Estimating the T-dependence of zero field splitting in the NV^--center using Neural Networks

Technical University Vienna

Center for Computational Materials Science

Diploma Thesis

Author:
Matthias Ebert, BSc.
1125488

Supervisor:
Univ.Prof. Dipl.-Ing. Dr.techn.
Peter Mohn
Contributors:
Dipl.-Ing. Dr.techn. Johannes
Gugler
Dipl.-Ing. Martin Pimon

March 20, 2020

TECHNISCHE
UNIVERSITÄT
WIEN
Vienna | Austria

Abstract

In this thesis we probe the feasibility of using Artificial Neural Networks (ANNs), trained on data computed using DFT methods (in VASP), to perform a molecular dynamics (MD) simulation of a many-particle system as well as estimate other system properties. The system chosen is a nitrogen-vacancy center (NV-center) in a diamond lattice (63 carbon atoms, 1 nitrogen atom) and the system property in question is the zero-field splitting (or D-tensor) at the NV-center. In the MD, ANNs are then used to estimate the forces acting on each atom as well as the diagonal D-tensor values at each step. Multiple such MDs are run at different temperature levels, introduced by a thermostat, to calculate the T-dependence of the D-tensor zz-component.
Performing 3 such simulations with varying number of MD-steps (100k, 300k, and 1000k), over temperatures ranging from 10 K to 550 K we were able to obtain a smooth and consistent estimate of the D-tensor T-dependence. We therefore demonstrate that such an approach has actual utility in computing properties of many-particle systems and deserves further investigation.

Kurzfassung

In dieser Arbeit untersuchen wir ob künstliche Neuronale Netzwerke (KNNs), die mit Ergebnissen aus DFT Rechnungen (in VASP) trainiert wurden, zu Druchführung von Molekulardynamik (MD) Simulationen von Vielteilchensystemen, sowie zu Schätzung von Systemeigenschaften verwendet werden können. Das untersuchte System ist eine Stickstoff-Fehlstelle (NV-center) in eineme Diamant Gitter (63 Kohlenstoff Atome und 1 Stickstoff Atom) und die Systemeigenschaft von Interesse ist das Zero-Field-Splitting (oder D-Tensor) an der Stickstoff-Fehlstelle. In den MD-Simulationen werden sowohl die Kräfte auf alle Teilchen, als auch die zz-Komponente des D-Tensors in jedem Schritt von KNNs geschätzt. Unter Verwendung eines Thermostats, führen wir mehrere solche MD-Simulationen bei verschiedenen Temperaturen durch um die Temperaturabhängigkeit des D-Tensors zu bestimmen.
Die Ergebnisse aus 3 solchen Simulationen über einen Temperaturbereich von 10 K bis 550 K, mit unterschiedlicher Anzahl an MD-Schritten (100k, 300k und 1000k), zeigen ein konsistentes Temperatur-Verhalten des D-Tensors. Damit zeigen wir, dass eine sinnvolle Schätzung von Systemeigenschaften auf diese Weise möglich ist und der beschriebene Ansatz weitere Untersuchung verdient.

Contents

Chapter 1

Introduction

Performing reliable computer simulations in physics, chemistry as well as materials science relies heavily on an accurate description of atomic interactions. The physics to describe such interactions is known and a great number of electronic structure methods using the Born-Oppenheimer approximation exist. These methods provide results of good accuracy, however, they require significant computational resources. Scaling with $O(n^3)$ in the number of particles (i.e., electrons) those methods impose limits on the system size as well as the number of simulated time-steps during such calculations. If it were possible to train an Artificial Neural Network (ANN) on data provided by an electronic structure method to estimate e.g. forces or energies from the atomic structure (i.e., all atomic positions and charges), one would obtain a relation mapping structure to forces or energies. This in turn would speed up the calculation of system dynamics significantly allowing for simulations of larger systems and vastly more dynamics steps, permitting either longer simulated time frames or finer granularity.

The purpose of this thesis is to test the feasibility of using such an approach to perform a molecular dynamics (MD) simulation of a medium sized system (64 atoms, 2 atom types) where the forces acting on each particle as well as other properties of the system are computed (estimated) by ANNs at each MD-step. The system chosen in this work is a NV^--center embedded in a diamond lattice and the system property in question is the zz-component of the zero-field splitting or D-tensor at the NV^--center. The zero-field splitting describes various interactions of energy levels resulting from the presence of more than one unpaired electron. The latter serves as a particularly good example, since it's computation via standard DFT has substantial variance requiring many such computations with different atomic configurations to obtain a statistically significant estimate of it's true value. Using NNs to estimate the D-tensor value could lead to a significant speed-up allowing to sample over a larger number of atomic configurations in a shorter amount of time.

Chapter 2

Theoretical Background

2.1 Density functional theory (DFT)

Developed in the 1960s, density functional theory (DFT) is a procedure for calculating the ground-state of a quantum many-body system of interacting electrons, which is based on the electron density as a function of position. Extensively employed in physics, chemistry and materials science, DFT is used to obtain electronic structure information of atoms, molecules as well as solid bodies. With this it is possible to calculate a multitude of properties, e.g., bond-lengths and bond-energies ab-initio.
While the computational effort to solve the full many-body Schrodinger's equation scales exponentially with the number of particles, DFT offers a procedure for finding the many-body ground-state (in principle without having to sacrifice any accuracy) that scales with $O(n^3)$, allowing for the simulation of much larger systems.
With this, it is no understatement to say, that DFT has revolutionized computational materials science as well as computational chemistry over the last couple of decades.

2.1.1 Prerequisites

The variational principle

Most generally, the variational principle can be used in the calculus of variations, for finding functions that give extremal values of qunatities that depend upon those functions (i.e., functionals).
Specifically, in the context of DFT, the Rayleigh-Ritz variational principle is used. It states that any normalized wavefunction used to calculate the expectation value of the energy is guaranteed to give an energy larger than the true ground-state energy E:

$$E = \min_{\Phi} \langle \Phi | \hat{H} | \Phi \rangle , \qquad \int_{-\infty}^{\infty} |\Phi(x)|^2 dx = 1 \tag{2.1}$$

This is a very powerful principle, which basically allows us to simply evaluate the energy of a set of normalized wavefunctions and find the best approximation to the ground-state by choosing the one which gives the lowest energy. Furthermore, this means that we can, at least in principle, approximate the ground-state to arbitrary accuracy by simply trying a sufficient number of wavefunctions.
In practice, one would use parameterized trial wavefunctions, e.g., a Gaussian

$$\Phi_G(x) = \beta e^{-\alpha x} \tag{2.2}$$

and choose the free parameters (α and β in this case) in order to minimize the energy.

Hohenberg-Kohn theorem

The Hohenberg-Kohn theorem relates to all systems where particles (electrons) move under an external potential. In those cases the theorem state [19]

1. The potential governing a system of electrons is a unique functional of that system's electron density ($n(\mathbf{r})$). Therefore, the ground-state density uniquely determines the potential and thereby the systems properties, including also the ground-state wavefunction.

2. For any positive integer N and potential $V(\mathbf{r})$, there exists a density functional $F[n]$ sucht that
$$E_{V,N}[n] = F[n] + \int_{-\infty}^{\infty} V(\mathbf{r})n(\mathbf{r})d^3r \tag{2.3}$$
obtains its minimal value at the ground-state density of N electrons in the potential $V(\mathbf{r})$ and the minimal value of $E_{V,N}[n]$ is then the true ground-state energy of the system. In other words, the functional delivers the ground-state energy if and only if the input density is the true ground-state density.

Kohn-Sham equations

The essence of the Kohn-Sham formalism is to treat a system of interacting electrons as a fictitious system of non-interacting electrons and to 'dump' all the interaction effects, beyond the Hartree-potential, into an additional exchange-correlation term in the ground-state functional. The orbitals Φ_i of such a system are then given by

$$\{-\frac{1}{2}\nabla^2 + V_{KS}(\mathbf{r})\}\Phi_i(\mathbf{r}) = \epsilon_i\Phi_i(\mathbf{r}) \tag{2.4}$$

and yield a density

$$n(\mathbf{r}) = \sum_{i=1}^{N} |\Phi_i(\mathbf{r})|^2. \tag{2.5}$$

Here, $V_{KS}(\mathbf{r})$ is the effective Kohn-Sham potential which has three contributions

$$V_{KS}(\mathbf{r}) = V_{ext}(\mathbf{r}) + V_H(\mathbf{r}) + V_{XC}(\mathbf{r}) \tag{2.6}$$

which are the external, Hartree and exchange-correlation potential respectively. Because of the Hohenberg-Kohn theorems we know that the independent particle equations have their own ground-state energy functional, which Kohn and Sham wrote as

$$E[n] = T_s[n] + E_{ext}[n] + E_H[n] + E_{XC}[n] \tag{2.7}$$

where $T_s[n]$ is the Kohn-Sham kinetic energy

$$T_s[n] = \sum_{i=1}^{N} \int_{-\infty}^{\infty} \Phi_i^*(\mathbf{r})(-\frac{1}{2}\nabla^2)\Phi_i(\mathbf{r})d^3r, \tag{2.8}$$

$E_{ext}[n]$ is the energy corresponding to the external potential

$$E_{ext}[n] = \int_{-\infty}^{\infty} V(\mathbf{r})n(\mathbf{r})d^3r, \tag{2.9}$$

$E_H[n]$ denotes the Hartree (or Coulomb) energy

$$E_H[n] = \frac{1}{2}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\frac{n(\mathbf{r})n(\mathbf{r}')}{|\mathbf{r}-\mathbf{r}'|}d^3r d^3r' \tag{2.10}$$

and finally, $E_{XC}[n]$ is the exchange-correlation energy functional, which, by definition, is the difference between the exact energy and all other known terms.

With the above equations one can construct an iterative procedure, where an initial electron density is chosen and used to calculate the Kohn-Sham potential via a variational approach

$$V_{KS}(\mathbf{r}) = V_{ext}(\mathbf{r}) + \int_{-\infty}^{\infty}\frac{n(\mathbf{r}')}{|\mathbf{r}-\mathbf{r}'|}d^3r' + \frac{\delta E_{XC}[n]}{\delta n(\mathbf{r})}. \tag{2.11}$$

By solving equation 2.4 and using equation 2.5 one can obtain a new electron density which can then be plugged in equation 2.11 to start the next iteration. This can be repeated until there is no longer any noticeable change in the density and we can be confident that we have found a minimum.

2.1.2 The DFT procedure

With the above, a basic DFT procedure boils down to the following:

1. Switch from describing a system via a many-body wavefunction to a description using the electron density. Here, the Hohenberg-Kohn theorems guarantee that finding the ground-state density is equivalent to finding the ground-state wavefunction.
2. Choose an approximation for the exchange-correlation energy functional (TODO: remove oveful box) $E_{XC}[n]$, which captures most of the relevant features of your system (i.e., only neglects effects that are sufficiently small).
3. Solve a Kohn-Sham procedure to obtain the ground-state density of your system with the desired accuracy.
4. The ground-state density can then be used to calculate the ground-state energy of your system, which in turn, allows you to derive most other relevant properties.

With this it becomes clear what DFT actually does. In simple words, DFT reformulates many-particle problems in a way such that everything that is unknown or expensive to compute (i.e., many particle interactions) is collected at one place, the exchange-correlation functional $E_{XC}[n]$. With this, approximations are confined to a single term which is generally easier to handle than having to use approximations at multiple steps in a procedure. Also, if required, doing empirical approximations for specific systems of interest becomes much easier.

