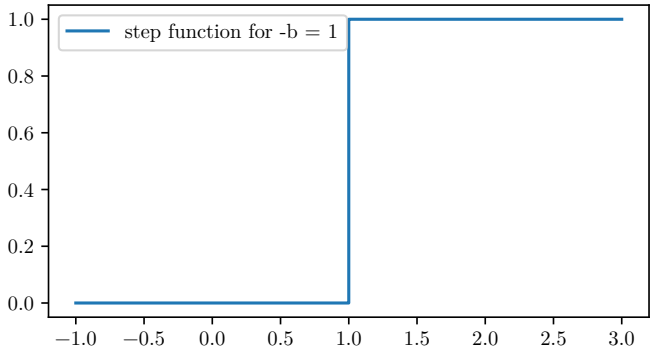


Figure 3: The shape of the step function.

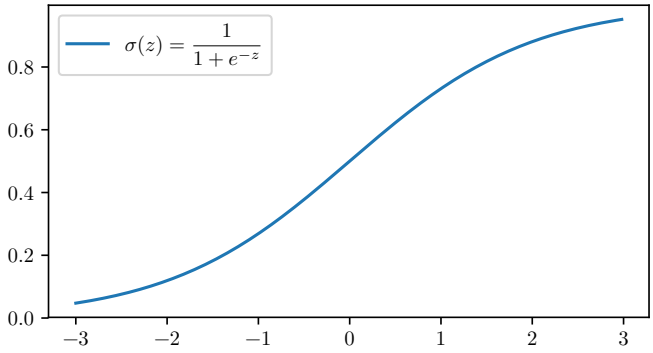


Figure 4: The sigmoid function σ, often called the logistic function.

changes in the parameters lead to only reasonably small changes in the output. In this section we will see some alternative activation functions. We start with the so called *sigmoidal* activation function, also called *logistic function*. Figure 4 shows it.
Another approach is to look for a function with similar shape, which is symmetric around the origin. Hence, we look for a sigmoidal shaped activation function that lies in the range $[-1, 1]$. A widely used activation function fullfilling all the criteria is the *hyperbolic tangent activation function*. It is depicted in Figure 5 on the following page.
Hyperbolic tangent activation functions are the default in the software developed within the thesis, while the user can also define a custom one.
Now that we know how to handle one artificial neuron, we will put many of them together to obtain a network of artificial neurons, a so called *artificial neural network*.

2.3 Putting perceptrons together: Artificial neural networks

In Figure 2 on the previous page, where an artificial neuron is schematically drawn, there is only one output arrow. In the future, we will be able to send the outgoing signal to many other neurons. The fact that there is only one arrow shows that the output is always the same. One artificial neuron has exactly one output, although we might send

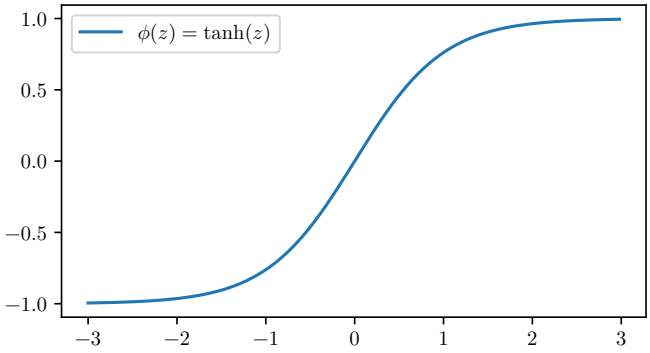


Figure 5: The hyperbolic tangent activation function.

the output signal to several other artificial neurons. This will in future be represented by many arrows going to the right, but still, the output transmitted along all these arrows is the same.

One artificial neuron alone isn't yet a powerful machine learning algorithm, at least not for complex tasks. The next step is to put many of them together to obtain a network of artificial neurons.

We will take a look at so called *fully connected feed forward artificial neural networks*, because this is the most basic network type and also the network type used the most in the accompanying software. Later, there will also be a short discussion on so called *convolutional neural networks* and how to use the idea of convolution to potentially improve the neural network architecture of the software.

A *fully connected feed forward artificial neural network* consists of several *layers*, which in turn consist of multiple artificial neurons. The first layer is called the *input layer*, the last layer is called the *output layer* and intermediate layers are called *hidden layers*. Note that the input layer is not a real layer as the other ones, the input is just passed through without any calculations taking place. The output from every artificial neuron from one layer is used as input for every single artificial neuron from the next layer, hence the term fully connected. The term feed forward is chosen because the signals are only passed from one layer to the next one, but never backwards. This architecture is depicted in Figure 6 on the following page.

It allows for some very powerful parallelization techniques, that enormously speed up computations. A derivation of them similar to the one given here, but more detailed, can be found in chapter 12 of [1] or chapter 1 of [2].

We start with some notation. We need a name for every single weight of the artificial neural network. The weight corresponding to the arrow from the j-th artificial neuron in layer k to the i-th artificial neuron in layer $k+1$ is called $w_{i,j}^{(k)}$. Note that the order of the indices is important.

When we have n artificial neurons in layer k and p artificial neurons in layer $k+1$, we can compress all weights between these two layers into a so called *weight matrix* $W^{(k)} \in \mathbb{R}^{p \times n}$ with $W_{i,j}^{(k)} = w_{i,j}^{(k)}$.

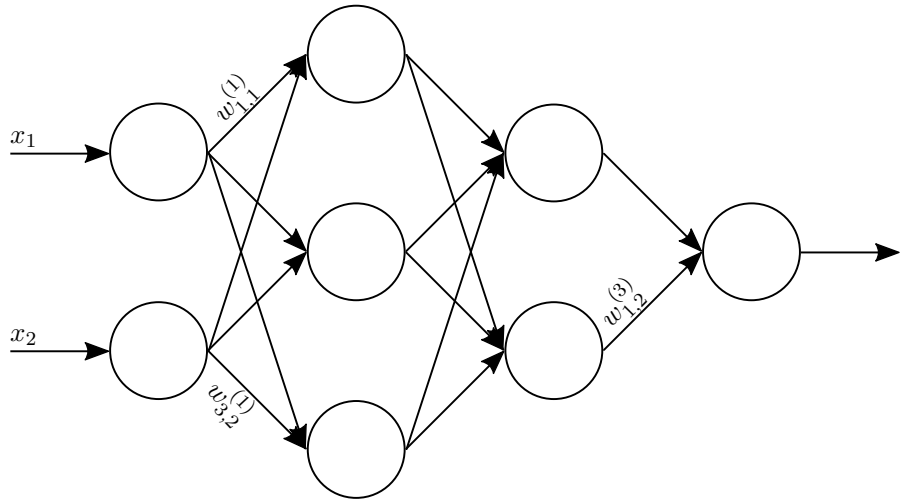


Figure 6: A scheme of a fully connected feed forward artificial neural network, with some exemplary weights to clarify the notation. The leftmost layer is the so called *input layer*. Its artificial neurons are a little different from the ones from the other layers, because the input is just passed through without calculations.

Similarly, the bias of the i-th artificial neuron in layer $k+1$ is called $b_i^{(k)}$. Actually, there is no bias for $k = 0$, because this corresponds to the input layer and nothing is calculated there. The vector of all biases of the n artificial neurons in layer k is called $\boldsymbol{b}^{(k)} \in \mathbb{R}^n$. The next step is to write the calculations taking place in the artificial neural network in a very compact way. Assume there are n neurons in layer k. For the j-th neuron of layer $k+1$, the output is called a_j^k and is calculated via

$$a_j^{(k)} = \phi((\sum_{i=1}^{n} w_{j,i}^{(k)} \cdot a_i^{(k-1)}) + b_j^{(k)}) \tag{2}$$

with $a_i^{(0)} = x_i$ being the i-th input. Similarly to the biases, the activations from all m artificial neurons in layer k are compressed to the vector $\boldsymbol{a}^k \in \mathbb{R}^m$. Hence, the above formula becomes

$$a_j^{(k)} = \phi(\langle W_{j,:}^{(k)}, \boldsymbol{a}^{(k-1)} \rangle + b_j^{(k)}) \tag{3}$$

where $W_{j,:}^{(k)}$ denotes the j-th row of $W^{(k)}$. Writing this out for the activation of every single artificial neuron from layer $k+1$ yields

$$\boldsymbol{a}^{(k)} = \overline{\phi}(W^{(k)} \cdot \boldsymbol{a}^{(k-1)} + \boldsymbol{b}^{(k)}) \tag{4}$$

where $\overline{\phi} : \mathbb{R}^m \to \mathbb{R}^m$ is given by applying ϕ componentwise. Equation 4 describes the mathematics taking place in a fully connected feed forward artificial neural network in

a short and elegant way. Since we chose a smooth activation function ϕ, the whole artificial neural network can be seen as a smooth function depending on the inputs, the weights and the biases. As mentioned earlier, the inputs will be given and hence fixed, so the weights and the biases are the parameters we want to optimize. How we do this is discussed in the next section.
To finish off this section, we will parallelize the description of a fully connected feed forward artificial neural network, depicting elegantly how even multiple outputs can be computed at once.
Let $X \in \mathbb{R}^{n \times m}$ denote the *input matrix*, where each column is one of the m input samples and each of the n rows corresponds to a *feature*. Hence the input layer consists of one artificial neuron for each feature. Now, instead of plugging one input at a time into Equation 4 on the preceding page, we denote $A^{(0)} = X$ and calculate

$$A^{(k)} = \overline{\phi}(W^{(k)} \cdot A^{(k-1)} + B^{(k)}) \tag{5}$$

where again $\overline{\phi} : \mathbb{R}^{l \times l} \to \mathbb{R}^{l \times l}$ is given by applying ϕ componentwise, l denotes the number of artificial neurons in layer $k+1$ and $B^{(k)} \in \mathbb{R}^{l \times p}$ is the matrix where each column is just $\boldsymbol{b}^{(k)}$ and p denotes the number of artificial neurons in layer k.

