Bachelor's Project

Enhancing Structure Search using Machine Learning

Author: Martin Knudsen, 201505931

Supervisor: Bjørk Hammer, Prof.

Department of Physics and Astronomy,
Aarhus University
Denmark

Bachelor of Physics

15.06.2018

English Summary

This report investigates how the use of machine learning techniques can improve the success rate of finding the global minimum of a two-dimensional structure at 0 Kelvin.

Structure search is of great importance in science and is applied in many areas ranging from catalysis over material science to protein folding [1]. Metal-Organic Frameworks (MOFs) is a new research area that could potentially benefit from structure search as presented here. However, the structures considered are simplified and therefore this report is a proof of concept.

In this report structures are constituted by molecules. This differs from usual structure search which focuses on atoms. The molecules contain atoms represented by fixed points that interact by the Lennard-Jones potential. The structure search is performed by utilizing a basin-hopping algorithm that alternates between displacing molecules, *rattling*, and finding a local minimum of the total potential. The latter is achieved by using the 'BFGS' [2] algorithm.

As a demonstration, a structure consisting of seven hexamer molecules with a highly symmetric global minimum is used as a test system. The choice of molecules to rattle is first performed stochastically with a success rate of 59% after 200 iterations. To perform a more informed rattling choice machine learning is applied. This is done by predicting fictitious local energies of the molecules using techniques such as feature vectors, clusters, k-means and ridge regression. Utilizing this energy information the most unstable molecules were chosen for rattling resulting in an improvement of the success rate to 73%.

Danish Summary

Denne rapport søger at undersøge, hvordan Machine Learning teknikker kan bruges til mere effektivt at finde det globale minimum af en todimensionel struktur ved 0 Kelvin.

Struktursøgning har mange anvendelser i naturvidenskaben. Det kan bruges til så forskellige ting som katalyse og materialevidenskab til proteinfoldning [1]. Et nyt forskningsområde, som muligvis kan drage nytte af struktursøgning som præsenteret her, er de såkaldte "Metal-Organic Frameworks" (MOFs). De undersøgte strukturer er dog simplificerede, og denne rapport er derfor et "proof of concept".

I stedet for at opbygge strukturer af atomer, som det er kutyme, består de her af molekyler. Atomerne er fikseret i molekylerne og interagerer med hinanden via Lennard-Jones potentialet. Selve struktursøgningen sker ved en såkaldt "basin-hopping" algoritme, hvor molekylerne skiftevis forskubbes, kaldet "rattling", hvorefter et lokalt minimum findes. Sidstnævnte udføres ved at bruge den såkaldte 'BFGS' [2] algoritme.

Som et eksempel kigges på en struktur bestående af syv "hexamer" molekyler, som har et meget symmetrisk globalt minimum. Først demonstreres algoritmen med et tilfældigt valg af, hvilke molekyler, der skal forskubbes med en succesrate på $\approx 59\%$ efter 200 iterationer. Dernæst bruges Machine Learning til at foretage et smartere valg af samme. Dette gøres ved at postulere fiktive lokale energier, der forudsiges ved brug af teknikker såsom "clustering", "k-means" og "ridge regression". Ved at bruge denne energi-information, kan de mest ustabile molekyler forskubbes, og dette resulterede i en forbedring af algoritmen til $\approx 73\%$.

Preface

This is the report of my bachelor's project completed in the third year of my bachelor degree in physics. It was written using the master thesis LaTeX- template "IT University Copenhagen" by Tony Beltramelli from `www.sharelatex.com`. Basic ideas and methods used are in-line with an article by S.A. Meldgaard et al. [1], however, the focus is on molecules rather than atoms.

Generally an undergraduate knowledge of physics should be sufficient to understand this report. However, the electives Computer Science and Machine Learning worth 30 ECTS which I attended proved to be to be very useful when writing this report. Most of the numerical python codes used in the report are attached in appendix A. To completely understand this section one needs an understanding of python and the NumPy package, yet, a physics undergraduate is likely to understand most since it is completely commented.

I wish to thank Mathias Siggaard Jørgensen and especially Søren Ager Meldgaard who endured my endless questions with patience as well as my supervisor Bjørk Hammer for giving me the opportunity to write about this subject and for his good counsel whenever I needed it.

Contents

1 Introduction

The overall purpose of this project is to show how applying machine learning techniques can accelerate the search for the global minimum of a two-dimensional structure at 0K. Structure search is important in many areas of science such as catalysis, material science and protein folding [1]. A new promising application area of structure search is the so-called Metal-Organic Frameworks (MOFs) which are introduced in section 1.1. Instead of focusing on single atoms forming a structure the focus is on molecules. Although the molecules used in this report could represent real molecules they are represented as points fixed to each other and are only interacting via the Lennard-Jones [3, p. 241] potential. This is of course a great simplification and this report therefore primarily serves as a proof of concept.

The global minimum is interesting because at a 0K the most likely structure is the one with the lowest energy so the search for the structure coincides with a search for the global energy minimum. Of course in reality it is not possible to obtain absolute zero because of quantum mechanics, but one can come close [4].

The rest of chapter 1 will show a potential application of global structure search, MOFs, for motivation. Chapter 2 will introduce the specific structures used and the traditional basin-hopping algorithm for finding the global minimum. In chapter 3 the basin-hopping algorithm will be revisited and improved with machine learning after a thorough introduction into this area. Most of the codes used throughout this report can be seen in appendix A.

1.1 Metal-Organic Frameworks

Metal-Organic Frameworks are structures consisting of metal atoms or clusters called *nodes* bound together by organic molecules called *struts* [5]. They can be build in an orderly crystalline fashion by repeating the fundamental metal-organic building blocks called secondary building unit [6]. An interesting thing about MOFs is that both the struts and the nodes can be exchanged resulting in different properties of the MOFs leading to more than $20,000$ different MOFs being discovered to date [6].

A famous example, the MOF-5, can be seen in figure 1.1. The nodes shown as the blue clover-shapes consist of Zn_4O and the struts shown as the six-membered rings are 1,4-benzenedicarboxylate (BDC^2). The spheres inside the structure are holes and highlight its porous nature. The pattern could be repeated indefinitely illuminating the crystalline nature of MOFS.

MOF-5 specifically can be used as storages of gasses such as hydrogen and methane in its holes [7]. Counterintuitively, because of the increased internal surface area, some MOFs can even be used to increase the amount of gas in a fuel tank by two to three times [6]. The use of organic rings allows different functional groups and some MOFs therefore have catalytic properties [6].