With this, one could say that the performance of DFT primarily depends on the quality of approximations to the exchange-correlation energy functional.

2.1.3 Exchange correlation energy

Local density approximation (LDA)

Local density approximations (or LDAs) are a class of approximations for the exchange-correlation energy functional $E_{XC}[n]$. As 'local' suggests, in LDA approaches $E_{XC}[n]$ depends solely on the value of the density at each point in space, which (in a spin-unpolarized system) takes a form like

$$E_{XC}^{LDA}[n] = \int_{-\infty}^{\infty} n(\mathbf{r})\epsilon_{XC}^{LDA}(n)d^3r. \tag{2.12}$$

Furthermore, the exchange-correlation energy density $\epsilon_{XC}(n)$ can be linearly decomposed into an exchange and a correlation contribution

$$\epsilon_{XC}^{LDA}(n) = \epsilon_{X}^{LDA}(n) + \epsilon_{C}^{LDA}(n). \tag{2.13}$$

Typically, the exchange term takes on an analytic form derived from the homogeneous electron gas (HEG) model [3]

$$\epsilon_{X}^{LDA}(n) = A_x * n^{4/3}(\mathbf{r}), \tag{2.14}$$

while there exist numerous different approximations for the correlation term.

Local spin density (LSD)

The extension to spin-polarized systems is rather straightforward, at least for the exchange term. In a spin-polarized DFT calculation there are two separated densities for both polarizations and the spin-polarized exchange term can be expressed using the spin-unpolarized one [3]

$$\epsilon_{X}^{LSD}(n_\uparrow, n_\downarrow) = \frac{1}{2}(\epsilon_{X}^{unpol}(2n_\uparrow) + \epsilon_{X}^{unpol}(2n_\downarrow)) \tag{2.15}$$

This result holds in general and not just in the LDA context. However, in case of LDA and the HEG the unpolarized exchange term would look as follows [3]

$$\epsilon_{X}^{LSDA}(n_\uparrow, n_\downarrow) = 2^{\frac{1}{3}} A_x(n_\uparrow^{\frac{4}{3}} + n_\downarrow^{\frac{4}{3}}). \tag{2.16}$$

Representation becomes even more straightforward, when introducing the concept of relative spin-polarization

$$\zeta = \frac{n_\uparrow - n_\downarrow}{n} \qquad and \qquad n_\uparrow = \frac{n}{1}(1+\zeta), \quad n_\downarrow = \frac{n}{1}(1-\zeta) \tag{2.17}$$

With this we can write the exchange energy functional as follows

$$E_{X}^{LSDA}[n_\uparrow, n_\downarrow] = A_x \int_{-\infty}^{\infty} n^{\frac{4}{3}}(\mathbf{r}) \frac{(1+\zeta(\mathbf{r}))^{\frac{4}{3}} + (1-\zeta(\mathbf{r}))^{\frac{4}{3}}}{2}. \tag{2.18}$$

Unfortunately, the correlation energy for spin-polarized systems cannot be expressed so simply, using the correlation energy of the unpolarized system. However, several forms were developed in the LDA context (i.e., in conjunction with different LDA correlation energy terms).

Advanced approaches

Improving on the performance of LDA in one way or another, there are a great number of more sophisticated approximations to the exchange-correlation energy functional, which are usually derived from fitting the exchange-correlation energy to results from quantum Monte-Carlo simulations. Those include generalized gradient approximations (GGA) or hybrid functionals, such as PBE or HSE.

2.2 NV-Center

The nitrogen-vacancy (NV) center is one of the many point defects in diamond, where nitrogen impurities, paired with a carbon vacancy, are present in the pure diamond lattice. This structure offers a variety of desirable properties, the most prominent of which include pholuminescence, detectable from individual NV centers as well as room-temperature spin-manipulation using mechanical stress, microwave radiation or electric/magnetic fields. These features make it suitable for room-temperature quantum information processing or room-temperature masers. Also, the production and manipulation of NV centers can be done via processes already present in current industries, which would reduce the adaptation time for solutions based on the NV center.

2.2.1 Structure

As mentioned, the NV center is a point defect in the diamond lattice, where a substitutional nitrogen atom is paired with a carbon vacancy at one of the nearest neighbor lattice points (see figure 2.1).

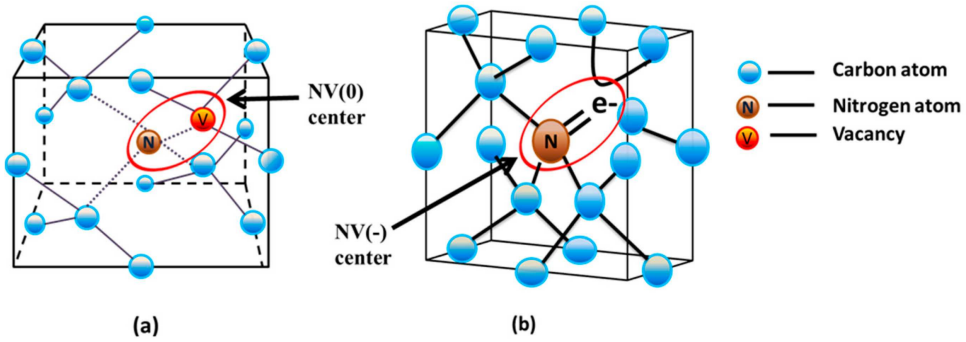

Figure 2.1: Schematic view of an NV^0 and NV^- in diamond. [6]

For the nitrogen, 3 of its 5 valence electrons are covalently bonded to its 3 remaining carbon neighbors, while the remaining 2 (called the lone pair) are left non-bonded. The vacancy, on the other hand, has 3 unpaired electrons, where 2 of them form a quasi-covalent bond and 1 is left truly unpaired. Overall, there is a C_{3V} symmetry (axial trigonal) which basically means, that the 3 vacancy electrons are perpetually exchanging their roles.

For the NV center two states were observed which differ in their charge. The NV^0 (figure 2.1a) has no charge and one electron is left unpaired, while for the NV^- (figure 2.1b) another electron is present at the vacancy site, which binds to the unpaired electron. However, it is not yet fully understood where these extra electrons come from.

Since the NV^- center is the more common version, the superscript will be omitted from here on unless explicit distinction is required.

2.2.2 Production

The production of NV centers is typically done via a rather straight-forward three-step process [6].

1. First, the desired quantity of nitrogen atoms are deposited in the diamond lattice via ion-implantation techniques, such that the spatial distribution of the substituents can be controlled fairly well.

2. After this step, the diamond target is irradiated to produce vacancies in the lattice. For this process, a variety of particles are suitable, including ions, protons, neutrons, electrons, and gamma photons. However most of the vacancies produced in this way will not be next to a nitrogen defect.

3. Since the vacancies are immobile at room temperature, the target is annealed at high temperatures which causes the vacancies to move. Fortunately, single substitutional nitrogen generates strain in the diamond lattice, which leads to vacancies being trapped preferably next to the nitrogen defects.

The first step can often be omitted, since in the majority of natural and artificial diamonds, nitrogen is already sufficiently abundant for most effects that are of scientific interest.

2.2.3 Optical properties and electronic structure

The ground state $^3A_2$ and first excited state ^3E of the NV center in diamond are triplets as shown in figure 2.2. Conveniently, both are situated within the diamond band gap, so it is possible to optically excite electrons, where resonance occurs for green light of $546nm$ wavelength. This transition is primarily spin-conserving and decays rather quickly ($\sim 10ns$ [5]), by emitting red light peaking at $689nm$ wavelength.

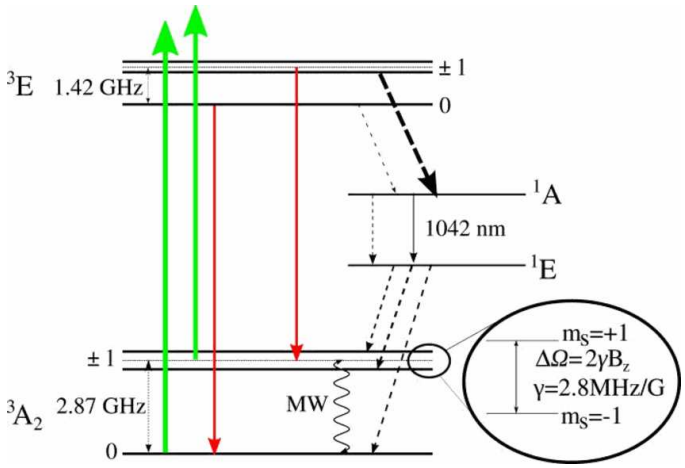

Figure 2.2: Energy level diagram of the NV^- center in diamond. Dashed lines represent transition that are believed to be non-radiative, whereas solid lines denote radiative ones. The transitions corresponding to the green and red lines have wavelength peaks of $546nm$ and $689nm$ respectively. [9]

Without an external magnetic field both the ground and excited state are split due to the magnetic interaction between the two unpaired nitrogen electrons, i.e. the energy is higher when spins are parallel ($m_s = \pm 1$) than when they are anti-parallel ($m_s = 0$). When a magnetic field is applied to the NV center it will affect the separation of the $m_s = +1$ and $m_s = -1$ state. In particular, if the component of the magnetic field along the NV defect axis reaches approximately 1027G, the $m_s = -1$ and $m_s = 0$ ground (and excited) states become equal in energy and will

start to interact.

Aside from the ground and excited triplet states there also exist two intermediate singlet states (^1A and ^1E) that allow for intersystem crossing (ISC) to occur. While optical excitations between ^3E and $^3A_2$ have to conserve the spin, it is possible for an electron in the $m_s = \pm 1$ state of ^3E to decay non-radiatively into the singlet state ^1A. In fact, a nice property of the NV center in diamond is, that the $m_s = \pm 1$ electrons have a much higher decay rate into the intermediate singlet state, than $m_s = 0$ electrons.
Between the two singlet states there is then the possiblity for either another non-radiative decay or a radiative (in the infrared range) decay, followed by one more non-radiative decay into the ground state triplet.

2.2.4 Applications

The properties of the NV center in diamond outlined above, allow for it to be used in a variety of interesting applications. One of the most important being it's use for quantum computing.

NV center as a qubit

The NV center looks to be a promising candidate for a room temperature qubit in quantum information processing, where the logical states of 1 and 0 are represented by the spin-polarization in the ground state triplet (i.e., 1 corresponds to $m_s = \pm 1$ and 0 to $m_s = 0$). Since the separation of the latter is in the microwave range it is easy enough to populate the $m_s = \pm 1$ states, i.e. flipping the qubit to logical 1, by irradiating the NV center with the corresponding frequencies.
On the other hand, a process for deterministically flipping the qubit to logical 0, i.e. populating the $m_s = 0$ state, is made possible due to the NV center's property that the non-radiative decay from ^3E to ^1A occurs preferably for electrons in the $m_s = \pm 1$ state. For this consider the following:

1. Excitation of electrons into the ^3E state with an off-resonance frquency above $546nm$. With this all spin states will be excited.

2. Now populating ^3E, electrons with $m_s = 0$ will simply decay radiatively back into $^3A_2$, thereby conserving spin. Electrons with $m_s = \pm 1$, however, now have a significant chance to decay into the intermediate state ^1A where their spin is flipped.

3. From ^1A they can decay into the $m_s = 0$ state of $^3A_2$.

After a sufficient number of iterations it can be guaranteed that the NV center is in the $m_s = 0$ state, i.e., the qubit has been set to logical 0.