2.4 Gradient Descent and Mini Batch Stochastic Gradient Descent

In this section we will make clear how an artificial neural network is trained. First, as per usual for optimization problems, we need to clearly define what we want to optimize. Of course, we want to minimize the *error* that the artificial neural network makes, but still we need to make clear how to *measure* the error.
Let, as in the previous section, $X \in \mathbb{R}^{n \times m}$ denote the *input matrix*. The corresponding *outputs* are denoted $Y \in \mathbb{R}^{s \times m}$, where column i of Y is the output corresponding to the input given in column i of X. Y is calculated by repeatedly applying Equation 5 to X. Let $\overline{Y} \in \mathbb{R}^{s \times m}$ denote the *target outputs*, it stores the outputs we would like to get when sending X through the artificial neural network.
For simplification, let's start with only one output, hence let $s = 1$. We will choose the *mean squared error*, in the future abbreviated by MSE, to assess the quality of the artificial neural network. In the case $s = 1$, it is calculated using

$$\text{MSE} = \frac{1}{m} \|Y - \overline{Y}\|_2^2 \tag{6}$$

where $\| \cdot \|_2$ denotes the euclidean distance. When the artificial neural network approximates the potential energy surface, the output is indeed a single number and Equation 6 is used. Later, we will also construct an artificial neural network that approximates the forces on each atom for each coordinate direction, hence there are three outputs. In this case, Equation 6 is applied to all outputs separately and the results are summed up in the end, yielding once again a single real number as measurement of the quality of the artificial neural network. We call this the *cost function* or the *loss*, assigning to it the letter E for future reference. In the case of multiple outputs, the error terms for each output will be called *losses*, while the cost function will be the sum of all losses.

Next, we need to find an algorithm that changes the weights and biases to decrease the error over time. There are different approaches to do this, in this work we stick to the classical one, the *gradient descent* and the *stochastic gradient descent*.
We already made clear that the *cost function* is a real-valued function depending on all the weights and all the biases, because we regard the inputs and outputs as fixed. Hence, the *gradient* of the cost function with respect to all the weights and all the biases can be calculated. Now we recall from analysis that the gradient is a vector pointing into the direction of the greatest rate of increase of a function, see for example [5]. Hence the negative gradient points into the direction of the greatest decrease of a function. We start by fixing one input sample, because working with the mean squared error and huge datasets might lead to memory issues and might also slow down learning a lot [6]. The equation for the cost function for a single input $X_{:,i}$, $i \in [m]$, becomes

$$\mathrm{E} = (Y_i - \overline{Y}_i)^2 \tag{7}$$

and is also called the *quadratic cost function*. We hope that taking a little step into the direction of the greatest decrease of the cost function E will indeed decrease the cost and hence improve the quality of our artificial neural network. Repeating this multiple times for all input samples one after another should gradually increase the cost until it is as small as desired.
Going through all input samples once is called an *epoch*. Training will consist of multiple epochs. Equation 8 shows how the gradient descent algorithm updates some weight w. Here, η, a real number, usually between 0 and 1, denotes the learning rate. Note that the parameters where the partial derivative is evaluated are ommited, they are the current weights and the current biases.

$$w \coloneqq w - \eta \cdot \frac{\partial E}{\partial w} \tag{8}$$

Of course, there is no guarantee that this algorithm works out, a simple counterexample is given in Figure 7 on the next page. Generally speaking, so called *hyper-parameters* like the length of the step into the direction of the negative gradient, below called *learning rate*, must be chosen carefully and only some experimentation will lead to success.
A useful modification of the gradient descent algorithm, with the idea to speed up learning [2], is the so called *mini batch stochastic gradient descent* algorithm. We don't use the data set as a whole at once for training, because it might be too large, but taking one sample at a time is also not necessary. We take a fixed number of, say, $b \in \mathbb{N}$ input samples chosen uniformly at random from all input samples and perform the gradient descent algorithm with Equation 8. Normally, we have b approximately between 8 and 128. Furthermore, according to [7], b should be a power of 2 to speed up calculations. Afterwards, we chose b samples, uniformly at random from the remaining training samples, and so on, until all input samples are being dealt with and an epoch is completed. In this case, b is called the *batch size* or *mini batch size*.
Note that the gradient of Equation 8 applied to a small set of randomly chosen samples might well differ from the gradient one would obtain if taking all samples at once. Hence

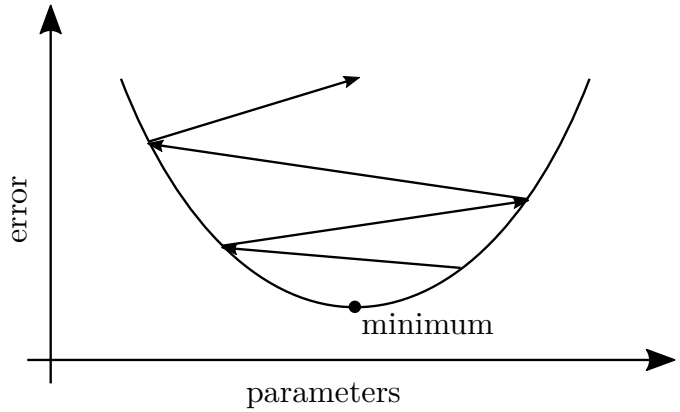


Figure 7: When the length of each step is chosen too large, one might go into the right direction, but overshoot the minimum. If, on the other hand, the step size would be chosen too small, learning could take much longer than necessary. Finding the right step length or *learning rate* is therefore an important task.

we are not guaranteed to make a step into the direction of the *greatest* decrease, but still we will most likely decrease the cost and, moreover, we do decrease the cost $\frac{m}{b}$ times as often as if we would use all samples at once. This factor is usually larger or even a lot larger than 100.
Mini batch stochastic gradient descent is also the algorithm used to train the artificial neural network in the accompanying software. There, an implementation of the so called *Adagrad* algorithm (from *Ada*ptive *Gra*dient) is used. Basically, the learning rate is tuned individually for each parameter, based on the current progress, to improve learning. Details can be found in [6] or [20].

2.5 Training data, validation data and test data

In this section we will deal with another very important topic of artificial neural networks, namely *overfitting*. Overfitting takes place when the artificial neural network gets better and better at the presented inputs used for training at the cost of getting worse and worse on unseen data which is used for testing. Figure 8 on the following page illustrates this point.
We separate the initial data set randomly into two parts, a *training set* and a *validation set*. Normally the training set is about three to four times as large as the validation set. This is done in general to assess the quality of a fitted artificial neural network, but it is especially helpful when trying to find out if and when overfitting takes place. We now train the artificial neural network using the training data set. Additionaly to recording the loss on the training data set after each epoch, we will also record the error on the validation data as a generalization measurement of the artificial neural network, in other words as a measurement of how good the network generalizes and hence how good it performs on unseen data.

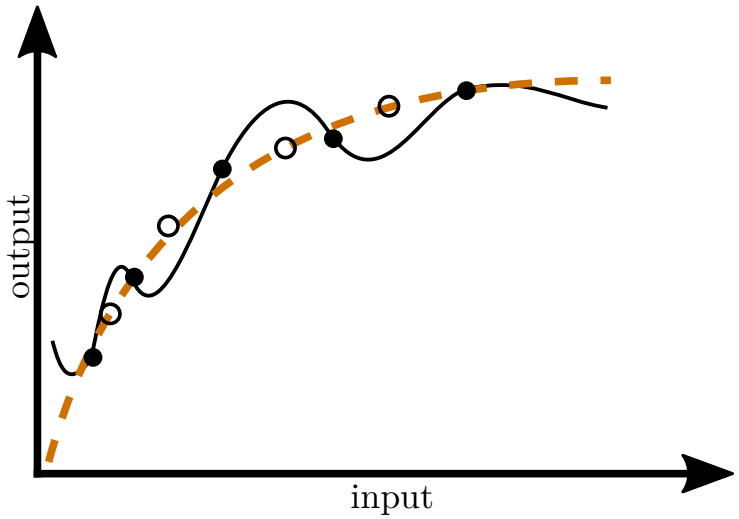


Figure 8: The filled circles denote the training data, the not filled ones the validation data. Although the black line goes perfectly through the training data points, it appears to be much worse on the validation data. The dotted line is a much better approximation, although it not even hits all the training data samples perfectly.