Although MOFs come in both 3D and 1D variations, the interesting structures for this report are 2D. The following is a recent example of a surface MOF from the article [8]. It forms a characteristic porous honeycomb pattern as shown in figure 1.2. It consists of a copper atom trimer (Cu(111)) connected via the organic six-membered ring tetrahydroxybenzene (THB). Figure 1.1.a) is a scanning tunneling microscope image of the MOF and shows the porousness of the structure in the form of the black voids. The light points on the picture are THBs and the Cu(111)s are not visible. b) shows the suggested theoretical model for the structure with the unit cell indicated as in a). This theoretical model was found using DFT calculations and several candidates were considered. It is for structure search such as in

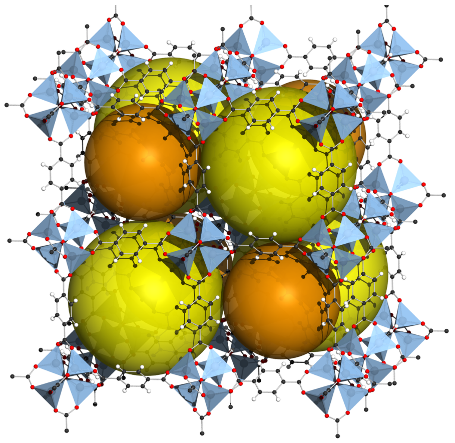


Figure 1.1: The MOF-5 structure. The organic linkers, BDC^2, are represented by the six-membered ring while the metals are shown as the blue clover-shape. The spheres contained in the structure represent the empty space contained within the structure. Taken from [7]

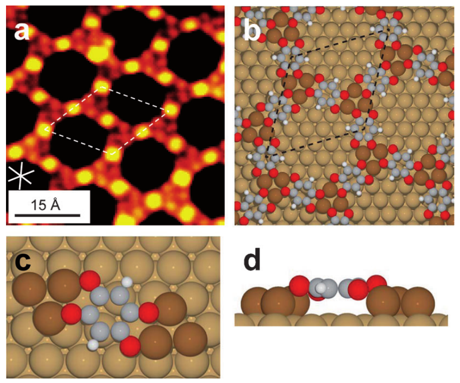


Figure 1.2: MOF consisting of Cu(111) and THB. a) A scanning tunneling microscope image of the MOF which shows the porousness of the structure in the form of the black voids. The bright lumps on the picture are THBs and the Cu(111)s are not visible. b) The suggested theoretical model for the structure with the unit cell indicated as in a). c) A zoom of the molecular interaction d) A sideways view of the structure. Taken from [8].

this case that the method presented in this report could become useful. Both a trimer molecule and a six-membered ring are used in this report and these could potentially be used to build such a network.

2

Traditional Structure Search

In this chapter the used molecules and their interaction is introduced. This allows the basin-hopping algorithm in its traditional incarnation for finding the global minimum structure to be presented presented.

2.1 Molecules

As this is a proof of concept, several structures with various molecules have been experimented with. The molecules tested consisted of 2,3,4 and 6 identical atoms which in chemistry jargon are called dimers, trimers, tetramers and hexamers respectively [9]. They can be seen in figure 2.1. They are

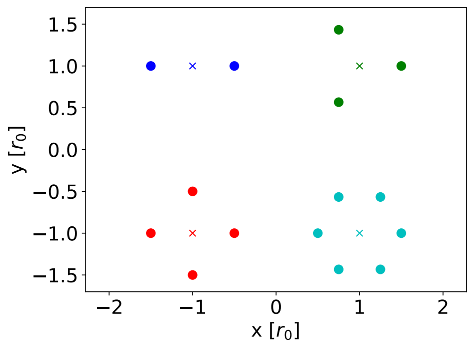


Figure 2.1: The different molecules used in the report: Dimers, trimers, tetramers and hexamers. The atoms are represented by dots and the "center of mass" of the molecules by crosses.

equilateral polygons with a "radius" from the center of mass of $r = 0.5$.

This choice is arbitrary and would in reality depend on the dimensions of the molecule used. To describe these molecules only three coordinates are needed as seen on figure 2.2: The Cartesian coordinates of the center of mass

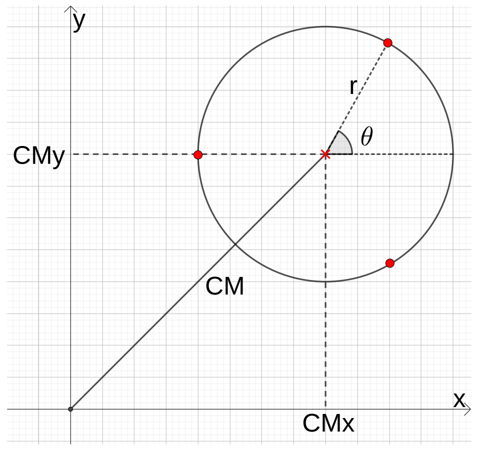


Figure 2.2: The coordinates of a molecule in this report illustrated by a trimer molecule. The molecule is an equilateral polygon with a center of mass. The distance from this point to the individual atoms is the radius, r. The coordinates are the x- and y-coordinates of this center of mass and the angle to the first atom counterclockwise.

CMx and CMy and an angle θ from the horizontal to the first atom counterclockwise. Once these coordinates are known the rest of the atoms position follow because they are fixed with respect to each other. In appendix A.1.1 the code of the hexamer molecule as a class can be seen. The molecules ”contain” atoms by simply giving the correct positions of them when the ”get_coordinates()” function is used. A structure consists of several of these molecules as in figure 2.3.

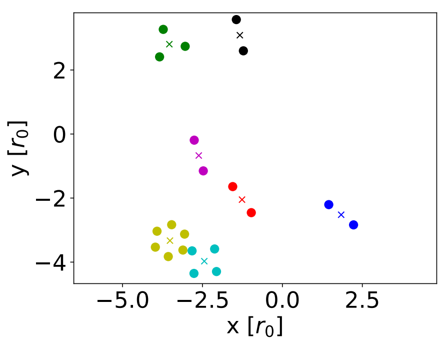


Figure 2.3: A random ensemble of molecules consisting of four dimers, one trimer, one tetramer and one hexamer.

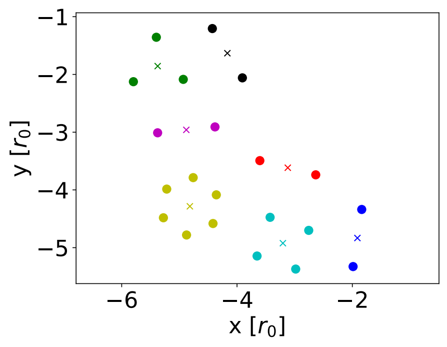


Figure 2.4: The same molecules after "relaxing" using BFGS minimization. Here the minimum of the Lennard-Jones potential was set to be at $r = 1$.