2.3 Artificial Neural Networks

Most generally[18], Artificial Neural Networks (ANNs), or often just Neural Networks (NNs), can be descirbed as computing systems which are vaguely inspired by biological neural networks that constitute human brains.
More specifically, for the purpose of this thesis, we will consider them as a nonlinear model for supervised learning, which uses a set of examples (i.e., the training data) to "learn" to perform a certain task. ANNs are made up by simple building blocks, called neurons.

2.3.1 Neurons

The smallest unit in an ANN is the neuron. Like their biological namesake, they also take several inputs denoted as $\mathbf{x} = (x_1, x_2, ..., x_d)$ and produce a single output $a(\mathbf{x})$.
Additionally, all the inputs are weighted by neuron-specific weights $\mathbf{w} = (w_1, w_2, ..., w_d)$ and the sum of all weighted inputs is offset by a single scalar neuron-specific bias b. In almost all cases the output-function $a_i(\mathbf{x})$ of neuron i can be split into a linear part (see figure 2.3) of the form

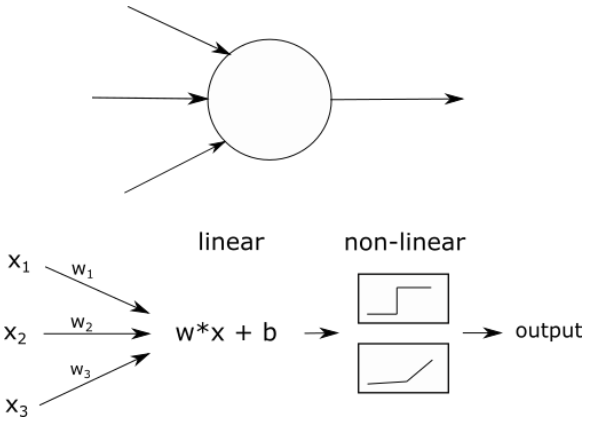

Figure 2.3: Neuron with 3 inputs in both graphical and decomposed functional form.

$$z^{(i)} = \mathbf{w}^{(i)} * \mathbf{x} + b^{(i)} \quad (2.19)$$

and a non-linear part

$$a_i(\mathbf{x}) = \sigma_i(z^{(i)}) \quad (2.20)$$

with a non-linear function $\sigma_i(z)$ also known as the activation-function (i.e., which determines the final output of the neuron).
The latter could be a step-function, sigmoid or hyperbolic tangent even though rectified linear units, leaky rectified linear units as well as exponential linear units are among the most common, nowadays. The choice of non-linearity mainly affects computational and training properties of the resulting ANN. The reason for this is that the training of ANNs involves gradient descent based methods which causes the derivative of the non-linearity to have a significant influence on the training behavior.

2.3.2 Feed Forward NNs

Neural Networks, as the name suggests, are created by layering neurons in a hierarchical fashion. The structure of this set of neurons, i.e., how the individual neurons are connected to each other, is called the network architecture, of which there exists a great variety. In this work the most basic architecture, a feed forward neural network (see figure 2.4), will be used. Here, the neurons are structured into layers starting with the input layer, followed by one or more, so called, hidden layers and terminated by an output layer. For the hidden layers, each individual neuron is

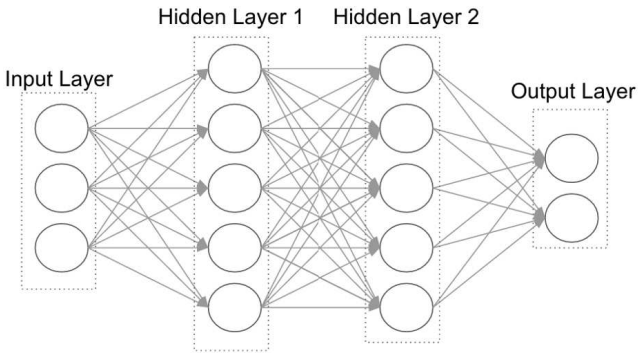

Figure 2.4: An example of a fully connected feed forward neural network with 3 inputs, 2 outputs and 2 hidden layers.

connected to one or more neurons in the previous layer and to one or more neurons in the following layer, i.e. the output of one layer is treated as the input for the next layer.
This type of NN works as follows:

- The input data is fed into the input layer in a one-to-one mapping (the number of neurons in the input layer has to be equal to the dimensionality of the input).
- The input is then propagated through the hidden layers making use of the neuron-specific activation functions, where the dimensionality of $\mathbf{w}$ and $\mathbf{x}$ correspond to the number of connections the neuron has to the previous layer.
- The output layer then uses a suitable activation function to generate the desired output format (e.g., discrete/continuous, bound/unbound, etc.). This is again done in a one-to-one manner, i.e. the number of neurons in the output layer and the number of required output dimensions has to match.

This means one can interpret the NN as a function which maps inputs of dimension n to outputs of dimension m

$$\mathbf{y} = f(\mathbf{x}) \tag{2.21}$$

with $\mathbf{y} = (y_1, y_2, ..., y_m)$ and $\mathbf{x} = (x_1, x_2, ..., x_n)$. The weights and biases of the individual neurons can be considered as free parameters of the function, i.e., can be tweaked to approximate a desired functional behavior. In fact, there exists a universal approximation theorem [7] which states, that even a single-layer NN, with a finite number of neurons, can approximate any function with arbitrary accuracy. However, this theorem is of limited practical relevance since it offers no bound on the number of neurons necessary to achieve this. Therefore, in real applications, a trade-off will always have to be made between accuracy of approximation and the necessary computational effort.

2.3.3 Training NNs

As mentioned in the beginning, NNs learn from examples. Therefore, one needs a (often quite large) set of samples consisting of pairs of inputs $\mathbf{x}$ and outputs $\mathbf{y}$, i.e. $\{(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), ..., (\mathbf{x}_N, \mathbf{y}_N)\}$.
The basic procedure for training NNs then is similar to most other, simpler, supervised learning algorithms. To start with, one has to construct a loss function (or cost function) combining the network output for the training input with the

actual outputs. This function, if minimized, should result in the desired behavior of the NN. Then, a gradient descent based method is used to change the weights and biases (for simplicity, from here on the biases will be considered to be part of the weights) of all neurons in a way that minimizes the loss function. E.g., one of the simplest (and also most common) loss functions is some form of mean squared error

$$E(\mathbf{W}) = \frac{1}{N}\sum_{i=1}^{N}(y_i - y_i(\mathbf{W}))^2 \tag{2.22}$$

where $\mathbf{W}$ is the set of all weights of all the neurons in the NN and N is the number samples in the data used to train the NN. The typical training procedure looks as follows

1. Choose a network architecture (e.g. feed forward NN) and a specific structure (i.e. number of hidden layers, number of neurons per layer, etc. ...).
2. Initialize all the weights in the network either randomly or using an educated guess.
3. Run all training samples through your network and compute your loss.
4. If the loss is smaller than some threshold then terminate, otherwise continue with step 5.
5. Use gradient descent to correct the weights of your NN according to the deviation of the predicted outputs from the real outputs.
6. Repeat from step 2.

Optimizers

In machine learning an optimizer is the procedure by which the loss function is minimized. They try to tweak the parameters in a model (i.e., weights and biases) in such a way that the error the model makes on predictions, quantified by the chosen loss function, is reduced.
The simplest optimizer would be a standard gradient descent approach, where the optimizer always follows the direction of the steepest slope until there is no longer any noticeable change in precision. However, in a high-dimensional parameter space, where the function to approximate is a multi-dimensional surface with many local minima, this naive approach will frequently lead to sub-optimal results.
Needless to say, because of this shortcoming, more sophisticated optimization schemes such as Adagard, RMSprop and Adam have been introduced. These advanced optimizers typically use one or more of the following techniques

- Introduce a stochastic component to the updating of parameters in order to escape a local minimum.
- Employ individual learning rates, i.e., factors defining the size of updates in each step, for all model parameters.
- Allowing the learning rate to vary between optimization steps.

The 'Adam' (Adaptive Moment Estimation) optimizer, which has been chosen for training the NNs in this work, employs all of the above and uses the following procedure to update model parameters [15]

$$\mathbf{g_t} = \nabla_{\mathbf{W}} E(\mathbf{W_t}) \tag{2.23}$$

$$\mathbf{m_t} = \beta_1 \mathbf{m_{t-1}} + (1 - \beta_1)\mathbf{g_t} \tag{2.24}$$

$$\mathbf{s_t} = \beta_2 \mathbf{s_{t-1}} + (1 - \beta_2)\mathbf{g_t}^2 \tag{2.25}$$

$$\hat{\mathbf{m}}_\mathbf{t} = \frac{\mathbf{m_t}}{1 - \beta_1^t} \tag{2.26}$$

$$\hat{\mathbf{s}}_\mathbf{t} = \frac{\mathbf{s_t}}{1 - \beta_2^t} \tag{2.27}$$

$$\mathbf{W_{t+1}} = \mathbf{W_t} - \eta_t \frac{\hat{\mathbf{m}}_\mathbf{t}}{\sqrt{\hat{\mathbf{s}}_\mathbf{t}} + \epsilon} \tag{2.28}$$

Here, $\mathbf{g_t}$ represents the gradient for the results achieved by the previous parameter values $\mathbf{W_t}$ (i.e., weights of the NN). $\mathbf{m_t}$ and $\mathbf{s_t}$ are estimates of the first and second moment, i.e., the mean and uncentered variance, of the gradient respectively. β_1 and β_2 are used to set the 'memory lifetime' for the first and second moment, i.e., influence how far into the future the values of the moments should have an effect as can be seen in equations 2.24 and 2.25. Since the latter were found to be biased towards zero, equations 2.26 and 2.27 are used to correct for this bias. Finally, the actual update of the parameters is done in equation 2.28 where the corrected moments and an adaptive learning rate η_t are used to perform a gradient descent. ϵ is just a small regularization constant used to avoid divergences.
If one pictures gradient descent procedures as a ball rolling down a slope, 'Adam' could be viewed as a heavy ball with friction. Thus it preferably 'looks for' flat minima in the error surface.

Batches and epochs

Two other terms that frequently occur in the context of machine learning are batches and epochs.
During the training phase, the entire training-set is randomly split into equally sized subsets, called batches. Then, the optimizer computes the predictions of the current model on all the samples in one batch and uses the errors of the entire batch to perform a single update on the model parameters. Doing this for all batches, i.e., going through the entire data-set once, is called an epoch. When starting the next epoch the training-set is randomly split into batches again.
The number of epochs and the batch-size can be considered parameters of the training procedure, that can be tweaked to improve performance, in the sense of finding a sweet-spot between accuracy and computational effort.

Chapter 3

Modeling & Simulation

3.1 Tools used

In order to perform the simulations described in this work, i.e., create the training data and train the machine learning models, various tools has been used. Those are described below.

3.1.1 The VASP-framwork

The Vienna Ab initio Simulation Package (VASP) is a program for modeling materials at the atomic scale. It can be used to calculate features such as the electronic structure or molecular dynamics from first principles. [16, 13, 12, 11]
The principal method underlying VASP is to use either density functional theory (DFT) for solving the Kohn-Sham equations or the Hartree-Fock approximation for solving the Roothaan equations in order to approximate the solution of the many-body Schrodinger equation.

Concerning this work, VASP was used to generate the raw data from which the training data for the NNs were derived.