At the beginning of the training process, both errors will decrease, while at some point, the error on the training data will still continue to decrease, while the error on the validation set increases again. This is visualized in Figure 9 on the next page. From this point on the artificial network might still improve on the training data set, but since we want to later use it on unseen data, further training is not benficial at all. Moreover, the input data usually is a little noisy itself. Hence, fitting an artificial neural network that perfectly predicts the training data does not mean that it has become an artificial neural network perfectly fitted for solving the problem it was designed to solve, because it most possibly is too sensible to noise in the input data. The technique of interrupting the training process when overfitting is detected is called *early stopping*.
One point is important to mention. Assume the validation set has been used to detect when to stop the training process and assume moreover that one wants to measure the quality of the trained artificial neural network. Using the validation data is not correct here, because the validation data has already been used to decide when to stop the training, hence it is very likely that we overestimate the quality of the neural network [7]. So when we want to measure the performance of the artificial neural network after the training process, we need to divide the whole data set into a training data set, a validation data set and a *test data set*. The test data set is then used after the training to determine the quality of the trained artificial neural network. When the number of samples is not *very* large, a usual split of the data would be 60% for the training data

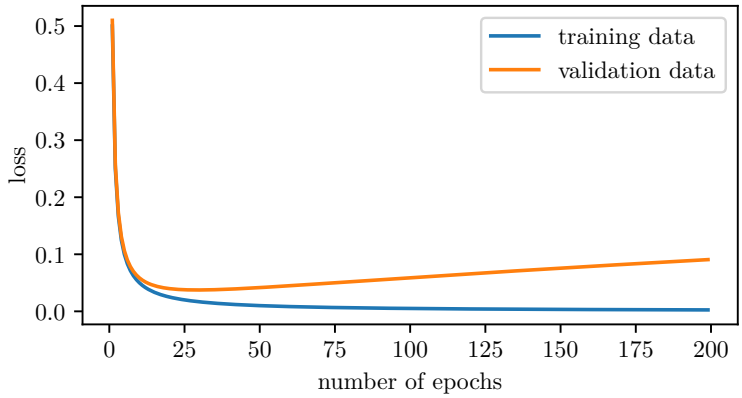


Figure 9: Overfitting starts at the point where the error on the validation data increases.

and 20% for the test and the validation data, each [7].
We have now covered all machine learning topics that are important for this thesis. The next chapter will give an introduction to molecular dynamics, which is not the primary topic of the thesis and hence will presented shortly. Afterwards, we will look at the software developed within the work on this thesis in detail.

3 Foundations of Molecular Dynamics

This section will give a very brief introduction to *Molecular Dynamics*, a type of so called *n-body simulations*. It's not a major part of this work, so we will not go into too much depth, but still it rounds off the whole material presented here.
Applications of Molecular Dynamics include investigations of reactions on surfaces with the idea of developing for example new catalysts [8] or research about high temperature superconductivity, with the idea of discovering high temperature superconductors [16], that is a material with zero electircal resistance, ideally at room temperature conditions. These are only two applications, among many others.
Assume one has a cluster of atoms and wants to simulate what happens with this cluster over time. When one knows all forces acting on the nuclei of all the atoms belonging to the atomic cluster at all times, one could use Newton's equations of motion to simulate a tiny time step and then again look at all the forces and repeat this process over and over again. This algorithm is called a *Molecular Dynamics simulation*.
To obtain the forces, it is sufficient to know the so called *Potential Energy Surface* at all time. The *Potential Energy Surface* maps any state of the cluster to the energy of the system. Now the forces acting on the nuclei of the atoms belonging to the atomic cluster can be obtained as partial derivatives of the energy with respect to the coordinate directions.
The problem with this approach is that the computation of the Potential Energy Surface with a high precision is very expensive, because first principle quantum mechanic calculations are used. Moreover, the number of time steps needed to get meaningful results ist often very large, most of the time larger than 10^6 [8].
Many times it is not possible to obtain the Potential Energy Surface in a closed analytical form at all [8]. Therefore one could calculate a discrete Potential Energy Surface, hence calculate the energy of the system for a finite set of points, and then *interpolate* in between to approximate the Potential Energy Surface, with the goal of obtaining about the same precision as the quantum mechanical calculations.
Evaluating a fitted artificial neural network for the interpolation of the Potential Energy Surface is very fast. Furthermore, to obtain enough data to fit an artificial neural network, about 10^4 data points are enough, dramatically increasing the number of first principle quantum mechanic calculations needed.
The interpolation step is exactly what we deal with in this thesis. One thing that is very beneficial for the application of machine learning techniques is that we can calculate exactly the sets of data that we need. This means that we can have data sets nicely spread across the space of possible features, in contrast to many other applications, where for example huge amounts of data had to be collected, and still some desired combination of features might be missing.
Somehow earlier applications of Machine Learning and artificial neural networks to this problem often dealt with the approximation of the Potential Energy Surface [8]. From this approximation, the forces where calculated as the negative of the derivatives of the energy, as explained above [17].
What we also did within the work on this thesis is directly approximating the forces

using a slightly modified artificial neural network. There is no extra computational cost for differentiating the energy, and moreover, the error might be smaller because if the Potential Energy Surface is only an approximation, it's derivatives might turn out to have even larger errors.

4 The software developed in this thesis

In this chapter we will take a closer look at an implementation of the ideas and algorithms we dealt with above. The software is written in the programming language *Python* [9], using the *Numpy* [10], *Theano* [12], *Keras* [13], *SciPy* [11] and *Matplotlib* [14] libraries. The respective software versions can be found in the introduction.
We first implement an artificial neural network for the classical purpose - approximating the Potential Energy Surface. Afterwards, we will discuss how to define and train multiple artificial neural networks, one for each atom type, that directly approximate the forces. At this stage we will also go through the feature calculation process. At the very end there will be a short discussion about deep learning and how convolutional layers could be benefitial for our network architecture.

4.1 Working with energy as output

In this section we will go through the first version of the software. We will mention the format of the input, the architecture of the artificial neural network and the so called *hyperparameters* that can be chosen to improve the performance of it.

4.1.1 Preparing the input data

The raw input data contains information about the coordinates of the atoms as well as their charge and the forces that act on their nuclei. A software was applied to calculate the features used to fit the artificial neural network as well as to calculate the target energy value of each frame. In the section where we discuss the forces as target outputs, we will see in more details how the features are calculated, because this part is then included in the second version of the software. But for now, let's assume that the input for our computer program is a file in the following format: First, there is one line containing a single number that tells how many atoms belong to the next frame. Afterwards, there is one line for each atom of the frame. Every such line consists of an integer, determining the atomic number of this atom, followed by floating point numbers, one for each feature. All numbers in a line are seperated by one or more whitespaces. After the lines for the atoms, it follows one line containing information about the energy of this frame. Afterwards, this pattern repeats, starting again with a line that consists of one single number that tells the number of atoms of the next frame and so on. It is important to mention that every atom of one type must have the same number of features, but different atom types may have different numbers of features. Also, the number of atoms may vary from frame to frame and the atoms do not need to be sorted by atom type in the input file.
The first part of the software consists of preprocessing the input. Imagine some features lie in the range of -0.1 to 0.1, but another feature lies in the range of -100 to 100. In this case, the artificial neural network would be more influenced by the features with larger values, which is undesirable. Figure 10 on the following page underlines this problem. To unify the inputs, all features are first shifted, so that their mean is zero,

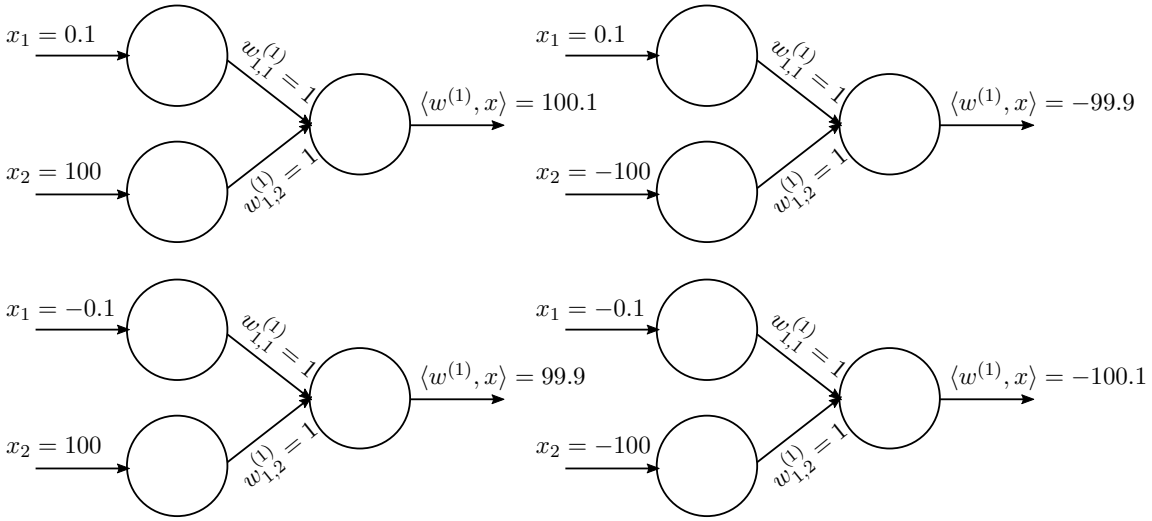


Figure 10: The drawing underlines that x_1 has almost no influence on the activation of the artificial neuron in layer two, compared to x_2, just because x_2 has much larger values. Of course, if $w^{(1)}_{1,1}$ would eventually become large enough to compensate for this, the effect would no longer harm learning. However, this takes time and not scaling the features beforehand would slow down learning. The same holds true for shifting and the bias, for simplicity the bias is assumed to be 0 and the activation is chosen to be the identity in the depicted example.