2.2 Lennard-Jones Energy

To find the global minimum structure composed of these or other molecules one must minimize the energy of the structure. Therefore a way of determining the energy of a structure is needed and it must be a function of the molecule coordinates as described above. The potential energy used in this report is the pairwise Lennard-Jones potential [3, p. 241]. It is defined in this report by the atom-wise interaction:

$$E_{pot}(r) = \epsilon_0 \left[\left(\frac{0.5r_0}{r} \right)^{12} - 2 \left(\frac{0.5r_0}{r} \right)^{6} \right] \tag{2.1}$$

This version of the potential has a minimum at $r = 0.5r_0$ and the value of the potential here is $E_{pot}(r_0) = -\epsilon_0$ as the reader can easily verify. ϵ_0 symbolizes the energy and is a measure of how much the particles attract each other. r_0 is the arbitrary unit of length. The potential used for the basin-hopping algorithm in units of ϵ_0 r_0 can be seen in figure 2.5 Since this potential describes the energy associated with two atoms the total energy of the structure is taken as the sum of every atom with all other atoms. However, because molecules were used the potential was specifically designed to ignore contri-

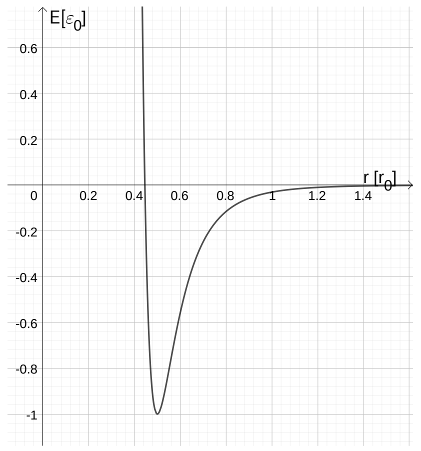


Figure 2.5: Lennard-Jones potential used in basin-hopping algorithm with minimum at $r = 0.5r_0$ and in units of ϵ_0 and r in arbitrary length unit r_0.

butions from atomic pairs within the same molecule. This doesn't matter for the algorithm, however, because the contribution is constant and independent of the coordinates of the molecule. Applying equation (2.1) the total energy of a structure now looks like:

$$E_{tot}(\mathbf{r_1}, ..., \mathbf{r_A}) = \frac{1}{2}\sum_{i=1}^{A}\sum_{j=1}^{A} E_{pot}(r_{ij}) \quad \text{for}\, m_i \neq m_j \tag{2.2}$$

where $\mathbf{r_k}$ is the position vector of the kth atom, A is the total number of atoms in the structure, m_i is the molecule atom i belongs to and m_j the molecule atom j belongs to. $r_{ij} = \|\mathbf{r_i} - \mathbf{r_j}\|$ is the Euclidean distance between the atoms. The division with 2 comes from having counted each contribution twice. Equation (2.2) is a function of all the Cartesian coordinates of the atoms, but since they are held fixated with respect to the molecules' center of mass it can be rewritten as a function of the molecular coordinates:

$$E_{tot}(CMx_1, ...CMx_M, CMy_1, ..., CMy_M, \theta_1, ..., \theta_M)$$

where M is the total number of molecules. The exact calculation can be seen in the code of appendix A.1.2.

The potential landscape as a function of the parameters could look like in figure 2.6. The global minimum is at point I. If the structure is at point

I it is very unlikely that it climbs over H or tunnels into G. Hence, it is possible to find structures in configurations other than their global minimum but very unlikely (unless the energy is very close to the global minimum or if either the temperature or the entropy is very high).

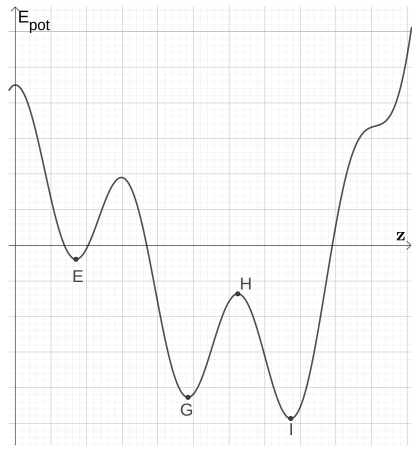


Figure 2.6: An example of the potential landscape of a structure as a function of all its parameters represented by $\mathbf{z}$. The global minimum is at point I.

The problem is now reduced to finding the global minimum of E_{tot}. However, E_{tot} is a $3M - 3$ dimensional surface if rotational and translational invariance of the whole structure is accounted for [10]. A naïve approach would be to simply minimize E_{tot} with respect to all its variables. There are many algorithms for multidimensional minimization, but in this report the 'BFGS' method [2] from the python package Scipy [11] is used as in appendix A.1.4. It minimizes E_{pot} with only a start guess of the variables. Providing the gradient of the function to be minimized improves the accuracy and speed of BFGS greatly, but it is not necessary and was not used. The effect of minimizing can be seen on figure 2.4. The molecules lump together as the angle and location of the molecules are adjusted to minimize E_{pot}. This is called "relaxing" the structure. Sometimes highly ordered structures can emerge from the chaos as can be seen in figure 2.7-2.8.

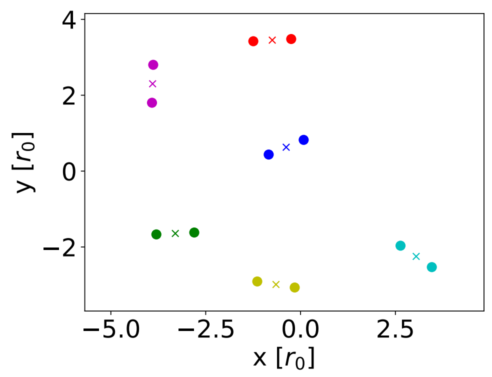


Figure 2.7: Structure consisting of six dimer molecules after initializing coordinates randomly.

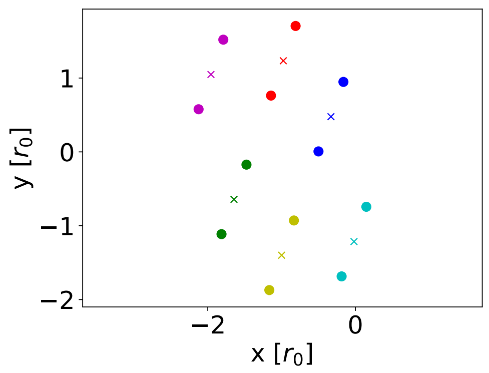


Figure 2.8: The same molecules after "relaxing" using BFGS minimization with a minimum of the Lennard-Jones potential was set to be at $r = \sqrt{1}$. The found minimum is probably not global.