3.1.2 Tensorflow

Tensorflow [1] is a programming framework, implemented in Python and C++, focused on dataflow programming and is used in Python-based programs. Originally developed by Google Brain Team, it is an end-to-end open source platform that has become quite popular in the data-science domain, primarily because it facilitates easy development and deployment of machine learning powered applications. Due to its popularity in the economic world as well as research communities alike, the Tensorflow ecosystem sports a large community providing a well maintained codebase as well as state-of-the-art (and nicely documented) functionality.

3.1.3 Libraries Used

Next to the Tensorflow Python library a couple of other libraries, that greatly reduce the effort of manipulating data as well as building and training NNs, were used.

Keras

Keras [4] is an open-source neural-network library written in and for Python, designed to run on top of, among other frameworks, Tensorflow. Its design focus are both user-friendliness as well as facilitating easy experimentation with neural

networks, minimizing the effort for going from a theoretical concept to a running program.
In essence, Keras provides an API (the Keras Functional API) with comprehensible functions for constructing and training neural networks.

However, as it is a wrapper library, Keras does not allow modifying everything in its backend which somewhat limits the customization of (especially more complex) NNs. Also, the error-messages thrown are sometimes not too useful.
Nevertheless, since this work did not require any advanced network structures, ease-of-use was deemed more important and Keras was chosen for implementing the NNs.

Pandas

Pandas [14] is an open-source Python library designed for data analysis and data manipulation. It features data structures to efficiently hold large amounts of data and offers high performance functions to operate on them.

Even though a steep learning curve due to Panda's extensive functionality, difficult syntax as well as bad documentation have to be mentioned as some notable handicaps when starting to use this library, the provided data structures and data manipulation tools are still worth the effort. Specifically for the time saved when operating on large data sets.

3.2 Data preparation & training the NNs

Since Machine Learning (ML) fundamentally relies on data for training models, two of the most important steps in every procedure involving ML is finding suitable data and restructuring them in a format that can actually be used as training input.

3.2.1 The data-set

The data-set used in this work was the VASP output (an OUTCAR file) from a 100k step Monte-Carlo simulation of a 64 atom cell of the NV-center, i.e., 63 carbon atoms and one nitrogen. For this, VASP was provided with 100k random (with some bounds) configurations for which the forces on each atom were, as well as the corresponding value of the D-tensor at the NV-center were computed.
This provided a data set with 100k mappings of cell configurations to D-tensor values which was used to gather the training data for the NNs.

3.2.2 Generating the training data

For the Molecular dynamics simulation in the next section, NNs of two kinds will be necessary. First, to actually do the MD, NNs that map atomic positions to forces on each atom are needed, which can then be used to compute the next position of each atom. Second, we will need a NN that maps atomic positions, specifically those around the NV-center, to a D-tensor value. Accordingly, two different types of training sets are required in order to train those NNs.

For the data-sets concerned with mapping position to forces, the following parameterization has been chosen. For each atom, its respective 4 nearest neighbors are determined in each of the 100k samples of the MC-simulation. Then, the relative distance (in Angstrom) to those neighbors is calculated. Also, since there are two chemical species present, i.e., nitrogen (N) and carbon (C), those have to be discriminated as well. For simplicity, this is achieved by just representing each atom type by its nuclear charge. With this, individual data points in a training set for one atom are structured as follows (see figure 3.1).

```
6.000000 7.132440 7.133620 7.133620 0.001625 -0.000204 -0.000202 6.000000 -0.891710 -0.891620 -0.891620 6.000000 0.892110 -0
6.000000 7.132630 0.000790 7.132630 0.014136 -0.050348 0.051306 6.000000 -0.893370 -0.889420 -0.892560 6.000000 0.891570 -0.
6.000000 7.132840 0.001870 7.131730 0.023599 -0.095940 0.097931 6.000000 -0.894890 -0.887460 -0.893410 6.000000 0.891140 -0.
6.000000 7.133090 0.002770 7.131010 0.027120 -0.131491 0.134534 6.000000 -0.896110 -0.885950 -0.894060 6.000000 0.890920 -0.
6.000000 7.133390 0.003440 7.130540 0.022611 -0.153671 0.157218 6.000000 -0.896950 -0.885030 -0.894450 6.000000 0.890980 -0.
6.000000 7.133730 0.003830 7.130350 0.010492 -0.161828 0.165328 6.000000 -0.897340 -0.884830 -0.894540 6.000000 0.891340 -0.
6.000000 0.000100 0.003930 7.130460 -0.010595 -0.152673 0.154582 6.000000 -0.897260 -0.885350 -0.894290 6.000000 0.892010 -0
6.000000 0.000440 0.003750 7.130850 -0.035843 -0.127267 0.128205 6.000000 -0.896780 -0.886540 -0.893730 6.000000 0.892880 -0
6.000000 0.000720 0.003340 7.131470 -0.062067 -0.092429 0.090849 6.000000 -0.895960 -0.888270 -0.892910 6.000000 0.893900 -0
6.000000 0.000890 0.002770 7.132260 -0.085059 -0.050673 0.045819 6.000000 -0.894930 -0.890350 -0.891880 6.000000 0.894920 -0
6.000000 0.000910 0.002100 7.133130 -0.100813 -0.006828 -0.002117 6.000000 -0.893820 -0.892570 -0.890730 6.000000 0.895830 -
6.000000 0.000740 0.001420 0.000000 -0.106065 0.034428 -0.047875 6.000000 -0.892750 -0.894700 -0.889570 6.000000 0.896490 -0
6.000000 0.000380 0.000810 0.000790 -0.098847 0.068950 -0.087111 6.000000 -0.891820 -0.896520 -0.888470 6.000000 0.896810 -0
6.000000 7.133850 0.000320 0.001420 -0.078739 0.093418 -0.116338 6.000000 -0.891110 -0.897850 -0.887540 6.000000 0.896740 -0
6.000000 7.133170 7.134000 0.001830 -0.046905 0.105661 -0.132882 6.000000 -0.890650 -0.898580 -0.886870 6.000000 0.896240 -0
6.000000 7.132410 7.133870 0.002010 -0.006172 0.104540 -0.135195 6.000000 -0.890440 -0.898630 -0.886490 6.000000 0.895360 -0
6.000000 7.131640 7.133930 0.001940 0.039135 0.090393 -0.123200 6.000000 -0.890420 -0.898030 -0.886460 6.000000 0.894160 -0.
6.000000 7.130940 0.000150 0.001650 0.083905 0.064755 -0.098283 6.000000 -0.890540 -0.896860 -0.886780 6.000000 0.892750 -0.
6.000000 7.130390 0.000490 0.001190 0.122617 0.030583 -0.063130 6.000000 -0.890700 -0.895250 -0.887410 6.000000 0.891260 -0.
```

Figure 3.1: The first few lines in the data set for a NN that computes the forces on a carbon atom. Structure: Atomic number of the carbon atom at the center, followed by it's absolute position as well as the forces acting on it in all directions. After this, a list of it's 4 nearest neighbors with their atomic nuber followed by their relative position towards the atom at the center.

The atomic number of the atom at the center, followed by its absolute position (order: x y z coordinates) and forces acting on it in all three directions (order: F_x F_y F_z). After this the 4 nearest neighbors are listed, again with their atomic

number followed by their position (order: $r_x\ r_y\ r_z$) relative to the atom at the center.

The data-set for training the NN mapping positions to d-tensor values, was structured quite similarly (see figure 3.2). This time the nitrogen atom is placed at the center and it's 16 nearest neighbors are used instead of only 4. Again, the atomic number is used to discriminate different atom types and relative distances towards the nitrogen atom are used to parameterize positions.

```
7.000000 3.476890 3.477120 3.476520 -589.706000 -1483.846000 2073.552000 6.000000 -0.936290 0.802370 0.800580 6.000000 0.800240 0
7.000000 3.476090 3.476540 3.475410 -597.118000 -1466.036000 2063.154000 6.000000 -0.936300 0.803650 0.800220 6.000000 0.798370 0
7.000000 3.475360 3.475880 3.474380 -603.911000 -1449.374000 2053.285000 6.000000 -0.936440 0.804580 0.799830 6.000000 0.796720 0
7.000000 3.474750 3.475130 3.473460 -609.494000 -1435.247000 2044.741000 6.000000 -0.936730 0.805030 0.799370 6.000000 0.795390 0
7.000000 3.474310 3.474260 3.472690 -613.493000 -1424.514000 2038.006000 6.000000 -0.937170 0.804890 0.798860 6.000000 0.794490 0
7.000000 3.474050 3.473290 3.472090 -615.609000 -1417.905000 2033.514000 6.000000 -0.937750 0.804160 0.798300 6.000000 0.794030 0
7.000000 3.473990 3.472230 3.471690 -615.695000 -1415.715000 2031.410000 6.000000 -0.938410 0.802850 0.797740 6.000000 0.794050 0
7.000000 3.474120 3.471130 3.471500 -613.875000 -1417.902000 2031.777000 6.000000 -0.939130 0.801070 0.797210 6.000000 0.794510 0
7.000000 3.474390 3.470050 3.471520 -610.445000 -1424.060000 2034.504000 6.000000 -0.939870 0.798970 0.796770 6.000000 0.795300 0
7.000000 3.474770 3.469060 3.471770 -605.726000 -1433.492000 2039.219000 6.000000 -0.940570 0.796740 0.796490 6.000000 0.796340 0
7.000000 3.475190 3.468260 3.472220 -600.235000 -1445.299000 2045.534000 6.000000 -0.941200 0.794600 0.796410 6.000000 0.797470 0
7.000000 3.475610 3.467720 3.472850 -594.552000 -1458.503000 2053.055000 6.000000 -0.941680 0.792780 0.796530 6.000000 0.798600 0
7.000000 3.475970 3.467510 3.473630 -589.173000 -1472.077000 2061.250000 6.000000 -0.942010 0.791460 0.796890 6.000000 0.799590 0
7.000000 3.476240 3.467690 3.474520 -584.500000 -1485.087000 2069.587000 6.000000 -0.942140 0.790810 0.797460 6.000000 0.800380 0
7.000000 3.476400 3.468300 3.475460 -580.906000 -1496.744000 2077.650000 6.000000 -0.942050 0.790920 0.798190 6.000000 0.800900 0
7.000000 3.476450 3.469340 3.476400 -578.534000 -1506.454000 2084.989000 6.000000 -0.941730 0.791830 0.799040 6.000000 0.801150 0
7.000000 3.476400 3.470800 3.477290 -577.460000 -1513.779000 2091.239000 6.000000 -0.941180 0.793500 0.799940 6.000000 0.801170 0
```

Figure 3.2: The first few lines in the data set for the NN that computes the diagonal D-tensor values at the NV-center. Structure: Atomic number of the nitrogen atom at the center, followed by it's absolute position as well as the forces acting on it in all directions. After this, a list of it's 16 nearest neighbors with their atomic nuber followed by their relative position towards the atom at the center.

Since, the original data-set was computed in a (inherently random) MD-simulation, it is not guaranteed that the N nearest neighbors (ordered by distance) around an atom appear in the same order in each MD-run. Therefore, to maintain consistency in the training data-sets, care has to be taken to ensure that neighboring atoms appear in the exact same sequence for each data point (line in the data-set), even if the, e.g. first atom, in the list of nearest neighbors is not always the one with the closest distance. This can be ensured by labeling each of the 64 atoms with an index and sorting the nearest neighbors by index rather than by absolute distance.