and afterwards scaled, so that they are in about the same range. More precisely, they are scaled so that the distance from the smallest to the largest sample is about 1. The target energy values are shifted and scaled in the same way. Note that the shifting and scaling numbers are stored so later, the obtained loss from the fitted artificial neural network can be transformed backwards to obtain the correct error measurement.
Also, the input data is grouped as *frame objects*. Every instance of the frame class has a list of Numpy arrays, each corresponding to the input of one atom type, as well as a single target output number, the energy value of the corresponding atomic cluster. Each row of the input matrices corresponds to one single atom, whereas the different columns correspond to the different features. In this case, one *sample* for version one of the artificial neural network consists of one frame, with all data for all the atoms of this frame as input and the Potential Energy value of the atomic cluster as corresponding target output value. Later, in version two of the software, we will use one single atom as one sample instead, dramatically increasing the number of samples.
In the next subsection, we will look at the architecture of our artificial neural network.

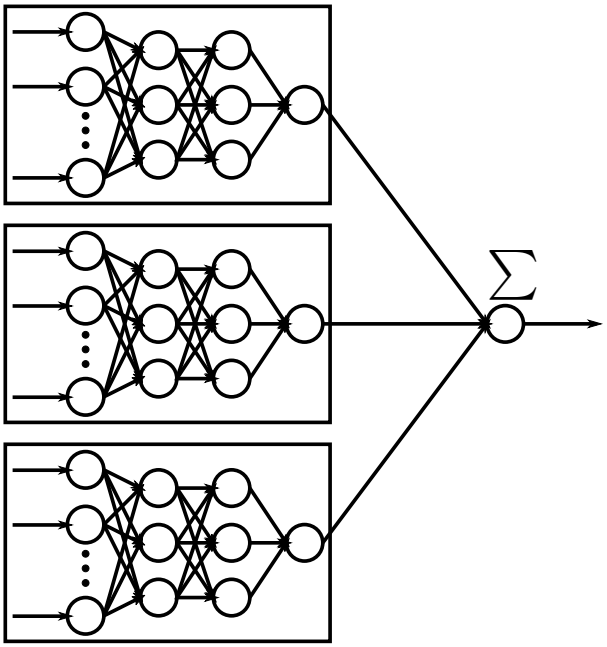


Figure 11: The architecture of the artificial neural network for approximating the energy. Each rectangle contains the scheme of a fully connected feed forward artificial neural network, there is one of these for every atom type. Their outputs are summed up at the very end to obtain the energy value for the frame.

4.1.2 Creating and fitting a model

We will start this subsection by explaining the architecture of the artificial neural network in detail. We have already discussed how a fully connected feed forward artificial neural network works. The network architecture chosen here is not quite a fully connected feed forward artificial neural network. However, it consists of multiple such parts that are glued together and form the network architecture.
More precisely, we will have one fully connected feed forward artificial neural network for each atom type. After all inputs are sent through their specific artificial neural network, the outputs can be interpreted as atomwise energy contributions. Hence, at the end all these outputs have to be summed up to calculate the predicted energy value of the frame. Figure 11 illustrates the explained architecture.
Now we will care about how exactly the feed forward calculations work. The picture seems to indicate that every frame consists of only one atom of each atom type. But this is usually not the case. In fact, for the crucial step, we will go back to Equation 5 on page 8. It describes how multiple inputs are sent through the artificial neural network in parallel. Instead of one input vector, that would lead to one output number, we send a whole matrix of input vectors through each partial artificial neural network to obtain a vector of outputs, where each entry of the vector corresponds to the energy contribution of a single atom. Summing up all entries of this output vector yields the total energy

contribution for all atoms of one type. In a second summation step, these total energy contributions are summed up again to obtain the predicted energy value of the whole frame.
Still though, for parallelization, we would like to send multiple samples at a time through the artificial neural network. Remember that one frame is one sample in this version of the software. Hence, the input for the network will be one 3-tensor for each atom type, with the Numpy, Theano and Keras libraries automatically expanding the parallelization of the calculations one dimension further, analogously to how we parallelized from dimension one to dimension two in the section about artificial neural networks. This is called *broadcasting* and is very useful for speeding up caluclation processes. Listing 1 shows how to define the network described above, using the Keras library. The class *My_layer* defines a layer that sums up it's inputs along a certain axis, as described above, to obtain the energy contribution for one atom type, given all energy contributions of the atoms of this type. The variable *params* is a dictionary containing all parameters provided from the user such as the specific number of atom types, the number of features for each atom type or the numbers of artificial neurons in the hidden layers.

```
models = []
inputs = []
for i in range(params['number_atom_types']):
    x = Input(shape=(None,params['number_features_per_type'][i]),
        name='Features_of_atoms_of_type_'+str(i+1))

    inputs.append(x)

    j = 1
    for layer in params['layers'][:-1]:

        # Defining activation functions and initializing weights and
        # biases for the hidden layers, and giving names.
        x = Dense(layer, activation = params['activation_function'],
                  kernel_initializer= params['weight_initial'],
                  bias_initializer = params['bias_initial'],
                  name='Hidden_layer_'+str(j)+'_for_type_'+str(i+1))(x)
        j = j + 1

    # Defining the activation function and initializing weights and biases
    # for the output layer, and giving a name.
    x = Dense(params['layers'][-1], activation = 'linear',
              kernel_initializer=params['weight_initial'],
              bias_initializer=params['bias_initial'],
              name='Energy_of_atoms_of_type_'+str(i+1))(x)

    # Defining the layer that adds up all energy contributions of the
    # single atoms.
    x = My_layer(1,name='Total_energy_of_atoms_of_type'+str(i+1))(x)
    models.append(x)

# Building the netwok model by combining all above defined layers.
```

```
33
34 final_model = Add(name='Neural_network_energy')(models)
35 model = Model(inputs = inputs, outputs = final_model)
```

Listing 1: This is the code that creates the artificial neural network for version one of the software, that is, with the goal of approximating the energy. Choosing *None* as the first parameter of the shape of the input tensor in line 4 means that this number might vary and hence ensures that the number of atoms of each type does not have to be the same in different frames.

Now that the model is defined, it has to be compiled and afterwards fitted to the training data. This is easily done using the following few lines of code shown in Listing 2.

```
1 model.compile(optimizer = optimizers.Adagrad(lr = params['learning_rate'],
2                 epsilon = params['epsilon'],
3                 decay = params['decay'],
4                 clipvalue = params['clipvalue']), loss = params['loss'])
5
6 history = model.fit(data_x,data_y,epochs = params['epochs'],
7             batch_size = params['mini_batch_size'],
8             validation_split = params['validation_fraction'])
```

Listing 2: In Keras, one first has to define a model, which was done above, afterwards compile it and finally fit it to the training data. The history variable stores all information necessary to plot the training progress, like the loss of each epoch.

In the next section, we will look at the results from a concrete example, namely an atomic cluster consisting of H_2S (*hydrogen sulfide*) molecules surrounded by He (*helium*) atoms.

4.1.3 Some exemplary results

When testing the implementation of version one of the software, we worked with data coming from a simluation using 155 hydrogen sulfide molecules, surrounded by 500 helium atoms. Hence we have 155 sulfur atoms, 310 hydrogen atoms and 500 helium atoms, making a total of 965 atoms, in each frame. The data consisted of 349 frames. Hydrogen sulfide is a *superconductor* at very high temperatures, namely about 203K, when put under high pressure [16]. A superconductor is a material with zero eletrical resistance and hence it is a very interesting area of research to find superconductors at feasible conditions. Putting the hydrogen sulfide molecules inside a sphere of helium atoms and decreasing the radius of this helium sphere simulates the high pressure.
For this test we chose the following network architecture parameters: Every network that was contained in a rectangle in Figure 11 on page 17 has two hidden layers with 15 artificial neurons in each of them. The activation function for the artificial neurons in

