MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

„Neural network acceleration techniques applied to SchNet"

verfasst von / submitted by

Maximilian Xaver Tiefenbacher, BSc

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien, 2021 / Vienna, 2021

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	UA 066 862
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Masterstudium Chemie
Betreut von / Supervisor:	Dr. Philipp Marquetand, Privatdoz.

Acknowledgements

First of all, I would like to thank my supervisor Priv.-Doz. Dr. Philipp Marquetand for the opportunity to work on my master thesis under his supervision. His expertise and understanding encouraged me throughout the time and guided me through the new challenges during my research and writing my thesis. I am especially grateful for his extraordinary support, given the extraordinary circumstances of COVID19.

I also want to thank head of the research platform ViRAPID, Univ.-Prof. Dr. Dr. h.c. Leticia González, for the chance to be part of the platform and being able to work with experts from different fields of science. Especially Brigitta Bachmair, MSc and Madlen Reiner, MSc have been a tremendous support during my research. Our weekly zoom calls have been a great source of encuragment in these challenging times. Furthermore I would like to thank Dr. Markus Oppel for providing his expertise in the field of informatics and all members of the Gonzales research group for their support.

Finally let me express my gratitude to my family for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis.

My sincere thanks to all of you!

Abstract

Neural networks have been shown to accelerate quantum mechanical dynamics simulations without a loss of accuracy. Even though the speedup is substantial, time scales beyond the realm of nanoseconds is still outside the reach of simulations. Recent developments in the acceleration of neural networks might hold the solution to this problem making neural networks predictions even faster, while promising the same accuracy as their non accelerated counter parts. This work focuses on the acceleration of such a neural network called SchNet. The main technique investigated in this work is the so called singular value decomposition, which is a special form of rank reduction. This technique is used to accelerate the matrix vector product, which is the most common operation in a neural network. The work analyzes mainly the error introduced and the speed up acchieved by this technique for different sizes of models and different amounts of reduced ranks. In addition also other techniques in the form of quantization and pruning were investigated following the trend of acceleration methods, which preserve the structure of SchNet.

Zusammenfassung

Neuronal Netze haben in der Vergangenheit gezeigt, dass sie in der Lage sind quantenmechanische Simulationen ohne den Verlust von Genauigkeit zu beschleunigen. Obwohl die Beschleunigung beträchtlich ist, sind Zeitskalen jenseits des Bereichs von Nanosekunden immer noch unerreichbar. Jüngste Entwicklungen in der Beschleunigung von neuronalen Netzen könnten die Lösung für dieses Problem sein, indem sie die Vorhersagen von neuronalen Netzen noch schneller machen, während sie die gleiche Genauigkeit wie das Original-Netzwerk versprechen. Diese Arbeit konzentriert sich auf die Beschleunigung eines solchen neuronalen Netzes namens SchNet. Die Haupttechnik, die in dieser Arbeit untersucht wird, ist die sogenannte Singulärwertzerlegung, die eine spezielle Form der Rangreduktion ist. Diese Technik wird verwendet, um das Matrix-Vektor-Produkt zu beschleunigen, welches die häufigste Operation in einem neuronalen Netz ist. Die Arbeit analysiert hauptsächlich den Fehler und die Beschleunigung, der durch diese Technik für verschiedene Modellgrößen und verschiedene Mengen an reduzierten Rängen eingeführt wird. Darüber hinaus wurden auch andere Techniken in Form von Quantisierung und Pruning untersucht, die dem Trend der Beschleunigungsmethoden folgen und die Struktur von SchNet erhalten.

Contents

1 Introduction

In the last few decades neural networks have become a versatile tool for tackling a plethora of problems such as image recognition [1], translation of languages [2] or protein folding [3]. Quantum chemistry also found use for this novel technique [4–7]. The main advantage from using neural networks arises from the speed with which neural networks are able to produce results with the same or similar accuracy as quantum mechanical methods. The acceleration of such predictions are a major advantage for techniques, which require multiple predictions of the same molecule such as molecular dynamics or Monte Carlo methods [8].
The acceleration a neural network is capable of depends on many factors such as the original method used or the size of the system in question. This work focuses on neural networks used for the acceleration of ab initio excited states molecular dynamics. Previous works have shown that neural networks are able to accelerate such simulations from picosecond up to nanosecond time scales [7].

This work focuses on SchNet [9], a neural network specifically used for the prediction of molecular properties such as potential energies, dipole moments, nonadiabatic couplings and many more. The network can be combined with SHARC (surface hopping including arbitrary couplings) [10], a molecular dynamics program for excited states, which treats the nuclei of atoms classically while usually performing a quantum chemical calculation for the electronic contribution of the potential energy. The combination of these two programs is named SchNarc and it uses the properties predicted by SchNet instead of the usual costly quantum chemical calculations [11]. However, the improvements of the prediction speed still leave simulation times longer than a few nanoseconds beyond our reach. Fortunately there are a few avenues available to make such time scales available. Promising techniques belong the field of rare event sampling such as forward flux sampling. Here, the overall concept being, that trajectories are accepted or rejected based on if they reach a certain area or not. Such methods usually involve a statistical element, which usually makes it necessary to run a large amount of trajectories in order to obtain meaningful results[12]. Therefore, such methods would also benefit from acceleration since even longer time scales would become possible. However the techniques focused on in this work fall under a different category, the acceleration of neural networks. Recent developments in the area of neural networks focus on this topic leading to faster predictions with less computational resources. The main reason for this surge in interest in this field is caused by the need to apply neural networks to smaller systems such as mobile phones or VR-devices. Since these systems have restrictions for computational power, storage space and even electrical power neural networks need be adapted in order to make them more accessible for such devices [13]. Even though this work does not focus on such devices the surge in different acceleration techniques can be used for other fields such as quantum chemistry as well. An additional advantage of accelerated neural networks is the access to faster derived properties such as gradients or Hessians since the underlying networks have less parameters. The reason these gradients are needed are firstly that forces are calculated as the first derivatives of the energy with respect to the atomic position and, secondly, nonadiabatic couplings can be approximated by using the second derivative of the energy with respect to the atomic positions [7].
The way acceleration of a neural network is achieved can vary greatly, ranging from simply deleting values to creating entire new neural networks, which learn to reproduce the same output from a bigger neural network. This work will focus mainly on one technique called rank reduction. The technique chosen for rank reduction is the singular value decomposition, for further details see Section 2.3.1. Previous works show, that rank reduction can lead to a speedup of a factor of 3 when

used for image classification [14].
This work will primarily focus on rank reduction but will also explore pruning and quantization. All these techniques focus on preserving the structure of the network while accelerating the matrix vector multiplication. The acceleration techniques will be tested on SchNet in order analyze the speedup achieved and error created by the method.

2 Theoretical background

2.1 Neural Networks

A detailed discussion of neural networks is beyond the scope of this work. However, a few key aspects are necessary for the understanding of the research conducted in this work.
The basic processing unit within a neural network is a so-called neuron. This neuron collects the output from other neurons, multiplied by a factor called weight. Also an external value, which is not dependent on previous neurons, can be added called bias. After collecting all the these values, a function is applied to the sum called activation function. The output of this activation function is then passed on to other neurons. We can write this value as:

$$y = f(b + \sum_{i=0} x_i \cdot w_i) \tag{1}$$

with x_i being the input from the i-th neuron and w_i being the corresponding weight. $f(x)$ is the activation function and b the bias. Finally y is the value, which the neuron passes to other neurons. The way these neurons are usually arranged is in so-called layers. Neurons within a layer share the same input and output neurons. Since we have multiple neurons within a layer we can describe them as a vector and the weights, which are needed in order to go to the next layer as a matrix. Therefore, we can describe the propagation from one layer to the next as a simple matrix vector multiplication. Since the bias is also added to every neuron, it can be added in form of a vector after the multiplication. The resulting formula for an entire layer can therefore be written as:

$$y = f(\boldsymbol{x}\boldsymbol{W} + \boldsymbol{b}) \tag{2}$$

with $\boldsymbol{W}$ being the weight matrix and $\boldsymbol{y}, \boldsymbol{x}$ and $\boldsymbol{b}$ being the output, input and bias vector, respectively. Training a neural network is the process of adjusting the values within the weight matrices and the bias vectors. The training requires inputs together with already known outputs. Algorithms such as gradient descent are used as a way to find the optimal values for weights and biases in order to predict the desired properties from the given inputs [15].

2.2 SchNet

SchNet[9] is a neural network used for the prediction of molecular properties such as potential energies, spin orbit couplings or dipole moments. It is also able to compute first and second derivatives with respect to the atomic coordinates, which make it possible to calculate forces or approximate nonadiabatic couplings [7]. It is programmed in python, built upon Pytorch[1] a python library used for the construction of neural networks. The program is available open source on

[1]https://Pytorch.org/[last access: 27.11.2020]

Github[2]. A schematic overview of how SchNet is structured can be seen in Figure 1. The several subunits will be explained in the following subsections. The overall concept is to use only the atomic numbers of each atom and its coordinates as input for the neural network usually referred to as descriptors. Using the descriptors, the neural network finds its own set of descriptors called features. These features are then used for the prediction of properties such as potential energies, dipole moments, nonadiabatic couplings or spin orbit couplings. Throughout this chapter the inner workings of SchNet will be illustrated with the CO_2 molecule as an example.