An illuminating example of a structure that is initially very unordered and becomes ordered after relaxing is seen on figures 2.9-2.10.

Even though this structure seems very symmetric it is actually *not* the global minimum, which can be seen in figure 2.12. This illustrates a potential pitfall when minimizing the coordinates of a structure; There is usually no guarantee that the minimum found is actually the global minimum and not just local. Looking back at figure 2.6 point I might represent the global minimum while G and E represent local minima. The global minimum was verified by performing several test-runs of the basin-hopping algorithm described in section 2.3 and choosing the structure with the lowest energy. This structure would typically get found in the largest majority of cases.

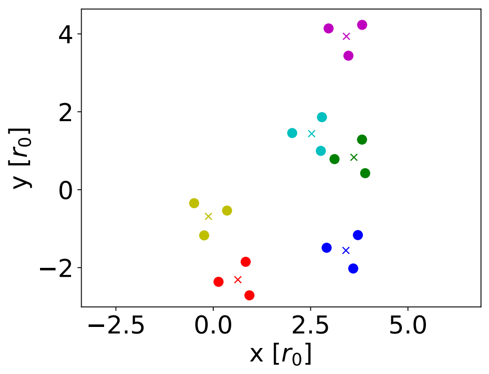


Figure 2.9: Structure consisting of six trimer molecules after initializing coordinates randomly.

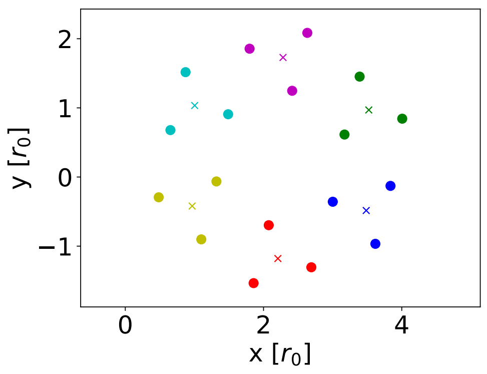


Figure 2.10: The same molecules after "relaxing" using BFGS minimization with a minimum of the Lennard-Jones potential set to be at $r = 1$. The found minimum is *not* a global minimum but a local one.

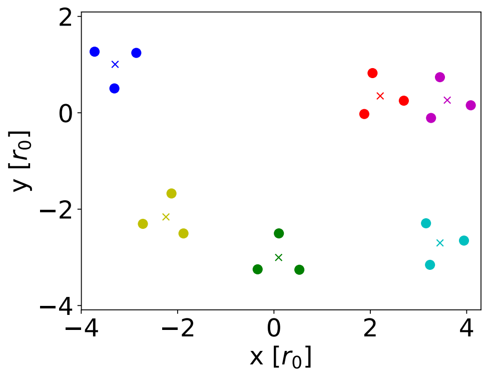


Figure 2.11: Structure consisting of six trimer molecules after initializing coordinates randomly.

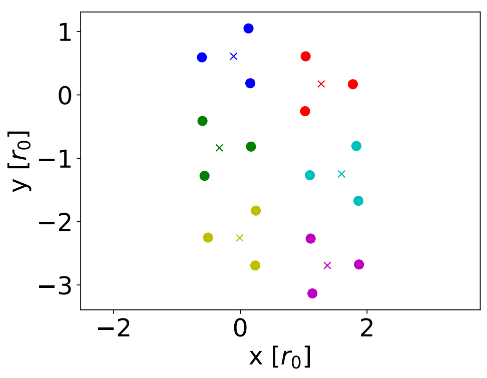


Figure 2.12: The same molecules after "relaxing" using BFGS minimization. The found *is* the global minimum.

2.3 Basin-Hopping

A systematic approach to finding the global minimum is the so-called basin-hopping algorithm [12]. Put simply it divides the potential landscape as in figure 2.6 into several "basins", each having their own local minimum. The global minimum is to be found in one of the basins. Relaxing simply amounts to going downhill until a local minimum is found. A basin can be escaped by perturbing the coordinates of the structure, thereby going to f.ex. basin E to G on figure 2.6. When this is done by randomly incrementing the coordinates of the atoms by a small amount it is called "rattling", but this term will be used for all kinds of basin-hopping. In this report the method of rattling consists of placing one or more molecules within a disc of radius $r = 3.5$ centered at the center of mass of the whole structure. The number of molecules to be rattled is chosen in accordance with table 2.1 so that it is most likely to rattle one molecule with declining probability of rattling more molecules. This is to combat difficult local minima. *Which* molecules are rattled is traditionally chosen at random and this will be called the *stochastic* basin-hopping algorithm. More details can be seen in appendix A.3.1.

#	1	2	3	4	5	6
$\mathbb{P}$	8.75×10^{-1}	9.38×10^{-2}	2.34×10^{-2}	5.86×10^{-4}	1.45×10^{-3}	3.66×10^{-4}

Table 2.1: Probability of choosing a certain number of molecules to rattle used in this report.

In order to find the global minimum a series of rattling and relaxations are performed until it is found. The exact process can be seen in figure 2.13. Initialization amounts to assigning random start values to the coordinates with certain limitations; they are contained within a 4 x 4 box centered at the origin and the center of mass of the molecules cannot be closer than $\|\mathbf{CM_a} - \mathbf{CM_b}\| \leq 1.3$ to prevent overlapping where a and b represent two distinct molecules. Furthermore, the molecules are relaxed before entering the basin-hopping loop. In the loop the structure is alternately rattled and relaxed until the global minimum is found. The upper limit to iterations is set to 200 in this report.

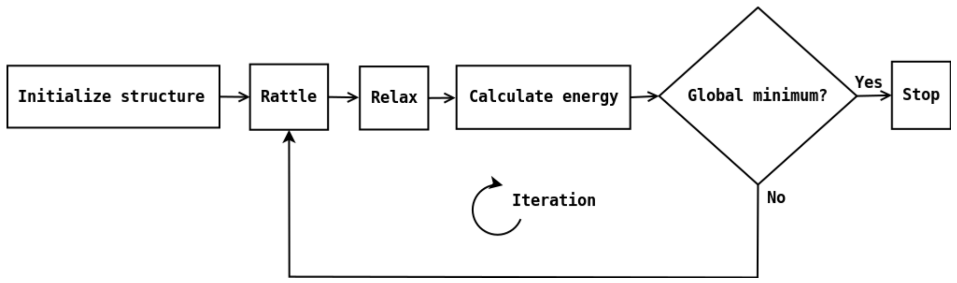
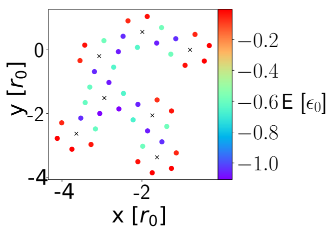


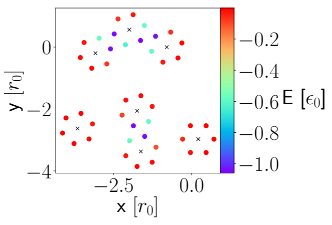
Figure 2.13: Basin-hopping algorithm for finding the global minimum structure. Initialization amounts to starting with random coordinates of the molecules and a relaxation. Rattling is the perturbation of the molecules and relaxing means finding the local minimum. The maximum number of iterations is set to 200.