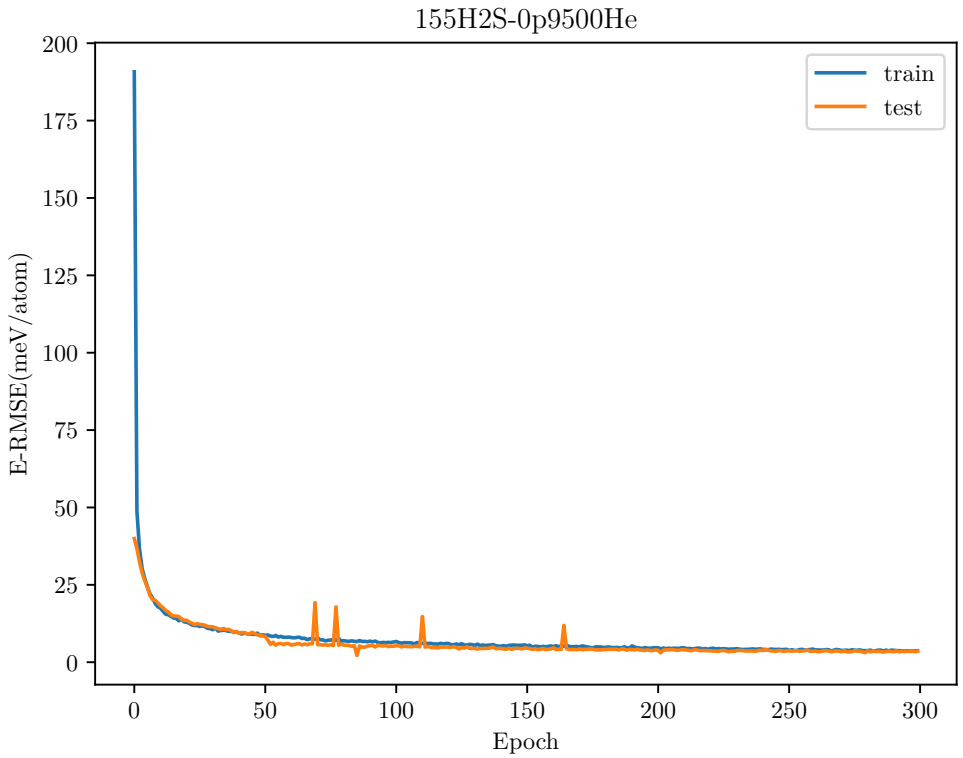


Figure 12: The train and test error decrease to about 3.6 meV/atom. One can see in the figure that no overfitting is taking place since the test error is never increasing towards the end of the training process.

these hidden layers is

$$\phi(z) = 1.7159 \cdot \tanh\left(\frac{2}{3}z\right) + 0.001z$$

while for the summation parts, no activation function is needed, which could also be written as the identity activation function. The constant factors in ϕ are chosen according to [18].

Figure 12 shows the result. The loss on the test data as well as on the training data, measured as *rooted mean squared error*, short RMSE, with milli electron volt per atom (meV/atom) as unit, decreases to about 3.6 meV/atom after 300 epochs. This is comparable to other state of the art results, see for example [17]. In that work, the best RMSE obtained is 13 meV/atom, but of course a different atomic cluster has been used there, so only the order of magnitude of the results is comparable. Figure 13 on the next page shows the same result on a double logarithmic scale. The slope of the error curves is very similar to that of $\sqrt{x}$. If for the moment we assumed that the training process is convergent to a desirable result, this would indicate an order of convergence of 0.5.

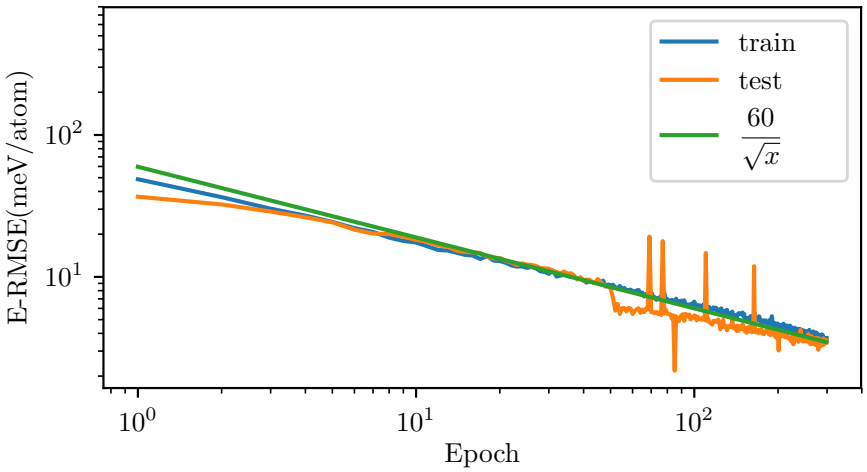


Figure 13: This is Figure 12 on the preceding page on a double logarithmic scale to get an idea of the order of the learning speed. For comparison, the function $f(x) = \frac{60}{\sqrt{x}}$ is also shown.

To make clear why we stopped after 300 epochs, Figure 14 on the following page shows the same run for 1500 epochs. One can clearly see that from about epoch 350 onwards, overfitting takes place and the test loss is larger than the training loss. Although the RMSE on the training data decreases to approximately 1.5 meV/atom, which would be a remarkable improvement compared to the result from above, it should not be overestimated because the RMSE on the test data is still about 2.4 meV/atom and thus larger by some margin.
Figure 15 on page 23 depicts this on a double logarithmic scale. One can clearly see the test loss increasing at a certain point around epoch 350 this picture as well. However, the loss on the training data continues decreasing with a similar slope as the function $f(x) = \frac{1}{\sqrt{x}}$, again indicating an "order of convergence" of 0.5.
The data shifting and scaling process and the fitting process afterwards took altogether 56 seconds for 300 epochs on a computer using an *Intel Xeon 1231v3* CPU (4 times 3.4 GHz) and a *MSI Nvidia GForce GTX 1070 Gaming 8G* GPU, which is the same computer system that is described in more detail in the introduction.

4.2 Working with forces as output

In this section, we will take a look at version two of the software. In this case, we directly try to approximate the forces acting on the nuclei of the atoms. The input and target output data has again been obtained by first principle quantum mechanic calculations, which are not part of this thesis. However, for this version of the software, we will take the data consisting of only the x- y- and z-coordinates of the nuclei and go through the feature calculation process as well. Afterwards, we will talk about the artificial neural network model architecture, which is a little different, namely simpler, than in

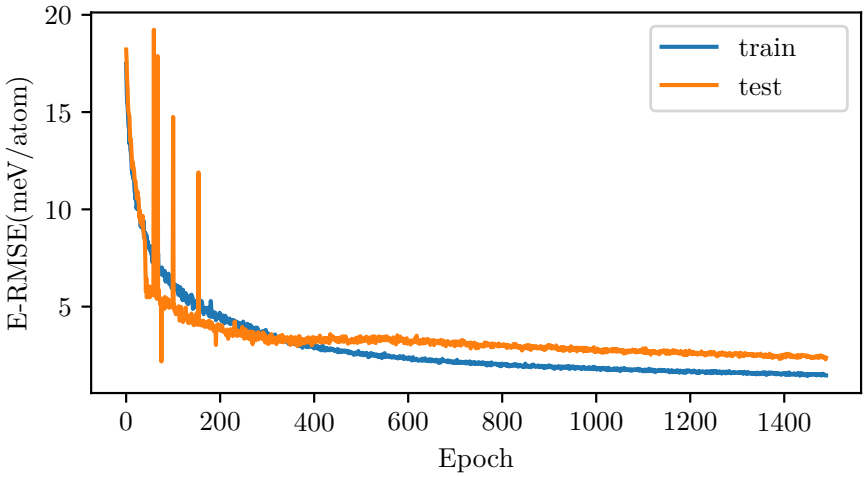


Figure 14: This Figure depicts the above run from Figure 12 on page 20, but extended to 1500 epochs. Overfitting can be detected after about 350 epochs, when the loss on the test data starts increasing and exceeds the loss on the training data.

version one from the previous section, because we can fit the networks for each atom type separately. In the last subsection, we will go through the basics of *deep learning* and especially *convolutional layers* and talk about how the idea of convolutional layers could be implemented to improve version two of the software.

4.2.1 Feature calculation

In this subsection we discuss the generation of the features for the artificial neural networks in detail. The software that was used to obtain the features for version one from above does the same calculations, with the exception that it also takes into account the contribution of each atom to the energy of the frame and generates the energy value of the atomic cluster.

Input data format

At first, we will take a look at the format of the input data. Listing 3 on the following page shows some exemplary lines of an input file.

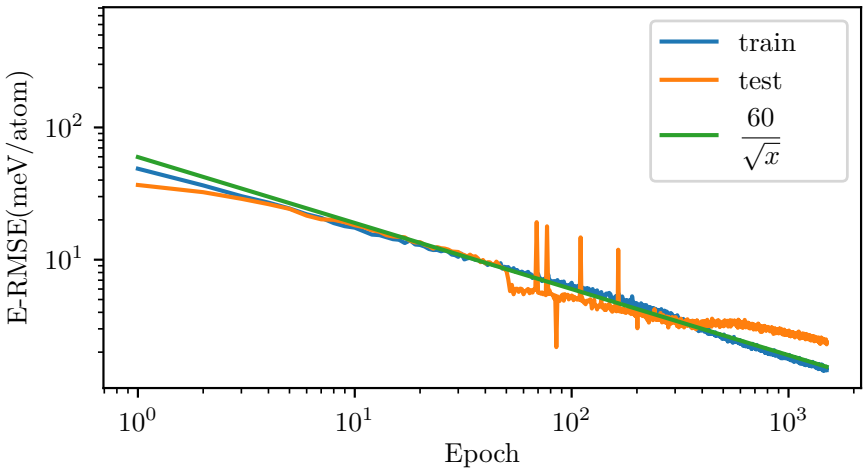


Figure 15: This is Figure 14 on the preceding page on a double logarithmic scale. One can once again see the overfitting taking place after about 350 epochs. One can also clearly see that the slope of the graph identifying the loss on the training data continues to be approximately the same as $\frac{1}{\sqrt{x}}$.

```
begin
 atom    9.59   -9.53   -10.59 H    0.34   0.00    0.017   -0.003   -0.002
 atom    8.39   -8.61    -9.54 O   -0.70   0.00   -0.014   -0.013   -0.017
 atom    6.62   -8.85   -10.14 H    0.38   0.00   -0.010   -0.0002   0.001
 ...
energy     -14413.3599431157
charge    0.000000
end
begin
 atom    9.58   -9.58   -10.60 H    0.34   0.00    0.010    0.0003   0.002
...
```

Listing 3: This is a short part of an input file to illustrate the format of the input data. All data for one frame lies between a *begin* and an *end*. All atoms are in a line starting with the keyword *atom* and at the end of every frame there are two additional lines, one for the energy of the system and one for the charge. Note that the real input data is given with much more precision, but these numbers have been cut off here to guarantee a formatting that is clear for the reader.