3.2.3 Network structure

For the NNs, computing the forces on individual atoms, a simple sequential structure (similar to the one described in section 2.3) with 17 input neurons, one hidden layer of 51 neurons and a rectified linear unit (with default parameters) as an activation function as well as 3 output neurons has been chosen. The weights for the connections between neurons are initialized using a normal distribution (i.e., kernel_initializer='normal'). The Keras implementation is shown in listing 3.1.

```
1 inputDim = 4*nearestN + 1
2 outputDim = 3
3
4 model = Sequential()
5 model.add(Dense(3*inputDim, input_dim=inputDim, kernel_initializer='
      normal', activation='relu'))
6 model.add(Dense(outputDim, kernel_initializer='normal'))
```

Listing 3.1: Python code for setting up the structure for the NNs used to map atom position to forces. Here 'nearestN' is 4 so the NN has 17 input parameters (atom type and 3 relative positions for each nearest neighbor plus the atom type of the central atom), followed by a hidden layer with 51 neurons and the results are mapped to 3 output neurons (one per force in each direction).

The NN for mapping atomic positions to D-tensor values is structured in similar fashion. Here, there are 65 input parameters (due to considering 16 nearest neighbors instead of just 4), 650 neurons in the hidden layer and again 3 neurons in

the output layer. As before, the activation function for the hidden layer is a rectified linear unit and the initial weights are set using a normal distribution. The corresponding Keras implementation can be seen in listing 3.2

```
inputDim = 4*nearestN + 1
outputDim = 3

model = Sequential()
model.add(Dense(inputDim*10, input_dim=inputDim, kernel_initializer='
    normal', activation='relu'))
model.add(Dense(outputDim, kernel_initializer='normal'))
```

Listing 3.2: Python code for setting up the structure for the NN used to map atom position to the diagonal D-tensor values at the NV-center. Here 'nearestN' is 16 so the NN has 65 input parameters (atom type and 3 relative positions for each nearest neighbor plus the atom type of the central atom), followed by a hidden layer with 650 neurons and the results are mapped to 3 output neurons (one for Dxx, Dyy and Dzz respectively).

3.2.4 Training

The NNs defined in the previous section are then trained on their respective datasets. In total there are 65 NNs, 64 for computing the forces on each individual atom and 1 for computing the diagonal D-tensor values at the NV-center. The Keras implementation for the training process is shown in listings 3.3 and 3.4.

```
inputDim = 4*nearestN + 1
outputDim = 3
batchSize = 1000
numEpochs = 2000
opt = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999,
    epsilon=None, decay=0.0, amsgrad=False)

model = Sequential()
model.add(Dense(3*inputDim, input_dim=inputDim, kernel_initializer='
    normal', activation='relu'))
model.add(Dense(outputDim, kernel_initializer='normal'))
model.compile(loss='mean_squared_error', optimizer='Adam')

model.fit(x_data[0:int(numSamples*8/10)], y_data[0:int(numSamples
    *8/10)], epochs=numEpochs, batch_size=batchSize)
score = model.evaluate(x_data[int(numSamples*8/10):numSamples], y_data
    [int(numSamples*8/10):numSamples], batch_size=batchSize)
```

Listing 3.3: Python code for training the NNs used to map atomic position to forces.

As a loss function a default 'mean squared error' loss is used and the Optimizer is of type 'Adam' (see section 2.3.3 for details on loss functions and optimizers in the context of training NNs).
For the NN, used to compute the D-tensor, the training process is given more time (i.e. more epochs), since it is larger and therefore has to vary significantly more weights in order to find an adequately performing configuration.

```
inputDim = 4*nearestN + 1
outputDim = 3
batchSize = 1000
numEpochs = 10000
opt = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999,
    epsilon=None, decay=0.0, amsgrad=False)

model = Sequential()
model.add(Dense(inputDim*10, input_dim=inputDim, kernel_initializer='
    normal', activation='relu'))
model.add(Dense(outputDim, kernel_initializer='normal'))
model.compile(loss='mean_squared_error', optimizer='Adam')

```

```
12 model.fit(x_data[0:int(numSamples*8/10)], y_data[0:int(numSamples
       *8/10)], epochs=numEpochs, batch_size=batchSize)
13 score = model.evaluate(x_data[int(numSamples*8/10):numSamples], y_data
       [int(numSamples*8/10):numSamples], batch_size=batchSize)
```

Listing 3.4: Python code for setting up the structure for the NNs used to map atom position to forces. Here 'nearestN' is 4 so the NN has 17 input parameters (atom type and 3 relative positions for each nearest neighbor plus the atom type of the central atom), followed by a hidden layer with 51 neurons and the results are mapped to 3 output neurons (one per force in each direction).

As can be seen in the code listings, a 20% cross-validation rate has been used to evaluate the performance of the NNs. Basically, this means that (randomly selected) 80% of the training samples are used as the actual training data on the NNs, while the remaining 20% of the samples are retained. The latter are then used to validate the performance of the trained NN.
This approach is considered good practice to avoid overfitting, since the NN has to perform well on data it has never 'seen' before and is especially useful if the training set is rather small, as with only 100k samples used here.

3.2.5 Further remarks on the selection and training of the NNs

Concerning the chosen network structure and training process, outlined above, some points deserve mentioning.

Even though a rather simple and straightforward structure for the NNs has been selected in the end, considerably more complex and larger network layouts have been tried as well. However, no significant improvements to the performance, measured only by the chosen loss function, not during the actual simulation, could be achieved. The simpler and smaller configuration was therefore chosen over the larger, more elaborate ones, since the gains in accuracy were deemed too insignificant to outweigh the vastly larger training effort and further, a reduction in model complexity also decreases the possibility for overfitting.
Similarly, using other loss-functions (including custom functions) or different parameters for the latter, as well as varying the Adam optimizer parameters did not lead to noticeable improvements in accuracy. Again, the default parameterization was chosen.
However, using the Adam optimizer leads to significant speed-ups and accuracy gains in the training phase, compared to the results achieved using other optimizers.

3.3 Simulation

In this work a Molecular Dynamics (MD) simulation is performed to compute the diagonal D-tensor values of the NV-center in a diamond lattice. The MD is augmented by a thermostat to introduce a temperature into the simulation which allows for evaluating the D-tensor at different temperatures, thereby getting an estimate for the D-tensor's temperature dependence.

3.3.1 Thermostat

In this work a Langevin thermostat is used to introduce a temperature to the simulation. Here, temperature is obtained by modifying Newton's equations of motion in the following way [17]

$$\frac{d}{dt}r = \frac{p_i}{m_i} \tag{3.1}$$

$$\frac{d}{dt}p = F_i - \gamma_i p_i + f_i \tag{3.2}$$

$$\sigma_i^2 = \frac{2M_i\gamma_i k_B T}{\Delta t} \tag{3.3}$$

where F_i is the force acting on atom i due to the interaction potential and γ_i is a friction coefficient. The f_i are random forces representing temperature dependent random 'kicks' to the atoms. The random values are sampled from a Gaussian distribution with variance σ_i which is dependent on the friction as well as the chosen temperature.
The Python implementation of the Langevin thermostat used in this work can be seen in listing 3.5

```
sigmaC = np.sqrt(2*gamma*massC*kB*T)
sigmaN = np.sqrt(2*gamma*massN*kB*T)

randP_C = np.random.normal(0, sigmaC, numSteps*(numAtoms - 1)*3)
randP_N = np.random.normal(0, sigmaN, numSteps*3)

# update positions of atoms using forces and temperature (via a
    Langevin thermostat)
for j in range(numAtoms):

  # Update for N-atom
  if(j == (numAtoms - 1)):

    P2[j][0] = P1[j][0] + forces[j][0]*del_t - gamma*P1[j][0] +
    randP_N[i*3]
    P2[j][1] = P1[j][1] + forces[j][1]*del_t - gamma*P1[j][1] +
    randP_N[i*3 + 1]
    P2[j][2] = P1[j][2] + forces[j][2]*del_t - gamma*P1[j][2] +
    randP_N[i*3 + 2]

    positions[j][0] += (P2[j][0]*del_t / massN)
    positions[j][1] += (P2[j][1]*del_t / massN)
    positions[j][2] += (P2[j][2]*del_t / massN)
  # Update for C-atoms
  else:

    P2[j][0] = P1[j][0] + forces[j][0]*del_t - gamma*P1[j][0] +
    randP_C[i*(numAtoms-1)*3 + 3*j]
    P2[j][1] = P1[j][1] + forces[j][1]*del_t - gamma*P1[j][1] +
    randP_C[i*(numAtoms-1)*3 + 3*j + 1]
    P2[j][2] = P1[j][2] + forces[j][2]*del_t - gamma*P1[j][2] +
    randP_C[i*(numAtoms-1)*3 + 3*j + 2]

```

```
    positions[j][0] += P2[j][0]*del_t / massC
    positions[j][1] += P2[j][1]*del_t / massC
    positions[j][2] += P2[j][2]*del_t / massC

  P1 = P2
```

Listing 3.5: Implementation of the Langevin thermostat in Python. Carbon and nitrogen atoms are considered differently since their different mass influences their dynamics as well as the random distribution for the thermostat.

3.3.2 Molecular Dynamics using NNs

The molecular dynamics in this work are performed using NNs. To achieve this, for each atom in the considered cell a NN was trained to map the relative position of the surrounding nearest neighbors to the resulting force acting on the central atom. With this it is possible to set up an iterative procedure where one starts with the 64 atoms at an initial position, determines the nearest neighbors and computes the forces acting on them using the 64 corresponding NNs. Using the calculated forces, together with a random 'push' from the thermostat, the positions of the atoms can be updated. Then, the procedure is repeated with the new atomic positions.
The Python implementation for computing the forces on each atom using NNs can be seen in listing 3.6, while the update of positions using those forces is shown in listing 3.5 in the previous section.

```
# Loop over all MD steps
for i in range(numSteps):
  for j in range(numAtoms):
    forces[j] = atomModels[j].predict(input[j:j+1])[0]

  # Perform position update using the forces
```

Listing 3.6: Python implementation for computing the forces on each atom using their respective NN. The update of position can be seen in listing 3.5.

3.3.3 Computing the D-tensor using a NN

Similarly to computing the forces, the diagonal components of the D-tensor were calculated after each MD step using the NN trained for estimating the D-tensor from the atomic configuration of the NV-center and its 16 nearest neighbors. The Python code for this is depicted in listing 3.7.

```
# Loop over all MD steps
for i in range(numSteps):
  for j in range(numAtoms):
    forces[j] = atomModels[j].predict(input[j:j+1])[0]

  d_tensor[i] = model_d.predict(input_d)[0]

  # Perform position update using the forces
```

Listing 3.7: Python implementation for computing the diagonal components around the NV-center using a NN. The update of position can be seen in listing 3.5.

3.3.4 Simulation setup

The general structure of the simulations performed for this work can be seen in listing 3.8.

```
temps = range(10,560,10)
# Loop over all considered temperatures
for m in range (len(temps)):
  T = temps[m]
```

```
5
6   # Preparations and initializations
7   #--------------#
8   ...
9   #--------------#
10
11  # Loop over all MD steps
12  for i in range(numSteps):
13
14    # Estimate D-tensor and forces using NNs
15    d_tensor[i] = model_d.predict(input_d)[0]
16
17    for j in range(numAtoms):
18      forces[j] = atomModels[j].predict(input[j:j+1])[0]
19
20    # update positions of atoms using forces and temperature (via a
      Langevin thermostat)
21    for j in range(numAtoms):
22      ...
23    P1 = P2
24
25    # updating input for predictor models
26    updateInput(input, atomicNumbers, positions, idx_nn, numNearestN,
      numAtoms, latticeX, latticeY, latticeZ)
27    updateInput(input_d, atomicNumbers, positions, idx_nn_d,
      numNearestN_d, 1, latticeX, latticeY, latticeZ)
28
29  # Compute D-tensor averages and variance
30  ...
```

Listing 3.8: Python implementation of the MD-simulations. First the range of temperatures is defined and then a full MD is done for each temperature. In each MD-step the D-tensor as well as the forces on each atom are estimated from the current configuration using NNs. The forces are used to update the position of all atoms and this new configuration is processed to generate the next input for the NNs. Finally, after each MD-run, i.e., for each tempearture, the estimated D-tensor values are averaged in order to obtain a D-tensor T-dependence.