2.4 Structure Search of Hexamer Structure

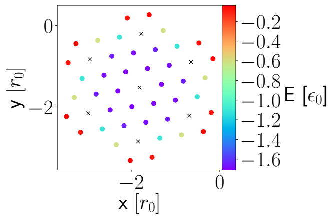
In order to test the basin-hopping algorithm a structure composed of seven six-membered rings, or hexamers, is used as can be seen in figure 2.14. The idea was actually to find the characteristic honeycomb pattern of graphene since it can be seen as a structure consisting of hexamers. But as one can see, the final global minimum in figure 2.14 does not have the characteristic shape of graphene. This is probably because the potential, equation (2.1), is too simplistic to describe the graphene system. The global minimum was found by inspecting structures of repeated basin-hopping runs and saving the ones with the lowest energy. An indication of the correctness of the found global minimum is its high symmetry and many "chemical bonds". On the figure one can see how the global minimum has several atoms with low energy especially towards the center of the structure. The choice of which molecule to rattle, however, does not seem very clever since the molecule chosen has the lowest energy. That this choice leads to the global minimum seems to be accidental which is typical of the traditional basin-hopping algorithm. In general, a better choice might have been the red molecules since they have a higher energy.



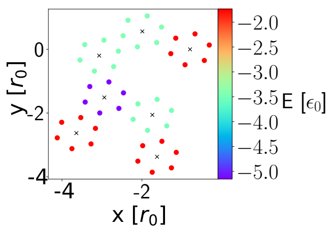
(a) Before rattle.



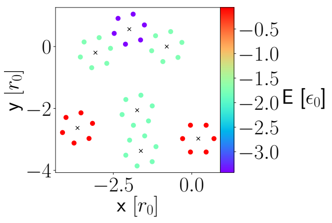
(b) After rattle.



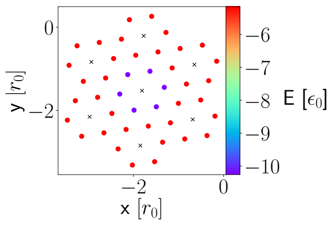
(c) Global minimum found.



(d) Before rattle.



(e) After rattle.



(f) Global minimum found.

Figure 2.14: An example of the last step of the stochastic basin-hopping algorithm before finding the global minimum for the hexamer structure. Figures a)-c) show the atomic energies as calculated by the Lennard-Jones potential and are marked by colour. Figures d)-f) show the same structure but with the energy of each molecule instead. The global minimum looks like in f) and was found by inspection.

In order to see how effective the algorithm was in finding the global minimum it was repeated with a new initialization 300 times. Because of the 21 coordinates necessary to describe it and the lack of an analytical gradient the relaxing took a considerable time. With a maximum of 200 iterations this became too costly for a normal PC. Therefore, the supercomputer, "Grendel" at the "Centre for Scientific Computing in Aarhus", was used for calculations. In order to enable parallel computation of the runs each run was send as a separate job to Grendel making the total computation time *only* several hours. The machine learning algorithm presented in chapter 3 took even longer, approximately 24 hours.

The result of the runs can be seen in figure 2.15. Evidently by just choosing a random molecule(s) by 200 iterations of the algorithm approximately 59% of the runs had found the global minimum. The graph has a slight negative curvature which means it is slightly more likely to find the global minimum at a low number of iterations. Looking back at figure 2.14 if the energy information calculated at each iteration could be used this could lead to a more informed rattling choice and possibly a better algorithm. This is the subject of the next chapter, but first a general description of machine learning is necessary.

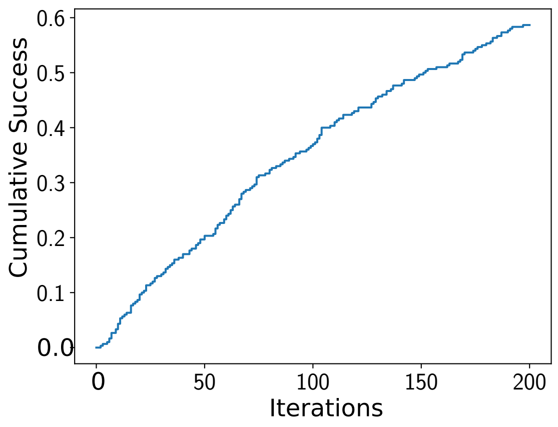


Figure 2.15: Cumulative success rate of 300 runs of the stochastic basin-hopping algorithm.

3 Structure Search Using Machine Learning

Machine learning is often used interchangeably with Artificial Intelligence but is in fact a subcategory of it in which data is used to "teach" the computer to solve a problem [13, p. 398]. To apply it to the basin-hopping algorithm machine learning in general is first introduced followed by the specific techniques utilized for improvement of the algorithm.

3.1 The Learning Problem

The fundamental assertion of machine learning is that an unknown underlying process can be uncovered by using data [14, p. 11]. Under what circumstances and to what degree this is possible is the essence of the learning problem [14, p. 1]. There are several paradigms in machine learning, but in this report the supervised and unsupervised paradigms will be used.