A line beginning with the keyword *atom* contains all necessary information for one atom. The first three numbers are the x-, y- and z-coordinates of it. It follows the atomic symbol. The next two numbers, the charge and energy, are not needed and hence skipped. The last three numbers are the forces acting on the nucleus of the atom,

again in x-, y- and z-coordinate direction. The lines starting with *energy* or *charge* are skipped in this version of the software, while the lines starting with *begin* oder *end* are used to determine when a frame is over.

Invariances

Of course, one could use the atom coordinates as features for an artificial neural network, but there is a major drawback: The forces acting on the nuclei are invariant under translation, and a rotated input cluster would result in rotated forces. We would like to keep these invariances. Moreover, the forces acting on the nuclei of the atoms are from a physical point of view not really depending on the coordinates in space, they are rather depending on other atoms nearby.
That's why we calculate so called *internal features*. For a very deep discussion about how to calculate the features in a sensible way, see [17]. In that work, the goal was first to construct an artificial neural network that approximates the Potential Energy Surface and then in a second step to obtain the forces acting on each nucleus via differentiating the Potential Energy Surface. We want to directly approximate the forces, but we will take this aforementioned work as some kind of a starting point for forces approximation and will suggest improvements. Unlike the forces, the energy of the whole system is indeed invariant under rotation. Hence, in the above mentioned work, features where created that are invariant both under translation as well as under rotation of the atomic cluster. Therefore, first for each atom a sphere of a certain radius Rc centered at this atom's nucleus was constructed. In a second step, all other atoms inside the sphere are taken into consideration, in other words, the radius of the sphere is a *cutoff radius* and atoms outside the sphere are assumed to have only negligible impact on the considered atom. It turns out that from a physical end chemical point of view, it makes sense to describe the interaction between the atoms inside the sphere and the atom at its centre in terms of so called *symmetry functions*.

Actual calculations

Given the fixed cutoff radius Rc as well as the distance $d < Rc$ from one atom inside the sphere to the atom at it's center, the symmetry function f_c that we chose to take from [17] is calculated via

$$f_c(d) = \frac{1}{2} \cdot \left(\cos\left(\frac{\pi \cdot d}{Rc} \right) + 1 \right) \quad . \tag{9}$$

This equation is applied to all atoms inside the sphere, while the results for all atoms of one type are grouped together, each. In the last step, these contributions from the single atoms are summed up for each atom type, respectively, and then form one feature of the atom we originally constructed the sphere around. We will do the summation according to Equation 6 from [17], so the feature F is calculated using

$$F = \sum_j e^{-\eta \cdot d_{i,j}^2} \cdot f_c(d_{i,j}) \tag{10}$$

where j goes through all atoms inside the sphere of one type, $d_{i,j}$ is the distance from the atom at the center of the sphere to atom j and η is a parameter that we are free to choose, it will determine the shape of the Gaussian term $e^{-\eta \cdot d_{i,j}^2}$. In fact, we will use different values for η to calculate multiple features. Equation 10 on the previous page is applied for the different atom types, one after another, and different values for η. This yields a total of the number of atom types multiplied by the number of η values many features for every atom. Listing 4 shows the implementation of the above equations in Python code using the Numpy and the SciPy libraries. The function *function_interaction_atoms* ist implemented so that multiple distances can be given as input and the calculations are then automatically parallelized by the Numpy library.
These features are now used as input for the artificial neural network.

```
def modify_data(points_tf, Rc, etas, number_atom_types, function_type):
    distances = pdist(points_tf[:,1:])
    number_atoms = points_tf.shape[0]
    # The distance from point i to point j is stored in
    # distances[get_pdist_index(i,j, number_atoms)]
    # (see function at the very bottom)

    features_by_atom = []
    for i in range(number_atoms):
        distances_in_sphere = [[] for _ in range(number_atom_types)]
        for j in range(number_atoms):
            if i!=j and distances[get_pdist_index(i,j, number_atoms)] < Rc:
                # now we have the case that another atom in the sphere is found
                distances_in_sphere[int(np.rint(points_tf[j,0]))].append(
                        distances[get_pdist_index(i,j, number_atoms)])
        # For this atom i, all atoms in the sphere have been detected and the
        # distances are put into seperate lists for each atom type.
        # Now the features can be calculated

        f_cs = [function_interaction_atoms(function_type,Rc,
                                           np.array(distances_in_sphere[k]))
                for k in range(number_atom_types)]
        # This implements equation 8, see below.
        features_this_atom = []
        for k in range(number_atom_types):
            features_this_atom.append([np.sum(np.exp(-eta*f_cs[k])) for eta in etas])
            # This implements equation 9
        features_this_atom = np.array(features_this_atom)
        features_by_atom.append(features_this_atom.flatten())
    return features_by_atom

def function_interaction_atoms(function_type,Rc,distances):
    if function_type == 1:
        return 0.5*(np.cos((np.math.pi*distances)/Rc) +1.0)
        # This is equation 8.
    if function_type == 2:
        return np.tanh(0.5*np.math.pi*(distances/Rc))
        # Another possible symmetry function mentioned in [17].
```

```
39      print("illegal_function_type,_0_returned")
40      return 0
41
42  def get_pdist_index(i1,j1, number_atoms):
43      index = 0
44      i = np.minimum(i1,j1)
45      j = np.maximum(i1,j1)
46      for c1 in range(i):
47          index += (number_atoms - (c1+1))
48      index += (j-i-1)
49      return index
```

Listing 4: This code depicts the calculation of the features. The function *scipy.spatial.distance.pdist* from the SciPy library is used, it stores all distances between the points in a *compressed distance matrix*, which is actually just a vector. The function *get_pdist_index* at the bottom shows how to index its entries correctly. The variable *points_tf* contains all information about the atom coordinates from this frame. It also stores the atomic numbers, so it has to be indexed in a way that they are not used for distance calculations in the second line of the code.

Ellipsoids instead of spheres

We could just take this method of feature calculation and put the forces as target outputs, instead of the energy. In fact, this is what we started with. The results are presented in the next section, but for now, we will take a look at a conceptional problem with this approach. As mentioned before, the calculated features are invariant under translation as well as rotation of the atomic cluster. However, the x-, y- and z-coordinates of the force vector are *not* invariant under rotation.
At first we will make clear why this is not a problem in [17]. There, the force in each coordinate direction is obtained as partial derivative of the Potential Energy Surface. And while the calculated features are the same for two atomic clusters where one is only a rotation of the other, the partial derivatives are not and hence there is no need to worry.
But for our case, assume that there are two given frames, where one is the rotation of the other. Now, the features calculated for these two frames are exactly the same. But, the target outputs, namely the forces, are not. So in this case we would have twice the exact same input pattern for training the artificial neural network, but with different target outputs. This is of course undesirable.
To fix this, we used *ellipsoids* instead of spheres. In fact, we tripled the number of features by doing the above calculation three times, once with the sphere replaced by an ellipsoid stretched along the x-axes, once with the sphere replaced by an ellipsoid stretched along the y-axes and finally once with the sphere replaced by an ellipsoid stretched along the z-axes.

The geometric reasoning behind taking ellipsoids stretched along the coordinate axes is the following: Assume that there are two frames, where the second one is only a version of the first one rotated by 90 degrees around a coordinate axes. In this case, the features would be the same, but swapped in positions, while the forces are also the same, but swapped in positions in the exact same way. So the features and the forces now behave similarly when rotating the atomic cluster.
As we will see in the next section, implementing the usage of the ellipsoids decreased the error only by a small amount. Calculating the features takes a little longer in that case, which is mostly based on the fact that the feature calculation based on the ellipsoids is not yet fully optimized in terms of speed. The time needed to fit the artificial neural network is about the same in both versions. Listing 5 gives the major difference compared to the calculations using the code in Listing 4 on page 25.

```
distances = [pdist(points_tf[:,1:],dx),
             pdist(points_tf[:,1:],dy),
             pdist(points_tf[:,1:],dz)]
# Different distances, corresponding to ellipsoids strechted
# along each of the coordinate axes.

# -- These are the distance functions used above.
def dx(u,v):
    return np.sqrt((((u-v)/np.array([1.5,1,1]))**2).sum())

def dy(u,v):
    return np.sqrt((((u-v)/np.array([1,1.5,1]))**2).sum())

def dz(u,v):
    return np.sqrt((((u-v)/np.array([1,1,1.5]))**2).sum())
```

Listing 5: This is the difference in the feature calculation when using ellipsoids instead of spheress. The *scipy.spatial.distance.pdist* function allows to choose a custom function for distance calculation. This allows the implementation of the ellipsoids replacing the spheres.