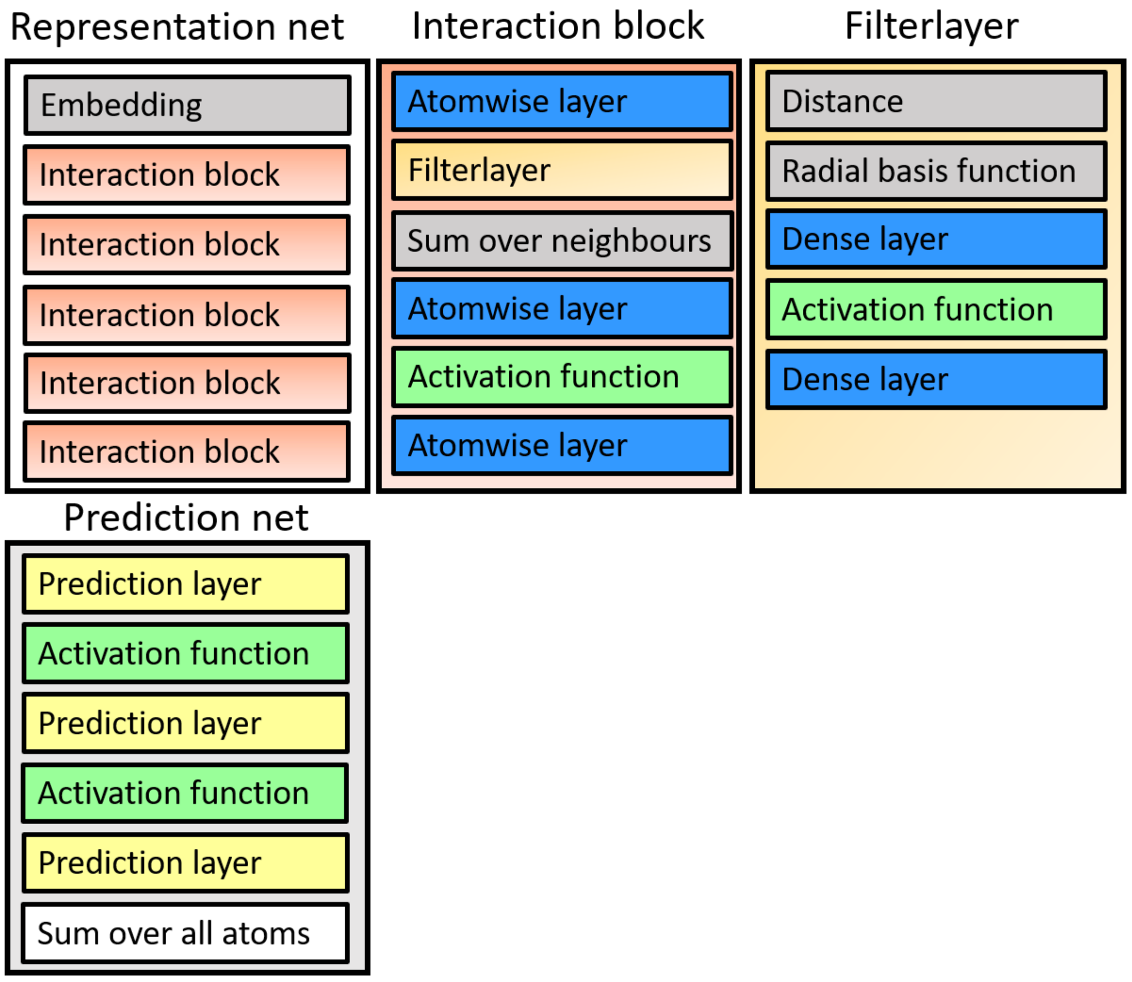

Figure 1: SchNet can be divided into two subnetworks, the representation and the prediction network. The representation network consist of the embedding and multiple interaction blocks. The interaction blocks themselves consist of multiple atomwise layers and a filter layer.

[2]https://github.com/atomistic-machine-learning/SchNetpack[last access: 19.03.2021]

2.2.1 Representation network

The representation network is used as a tool to find a set of features, which are used as inputs for a second neural network. This second neural network then predicts the properties of the molecule we are interested in. The features are produced by adding multiple layers of interaction blocks (see Section 2.2.4) to an initial set of features. These features are a vector of a certain length for each individual atom. A more detailed description will follow in section 2.2.2. Written as an equation, we can formulate the representation network as:

$$\boldsymbol{x_N} = \boldsymbol{x_0} + \sum_{l=0}^{\mathrm{N_{interactions}}} \boldsymbol{v_l(x_l)} \tag{3}$$

With $\boldsymbol{x_l} = \boldsymbol{x_{l-1}} + \boldsymbol{v_{l-1}(x_{l-1})}$ and $\boldsymbol{x_0}$ being the starting features. The dimension of these feature vectors is a parameter, which can be changed by the user and has a default value of 256. The dimension of the features is the largest factor when it comes to computational costs since it alone determines the size of almost every matrix within the network. $\boldsymbol{v_l(x_l)}$ is the contribution from the l-th interaction block, which is dependent on the feature set generated by the former interaction block. Finally, x_N is the output of the representation network. The number of interaction layers $\mathrm{N_{interactions}}$ can be userdefined and by default, six interaction blocks are used, which is also the given as input to SchNet. Since the sum starts to count at zero in the formular given above the mathematically correct number for $\mathrm{N_{interactions}}$ would be five, since the sum starts to count at zero, but six needs to be entered as the parameter for the code.

2.2.2 Embedding

The embedding is the very first section of SchNet. It is used to assign the feature vectors x_0 to each atom, which are the starting features mentioned in Section 2.2.1. For each atom passed through this section a set of features is generated. Each feature vector starts out as a set of random numbers for every atom type at the beginning of the training. With every training cycle SchNet will adapt these vectors as it sees fit in order to generate an optimized representation of the compounds contained in the training set. These features are then stored in a matrix. Accessing the features for a certain atom can be done by simply taking the row vector corresponding to the atomic number. Since the first row in the matrix corresponds to the atom with the atomic number of zero this line will always be a row of zeros. Therefore for the given example the 7th row would be taken once for the carbon atom and the 9th row twice for the two oxygens generating a 3×256 matrix if the length of the feature vectors is 256. Each of these vectors will be treated separately by the neural network.

2.2.3 Atomwise layer

The atomwise layer is the next and most common layer of SchNet. It is applied to every atom of the molecule always using the same weights and biases. The layer can be seen as a normal fully connected layer and can therefore be written as :

$$\boldsymbol{x^i_{k+1}} = \mathrm{f_{activation}}(\boldsymbol{W_k x^i_k} + \boldsymbol{b_k}) \tag{4}$$

With $\boldsymbol{x^i_k}$ being the feature of the atom i at the layer k with the corresponding weights $\boldsymbol{W_k}$ and bias $\boldsymbol{b_k}$. $\boldsymbol{x^i_{k+1}}$ is therefore the feature, which will be fed into the next layer. Please note, that each atomwise layer is within an interaction block, which would make it necessary to add the subscript

l. For sake of readability this subscript was neglected. The activation function is in our case the so-called softplus function shifted by a constant factor of $ln(2)(\mathrm{f}_{\mathrm{activation}}(\mathrm{x}) = ln(1+e^x) - ln(2))$. This function is applied only after certain layers, which will be indicated by $\mathrm{f}_{\mathrm{activation}}$ [16]. Continuing the example, the calculation performed by an atomwise layer is:

$$\boldsymbol{x^i_{k+1}}(256) = \mathrm{f}_{\mathrm{activation}}(\boldsymbol{W_k}(256 \times 256)\boldsymbol{x^i_k}(256) + \boldsymbol{b_k}(256)) \tag{5}$$

with the dimensions of the individual variables written in brackets. It is important to note that an activation function acts on each element in the feature vector individually, which means the activation function is applied 256 times per atom. Since CO_2 has 3 atoms the layer needs to be applied three times in order to go to the next layer.

2.2.4 Interaction block

The interaction blocks can be seen as a way to describe how a certain atom interacts with their surrounding neighbours. We define neighbours as atoms, which are within a certain radius called the cut-off radius. However, the systems we are investigating in this work are rather small, which is why this cut-off is usually chosen in a way, that all atoms of the molecule are within this radius. The interaction blocks take the features from the last layer x_l as an input and add their output values v_l onto them. Interaction blocks consist of three atom-wise layers with a so-called generated filter layer between the first and the second atom-wise layer. The second layer also has an activation function applied to its output. In the following formula the dimensions of the corresponding weight matrices are given in brackets. The atom wise layer marked with a star is the only layer within the network, which does not have a bias, because it is mainly used for shuffling the features before applying the filter layer.

$$\begin{aligned} v_l(\mathrm{n}_{\mathrm{features}}) =& \mathrm{L}_{\mathrm{atom\_wise}}(n_{\mathrm{features}} \times n_{\mathrm{features}}) \times \\ & \mathrm{f}_{\mathrm{activation}}(\mathrm{L}_{\mathrm{atom\_wise}}(n_{\mathrm{features}} \times n_{\mathrm{features}})) \times \mathrm{L}_{\mathrm{filter}} \times \\ & \mathrm{L}_{\mathrm{atom\_wise}}(n_{\mathrm{features}} \times n_{\mathrm{features}})^* \times x^l(n_{\mathrm{features}}) \end{aligned} \tag{6}$$

L represents the different layers with their name being indicated as a subscript.

With taking the standard value for $\mathrm{N}_{\mathrm{interactions}}$ this block would be executed 6 times generating the vector $v^l(256)$ for a network with 256 features.

2.2.5 Filter layer

The filterlayer is the heart of SchNet. A filter is a tool usually used for discrete signals such as pixels or digital audio data, which are placed along a grid in convolutional networks [17]. However using this kind of filter on atomic positions would make it necessary to impose such a grid onto them. While there are techniques to accomplish such a goal, SchNet takes a different approach. The solution is to use a small, specially designed neural network to create a continuous filter on its own. The input the filter layer requires is the distance between the atom it currently investigates and each atom within a certain radius also called cutoff radius. Atoms outside this radius will not be take into consideration, which ensures the method can scale better with system size. The filter layer will analyze each atom eventually calculating the distance between each atom discarding the ones with a distance higher than the cutoff radius and applying the process described below for the rest of them. The filter layer also requires the features from the last interaction block. The obtained distances are first expanded in a basis of Gaussians. These Gaussians take the shape of:

$$g(d_{ij}) = e^{-\gamma||\mu_h - d_{ij}||_2^2} \tag{7}$$

with μ_h being the different centers of the Gaussians and d_{ij} being the distance between the atoms i and j. γ is calculated in the following way: $\gamma = -0.5/(||\mu_0 - \mu_1||_2^2)$. The centers of these Gaussians are equidistantly placed between 0 and the cut-off radius.

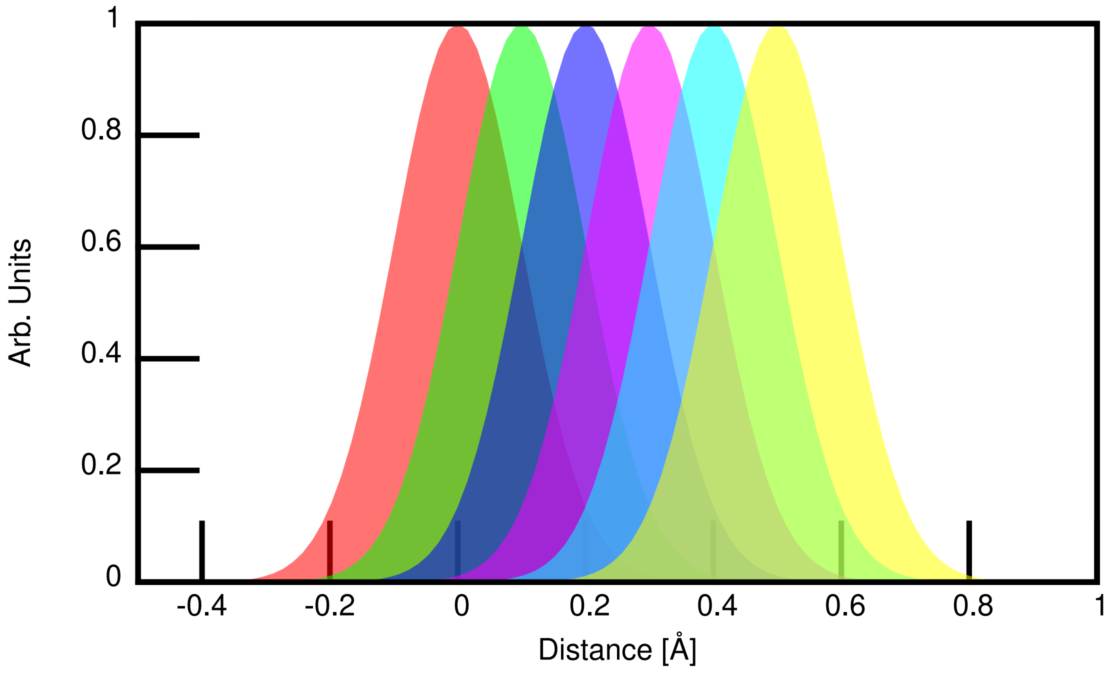

Figure 2: Examplary Gaussian functions, which are used in the expansion of the distances.