3.1.1 Supervised Learning

In one sentence the problem of supervised learning can be defined as follows: Given the observed data is it possible to predict the outcome of new data? If so, learning has taken place. More schematically one can draw the problem as shown in figure 3.1. One typically has some data $(\mathbf{x_1}, y_1), \cdots (\mathbf{x_N}, y_N)$

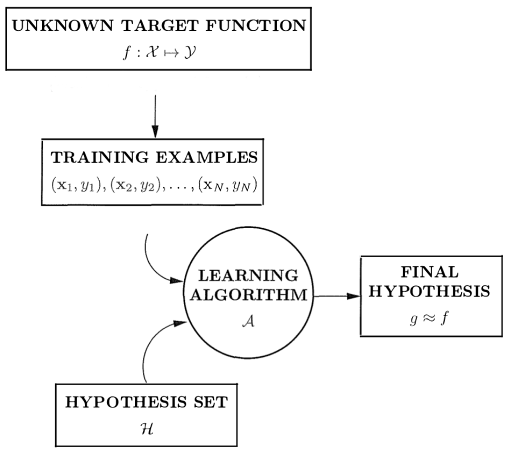


Figure 3.1: Schematic drawing of the learning problem from [14, Fig. 1.2]

contained in a dataset $\mathcal{D}$, where $\mathbf{x_i}$ is a vector with the parameters or *features* of the problem and a corresponding result y_i. It is then assumed that there is a connection between the parameters and the result represented by an unknown target function $f(\mathbf{x})$. The learning process now corresponds to picking a function $g(\mathbf{x})$ that best approximates $f(\mathbf{x})$. Possible candidates for $g(\mathbf{x})$ are contained within the hypothesis set $\mathcal{H}$. The learning algorithm $\mathcal{A}$ as well as the contents of $\mathcal{H}$ are specific to the particular problem and can be changed to fit particular needs.

A simple example is trying to establish a connection between types of customers in a store and whether they will buy a certain product. The parameters could be $\mathbf{x} = (\text{age}, \text{gender}, \text{salary})$ and the result is a simple binary $y = 1 \wedge -1$ for yes and no, respectively. Here $\mathbf{x}$ is called a *feature vector*. The goal now is to find a connection between $\mathbf{x}$ and y. There is obviously no simple analytical expression covering all costumers, because two different costumers of same age, gender and salary might still choose different decisions. A possible simple connection could be made by weighting the parameters differently and sort on the sign. If $\mathbf{x}$ is changed to include an offset $\mathbf{x} = (1, \text{age}, \text{gender}, \text{salary})$ this looks like

$$g(\mathbf{x}) = \text{sign}(\boldsymbol{\theta}^T \mathbf{x}) \tag{3.1}$$

where $\boldsymbol{\theta} = [\theta_0, \theta_1, \theta_2, \theta_3]$ is a vector containing the weights of all the parameters and an offset θ_0. Learning is now reduced to the palpable problem of finding the $\boldsymbol{\theta}$ that best represents the data points already seen. $\mathcal{H}$ in this case is just the infinite set $\mathcal{H} = \{\boldsymbol{\theta} \in \mathbb{R}^4\}$. This particular $\mathcal{H}$ is actually called a *perceptron* [14, p. 5] and the corresponding learning algorithm is known as the *perceptron learning algorithm.* The weights are found by the simple recursion [14, p. 7]

$$\boldsymbol{\theta_{i+1}} = \boldsymbol{\theta_i} + y_i \mathbf{x_i} \tag{3.2}$$

offered here without proof. To find the weights one simply initializes $\boldsymbol{\theta_0}$ with random appropriate values and run through all data points using the recursion. The resulting $\boldsymbol{\theta_N}$ corresponds to the learned model and if successful this model generalizes to new data. Meeting a new customer a prediction of whether or not he will buy can be obtained by simply plugging his parameters in to equation (3.1).

Another prominent example of supervised learning is artificial neural networks [15, p. 2]. It won't be covered in detail here, but training neural networks is very similar to the perceptron in that it ultimately amounts to finding weights using known data [15, p. 7].

3.1.2 Unsupervised Learning

In supervised learning a complete set of parameters and results, $(\mathbf{x}_i, y_i)$, is available. When only the parameters $\mathbf{x}_i$ are given, the unsupervised learning paradigm is entered [14, p. 13]. Instead of trying to predict a result from the parameters the goal is now to find structure and order in the data itself.

An example of this is *clustering.* To return to the example of the previous section a store might want to group its customers depending on the data it has about them. Clustering amounts to finding the customers who are closest to each other in the feature space of customers. The 2D case of age and salary might look as in figure 3.2. On the figure, the data has

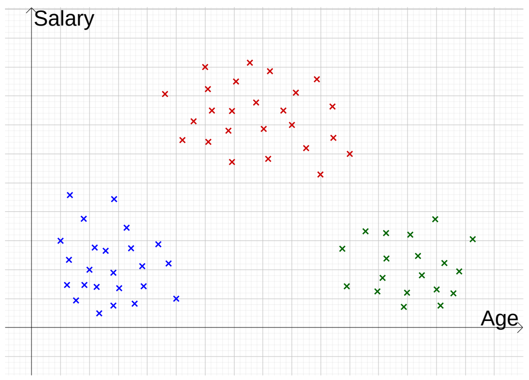


Figure 3.2: Example of two-dimensional data that can be clustered. The clustering is indicated by the colours.

been clustered according to the two dimensions into three clusters. The information contained in the clusters, namely their centers, could be valuable for the store. It might turn out that most of the customers belong to a certain age, gender and salary group. While this might seem trivial it is by no means the case with tens or hundreds of dimensions. The number of clusters can be altered to fit the needs of the problem at hand.

3.1.3 Generalization

Generalization describes under what circumstances it is possible to learn outside $\mathcal{D}$. On a first glance it might actually seem impossible to ever learn. The dataset $\mathcal{D}$ that is trained on is normally only a fraction of the enormous/infinite possibilities of datasets. So it is not possible to predict patterns outside of $\mathcal{D}$ with a 100% certainty [14, p. 18]. There is always the possibility that the observed data is "lucky" and that $\mathcal{D}$ isn't representable of the problem. This is a well-known problem in statistics [16]. To return to the store example a $\mathcal{D}$ as in figure 3.2 could be observed, but this might be the only time in the history of the store that it sees those groups. Hence, whatever pattern is found from it is irrelevant.

Fortunately for machine learning as long as the *probability* of a representative $\mathcal{D}$ is high enough the *chance* of learning is reasonable. To explain this probability one can look at the Hoeffding inequality with a union bound. Offered here without proof it can be written in the form [14, p. 24]:

$$\mathbb{P}[|E_{in}(g) - E_{out}(g)| > \epsilon] \leq 2Le^{-2\epsilon^2 N} \tag{3.3}$$

Here $\mathbb{P}[\cdot]$ denotes the probability of the event, g is the learned hypothesis from $\mathcal{H}$, ϵ the tolerance, L the size of the hypothesis set $\mathcal{H}$ and N the sample size. $E_{in}(g)$ is the fraction of $\mathcal{D}$ where the hypothesis g predicts the correct result. $E_{out}(g)$ is the error of predicting the result of new data points outside $\mathcal{D}$. Typically $E_{in}(g)$ is known, but $E_{out}(g)$ is unknown. In words this inequality expresses that the probability of observing an out-of-sample error that deviates more than the tolerance from the in-sample error is upper bound by the right-hand side of (3.3). In order for the prediction to be as good as possible $E_{out}(g)$ has to be minimized. There are two steps necessary to achieve this: Making the right-hand side of (3.3) as small as possible and making $E_{in}(g)$ as small as possible. Intuitively the first step consists of increasing the number of data points N and decreasing the number of available hypothesis L. The second step consists of predicting as well as possible inside $\mathcal{D}$.