Other geometric objects

In crystallography, where the atomic clusters are given in crystalline structures, it might be much more convenient to not use smooth geometric objects at all, but rather polyhedra like an octahedron. It should be mentioned that when the features are approximated as derivatives of the energy, it is important that all functions that are used to calculate the features are differentiable, so that later the forces can be obtained. However, with our approach, that is when the forces are approximated directly, no differentiability of the functions that are used to generate the features is needed anymore. This means that with some knowledge of the structure of the observed material, non smooth symmetry

functions as well as non smooth geometrical objects for determining the cut off region might further improve results.

4.2.2 Creating and fitting the models

The sytem we use for testing in this section consists of approximately 9600 frames. In each frame, there are Hydrogen, Oxygen and Aluminium atoms, their number is varying, but altogether there are approximately 200 atoms each frame.
Because the force—in contrast to the energy of the whole cluster—is given for every single atom, we will regard every atom as one sample and we will construct one artificial neural network for each atom type.
The construction of the neural networks for each atom is similar to the creation of the neural network we used for approximating the energy. Listing 6 shows the code that was used to create, compile and fit the models. As in the sections above, the Keras library is used again.

```
# Construct one model for each atom type
models = []
histories = []
for i in range(len(params['atom_types'])):
    # -- INPUT LAYER --
    x = Input(shape = (len(features[i][0]),))
    inp = x # store a copy for later use
    # -- -- -- -- -- --

    # -- HIDDEN LAYERS ACCORDING TO THE PARAMTER LAYERS --
    for layer in params['layers']:
        x = Dense(layer, activation = params['activation_function'],
            kernel_initializer = params['weight_initial'],
            bias_initializer = params['bias_initial'])(x)
    # -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

    # -- OUTPUT LAYER --
    x = Dense(len(forces), activation = 'linear',
        kernel_initializer = params['weight_initial'],
        bias_initializer = params['bias_initial'])(x)
    # -- -- -- -- -- ---

    # -- MAKE AND COMPILE MODEL --
    model = Model(inputs = inp, outputs = x)
    model.compile(optimizer = optimizers.Adagrad(lr = params['learning_rate'],
                        epsilon = params['epsilon'],
                        decay = params['decay'],
                        clipvalue = params['clipvalue']), loss = params['loss'])

    models.append(model)
    # -- -- -- -- -- -- -- -- -- --

    # -- FIT THE MODEL --
```

```
34      histories.append(model.fit(features[i],forces[i],epochs=params['epochs'],
35                  batch_size=params['mini_batch_size'],
36                  validation_split = params['validation_fraction']))
37      # -- -- -- -- -- -- -
```

Listing 6: This code shows the creation of one network for each atom type as well as the fitting process.

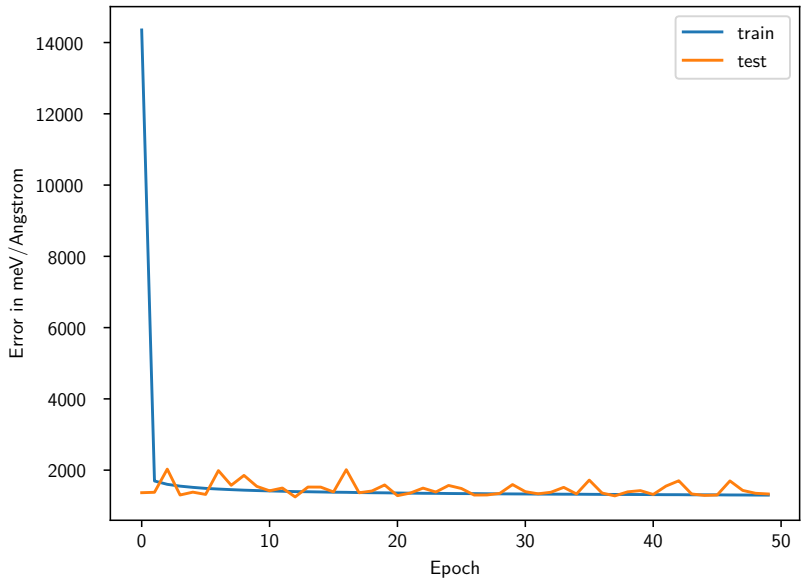
In the following, we will first talk about the approach using spheres and the corresponding results. Afterwards, we also see a first test using the ellipsoid approach.

Results using spheres for feature generation

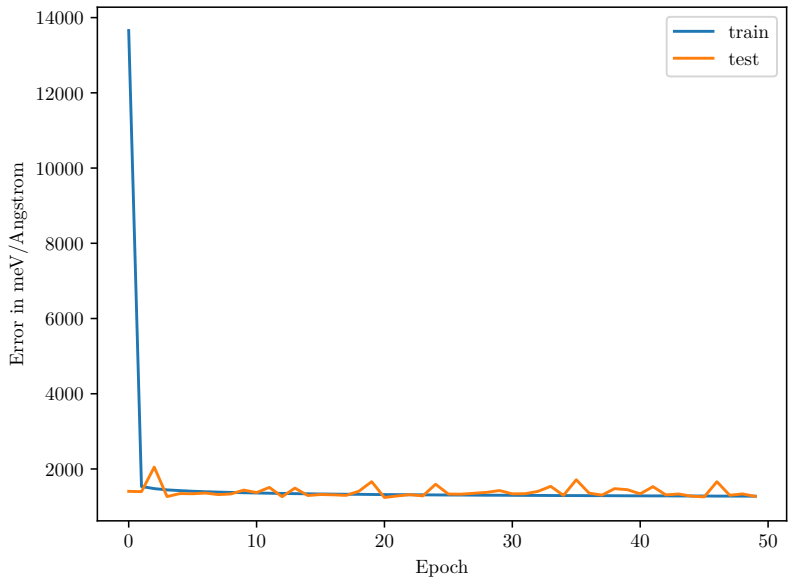
We will measure the error of the forces acting on the nuclei of the atoms in terms of milli electron Volt per Angstrom, for short meV/Angstrom. In [17], an error of less than 100 meV/Angstrom was obtained approximating the forces by differentiating the approximation of the Potential Energy Surface that has been obtained before by fitting an artificial neural network. Of course, we can not compare the numbers directly to our results, because in [17], a completely different set of data has been used. However, when approximating the Potential Energy Surface, we obtained an error which was about half as large as the error in [17], so we would of course like to get about the same results for the forces.
The error that we obtain using the second version of the software lies between 1000 meV/Angstrom and 1500 meV/Angstrom after 50 epochs, depending on the atom type. This is of course too large compared to what we would expect. However, we spent a lot of effort on optimizing the parameters in version one of the software to obtain the good results, while there is a lot of room for improvement in the second version. The second version should rather be seen as a feasibility study. It is possible to construct an artificial neural network that, given an atomic cluster, approximates the forces that are acting on the nuclei. We used simple fully connected feed forward artificial neural networks. In the last subsection of this section we will shortly discuss convolutional neural networks. Changing the network architecture to a more deep and complex one like this and using the techniques of deep learning might turn out to improve these results a lot in the future.
Figure 16 on the following page illustrates the fitting process for the network that approximates the forces acting on the nuclei of the hydragen atoms. Subfigure (a) depicts the fitting process when the features where generated using spheres. In this case, both the loss on the training data as well as the loss on the test data decrease to about 1320 meV/Angstrom. The network architecture is a fully connected feed forward artificial neural network with two hidden layers consisting of 15 artificial neurons, each. The activation function is the familiar

$$\phi(z) = 1.7159 \cdot \tanh\left(\frac{2}{3}z\right) + 0.001z$$

for all artificial neurons in the hidden layers and the identity in the output layer.



(a)



(b)

Figure 16: You can see an example fitting progress for the artificial neural network approximating the forces acting on the nuclei of hydrogen atoms. Subfigure (a) contains the information for using spheres, subfigure (b) for using ellipsoids, instead. Both plots look very similar, and indeed the usage of ellipsoids instead of spheres only slightly improves the results.

Results using ellipsoids for feature generation

When using ellipsoids instead of spheres for the generation of the features for the artificial neural network, we only get slightly better results. The network architecture was kept the same, one fully connected feedforward artificial neural network for each atom type, with all of them having two hidden layers with 15 artificial neurons in each of them. The activation function is again

$$\phi(z) = 1.7159 \cdot \tanh\left(\frac{2}{3}z\right) + 0.001z$$

for all artificial neurons in the hidden layers and the identity in the output layer. The only thing changes is the number of features, because using the ellipsoids creates three times as many features than using the spheres. Hence there are three times as many artificial neurons in the input layer. Figure 16 on the previous page (b) depicts the fitting process when using the ellipsoids. The loss on the training data as well as the loss on the test data decrease to about 1270 meV/Angstrom in this case, which is about 50 meV/Angstrom smaller than the result using spheres. Potentially, another kind of cut off region, as well as a different choice of the hyperparameters such as the learning rate or the number of artificial neurons in the hidden layers might further improve the results.