Figure 2 shows a few of those functions and how they overlap with each other. Since the values of the Gaussians stay the same for the entire network for a given molecular geometry, the values are calculated once at the beginning of the pass through the network and then stored and used for every filter layer. The resulting vectors have only a few entries with values close to 1, if the corresponding Gaussians are centered around a given bond distance, while the values grow smaller and eventually reach 0 further away from the bond distance. Consider the example of CO_2 with a bond distance of 1.16 Å between carbon and oxygen atoms with 50 Gaussians placed between 0 and 5 Å : The Gaussians centered at 1.1 and 1.2 Å show values of 0.83 and 0.92, respectively, while the values for the Gaussians centered 0.5 Å further away have already decreased to 10^{-7}. After the expansion of one distance, two fully connected layers are applied to this vector. Since the number of features and Gaussians are rarely the same, the first dense layer is a rectangular matrix that changes the size of the vector to the length of the features. After applying an activation function, the second layer is added as a normal square matrix. In a last step, we multiply the vector generated by the filter layers output elementwise with the features of the atom from the last interaction block. After performing the procedure for every distance for one center atom these vectors are summed up to get one vector. The new vector generated this way can be passed forward to the next layer in the network. The filter layer can be written as[18]:

$$\begin{aligned} x_i^{l+1} = \sum_{j=0}^{\mathrm{N_{neighbours}}} & x_i^l \cdot \mathrm{L_{dense\_layer}}(\mathrm{n_{features}} \times \mathrm{n_{features}}) \times \\ & \mathrm{f_{activation}}(\mathrm{L_{dense\_layer}}(\mathrm{n_{features}} \times \mathrm{n_{Gaussians}})) \\ & \times \mathrm{G_{expansion}}(d_{ij})(\mathrm{n_{Gaussians}}) \end{aligned} \tag{8}$$

With $\mathrm{G_{expansion}}$ being the Gaussian expansion and $\mathrm{L_{dense\_layer}}$ being the dense layers. $\mathrm{n_{Gaussians}}$ corresponds to the number of Gaussians used for the system. The default number is 50.

2.2.6 Prediction network

After applying all aforementioned representation layers, we are left with a set of features that we can use in order to predict the properties we are interested in. The prediction is done by the so-called prediction network. This network is a feed-forward neural network. It consist of rectangular matrices, which reduce the amount of features by a factor of 2 in each step eventually reaching the number of output neurons, which is equal to the number of values the network is supposed to predict. The reduction of neurons through the network can also be changed, if the default architecture is not beneficial for the number of values that should be predicted. Between the layers, except for the last layer, there is an activation function. At the end of the prediction network the desired values can be obtained by summing over the contributions of all atoms. A set of properties can therefore be obtained in the following way.

$$\begin{aligned} Property = \sum_{i=0}^{\mathrm{N_{Atom}}} & \mathrm{prediction\_layer\_4}(\mathrm{N_{properties}} \times 16) \\ & \cdot \mathrm{f_{activation}}(L_{\mathrm{prediction\_layer\_3}}(16 \times 32)) \\ & \cdot \mathrm{f_{activation}}(L_{\mathrm{prediction\_layer\_2}}(32 \times 64)) \\ & \cdot \mathrm{f_{activation}}(L_{\mathrm{prediction\_layer\_1}}(64 \times 128)) \\ & \cdot \mathrm{f_{activation}}(L_{\mathrm{prediction\_layer\_0}}(128 \times 256)) \\ & \cdot (x_0 + \sum_{n=0}^{\mathrm{N_{interactions}}} v_n) \end{aligned} \tag{9}$$

In this example we start with 256 features and continue to reduce the number of neurons in each prediction layer by a factor of 2. The default number of layers is just 3 however we decided for a larger number in order to see the structure of the prediction network more clearly.

2.3 Acceleration of NNs

The acceleration of neural networks has gained a lot of interest in the last few years. The main reason is the need to transfer neural networks to mobile devices such as smart devices, which have to work with limited resources for computational power or energy. Since the size of these networks have increased to several millions sometimes even billions of parameters in past years it is crucial to find ways to reduce their storage and computational costs [13]. While this work does not focus on such applications, for the latter might still be beneficial for the acceleration of SchNet.

2.3.1 Rank reduction

The rank of a matrix can be defined as the maximum numbers of linearly independent rows or columns within a matrix. A matrix is called full rank if it only has linearly independent rows/columns. Note that a rectangular matrix $A \in \mathbb{R}^{n \times m}$ can only have a rank of $min(m, n)$. Matrices with lower ranks can be decomposed into a form, which makes it possible to make the matrix vector multiplication more efficient, than using the original matrix.

The singular value decomposition is the main technique used in this work to accelerate SchNet and also the most basic form of rank reduction. The technique will be explained in order to understand how and under what circumstances an acceleration with this method is achievable.

The theorem of the singular value decomposition (SVD) is as follows[3]: For a given matrix $A \in \mathbb{R}^{n \times m}$ there exist two orthonormal matrices $U \in \mathbb{R}^{m \times m}$ and $V \in \mathbb{R}^{n \times n}$ and a diagonal matrix Σ such that $U \Sigma V^T = A$. Σ stores the so-called singular values from the highest to the lowest along its diagonal. The number of these positive values is equivalent to the rank of the matrix A. If the rank is smaller than the dimension of the diagonal matrix the values, which follow the last singular value, are zero [19]. A different way of writing the SVD is:

$$A = \sum_{k=1}^{rank} \sigma_k u_k \times v_k^T \tag{10}$$

with $\sigma_k$being the k-th singular value and $v_k \times u_k$ being the outer product of the k-th vector in the corresponding matrices V and U. One can now reduce the rank of matrix A by setting singular values to 0. The error introduced this way in the Frobenius norm, which is $||A||_F = \sqrt{\sum_{i=0}^{m} \sum_{j=0}^{n} a_{ij}^2}$ for $A \in \mathbb{R}^{n \times m}$, can be bound by $E = \sqrt{\sum_{k=t+1}^{rank} \sigma_k^2}$ if we want to reduce the rank of A by $rank - t$ with t being a natural number smaller than the rank of the matrix[20]. The operations needed to compute the SVD usually scale with mn^2 [21]. Figure 3 shows how a reduction in rank can be used for the acceleration of a matrix vector product. The left part of the figure shows the normal propagation through the network. In order to propagate the value from one neuron to the next layer one needs to perform one multiplication for every neuron in the next layer as long, as both layers are fully connected with each other. Assuming that the green layer has n neurons and every layer is connected to every neuron in the blue layer, which has m neurons the total amount of operations scales with $O(n \cdot m)$. In comparison when using a reduced rank structure there is a first propagation to a layer with r neurons equivalent to the rank of the matrix. This process takes for one neuron only r calculations resulting in only $m \cdot r$ operations. In a second step the values from the orange layer are propagated to the green layer using $r \cdot n$ operations. Therefore in total the amount of calculations scales with $O((m + n) \cdot r)$. The number of additions performed for both layers scales similarly. For the normal case the operations scale again with $O(n \cdot m)$ since for all n entries in the green layer m additions need to be performed. For the other layer again only $m \cdot r$ additions need to be performed for the first matrix vector multiplication and only $r \cdot n$ additions for the second one ending up with the same scaling $O((m+n) \cdot r)$ as for the multiplications. Comparing the scaling of both layers show that a rank reduced layer can lead to a reduction in operations if

[3]We will only consider the case of a real matrix here since weight matrices for SchNet only have real values.

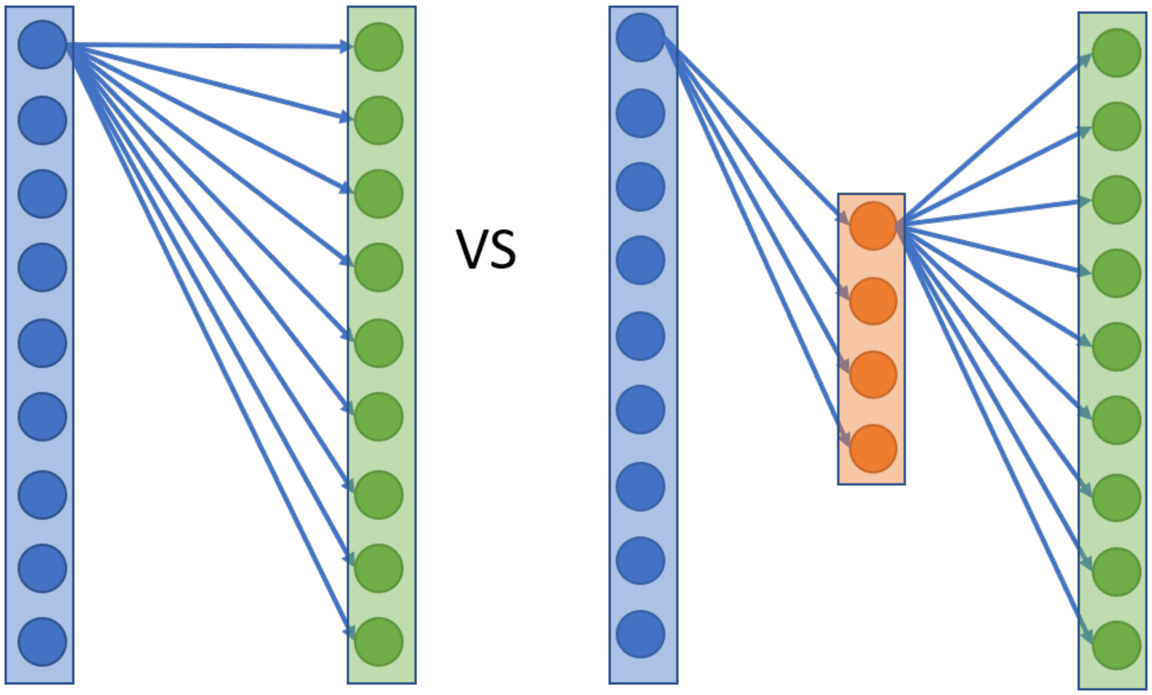

Figure 3: Schematic structure of a normal layer (left) versus a layer that uses a rank reduced structure for a $n \times m$ weight matrix (right).

$r < \frac{n}{2}$. An acceleration can therefore only be accomplished if the rank of the matrix is lower, than half of its full rank.

As stated, SVD is only applicable to matrices. However for other neural networks this method is not sufficient, since their inputs might not be the shape of a vector but a matrix or even higher dimensional constructs. Therefore more complex rank reduction methods such as Tucker or tensor train decomposition are needed for such neural networks, since SVD is not applicable for tensors in general. However SchNet uses only vectors, which means that its weights are in the shape of matrices, which makes it possible to apply the SVD. As mentioned in Section 1 rank reduction can lead to a speedup of a factor of 3 when used for image classification [14].