Looking back at the supervised store example there now is a problem: Here, $\mathcal{H}$ is infinite. So according to (3.3) generalization is impossible. Fortunately, because $\mathcal{D}$ is finite the number of ways dividing the data, the *dichotomies*, are finite as well. If 10 data points are observed and the only results are 1 and -1, the number of different dichotomies are $2^{10} = 1024$. This speculation leads to the *Vapnik-Chervonenkis Generalization Bound* again offered here without proof [14, p. 53]

$$E_{out}(g) \leq E_{in}(g) + \sqrt{\frac{8}{N} \ln \frac{4l_{\mathcal{H}}(2N)}{\epsilon}} \tag{3.4}$$

that is true with probability $\geq 1 - \epsilon$. $l_{\mathcal{H}}(N)$ is called the *growth function* and is defined as the number of dichotomies possible by using $\mathcal{H}$ on N points. It is always upper bound by $l_{\mathcal{H}}(N) \leq 2^N$ [14, p. 42].

Equation (3.3) or 3.4 will not specifically be used in this report but they are important as they provide the theoretical justification of machine learning.

3.1.4 Overfitting

One might think that decreasing E_{in} is always a good idea and will decrease E_{out} as well. But according to equations (3.3) and (3.4) this is actually not the case. In order to decrease E_{in} one either has to increase the *size* of $\mathcal{H}$ or the *complexity* of $\mathcal{H}$. This in turn makes L or $l_{\mathcal{H}}$ larger and hence make the bounds looser. This is a well-known phenomenon in mathematics. As an example look at the Fourier series of a function describing it perfectly within an interval [17, p. 42-46]. Outside that interval the series completely diverge from the function. Another example is polynomials which can completely recreate the data points if the degree is high enough. In fact there always exists a polynomial of order n that can pass through any $n + 1$ points [18, p. 1]. When E_{in} is minimized at the expense of E_{out}, the data is said to be a victim of *overfitting*. At the other extreme E_{in} could be completely disregarded and the focus could lie entirely on minimizing L or $l_{\mathcal{H}}$. Looking at (3.3) and (3.4) this would lead to $E_{out} \approx E_{in}$, but this is no improvement if E_{in} is very high. This phenomenon is called the *approximation-generalization trade-off* [14, p. 62].

An easy way to combat this problem is by using *regularization*. This basically means "punishing" the learning algorithm $\mathcal{A}$ for the complexity of the hypothesis h therefore encouraging it to choose a simple model [14, p. 126].

3.2 Machine Learning Techniques

Building on the theory from section 3.1 these are the particular machine learning methods used in this report.

3.2.1 Local Feature Vector

Often it is easier to learn if the data is transformed to a different space with particular properties [14, p. 99-100]. In this report a way to present the ensemble of atoms that is invariant to rotation and translation is needed, since the only thing that matters for the local atom/molecule is its environment — independent of where in space it is (or atleast this is a reasonable assumption in everyday life). The Cartesian coordinates of the atoms themselves don't have these properties; if the structure is moved or rotated they change. There are many ways of doing this, but the one used here is the Behler-Parrinello feature [19]. It is an atomic feature with a radial and an angular component for each atom, equations (3.5) and (3.6)

$$f_i^1 = \sum_{j\neq i}^{all} e^{-\eta(R_{ij}-r_s)^2 f_c(r_{ij})} \tag{3.5}$$

and

$$f_i^2 = 2^{1-\xi} \sum_{j,k\neq i}^{all} (1+\lambda\cos\theta_{ijk})^{\xi} e^{-\eta(r_{ij}^2+r_{ik}^2+r_{jk}^2)} f_c(r_{ij}) f_c(r_{ik}) f_c(r_{jk}) \tag{3.6}$$

with

$$f_c(r_{ij}) = \begin{cases} 0.5\left[\cos\frac{\pi r_{ij}}{r_c}+1\right] & \text{for}\, r_{ij}\leq r_c \\ 0 & \text{for}\, r_{ij} > r_c \end{cases} \tag{3.7}$$

where $\theta_{ijk} = \frac{\mathbf{r_{ij}}\mathbf{r_{ik}}}{r_{ij}r_{ik}}$ is the angle between atom i, j and k with i in the center with $\mathbf{r_{ij}} = \mathbf{R_i} - \mathbf{R_j}$ is the vector from the position of atom j to atom i. Equation (3.7) is called the cutoff function and ensures that only atoms within the distance r_c are contributing to the terms of the features (3.5) and (3.6). Here η, r_s, ξ, r_c and λ are parameters. These functions can then be combined with different values for the parameters. In this report two 12-

dimensional feature vectors were used $\mathbf{f} = \left[f_1^1, f_1^2, ..., f_6^1, f_6^2\right]$ consisting of all possible permutations of the parameters which can be seen in table 3.1. The

	r_C	r_s	η	ζ	λ
$\mathbf{f_1}$	2	0	0.05, 2, 8	0.005	1, −1
$\mathbf{f_2}$	2	0	0.05, 2, 8	0.005, 0.05	1

Table 3.1: Parameters of the two 12D Behler-Parrinello features used. The features consist of all possible permutations of the parameters.

values of the parameters were inspired by [20].