4.3 Deep Learning and Convolutional Neural Networks

Convolutional Neural Networks, for short *CNN* or *ConvNet*, have become very popular for image and speech recognition tasks because of their incredibly good performance [1]. The paper that is said to have made Convolutional Neural Networks popular (for example in [1] and [2]) is [19].
The idea of Convolutional Neural Networks is depicted in Figure 17 on the following page. The so called *kernel* consists of weights that are applied to a small portion of the input data, just as it would be the case in a fully connected feed forward artificial neural network that we discussed earlier with the only difference that we do not apply a weight matrix to all of the data, but as mentioned to a small portion of it, depicted in the aforementioned figure by a bold rectangle. Now, the kernel is not just applied to one single portion of the data, but to many portions of the data. Imagine the bold square in Figure 17 on the next page to move from left to right over the input data, row by row. It is important that always the same weights are applied to the selected portion of the input data, hence the term *shared weights*. This way, a certain feature is detected in the data, and the values in the feature map basically state in which part of the input data a feature is more likely to have been detected and in which parts it is not. This type of convolutional layer is often called *local receptive field*.
The approach of a local receptive field is then extended, using not only one feature map, but many feature maps, so the above process is repeated for different kernels to obtain multiple outputs and hence multiple feature maps. This way, a lot of different features can be detected in the input data. Moreover, multiple convolutional layers may

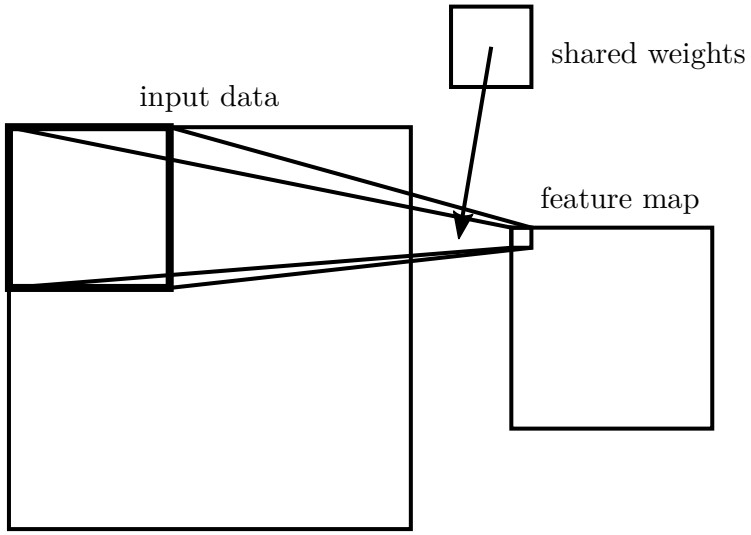


Figure 17: The way a convolutional layer, more precisely a *local receptive field*, works.

be stacked after another, so another convolutional layer is used to detect features in the feature maps that have been the outputs from the first convolutional layer, and so on. This can yield quite a deep architecture and hence makes up the term *deep learning*.
In a fully connected feed forward artificial neural network, having many layers would dramatically increase the number of parameters. Using the shared weights approach is hence very helpful to keep control of the number of parameters even for deep architectures.
Applying Convolutional Neural Networks for image recognition makes great use of the spatial structure of the input data, namely a two dimensional image in the black-and-white case, or a three dimensional one, when the red, green and blue values are treated separately. In our case, when trying to construct an artificial neural network that predicts the forces acting on the nucleus of an atom, we just regarded the input as a long vector of features that have been calculated using symmetry functions and either spheres or ellipsoids as cut off region. But let us go back to how the features where calculated. We first constructed a cut off region around a certain atom, and then detected all other atoms within that cut off region. Afterwards we applied Equation 9 on page 24 to all atoms inside the cut off region. This yielded one number for each atom inside the cut off region, and these numbers have been grouped together by atom type. An actual feature was then calculated applying Equation 10 on page 24 to these values. More precise, for one fixed value of η, we obtain one feature per atom type. We could write these features into a vector. We already explained that Equation 10 on page 24 is applied for multiple values of η, giving us many of these vectors that could be put together into a matrix.
This way, we could regard our input data as two dimensional, just as the input data for the black-and-white case in image recognition. If we would run the aforementioned calculations for several different cut off regions, it yields one matrix for every cut off region and hence three dimensional input data, just as in the multicolored case.

Apart of local receptive fields, there are other types of convolutional layers. One example is a so called *pooling layer*. We will take a short look again at Figure 17 on the previous page. Imagine that we regard a small top left portion of the feature map. The values represent how likely a certain feature has been detected either in the top left corner or close to the top left corner of the input data. We apply a pooling layer to combine these values to a single one, describing how likely it is that a certain feature has been detected somewhere close to the top left corner of the input data. One way of doing this is a *max pooling layer*, in this case we would just pick the maximum value of the small portion of the feature map. There are other pooling layers depending on how one wishes to combine the values from the feature map, for example *min pooling* or L_2 *pooling* [2].
Of course, it requires a lot of testing to find out valuable cut off regions as well as the convolutional architecture in general, but applying the convolutional techniques to the problem of predicting the forces acting on the nuclei of an atomic cluster could prove to be very powerful and might improve the results presented above a lot.

5 Conclusion

After having introduced some ideas of Machine Learning, especially the important ideas of Artificial Neural Networks, and also the basic ideas of Molecular Dynamics, the main part of this work was devoted to a modern, efficient implementation of an artificial neural network that is able to predict the Potential Energy of an atomic cluster with state of the art accuracy.
We chose the programming language Python and used the Numpy, SciPy, Theano and Keras libraries that provided us an efficient use of both the CPU and the GPU.
Using the testing computer system which is described in detail in the introduction, we gained an improvement in terms of computation speed of by a factor of about 50 to 60.
The final error measurement of the artificial neural network is comparable to the results of other state of the art software packages that provide an artificial neural network approach to predicting the Potential Energy of an atomic cluster.
Normally, the final goal for running Molecular Dynamics simulations is to predict the forces acting on the nuclei of the atoms in the atomic cluster. They have been obtained in previous works by differentiating the Potential Energy Surface with respect to the coordinate axes.
In the last part of this work we did something new, namely developing artificial neural networks that directly approximate the forces acting on the nuclei of the atoms in an atomic cluster. The first results of this approach yield an error that is larger than the error in comparable state of the art software packages, that obtained the forces by differentiating the Potential Energy Surface, by some margin. However, there is a lot of room for improvement in future works. We discussed that besides the optimization of hyper parameters, the application of Convolutional Neural Networks and Deep Learning could significantly improve the results that we have obtained. It will be very interesting to see the results of future research in this direction.

References

[1] S. Raschka. Python Machine Learning. Packt Publishing 2015

[2] M. Nielsen. Neural Networks and Deep Learning. Online book www.neuralnetworksanddeeplearning.com, 21/11/2017

[3] R. Rojas. Neural Networks A Systematic Introduction. Springer-Verlag, Berlin, 1996

[4] W. S. McCullock, W. Pitts. A Logical Calculus of the Ideas Immanent in Nervous Activity. The bulletin of mathematical biophysics, 5(4):115-133, 1943

[5] H. Amann, J. Escher. Analysis II. Birkhäuser 1998

[6] S. Ruder. An overwiev of gradient descent optimization algorithms. Internet source http://ruder.io/optimizing-gradient-descent/, 21/11/2017

[7] A. Ng. Machine Learning. Online course from coursera available at www.coursera.org/learn/machine-learning, 21/11/2017

[8] S. Lorenz. Reactions on Surfaces with Neural Networks. Dissertation 2001

[9] Python Software Foundation. Python Language Reference, version 3.5.2. Available at www.python.org, 21/11/2017

[10] Numpy software library. Version 1.12.1. Available at www.numpy.org, 21/11/2017

[11] SciPy software library. Version 0.19.1. Available at www.scipy.org, 21/11/2017

[12] Theano software library. Version 0.9.0.
Available at http://deeplearning.net/software/theano/, 21/11/2017

[13] Keras software library. Version 2.0.2. Available at https://keras.io/, 21/11/2017

[14] Matplotlib software library. Version 2.0.2. Available at https://matplotlib.org/, 21/11/2017

[15] TensorFlow software library. Version 1.4. Available at www.tensorflow.org, 21/11/2017

[16] A.P. Drozdov, M. I. Eremets, I. A. Troyan, V. Ksenofontov, S. I. Shylin. Conventional superconductivity at 203 K at high pressures. Nature 525, 73–76, 03/09/2015

[17] 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

[18] G. Montavon, G.B. Orr, K.-R. Müller (Eds.). Neural Networks: Tricks of the Trade. Lecture Notes in Computer Science, second ed., vol. 7700, Springer, Berlin Heidelberg, 2012.

[19] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner. Gradient-Based Learning Applied to Document Recognition. Proceedings of the IEEE, 86(11):2278-2324, 1998

[20] J. Duchi, E. Hazan, Y. Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. Journal of Machine Learning Research, 12, 2121–2159. Retrieved from http://jmlr.org/papers/v12/duchi11a.html, 2011