2.3.2 Pruning

Pruning is a very common approach when it comes to reducing the parameters in a neural network. The concept is to remove weights from the network. Pruning can be performed with a lot of different strategies in mind and even randomly deleting weights is an approach, which has been tried in the past. One idea for this technique is the so-called magnitude-based pruning. This magnitude is a constant value that is used as a threshold for deleting weights with lower absolute values. Another way is to delete a certain percentage of weights with the lowest magnitudes. These tactics can either be used on a global scale or just be applied layer by layer. Other techniques are more focused on the training of the neural network. One can put a penalty on large weights in the loss function and prune the smallest values for a fast and sometimes effective approach to the problem. Another approach is to use first or second order derivatives to determine, if a change for a certain weight shows significant contribution to the error. Values that show a small change can be deleted leaving only weights that contribute to the predictions [22]. Theoretical accelerations by a factor of 2-3 per weight matrices have been shown in the past to be possible [23].

2.3.3 Quantization

Quantization is a very straightforward approach to reducing the size of neural networks. It aims to reduce the number of bits of the weights and biases used in the neural network. Common storage formats (before quantization) are 32-bit, which is the normal floating point format, or 16-bit, which can be either half precision floating point or 16-bit integer format [22]. The main difference between integer and floating point numbers is how they are stored within the computer system. The smallest storage unit in a computer is the so-called bit, which can either be a 1 or a 0. Using multiple of these bits results in 2^p different combinations with p being the number of bits used. When storing an integer every bit but one is used in order to represent a unique number. The last bit is used to store the sign of the number defining if the number is positive or negative. The total amount of numbers, that a certain integer format can therefore store is 2^{p-1}. Floating point numbers take a different approach. They use a certain amount of bits to represent an integer value but use the remaining bits to represent a second integer, which is then used as an exponent with the basis of 2, which is then multiplied with the first integer value. Again a single bit is used for the storage of the sign. Floating point numbers can therefore be written as $m \cdot 2^e$ with m and e being integers represented by a number of bits [24]. The number of bits for each of these variables have been standardized for certain amounts of bits such as 64 or 32 bits [25]. Currently there is no standard for lower floating point formats than half precision or 16 bit floating point formats [26]. Therefore, lower bit formats are integer formats such as eight, four or two bit [22].
The general idea of quantization is to find a range, in which most of the values, which are used for weights and biases or are the outcome of a layer lie in. With this information one can then use a constant factor to multiply the values with after applying the layer to it and transform the integer values back to floating point values. This approach can be applied to the entire network at once, which is then called fixed point since the factor is fixed for the entire system. On the other hand one can use dynamic fixed point to adjust these constant values for example for every weight matrix, bias, results from activation functions and other values from or created by a single layer [26]. It is also possible to fuse certain layers with their activation function together needing less scaling factors [27]. Accelerations in this field are dependant on type of storage chosen. However since this work will focus on an 8-bit quantization it has been shown, that networks can double their prediction speed [28].

3 Development of a SVD layer

This section will focus on the implementation of a so-called SVD layer, which is the tool used in for the reduction of operations in SchNet. The layer was built with the basic layer provided by SchNet. This layer is called the Dense layer. It acts as a fully connected layer and is used usually in the network for the atomwise layers and also in the filter layer. The Dense layers always have the parameters n_in and n_out. These parameters correspond to input and output neurons of the layer. Additionally a parameter called activation exists, which corresponds to the activation function used

after the layer is applied. The default parameter for activation is None, which corresponds to not having an activation function. If an activation function is required, it needs to be imported in order to be available. Another parameter is used for a possible bias for the Dense layer. Again the default parameter is None, which would correspond to the layer not having a bias.
The SVD layer developed in this work is created in a way to emulate the same behaviour as a Dense layer, while reducing the amount of operations. The approach chosen for this task was to substitute the Dense layer by two smaller ones. Since the layer still needs to propagate the same information from one layer to the next the parameters n_in and n_out need to also exist for the SVD layer and need to stay consistent with the Dense layer it aims to substitute. Additionally a parameter n_rank is used in order to determine the rank of the weight matrix. The parameters activation and bias also need to be used by the SVD layer. The layer is constructed by using the two Dense layers to represent the matrices U and V mentioned in Section 2.3.1. Since the propagation from one layer to the next can be seen as a matrix vector multiplication it is possible to do such a substitution. Therefore the first layer corresponds to the matrix V. This Dense layer has the same n_in as the original Dense layer but its n_out is equal to the parameter n_rank. It also does not have a bias or an activation function since it is simply used as a tool for the first matrix vector multiplication. The second Dense layer corresponds to the matrix U. This Dense layer has the allowed rank as n_in and the n_out is the same as for the original Dense layer. This layer also has an activation function and/or a bias based on the original Dense layer.
The SVD layer has two different tasks to fulfill. On the one hand, it should be able to decompose already existing layers and transferring them into their rank reduced form. On the other hand it should also be possible to create SVD layers without any previous weights. Since for the first case a weight matrix is required it can simply be checked, if a matrix is given to the SVD layer. After performing the singular value decomposition for the weight matrix the given amount of singular values are deleted from the matrix Σ together with the corresponding vectors in V and U. In order to include the singular values, the matrix Σ is multiplied with U. Making this multiplication includes $Sigma$ in the layer without contributing to the number of operations needed while using the network for predictions.
The SVD layers were directly inserted into SchNet, substituting the Dense layers. The rank of the weight matrix was set to a percentage of the full rank in order to make trainings with different amounts of features possible, while maintaining the relative rank.

4 Results and Discussion

In the following section, the different acceleration techniques will be analyzed when applied on SchNet. There are two important quantities, which will be focused on in order to evaluate the success of the methods.
The first one is the speedup. It is usually measured as a factor since the actual prediction speed is highly dependent on the system the calculation is performed on. However, the data that is shown in the following sections will be measured in seconds since they are all calculated on the same machine (CPU: 2x Intel Xeon E5-2650 v3, 20 cores).
The second property is the error introduced by the method at hand. Generally speaking, the ideal method would not introduce any kind of additional error. However, since an accelerated network has less information than a normal one, it might not be realistic to search for such a method. The error will be shown in kcal/mol since 1 kcal/mol is the maximum error of the system we want to achieve.

4.1 Singular value decomposition

The following section will focus on the results obtained by the singular values decomposition. The first section will focus on the analysis of already trained networks and their behaviour. A second section will investigate how trainings that only allow a certain rank affect the accuracy of SchNet.

4.1.1 Rank reduction on trained networks

In order to evaluate the error, which is introduced by the SVD, first the decomposition was performed on every weight matrix within SchNet. The implementation from the numpy[4] package was chosen for performing the SVD. Note that the scipy[5] package could also be employed for this task, where the underlying routines are the same.

A first training set used, consists of 6585 data points of diiodomethane, which were generated by CASPT2/ano-rcc-vdzp calculations with an activate space of (12,8). The properties this network is trained on are energies, forces and spin orbit couplings. These quantities stem from five singlet and four triplet states. The sums of errors for each property are weighted equally throughout the training process. This data set was split in 6200 data points for the training set and 300 for the validation set leaving 285 points left for the test set. Additionally a cutoff of 6 Åwas chosen. For the prediction network, five layers were chosen. The trainings were performed on GPUs for faster training times. The loss function was specialised on training with phase dependant properties such as spin-orbit couplings. The batch size for the training was 20.

Additionally the analysis was performed for a second test system, $CH_2NH_2^+$. The quantum chemical data was obtained by SA-CASSCF calculations with an active space of (6,4) and MRCI-SD both with a basis set of aug-cc-pVDZ. The neural network parameters for this system were the same with the exception of the training set, which consisted out of 3800 data points, 200 data points for the validation set and 200 as a test set. Additionally this system was only trained on energies and forces. The prediction times were done with all the data points combined. The results can be seen in figure 8. The trends, which were observed for the diiodomethane molecule, hold true for this test system as well.

Figure 4 shows the singular values of every weight matrix within the network and how they increase. As can be seen they share very similar values for the different types of weight matrices. The ones with the highest possible rank shown in black correspond to the atomwise/dense layers within the representation network. They start with the lowest values in the range of 10^{-2} to 10^{-3}depending on the size of the network. The overlap of the different singular values show, how similar the different weight matrices behave with respect to the rank. All of these values have a linear increase until approximately the highest five percent of the singular values, where they increase very fast. The lines starting from 50%,25%,12.5% of the rank of the atomwise layers correspond to the layers in the prediction network shown in orange. The increase for these values is steeper than for the square matrices. Since two different prediction networks are used for the prediction of energies and spin orbit couplings two matrices with the same exist in the network an can therefore be seen in the figure. A last group of lines shown in purple represent matrices with a

[4]https://numpy.org/doc/stable/reference/generated/numpy.linalg.svd.html [last accessed: 24.03.2021]
[5]https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.svd.html [last accessed: 24.03.2021]

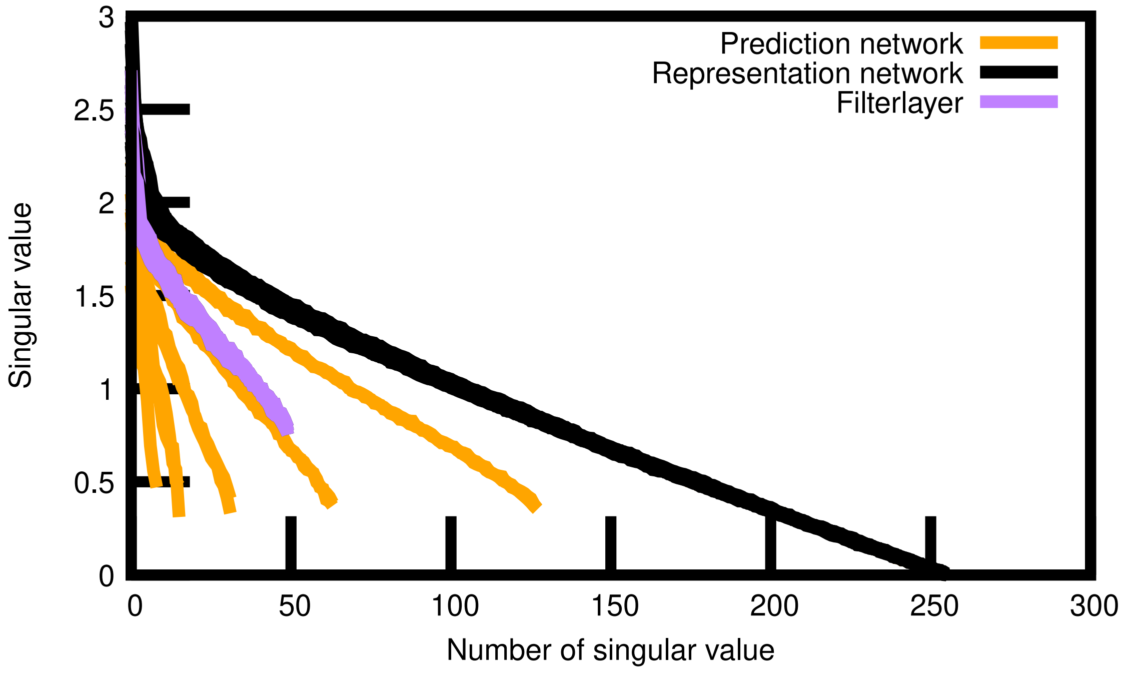

Figure 4: Singular values of each weight matrix within a 256 feature neural network trained with the diiodomethane data. Each line represents the singular values of one matrix and how they increase. The x value corresponds to the place within the diagonal of the Matrix Σ. The y value represents the numerical value for the singular value. The homogeneous behaviour for most of the matrices lets them mostly overlap. Since the rank of all weight matrices is not the same different lines start at different numbers.

rank of 50. These matrices are the first dense layer from the filter network. Since one dimension for these matrices is restricted by the number of Gaussians for every network the maximum rank for these types is the same. The increase for these matrices is even steeper.