First, the number of MD-steps, i.e. the number of different configurations the D-tensor values will be averaged over, is fixed. Then, the temperature range that will be considered is defined. In this work, the temperature is set to rise from 10K to 550K in steps of 10K. For each of these temperatures, a MD-simulation with the desired number of steps is then run and in each step, the D-tensor at the NV-center as well as the forces on all the atoms are estimated from the current configuration using NNs. Using those forces, together with a temperature-dependent random force provided by the thermostat, the positions of all the atoms are updated and used to generate the input for the NNs for the next step. Finally, after each MD run at a given temperature the collected D-tensor values are averaged and saved in order to obtain their T-dependence.

The '*updateInput*' function from listing 3.8 is shown in listing 3.9. Here, the new atomic configuration obtained after each MD-step is used to compute new input (the structure of which is explained in section 3.2.2) for the NNs. As mentioned in section 3.2.2, the sequence of nearest neighbors should remain the same in order to provide consistent input to the NNs. For this reason the nearest neighbors are not computed by their absolute distance to the central atom in each new configuration but rather provided statically in '*list_nn*'. Also, one has to consider the periodicity of the considered atomic structure and project atoms leaving the boundaries back into the cell.

```
1 def updateInput(input, atomicNumbers, positions, list_nn, numNearestN,
      numAtoms, latticeX, latticeY, latticeZ):
2 for i in range(numAtoms):
3   if(numAtoms == 1):
4     # inputs for D-tensor model are nearest neighbors to N atom
```

```
    x = positions[len(atomicNumbers)-1][0]
    y = positions[len(atomicNumbers)-1][1]
    z = positions[len(atomicNumbers)-1][2]
  else:
    x = positions[i][0]
    y = positions[i][1]
    z = positions[i][2]
  for j in range(numNearestN):
    if(numAtoms == 1):
      idx = int(list_nn[0][j])
    else:
      idx = int(list_nn[i][j])

    if(idx < len(atomicNumbers)):
      rx = x - positions[idx][0]
      ry = y - positions[idx][1]
      rz = z - positions[idx][2]
      if (abs(rx) > latticeX/2):
        rx = rx - latticeX*np.sign(rx)
      if (abs(ry) > latticeY/2):
        ry = ry - latticeY*np.sign(ry)
      if (abs(rz) > latticeZ/2):
        rz = rz - latticeZ*np.sign(rz)
      input[i][1 + 4*j:1 + 4*j + 4] = np.array([atomicNumbers[idx], rx
    , ry, rz])
```

Listing 3.9: Python implementation for generating input to the NNs using an atomic configuration. In order to maintain the nearest neighbor sequence for each atom, the atomic indices of the nearest neighbors are provided by 'list_nn' instead of being computed each time (see section 3.2.2). Also, an atom leaving the boundaries of the considered cell has to be projected back into it.

Lastly, the averaging of the estimated D-tensor values is done using a standard mean and the variance is computed as the expectation of the squared deviations from the mean (see listing 3.10).

```
# Compute average and variance of D-tensor values
sum1 = 0.0
sum2 = 0.0
sum3 = 0.0
for i in range(numSteps):
  sum1 += d_tensor[i][0]
  sum2 += d_tensor[i][1]
  sum3 += d_tensor[i][2]


sum1 /= numSteps
sum2 /= numSteps
sum3 /= numSteps

var1 = 0.0
var2 = 0.0
var3 = 0.0
for i in range(numSteps):
  var1 += (sum1 - d_tensor[i][0])*(sum1 - d_tensor[i][0])
  var2 += (sum2 - d_tensor[i][1])*(sum2 - d_tensor[i][1])
  var3 += (sum3 - d_tensor[i][2])*(sum3 - d_tensor[i][2])

var1 /= numSteps
var2 /= numSteps
var3 /= numSteps
```

Listing 3.10: Python code for averaging the estimated D-tensor values. Here, a standard mean and variance are used.

Chapter 4

Results

In this chapter the results of the simulations described before are displayed and a short interpretation is given. Finally, this work is closed by outlining some options for further research as well as providing some thoughts concerning the long-term effects the use of machine learning in science might have on the field in general.

4.1 D-tensor Temperature dependence

For this work three MD simulations, facilitating the use of NNs for computing (estimating) forces on atoms as well as D-tensor values, as described in section 3.3.4, were done. The individual simulations differed only in the number of MD-steps performed, which were 100k, 300k and 1000k respectively. The simulated temperature ranged from 10 K to 550 K in steps of 10 K in all MD runs. Below, the results are displayed for estimating the zz-component of the D-tensor.

4.1.1 100k average

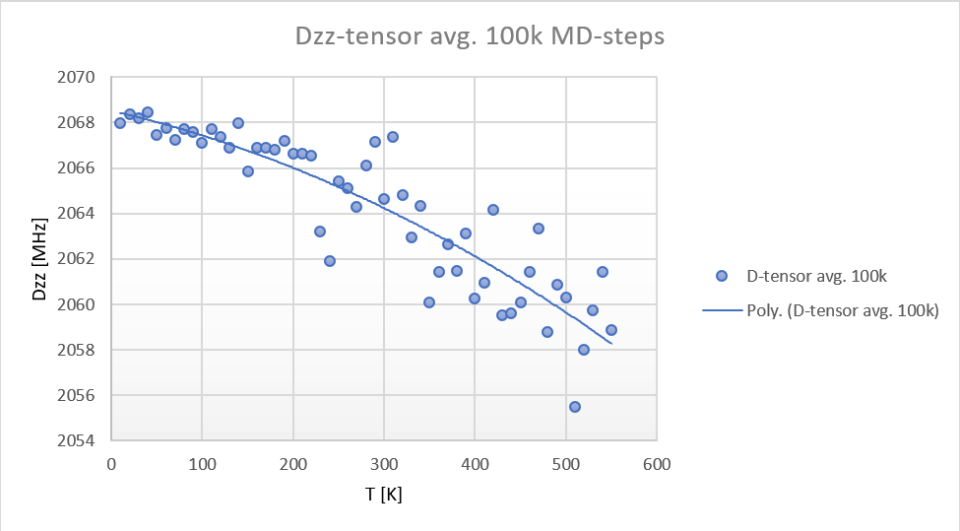

Figure 4.1: Averaged estimates of the zz-component of the D-tensor at the NV-center for temperatures ranging from 10 K to 550 K. 100k MD-steps were performed at each temperature. The trendline is a fitted 2nd order polynomial and the mean-squared error of this fit is 2.20 MHz^2.

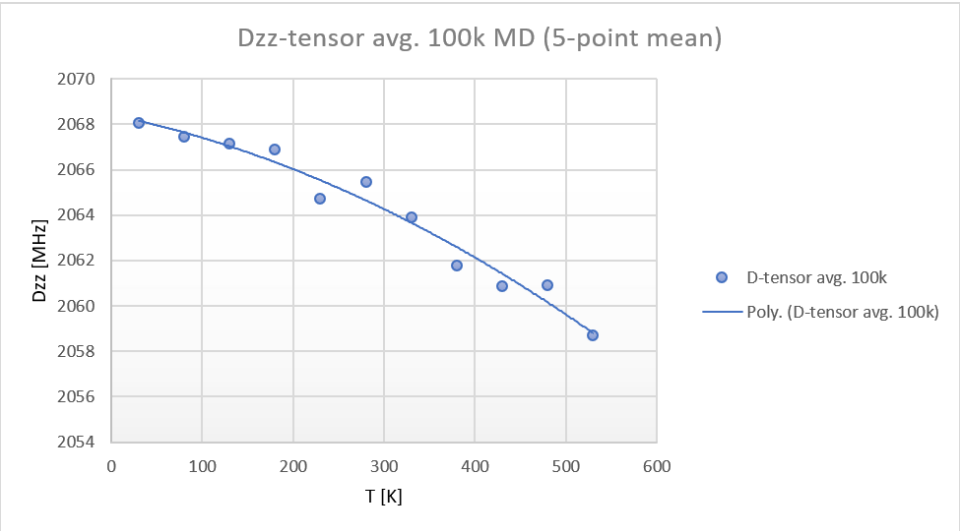

Figure 4.2: The same dataset as in figure 4.1. However, here the means of 5 consecutive datapoints are shown. The trendline is a fitted 2nd order polynomial and the mean-squared error of this fit is 0.29 MHz^2.

4.1.2 300k average

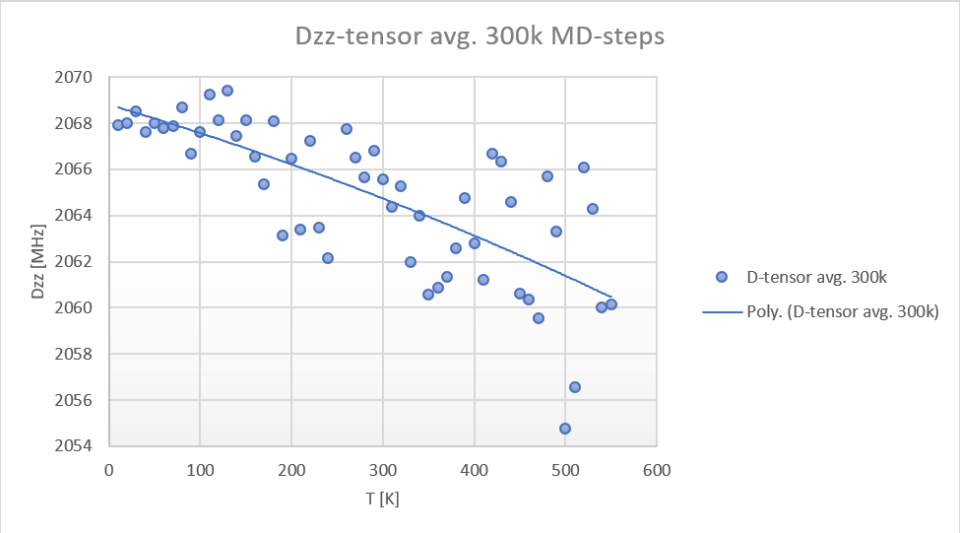

Figure 4.3: Averaged estimates of the zz-component of the D-tensor at the NV-center for temperatures ranging from 10 K to 550 K. 300k MD-steps were performed at each temperature. The trendline is a fitted 2nd order polynomial and the mean-squared error of this fit is 5.41 MHz^2.