Figure 5 shows the analysis of the singular values for a network trained on the methylenimmonium cation data. The behaviour of the individual matrices is very similar to the ones of the diiodomethane network shown in figure Figure 4. Since this network was only trained for energies only a single set of matrices exist in the prediction network .
Generally speaking it can be said the more singular values with low values exist the easier it will be to achieve a speedup with a small error. If a matrix would show a rank lower than its full rank the singular values would be zero. Since non of the matrices show such values it can be concluded, that all weight matrices are full rank. It can therefore be concluded, that an acceleration with no increase in error, without any modifications after the rank reduction, is impossible. However a speedup could still be possible with an acceptable error. Since the square matrices show the slowest increase in singular values in a first analysis, only these matrices were decomposed and approximated. Since only matrices within the representation network are square they were chosen

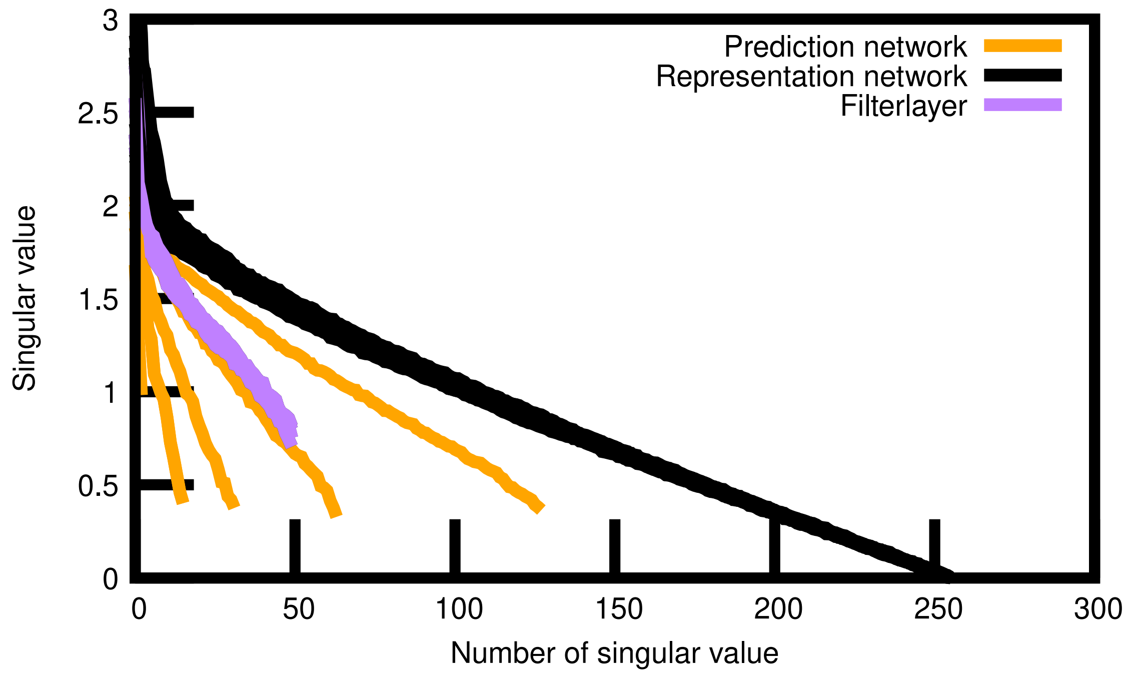

Figure 5: Singular values of each weight matrix within a 256 feature neural network trained with the methylenimmonium cation data. Each line represents the singular values of one matrix and how they increase. The x value corresponds to the place within the diagonal of the Matrix Σ. The y value represents the numerical value for the singular value. The homogeneous behaviour for most of the matrices lets them mostly overlap. Since the rank of all weight matrices is not the same different lines start at different numbers.

for a first investigation. By deleting singular values as described in chapter 2.3.1 and reconstructing the original matrices with a lower rank the effect the of the SVD can be seen.

Figure 6 shows the error in the prediction of energies plotted against the number of deleted singular values for networks trained with the data from diiodomethane. In total four different networks were trained with 128, 256, 512 and 1024 features. In order to make the errors across multiple network sizes comparable, the number of singular values were adjusted to generate curves of the same length. In other words, for each eight singular values deleted in the 1024 feature network, four were deleted in the 512 feature network, two in the 256 and only a single one in the 128 feature network. The error shown in this plot only considers the change compared to the full rank network and not to the reference values, which the network should predict. However, since the error from the network itself is under 1 kcal/mol the overall behaviour especially in the area with less than 50% of the full rank. As can be seen in this plot the error increases exponentially. Therefore the minimum error, we would introduce into our predictions in order to obtain a theoretical speed up would be about 5 kcal/mol, which is five times the error the neural network as a whole should be able to achieve.

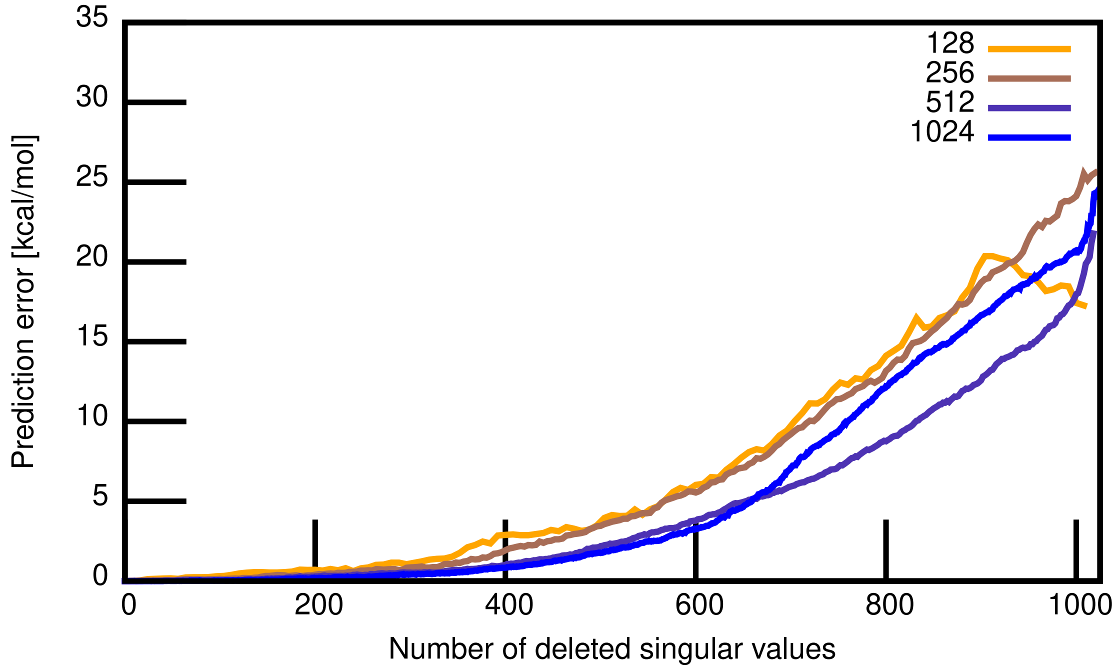

Figure 6: Increasing error for the energy against the amount of deleted singular values for the networks trained with the data from diiodomethane. In order to make the curves comparable for one singular value in the 128 feature network two, four or eight singular values were deleted in the 256, 512 and 1024 feature network respectively.

4.1.2 Rank restricted training

In order to give SchNet the possibility to react to the reduction of the rank, the SVD layer was implemented. The concept of this layer was to substitute the original atomwise layer with two dense layers see Section 3. The shape of these layers can be seen as the matrices V and U from Section 2.3.1 with Σ being multiplied with either of them. Using such SVD layers, trainings were performed with these layers substituting the normally used atomwise layers. In order to observe how the technique reacts to increasing network sizes.

Figure 7 shows a bar chart for the prediction times using this altered version of SchNet. It shows the prediction time for the unaltered version of SchNet in purple and the rank reduced version in orange. For the predictions the training set of diiodomethane was used with its 6585 data points as a data set to be predicted by the trained network. The rank of the rank restricted network was restricted to 33 % of the full rank of the weight matrices. The trainings were performed for a 256, 512 and 1024 feature network. It can be seen, that smaller networks show a small decrease in prediction time. However, this speedup gets smaller for the 512 feature network and the 1024 network shows a slight increase in calculation time.

Table 1 shows the errors and the time the predictions need. Two additional times are listed. The first one is the so-called user time, which is the time a user would need to wait in order to get the results. The second time included the so-called CPU time, which is the time required by the

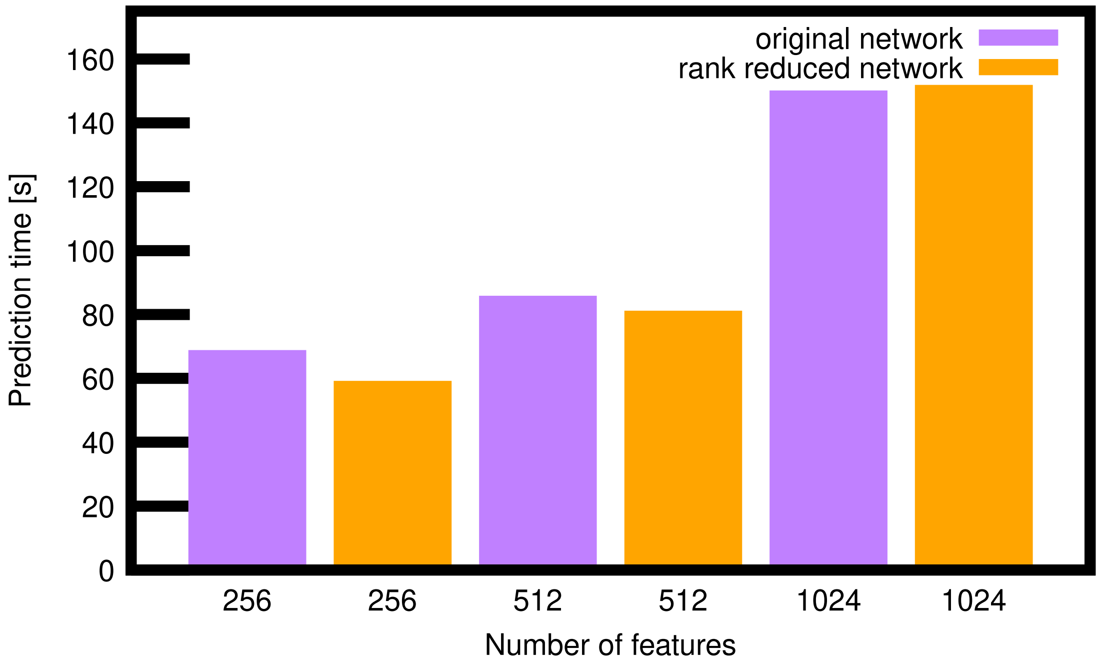

Figure 7: This bar plot shows the prediction time trained with the diiodomethane dataset of SchNet with the feature sizes given below them. The purple bars represent the original network, while the orange bars show the prediction times from the rank reduced network.

CPU to calculate the results. Since the predictions were done on a multi-core machine, the CPU time is much larger than the user time. Generally both of these times are important to look at since the CPU time gives insight on the time used for calculations and how much is needed for writing files or loading data.

It can actually be seen, that the rank reduced network used up more CPU time, than the normal version of SchNet for the 256 and 1024 feature network. The increase however is rather small for both cases with about 5 % for the 256 feature network and even less for the 1024 feature network. Also the difference for the 512 feature network is below 5%. When comparing the error between a normal network and the one with the substituted layers the differences show not a clear picture. While the error does increase for the smaller networks the error for the 1024 feature network shows no significant change. This behaviour might not be intuitive at first but a possible explanation to could be that the optimisation algorithm has a much harder time finding minima since there are much more variables to consider. Since the error is higher than for the 256 feature network there is no incentive to use a bigger network especially since the computational cost increases significantly. Therefore further investigations were performed for the 256 feature network only since the increase in size did not lower the error.

Figure 8 shows the prediction times for the methylenimmonium cation dataset. The rank for the rank restricted networks has been limited to 20 %. As can be seen the prediction times over all are much smaller. There are two reasons for the decrease. The first being, that the dataset consists

feature size	real time [s]	CPU time [s]	error [kcal/mol]
1024	149.909	2486.479	0.83325725
1024 SVD	151.644	2501.534	0.82798625
512	85.605	1201.656	0.794917
512 SVD	80.913	1102.897	1.0237035
256	68.587	628.206	0.593
256 SVD	58.95	660.754	0.7852535

Table 1: The calculation times and the errors of in kcal/mol of diiodomethane networks with different system sizes. The SVD represents the networks with only a third of the full rank. Real time refers to the time a user would wait for the predictions while CPU time represents accumulated core time from the CPU.

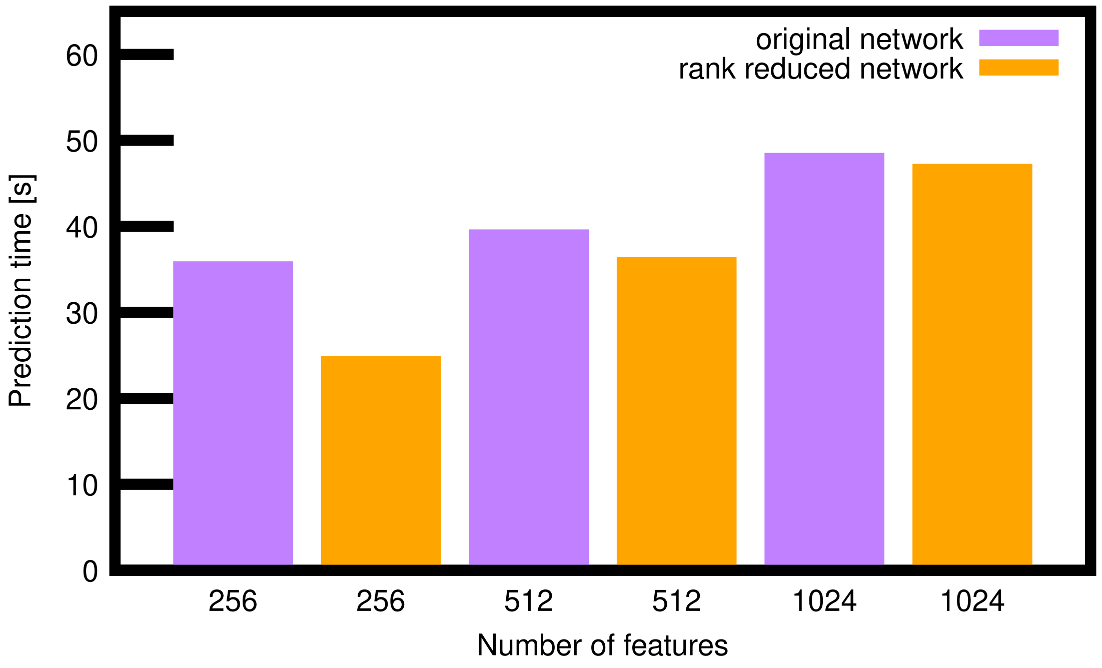

Figure 8: Prediction time for networks trained with the methylenimmonium cation dataset of SchNet with the feature sizes given below them. The purple bars represent the original network, while the orange bars show the prediction times from the rank reduced network.

of more than 2000 datapoints less and the second one is, that it was only trained for energies and not also for spin orbit couplings. The acceleration gained by SVD layers in this network is much higher than for the diiodomethane networks. A reason is probably that the percentage of rank reduced networks is much higher since there are less layers in the prediction network, because only one property is predicted instead of two. As mentioned before, the rank reduction is more limiting, than the previous one. However we can observe the same trend as before with larger networks showing smaller accelerations than smaller ones.

feature size	real time [s]	CPU time [s]	error [kcal/mol]
1024 SVD	47.18	636.218	0,45725925
1024	48.456	752.397	0,595623
512 SVD	36.314	345.562	0,3349595
512	39.557	378.922	0,434104
256 SVD	24.843	221.931	0,5563415
256	35.849	219.993	0,319774

Table 2: The calculation times and errors in kcal/mol of methylenimmonium cation networks with different system sizes. The SVD represents the networks with only a fifth of the full rank. Real time refers to the time a user would wait for the predictions while CPU time represents accumulated core time from the CPU.

Table 2 shows in addition to the wall prediction time also the CPU time and the error given in kcal/mol. It can be seen, that the CPU times are getting smaller by a significant amount for the larger networks, while reaching similar values for the smallest one. It can therefore be concluded that the reduced rank does reduce the calculation time. However, other operations seem to take more time for larger networks making the acceleration of the matrix vector multiplication less impactful. When comparing the error between the original and the rank reduced networks we can actually see that the error decreases for the larger networks, while it increases for the smallest one. Comparing the two different datasets and their respective networks it can be concluded that acceleration is possible. However, the speed up was negligible for rank reductions, which should have theoretically been able to accelerate the network. Since the 256 feature network showed an error below 1 kcal/mol the investigations focused on this feature size.

In order to increase the speedup further, not only the square matrices but also the matrices in the prediction network were included in the rank reduction. With this change every weight matrix within SchNet will be rank reduced with the exception of the first dense layer within the filter layer. This layer has is responsible for the transformation of the vector gained from the Gaussians expansion into the shape of the features used in the network. This process might be more sensitive to changes than the rest of the network. Additionally, that since the layer has the shape of 256×50 it can only have a rank of 50. A reduction to under 25 would be necessary in order to achieve a speedup. The potential of a speedup is therefore very limited and the rank reduction would most likely introduce a high error. Given these reasons, this layer was left untouched for the rest of the investigation. In a next step, multiple different rank restrictions were tried in order to evaluate how the error behaves.

Figure 9 shows the error plotted against the percentage of rank allowed for a weight matrix. At first an increase for the error can be seen for up to over 1 kcal/mol at 20%. However this error decreases again for the following 2 rank restrictions before finally increasing up to more than 2 kcal/mol at 10 % of the rank. The last increase is most likely caused by the network not having enough parameters to produce accurate results. Lower ranks were not investigated since the error would only increase even further. The local increase at 20 % however is a strange phenomena and harder to explain. A first explanation could be, that the weight initialisation had a bad seed, which resulted in a local minimum, that the optimization algorithm was not able to escape from. However

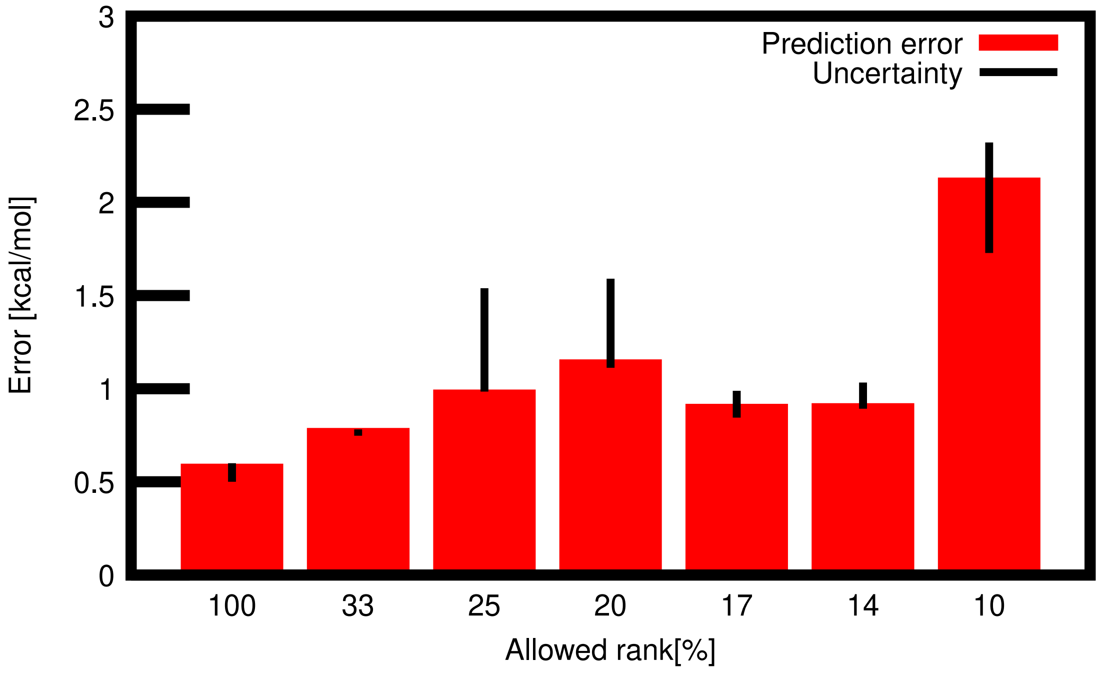

Figure 9: Prediction errors of networks with different amount of maximum rank given in percentages of the full rank. The different data points were plotted additionally in order to make then more visible. The uncertainty shows the fluctuation of the error within the last 1000 training steps.

also 25 % showed a larger error than both 17 and 14%. Last but not least it needs to be stated, that while some of these errors are smaller, none of these errors are close to the error of the original network. Therefore, the accuracy will always decrease when the prediction speed increases. While performing the trainings it was observed that networks with a rank restriction of lower than 20 % took a significantly longer time to converge and the networks with lower ranks of 25 and 20 % had trouble converging. The black bar shows the area in which the error fluctuated in for approximately the last 1000 training steps before the training was stopped. It can be seen, that this uncertainty is the largest for networks with 25, 20 and 10 % rank restriction. The large uncertainty for 10% can be explained by the network not being able to describe the chemical system accurately anymore.
Figure 10 shows the prediction times with the same rank restrictions as for the error plot. Predictions up until this point have been performed on the aforementioned CPU but the data for it was stored on a centralised file system (from now on referred to as global file system). In order to evaluate the contribution of the data transfer to the calculation time the data was transferred to the local storage of the machine and the results can also be seen in the figure. Comparing the two timings on a first look the local storage does decrease the prediction times. However, already the second data point at 33 % of the full rank shows a different behaviour for the two file systems. While the time decreases for the global file system the prediction time for the local file system increases by 10 seconds. For the local case the time decreases steadily up until to 17 %. After that the time stagnates at 42 seconds. For the global case the behaviour is more erratic. While the