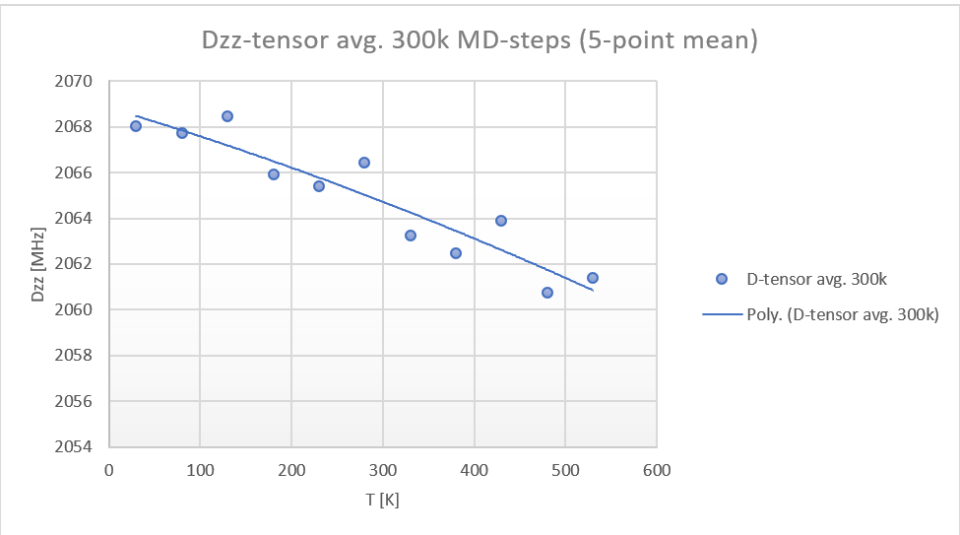

Figure 4.4: The same dataset as in figure 4.3. However, here the means of 5 consecutive datapoints are shown. The trendline is a fitted 2nd order polynomial and the mean-squared error of this fit is 0.85 MHz^2.

4.1.3 1000k average

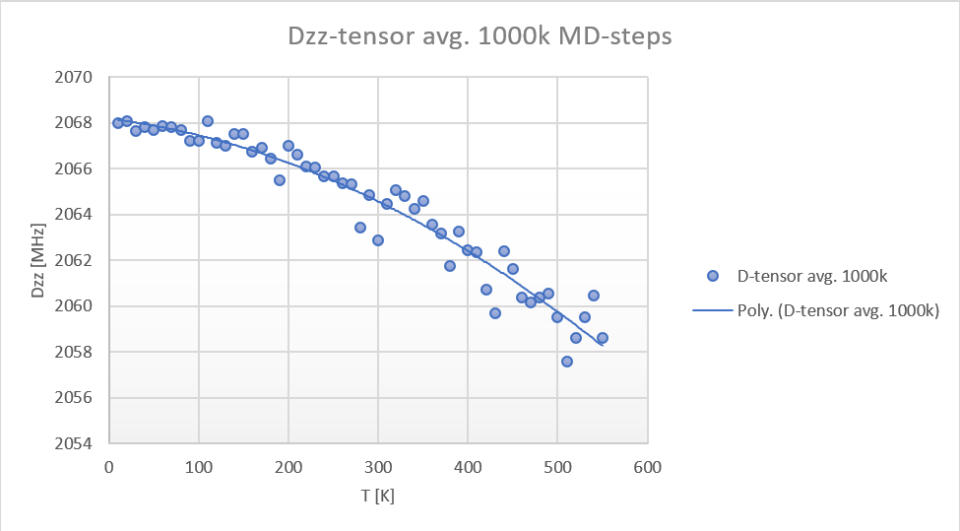

Figure 4.5: Averaged estimates of the zz-component of the D-tensor at the NV-center for temperatures ranging from 10 K to 550 K. 1000k MD-steps were performed at each temperature. The trendline is a fitted 2nd order polynomial and the mean-squared error of this fit is 0.52 MHz^2.

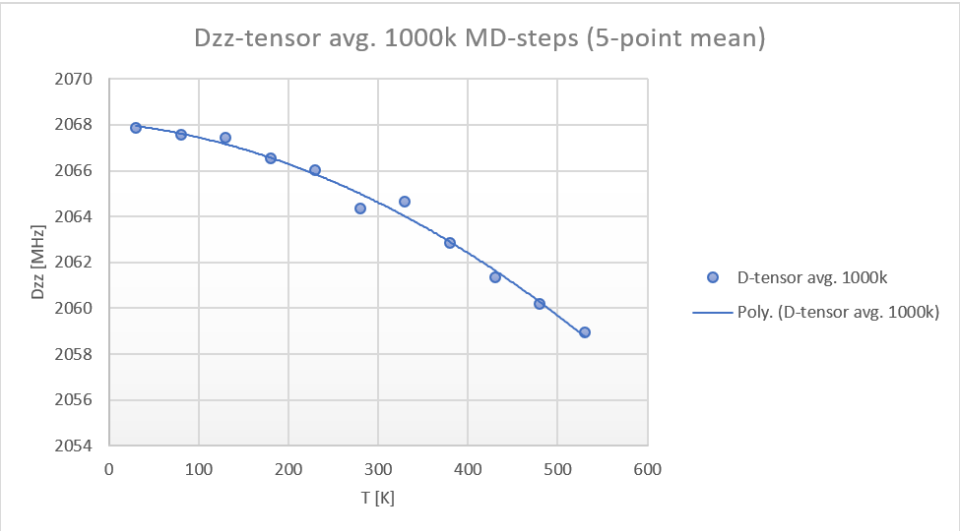

Figure 4.6: The same dataset as in figure 4.5. However, here the means of 5 consecutive datapoints are shown. The trendline is a fitted 2nd order polynomial and the mean-squared error of this fit is 0.10 MHz^2.

4.1.4 Variance and fit quality

In order to get a feeling on how precise the predictions of D-tensor values are, the variance of the latter has also been computed and is displayed in figure 4.7 below.

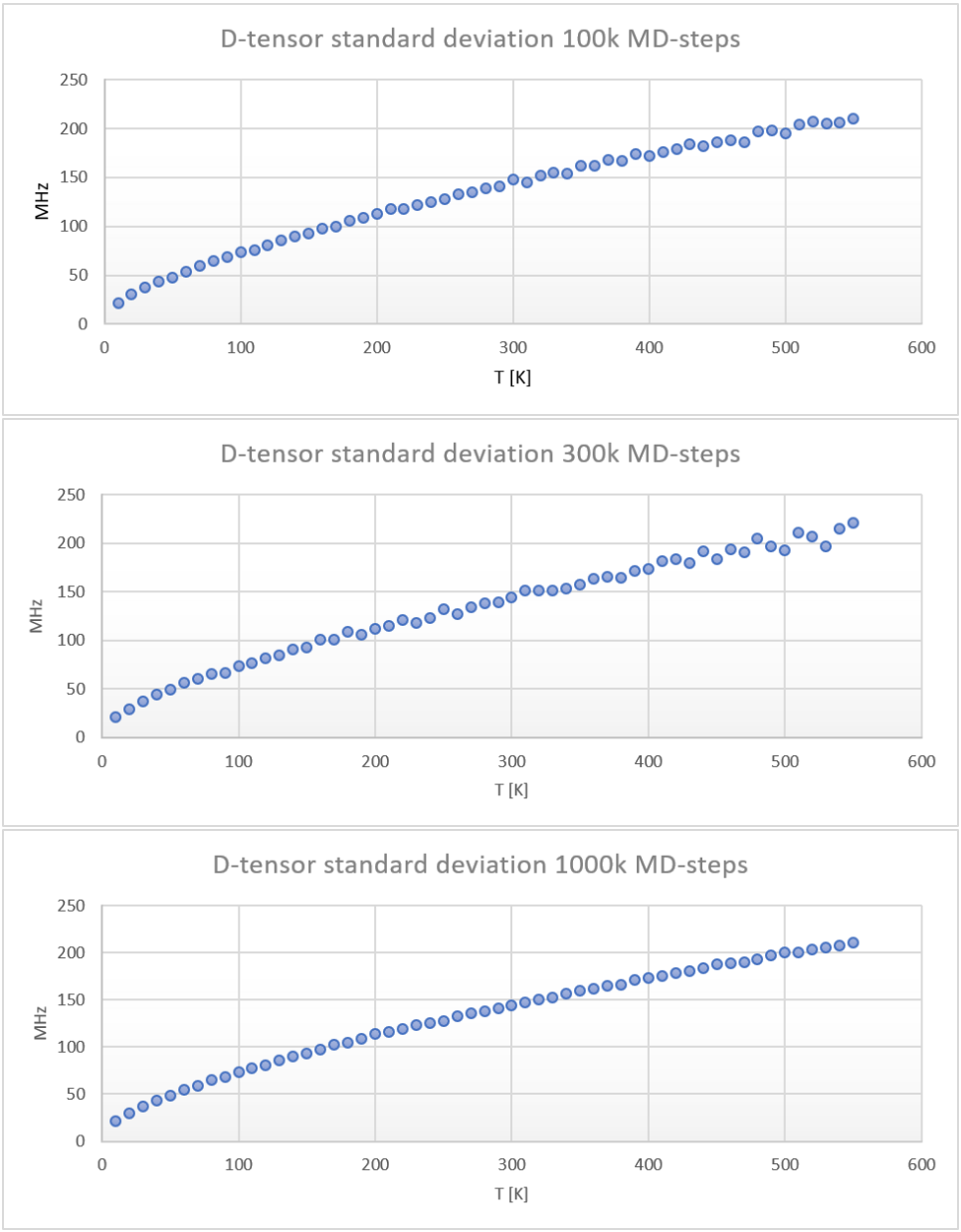

Figure 4.7: Variance of the estimated D-tensor zz-component as a function of temperature for simulations performed with 100k, 300k, and 1000k MD-steps respectively.

Also, table 4.1 lists the second order polynomials as well as the mean-squared-error relative to the estimated values shown in figures 4.1, 4.3 and 4.5.

MD-steps	coeff(x^2)	coeff(x^1)	coeff(x^0)	MSE
100k	-1.73e-5	-0.0091	2068.5	2.119
300k	-6.48e-6	-0.0116	2068.8	5.410
1000k	-2.40e-5	-0.0049	2068.2	0.519

Table 4.1: Coefficients and mean-squared error of 2nd order polynomial fits to the D-tensor zz-component estimates shown in figures 4.1, 4.3 and 4.5.

And table 4.2 shows the polynomial coefficients and MSEs for the 5-point mean of the D-tensor estimates depicted in figures 4.2, 4.4 and 4.6.

MD-steps	coeff(x^2)	coeff(x^1)	coeff(x^0)	MSE
100k	-1.87e-5	-0.0082	2068.4	0.29
300k	-5.90e-6	-0.0119	2068.8	0.85
1000k	-2.61e-5	-0.0038	2068.1	0.095

Table 4.2: Coefficients and mean-squared error of 2nd order polynomial fits to the D-tensor zz-component 5-point means shown in figures 4.2, 4.4 and 4.6.

4.2 Comparison & interpretation

Looking at the estimates of the D-tensor zz-component, one can see a consistent downward trend in all three simulations, which is already quite remarkable given that the data-set used was not well converged (i.e., forces and D-tensor values were not computed very accurately) and computed at a single temperature of 10 K. Especially, the results for the run with 1000k MD-steps follow a polynomial fit quite well, suggesting a possibility of finding a reasonably simple analytic form for the T-dependence of the zz-component of the NV-center D-tensor.

Looking at the variance (figure 4.7) one can see that it monotonically increases with temperature, basically independent of the number of MD-runs performed. This suggests that the increase in variance might be governed primarily by the thermostat.

Nonetheless, increasing the number of MD-runs appears to lead to less chaotic results, i.e., the predicted values seem to approach some analytic form as can be seen in table 4.1 and 4.2. Here, the mean squared error (MSE) for a 2nd order polynomial fit is significantly smaller for the simulation with 1000k MD-steps per temperature point compared to performing only 100k or 300k MD-steps.