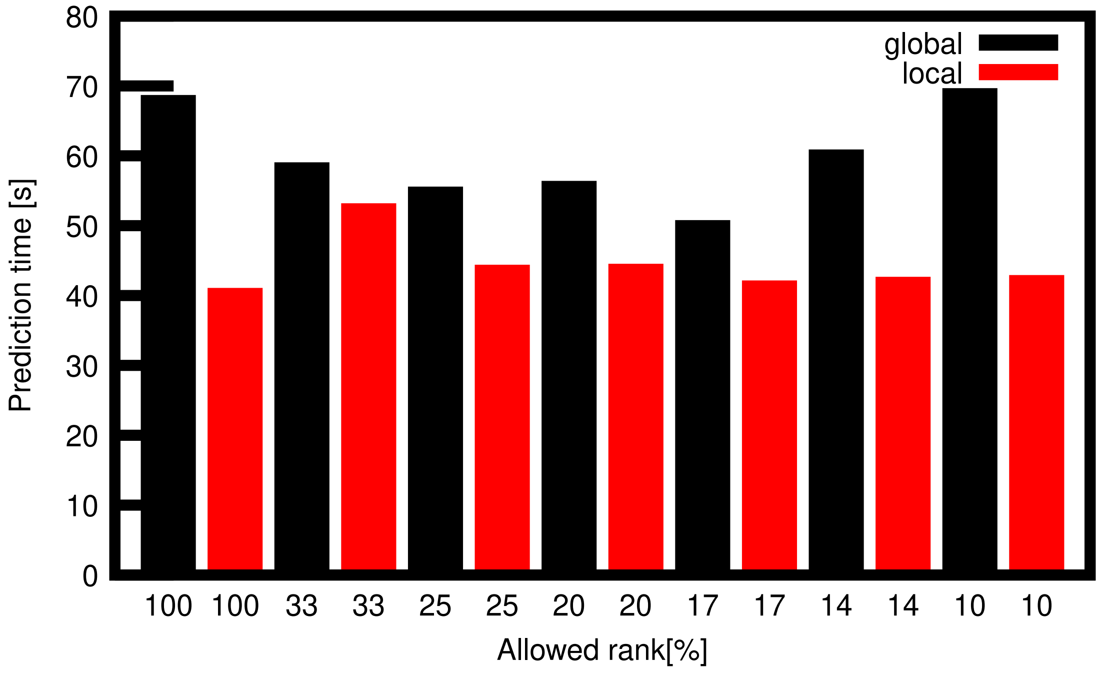

Figure 10: Prediction time of networks with different amount of maximum rank given in percentages of the full rank. The black bars labeled as global correspond to the prediction time, when data was stored on a centralised file system. The red bars show the prediction time if the same data for the same networks is stored locally instead.

decrease is the same as for the local case, a sharp increase can be seen for the 2 lowest percentages.

The initial increase for the local file system might be caused by almost doubling of the matrix vector multiplications caused by the substitution of normal layers by the SVD layers. As described in Section 2.3.1, the number of operations decreases. However the initialisation of a matrix vector multiplication also costs time. It is possible that the time needed for the initialisation for two instead of one matrix vector multiplication outweighs the benefits of the reduced floating point operations. For lower rank percentages, this disadvantage seems to get outweighed by the lower operations eventually leading to a speedup. The stagnation of the acceleration can also be explained by this bottleneck, since at a certain number of floating point operations the calculations are done as soon as the data is ready to be calculated. Instead the main factor is the loading of data and the initialisation of the calculations. The global prediction times have an additional factor, which contributes to the computational speed, which is the speed at which data can be accessed. Since a rank reduced matrix has less parameters than a normal one the continuous decrease in prediction time could be attributed to the smaller amount of data having to be transferred. This problem does also exist in the local case however the time needed is much smaller and might simply play a minor role.

Table 3 and 4 shows again not only the real time but also the CPU times. The behaviour of the CPU time supports the hypothesis of the initialisation being a time relevant step. It is important

rank size [%]	real time [s]	CPU time [s]	error [kcal/mol]
full	68.587	628.206	0.593
33*	58.95	660.754	0.7852535
25	55.471	682.301	0.99088525
20	56.293	623.413	1.1534705
17	50.672	616.746	0.9142675
14	60.802	603.957	0.9182835
10	69.564	595.72	2.12904475

Table 3: The calculation times are shown here together with the error in kcal/mol. The rank size is given in percentage. Real time refers to the time a user would wait for the predictions while CPU time represents accumulated core time from the CPU. The 33 % network is the same as in Table 1, which means is does not have a rank reduction in the prediction net.

rank size [%]	real time [s]	CPU time [s]	error [kcal/mol]
full	40.977	554.977	0.593
33*	53.09	613.403	0.7852535
25	44.293	576.6	0.99088525
20	44.443	570.25	1.1534705
17	42.053	539.685	0.9142675
14	42.572	540.836	0.9182835
10	42.812	543.614	2.12904475

Table 4: The calculation times are shown here together with the error in kcal/mol. The calculation times are shown in this table for the calculations with data stored on the local hard drive. The rank size is given in percentage. Real time refers to the time a user would wait for the predictions while CPU time represents accumulated core time from the CPU. The 33 % network is the same as in Table 1, which means is does not have a rank reduction in the prediction net.

to note, that the 33 % network given in the tables does not have reduced matrices in the prediction network, which could be the reason why it has a slightly lower CPU time, than the next lower network, which does have rank reduced matrices. After these two data points a steady decrease can be seen with the same stagnation as for the real time in the case of the local file system. The global file system Here a very small decrease can still be observed even for the lowest rank networks. quantity predictions for 256 features networks with different amount of ranks allowed.

The size of the matrices used in SchNet are from a computational stand point rather small. The small size is of course a big advantage when it comes to computational costs. However the small size also means, that initialisation steps for matrix vector multiplications are relatively seen rather expensive and might lead to an actual increase in calculation time as can be seen in figure 10. It is also important to note, that while under certain circumstances accelerations are possible these networks do show an increase in error.

4.2 Other acceleration techniques

4.2.1 Pruning applied to SchNet

Pruning was performed with the tools provided by Pytorch. The package provides different methods for pruning networks. They can be divided into structured and unstructured methods. Structured methods focus on imposing or finding a structure of important weights. Using these structures, matrix vector multiplications can be manipulated taking only the non-zero elements into account [22]. This work focuses on the unstructured methods, since no artificial structure is imposed onto SchNet. The search for such structures would have lead to further investigations, which are beyond the scope of this work. The two Pytorch routines of which to choose from are either "random_unstructured" or "L1_unstructured". The first one chooses weights randomly and deletes them, while the second one focuses on the smallest error in the L_1 norm, which means it simply deletes the values closest to zero. For both of these functions, a parameter can be chosen, which either deletes a certain amount of weights or a certain percentage of them.

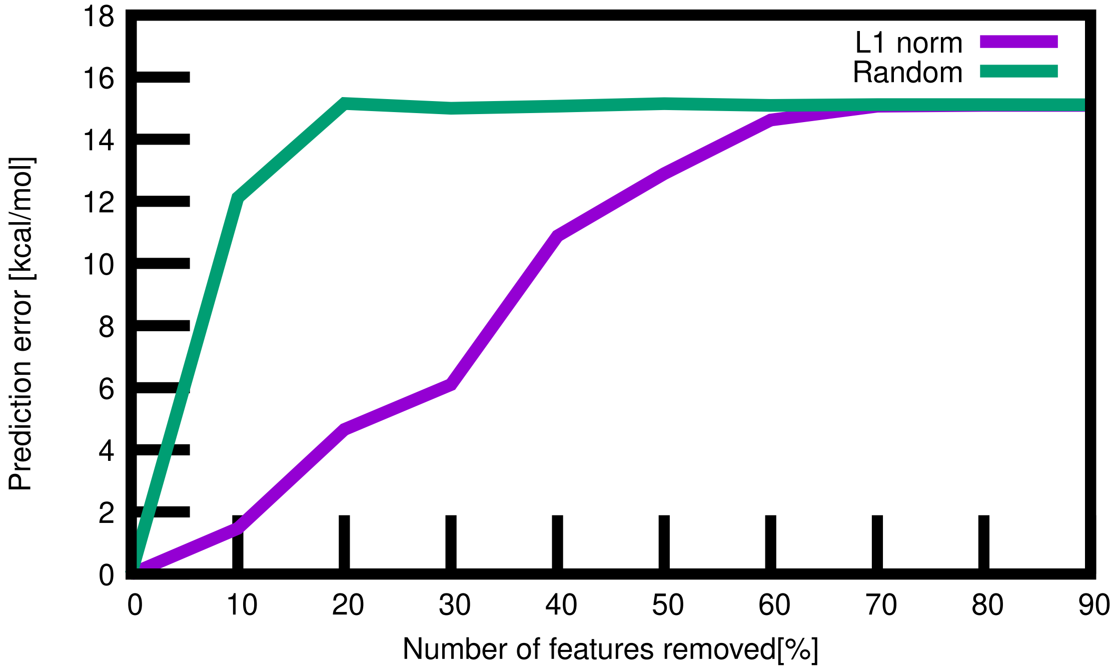

Figure 11: The error of the pruned networks for different percentages using the built in functions provided by Pytorch. The purple line corresponds to the error produced by random pruning. The blue line corresponds to the error introduced by deleting the lowest absolute values.

Both functions were applied to the network. Figure 11 shows the error caused by the deletion of values from 10 up to 90 %. A first look shows quickly , that the error for random deletion is higher than the error for the deletion of smallest values. It can also be seen that the error introduced by this technique is higher than the desired chemical accuracy even if only 10 % of the weights are pruned. It is also apparent, that there is a threshold for the introduced error at 15 kcal/mol.

removed weights[%]	error L_1 [kcal/mol]	error random [kcal/mol]
10	1.46577725	12.11959775
20	4.655297	15.15293275
30	6.09961375	14.99806575
40	10.888631	15.06683975
50	12.89104625	15.1474735
60	14.61955775	15.0904965
70	15.0643925	15.11352575
80	15.08779825	15.1178555
90	15.092379	15.11151775

Table 5: The errors caused by pruning for different percentages of weights are shown in this table. Error L_1 corresponds to the error produced by pruning using the L_1 norm. Error random therefore is the error corresponding to the pruning of random values.

This error is reached with random deletion rather quickly at already 20 %, while the deletion of the lowest values only reach this error after deleting 60 to 70 % of all values. When analysing the acceleration caused by pruning no speedup is found. The cause is most likely the same as has been experienced for the singular value decomposition, which is that loading data and initialising calculations take up most of the computational time.