Of course, one has to consider if this boost in accuracy is worth the linear increase in computational effort. On the same note, using 5-point means of consecutive temperature points allows an even smoother fit to a 2nd order polynomial as shown in figures 4.2, 4.4 and 4.6. However, again this comes at the cost of a 5-fold increase in computation time.

One point of concern is the noticeably bad performance of the simulation with 300k MD-steps per temperature point, since it would be expected to produce results somewhere in between the simulations with 100k and 1000k MD-steps. There is no apparent explanation for this discrepancy and it requires further testing, e.g., increasing the number of MD-steps more slowly.

4.2.1 Shortcomings of the current approach

Even though the results above suffice as a proof of principle, the chosen approach has a number of shortcomings that should be addressed.

Inflexible

The chosen approach in this work is rather inflexible and generalizes badly, since the NNs are trained for each atom individually and for a specific neighborhood configuration, i.e., types of surrounding atoms. This means that the trained NNs cannot be reused in different cell configurations, only in expansions of the cell structure they are trained in. E.g., the NNs trained in this work could be used (with minor adaptations) to perform a MD-simulation of a larger cell of carbon atoms and NV-centers.
However, one would ideally like to train one network for each atom type that can be used in any surrounding, i.e., one for carbon and one for nitrogen in the setup used in this work. An approach that explores this uses atom-centered symmetry functions to model the neighborhood of an atom (see section 4.3.7).

Computationally expensive

Another inconvenience of the approach used in this work is the computational effort it takes to reach useful results. Looking at the results in the previous section,

only the simulation with 1000k MD-steps at each temperature point gave decent results, in the sense of providing a somewhat smooth T-dependence of the D-tensor zz-component. However, a MD-simulation with that number of steps took almost 2 days per temperature point (running on a single GPU). Even though, the smoothness of the estimated T-dependence could probably be increased by using a more elaborate thermostat such that a 100k step MD gives similar results, this approach is probably still too inefficient (though significantly better than DFT methods) to use in production, especially for larger systems.

4.3 Further Research

The present work was meant as a feasibility study for data-driven computing to support DFT-calculations. Therefore, the search for a suitable ML model, training process and structure of the input data was far from exhaustive leading to simulation results that are, in all likelihood, sub-optimal.
The following sections will give some ideas on how to improve on the procedure described in this work as well as outline other possible approaches.

4.3.1 Get a better dataset

Since a ML-approach is always only as good as the underlying data it is trained on, the first and most important improvement would be to generate a better data-set, i.e. one featuring more accurate computed values for forces and D-tensor. This might allow to either get more accurate predictions of NNs trained with this data-set or require less samples to train the NNs to a given level of accuracy.

4.3.2 Increase the number of nearest neighbors

One low effort way to possibly increase the accuracy of the simulation results would be to tinker with the number of nearest neighbors used in predicting forces and D-tensor values. E.g., considering more than just the 4 nearest neighbors for predicting forces should allow the ML model to also learn interactions, atoms that are further apart might have with each other. Naturally, it could easily be tested whether those long-distance interactions are negligible in a solid crystal (or at least not worth the additional computational effort).

4.3.3 Network structure & training process

Even though some effort has been undertaken trying different structures and sizes as well as varying optimizer parameters and optimizer types, there is still more to be done. Most importantly, the performance of the above variations were only evaluated with respect to achieved network accuracy according to the performance on the data-set (split into a training and validation set; see section 3.2.4) not on the performance of the resulting NNs in actual MD simulations. So it might be worthwhile to test the above mentioned changes with regards to the predicted D-tensor values as well.
Here, approaches could include

- **Varying number of hidden layers and layer sizes:** Here one could try increasing network complexity for gains in accuracy or decreasing network complexity for reducing computational effort. If the latter results in only a slight drop in accuracy this loss might be acceptable in favor of allowing for more simulation steps and better statistical smoothing.
- **Varying the optimizer type and optimizer parameters:** As mentioned, there are many different types of optimizers for converging NNs and many of them feature a multitude of parameters that can be tweaked individually. So, as with network structure, there is most likely some room for improvement down this avenue, such as speeding up the training process. This might prove to be especially useful if a larger, more complex, network structure is chosen.
- **Trying different ML approaches other than neural networks:** Neural Networks are by no means the only tools the field of machine learning has to offer. So one could try to use other techniques like regression or K-means and

potentially benefit from reductions in computational load by using a simpler yet sufficiently accurate approach.

4.3.4 Introduce a feedback-loop into VASP

Another way of improving performance would be to introduce a feedback-loop into VASP. Here, the idea would be to train a model on an initial data-set and evaluate its performance for various atomic configurations. After finding those configurations on which the current model performs badly (i.e., has a large error), VASP is used to compute new data-points that are similar to those configurations where the error was largest. The new data-points could then be used to retrain the model in the hope, that the resulting new model will give better predictions for those configurations. This process is then iterated until the final model performs sufficiently well on the entire input-space. An approach of this kind is tried in [10].

4.3.5 Reformat training data

Looking at the training data (figures 3.1 and 3.2) we can see that the values for relative positions as well as for the forces do not differ too much between individual samples. One could try rescaling those values in such a way that those differences become more significant, i.e., increasing the 'distance' between samples in the input-space. Of course, it should be considered that the effect of atomic charge might be of different magnitude compared to changes in position, so this could also be reflected by the absolute size of the corresponding values in the samples.

4.3.6 Combinations of approaches

Of course, the approaches mentioned above do not have to be tried in isolation. Indeed, they will, in all liklihood, yield the best results when combined. E.g., increasing the number of nearest neighbors considered could benefit from the increased versatility of more complex network structures.

4.3.7 Different approach: Atom-centered symmetry functions

One might also pursue an entirely different approach where the NNs are not trained simply on relative distances and nuclear charges surrounding a central atom, but rather on a more elaborate representation of the atomic neighborhood. One such representation are atom-centered symmetry functions (ACSFs) pursued by e.g. Behler et al. (see [2]).
The basic idea here is to map the neighborhood of an atom to various overlaps of Gauss-kernels each centered on the distance of one of the neighboring atoms to the central particle. The machine learning model, in Behlers case also NNs, are then trained on the parameters of these Gauss-kernels.
This approach not only promises to be more general, allowing to train only a single NN for each atom type but also represents the physical reality better, i.e., considering atoms and there charge-distributions not as something localised but rather as a 'smeared out' distribution.

4.4 Conclusion

The idea behind this work was to explore the feasibility of using neural networks (NNs) to support calculations on quantum many-body systems. More specifically, to compute the evolution of such a system via a molecular dynamics simulation, where NNs are employed at each MD-step to map the atomic configuration onto forces acting on each atom as well as estimating an additional property of interest (in this case the zz-component of the D-tensor in the negatively charged nitrogen vacancy center in diamond).

As detailed in chapter 3.2, the chosen approach consisted of training one NN for each atom in the considered structure to compute the acting forces as well as one additional NN estimating the D-tensor value at the NV-center. The training data contained information on a fixed number of nearest neighbors, represented in a rather straight forward fashion, i.e., one parameter for atomic charge and 3 parameters for relative distance for each considered atom in the neighborhood.

The results (presented in section 4.1) suggest that useful information about a physical quantity can be extracted via this approach that would be hard to retrieve via other methods. In this case a temperature dependence, in a range of more than 500K, of the zz-component of the NV-center D-tensor could be obtained using a data-set computed for a fixed temperature of 10 K. Also, the general trend of the estimated T-dependence has a similar form as one that was obtained by fitting experimental data (see ([8], Fig. 6).

However, the approach chosen in this work proved to be too inflexible and computationally expensive to use in any real application as described in section 4.2.1. Therefore, Section 4.3 elaborates on a number of possible ways to expand on the presented approach such as

- Generate a better data-set to train on
- Vary network complexity and input size
- Parameterize the atomic neighborhood differently, e.g., using ACSFs

possibly tackling some of the shortcomings of the method described in this work.

Finally, one should ponder the implications using machine learning techniques such as NNs might have on the way science is done in general. For employing tools, that are essentially black-boxes, to study events might diminish our ability to perceive and contemplate the very nature of the processes we are trying to understand.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Jörg Behler. Perspective: Machine learning potentials for atomistic simulations. *The Journal of Chemical Physics*, 145:170901, 11 2016.

[3] K Burke. The abc of dft. 04 2007.

[4] François Chollet et al. Keras. `https://keras.io`, 2015.

[5] H. Hanzawa, Y. Nisida, and T. Kato. Measurement of decay time for the NV centre in Ib diamond with a picosecond laser pulse. *Diamond and Related Materials*, 6:1595–1598, October 1997.

[6] Ariful Haque and Sharaf Sumaiya. An overview on the formation and processing of nitrogen-vacancy photonic centers in diamond by ion implantation. *Journal of Manufacturing and Materials Processing*, 1(1), 2017.

[7] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Netw.*, 2(5):359–366, July 1989.

[8] Viktor Ivády, Tamás Simon, Jeronimo R. Maze, I. A. Abrikosov, and Adam Gali. Pressure and temperature dependence of the zero-field splitting in the ground state of nv centers in diamond: A first-principles study. *Phys. Rev. B*, 90:235205, Dec 2014.

[9] A Jarmola, Andris Berzins, J Smits, Krisjanis Smits, Juris Prikulis, Florian Gahbauer, R Ferber, Donats Erts, Marcis Auzinsh, and Dmitry Budker. Longitudinal spin-relaxation in nitrogen-vacancy centers in electron irradiated diamond. *Applied Physics Letters*, 107, 11 2015.

[10] Ryosuke Jinnouchi, Jonathan Lahnsteiner, Ferenc Karsai, Georg Kresse, and Menno Bokdam. Phase transitions of hybrid perovskites simulated by machine-learning force fields trained on the fly with bayesian inference. *Phys. Rev. Lett.*, 122:225701, Jun 2019.

[11] G. Kresse and J. Furthmüller. Efficient iterative schemes for ab initio total-energy calculations using a plane-wave basis set. *Phys. Rev. B*, 54:11169–11186, Oct 1996.

[12] G. Kresse and J. Furthmüller. Efficiency of ab-initio total energy calculations for metals and semiconductors using a plane-wave basis set. *Computational Materials Science*, 6(1):15 – 50, 1996.

[13] G. Kresse and J. Hafner. Ab initio molecular-dynamics simulation of the liquid-metal–amorphous-semiconductor transition in germanium. *Phys. Rev. B*, 49:14251–14269, May 1994.

[14] Wes McKinney. Pandas. `https://pandas.pydata.org/`, 2008.

[15] Pankaj Mehta, Marin Bukov, Ching-Hao Wang, Alexandre Day, Clint Richardson, Charles Fisher, and David Schwab. A high-bias, low-variance introduction to machine learning for physicists. *Physics Reports*, 810, 03 2018.

[16] LLC RocketTheme. About vasp. `https://www.vasp.at/index.php/about-vasp/59-about-vasp`. Accessed: 2018-07-02.

[17] VASP. Langevin thermostat. `https://www.vasp.at/wiki/index.php/Langevin_thermostat`. Accessed: 2020-11-02.

[18] Wikipedia. Artificial neural networks. `https://en.wikipedia.org/wiki/Artificial_neural_network#Components_of_an_artificial_neural_network`. Accessed: 2019-01-07.

[19] Wikipedia. Density functional theory. `https://en.wikipedia.org/wiki/Density_functional_theory#Hohenberg\OT1\textendashKohn_theorems`. Accessed: 2019-04-16.

List of Figures

List of Tables

Listings