4.2.2 Quantization applied to SchNet

For quantization, also the tools provided by Pytorch were used. However these tools are only applicable to structures already provided by Pytorch. Therefore custom structures such as newly created layers can not be used by those tools. As an example, take the most basic layer in SchNet called Dense layer. This layer is modeled after the linear dense layer `torch.nn.Linear` provided by Pytorch. The function of these two layers is very similar. The Dense layer provided by SchNet comes with the additional option of adding an activation function, while for the Pytorch case an additional layer would be required. However since the Dense layer is not known by Pytorch the functions provided by it can not be applied.
Currently the only workaround, which was tested, is to transform the Dense layer, which doesn't have any activation functions, into Linear layers. This change is only possible for 3 out of the 5 Dense layers included in the interaction blocks and only for the very last layer in the prediction network. The reason why this treatment is not possible to apply to the Dense layers with an activation function is a slight change in the softplus function used in SchNet. The activation function used in SchNet is not exactly the softplus function ($f_{activation}(x) = ln(1 + e^x)$) but it is shifted by a constant value of $ln(2)$. While the non-shifted softplus function is implemented, the shifted one is making it again necessary to create a custom layer. The reason this is again not beneficial since Pytorch would provide the option to merge layers as briefly mentioned in Section 2.3.3. Pytorch right now does not provide the tools to merge the softplus function with a linear layer. However since the shifted softplus function is not implemented on its own makes it less likely, that it will be possible in the future, while the non shifted version might be available in the future.
Pytorch usually stores models in so-called model files, which contain all the information necessary such as architecture and values of weights and biases for using the model. However Pytorch does

not support this option for quantized models. The only way to store such a model is in a so-called state_dict file, which consists only of the values for weights, biases and other values, which might be necessary for the network. Since the architecture is missing one needs to initialise it right before the calculations, including the quantization steps, with the exception of a calibration step, since these values can be stored. This extra step might cause quite a significant overhead for the usage of such a network.
Given the above mentioned problems and the time needed for the implementation of making quantization work for SchNet this approach was not pursued any further.

4.2.3 General remarks on the Acceleration of SchNet

In the previous chapters, reasons were discussed on why SchNet does not respond as expected to certain acceleration techniques. It can be concluded that SchNet in its current form is very compact and therefore hard to further optimize. The main advantage reason is that SchNet uses rather small matrices from a computational point of view. This property leads to short calculation times for operations such as the matrix vector product. Therefore, acceleration techniques which focus on accelerating these operations have only a small potential. It is also important to state, that while accelerations have been found for certain systems under certain circumstances a decrease in accuracy has also been found. Therefore, it always needs to be evaluated, if a certain factor of acceleration is worth the additional error introduced into the system. The SVD shows the most potential for acceleration with the aforementioned problems of system dependencies and additional errors. The rest of the section will focus on other problems encountered while applying different acceleration techniques.
A first problem is related to the architecture of SchNet. It consists of two neural networks, the representation and the prediction network, which themselves consist of multiple subunits. This rather complex design leads to problems for some functions to access the layers, which store parameters such as weights of the network itself. It might seem like a small problem but again tools provided by Pytorch suffer from this problem making certain task much more demanding from an implementation standpoint.
Another problem stems from the input, which SchNet requires. The input is not a single matrix or tensor but a dictionary of different matrices such as the coordinates of atoms, the types of atoms and a mask, which determines, which atoms are considered neighbours and which are not. This property becomes a problem, when trying to use libraries outside of Pytroch such as the distiller package [29].This library would be able to perform multiple acceleration methods on its own but is not able to since the network requires a test set in the form of a single matrix or tensor.
As discussed before, the main origin of the unsuccessful acceleration attempts are caused by the small size of the matrices used in SchNet. With this information in mind the question arises if it is not possible to accelerate SchNet or if there are techniques, which might still be a possible solution to this task. SchNet uses the representation network in order to create a set of features, which is then used by the prediction network to predict properties. While the careful construction of the first network produces feature sets that are able to predict properties precisely, the computational costs for this network is rather high. Therefore, an acceleration might be possible by reducing this network to a much smaller size. This technique is called knowledge distillation. In general, this technique requires a so-called teacher network and a student network. As the name suggests the teacher network “teaches” the student network to produce the same output. Since the student network consists of less parameter than the teacher network eventually the student network should

be able to reproduce the results from the teacher network with much less computational costs [22]. When applied to SchNet the case is clear: The representation network would act as the teacher network and a smaller network would substitute it in SchNet after it is trained by the former. While such an approach would most likely lead to an acceleration of the network, the bigger question lies in the if and how the accuracy of the SchNet as a whole would be changed. It remains to be seen if this structural change performs better than the techniques used in this work.

5 Conclusion and Outlook

This work focused on the acceleration of SchNet a neural network used for the prediction of quantities such as potential energies, forces, dipole moments and many more. The main technique used for acceleration was singular value decomposition. This method focuses on the reduction of the rank of matrices. The technique promises an acceleration, if the rank is below half of the dimension of the matrix. The acceleration is achieved by substituting the original matrix by two smaller ones. A matrix vector multiplication is then performed by multiplying the vector with both of those matrices. Other techniques investigated were pruning, which focuses on the deletion of weights, and quantization, which aims to store values such as weights and biases in formats, which consume less storage. The acceleration of SchNet proofed to be of minimal success. The main reason for this is the fact, that SchNet operates with rather small matrices. When accelerating such small matrices the amount of operations, which can be reduced, is rather small. Also additional computation time is required since one matrix vector multiplication is substituted by two, which introduces an additional initialisation step. The additional time the second initialisation needs in combination with the small potential of operation reduction lead an increase in calculation time rather than a decrease. Other acceleration techniques investigated in this work suffered from similar problems with the addition of difficulties in implementation. It can therefore be concluded, that SchNet in its current structure is already highly optimized and therefore difficult to accelerate.
Acceleration techniques used in this work focused on the conservation of the structure of SchNet, which leads to techniques only using numerical approaches. However limiting oneself to this approach excludes the possibility of optimizing the structure of SchNet itself. In this work this limitation was chosen deliberately however future work should drop this constraint and focus on more drastic changes of the network.
A first promising candidate is the so-called knowledge distillation. This approach focuses on a neural network to learn the output of another neural network . The main target of this approach for SchNet could be the learning of the representation network with a smaller network than the original representation network. The resulting network would then be used for the prediction network and bypass the majority of operations of SchNet located in the representation network.

References

[1] Boukaye Boubacar Traore, Bernard Kamsu-Foguem, and Fana Tangara. Deep convolution neural network for image recognition. *Ecological Informatics*, 48:257–268, 2018.

[2] W. Williams, N. Prasad, D. Mrva, T. Ash, and T. Robinson. Scaling recurrent neural network language models. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5391–5395, 2015.

[3] Mohammed AlQuraishi. AlphaFold at CASP13. *Bioinformatics*, 35(22):4862–4865, mai 2019.

[4] Julia Westermayr and Philipp Marquetand. Machine learning for electronically excited states of molecules. *Chemical Reviews*, november 2020.

[5] Sergei Manzhos and Tucker Carrington. Neural network potential energy surfaces for small molecules and reactions. *Chemical Reviews*, 2020.

[6] Sergei Manzhos and Tucker Carrington, Jr. Using neural networks to represent potential surfaces as sums of products. *Journal of Chemical Physics*, 125(19):194105, 2006.

[7] Julia Westermayr, Michael Gastegger, Maximilian F. S. J. Menger, Sebastian Mai, Leticia González, and Philipp Marquetand. Machine learning enables long time scale molecular photodynamics simulations. *Chemical Science*, 10:8100–8107, 2019.

[8] Pavlo O. Dral. Quantum chemistry in the age of machine learning. *The Journal of Physical Chemistry Letters*, 11(6):2336–2347, 2020.

[9] K. T. Schütt, P. Kessel, M. Gastegger, K. A. Nicoli, A. Tkatchenko, and K.-R. Müller. SchNetpack: A deep learning toolbox for atomistic systems. *Journal of Chemical Theory and Computation*, 15(1):448–455, 2019.

[10] Sebastian Mai, Philipp Marquetand, and Leticia González. Nonadiabatic dynamics: The sharc approach. *WIREs Computational Molecular Science*, 8(6):e1370, 2018.

[11] Julia Westermayr, Michael Gastegger, and Philipp Marquetand. Combining SchNet and SHARC: The SchNarc machine learning approach for excited-state dynamics. *The Journal of Physical Chemistry Letters*, 11(10):3828–3834, 2020.

[12] Mauro Ferrario, Giovanni Ciccotti, and Kurt Binder. *Computer Simulations in Condensed Matter Systems: From Materials to Chemical Biology*, volume 1. january 2006.

[13] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. A survey of model compression and acceleration for deep neural networks. *CoRR*, abs/1710.09282, 2017.

[14] Cheng Tai, Tong Xiao, Xiaogang Wang, and Weinan Ee. Convolutional neural networks with low-rank regularization. november 2015.

[15] Ben Kröse and Patrick van der Smagt, 1996. URL:http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=50EEC7375F8D57842F4462992169C6D9?doi=10.1.1.18.493&rep=rep1&type=pdf.

[16] K. T. Schütt, H. E. Sauceda, P.-J. Kindermans, A. Tkatchenko, and K.-R. Müller. SchNet – a deep learning architecture for molecules and materials. *The Journal of Chemical Physics*, 148(24):241722, 2018.

[17] S. Albawi, T. A. Mohammed, and S. Al-Zawi. Understanding of a convolutional neural network. In *2017 International Conference on Engineering and Technology (ICET)*, pages 1–6, 2017.

[18] Kristof T. Schütt, Pieter-Jan Kindermans, Huziel E. Sauceda, Stefan Chmiela, Alexandre Tkatchenko, and Klaus-Robert Müller. SchNet: A continuous-filter convolutional neuralnetwork for modeling quantum interactions. *Advances in Neural Information Processing Systems*, 30:992–1002, september 2017.

[19] V. Klema and A. Laub. The singular value decomposition: Its computation and some applications. *IEEE Transactions on Automatic Control*, 25(2):164–176, 1980.

[20] G. W. Stewart. On the early history of the singular value decomposition. *SIAM Review*, 35(4):551–566, 1993.

[21] Tony F. Chan. An improved algorithm for computing the singular value decomposition. *Association for Computing Machinery*, 8(1):72–83, 1982.

[22] James O' Neill. An overview of neural network compression. 2020.

[23] Ruichi Yu, Ang Li, Chun-Fu Chen, Jui-Hsin Lai, Vlad I. Morariu, Xintong Han, Mingfei Gao, Ching-Yung Lin, and Larry S. Davis. Nisp: Pruning networks using neuron importance score propagation. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9194–9203, 2018.

[24] W. Kahan and J. Palmer. On a proposed floating-point standard. *SIGNUM Newsl.*, 14(si-2):13–21, 1979.

[25] Shmuel Gal. An accurate elementary mathematical library for the ieee floating point standard. *ACM Trans. Math. Softw.*, 17(1):26–45, 1991.

[26] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications. *International Conference on Learning Representations*, 2015.

[27] Adrian Bulat and Georgios Tzimiropoulos. Xnor-net++: Improved binary neural networks. *British Machine Vision Conference*, 2019.

[28] Tim Dettmers. 8-bit approximations for parallelism in deep learning. *International Conference on Learning Representations*, 2016.

[29] Neta Zmora, Guy Jacob, Lev Zlotnik, Bar Elharar, and Gal Novik. Neural network distiller: A python package for dnn compression research. october 2019. URL: https://arxiv.org/abs/1910.12232.