3.2.2 K-means

In order to predict energies of the atoms, the feature vectors have to be divided into different groups that can be assigned a particular energy. This is done by using unsupervised learning, specifically clustering, as explained in section 3.1.2. The objective is now to identify the correct k clusters for N datapoints among the many possibilities. This of course depends on the definition of correct. A simple definition is the clustering that minimizes the total Euclidean distance to the cluster centers [21, p. 334]. The cluster centers , or centroids, are defined by [21, p. 333]

$$\boldsymbol{\mu_i} = \frac{1}{|C_i|} \sum_{\mathbf{x_j} \in C_i} \mathbf{x_j} \tag{3.8}$$

where $|C_i|$ is the number of points contained in cluster C_i. In order to not exhaustively try all clusterings the k-means algorithm is used [21, p. 335] which can be seen in algorithm 1. Here d is the dimension of the data points and ϵ is the chosen tolerance. It starts out by initializing the centroids at random, and then repeating the two following steps until the total change of the centroids between iterations is less than the tolerance:

1. Assigning data points to the closest cluster.

Algorithm 1 K-means Algorithm for finding the best clustering of $\mathcal{D}$. Taken with a few modifications from [21, p. 335].

```
1: $t = 0$
2: Initialize $k$ centroids randomly $\boldsymbol{\mu_1}, ..., \boldsymbol{\mu_k} \in \mathbb{R}^d$
3: while $\sum_{i=1}^{k} \left\| \boldsymbol{\mu_i^t} - \boldsymbol{\mu_i^{t-1}} \right\|^2 > \epsilon$ do
4:     $t \leftarrow t + 1$
5:     $C_i \leftarrow \emptyset$ for $i = 1, ..., k$          ▷ Initialize all clusters as empty sets
6:     for $\mathbf{x_i} \in \mathcal{D}$ do          ▷ Assign data points to closest cluster
7:         $i^* \leftarrow \arg\min_j \left\| \mathbf{x_i} - \boldsymbol{\mu_j^{t-1}} \right\|$
8:         $C_{i^*} \leftarrow C_{i^*} \cup \mathbf{x_i}$
9:     end for
10:    for $i = 1$ to $k$ do          ▷ Find new centroids
11:        $\boldsymbol{\mu_i^t} \leftarrow \frac{1}{|C_i|} \sum_{\boldsymbol{x_j} \in C_i} \left( \mathbf{x_j} \right)$
12:    end for
13: end while
```

2. Calculating the new centroids of each cluster.

Which cluster a new atom belongs to can now be predicted, by assigning it to the cluster with the closest centroid in feature space.

In this report a version of k-means from the Scikit-Learn package [22] that initializes the centroids in a more clever way was used [23].

3.2.3 Ridge Regression

This section builds on ideas from [1]. In order to assign energies to the different clusters, a form of linear regression on a new global feature vector is used. This feature vector $\mathbf{f_G} = (n_1, ..., n_k)$ is the number of atoms n_i in a structure that belongs to each cluster C_i from the k-means algorithm. The energy of a structure is now divided into local energies:

$$E(\mathbf{r_1}, ..., \mathbf{r_A}) = \sum_{i=1}^{k} n_i \epsilon_i \tag{3.9}$$

here ϵ_i is the local energy of C_i. In words the total energy consists of the sum of the energy of each atom belonging to one of the k clusters. The reader might object that this isn't physically feasible in quantum mechanics. The total energy of the system is given as the eigenvalue of the Hamiltonian by the time-independent Schrödinger equation [24, p. 27]. But the Hamiltonian of the system contains the potential energy $V(\mathbf{r_1}, ..., \mathbf{r_N})$ which is not a simple sum of the individual potentials because the particles interact with each other [25]. So one cannot divide the Hamiltonian of a system into it's constituent parts in general and doing so is an approximation. In order to find the energies ϵ_i, some known examples of n_i's and the corresponding energy must be used and then the ϵ_i's that fit all the structures best can be found. This can be done by establishing a system of equations:

$$\begin{bmatrix} n_{11} & \dots & n_{1k} \\ \vdots & \ddots & \vdots \\ n_{N1} & \dots & n_{Nk} \end{bmatrix} \begin{bmatrix} \epsilon_1 \\ \vdots \\ \epsilon_k \end{bmatrix} = \begin{bmatrix} E_1 \\ \vdots \\ E_N \end{bmatrix} \tag{3.10}$$

Here N is the number of structures used. To find an approximate "solution" to this system, linear regression is used. Let $\mathbf{X}$ be the data-matrix of n_{ij}'s, $\mathbf{w}$ be the column-vector of ϵ_i's and $\mathbf{y}$ be the column-vector of E_i's. The system now becomes

$$\mathbf{Xw} = \mathbf{y} \tag{3.11}$$

the letter $\mathbf{w}$ here signifies weights. Although very well-known in statistics, linear regression is actually a machine learning technique [14, p. 82]. It is a classic case of supervised learning as introduced in section 3.1.1 because a model of how to predict on future data using data already seen is learned. An approximate solution can be found by finding the $\mathbf{w}$ that minimizes

$$\mathbf{w_{lin}} = \arg\min_{\mathbf{w}} \|\mathbf{Xw} - \mathbf{y}\|^2 \tag{3.12}$$

However, because $\mathbf{X}^{\mathrm{T}}\mathbf{X}$ might be singular [1], meaning it is not invertible [26], ridge regression is used instead. Aside from being able to deal with singular a $\mathbf{X}^{\mathrm{T}}\mathbf{X}$ it regularizes the learned $\mathbf{w}$ to prevent overfitting as described in section 3.1.4 [27][28]. This means that the weights are punished for being large. The ridge regression is identical to the linear except for the regularization term

$$\mathbf{w_{ridge}} = \arg\min_{\mathbf{w}} \left(\|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2 + \lambda \|\mathbf{w}\|^2 \right) \tag{3.13}$$

The solution offered without proof can be written as [1]

$$\mathbf{w_{ridge}} = (\mathbf{X}^{\mathrm{T}}\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{y} \tag{3.14}$$

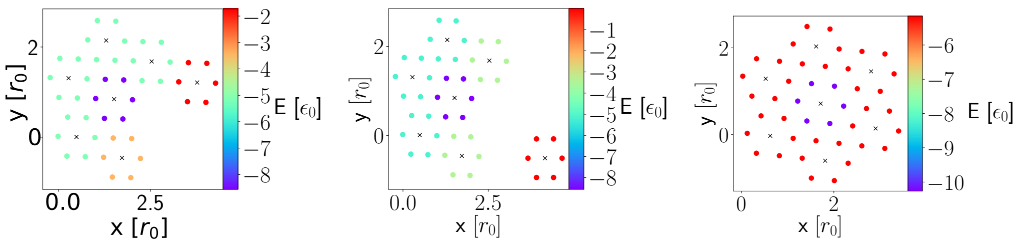
where λ is the regularization parameter and $\mathbf{I}$ is the identity matrix. The bigger λ is the bigger the punishment for choosing large weights. In this report a value of $\lambda = 0.3$ was used.

3.3 Global Minimum of Hexamer Structure Using Machine Learning

Fully equipped the problem of structure search using machine learning can now be revisited. As hinted in section 2.3 a way to improve the basin-hopping algorithm is to make a clever choice of which molecule(s) to rattle. An example of such a choice can be seen in figure 3.3a-3.3b. Here the most unstable molecule is rattled to a location that finds the global minimum after relaxing. Even though the location of the molecule after rattling is lucky a bad location could have been corrected within a couple of iterations.

Because the Lennard-Jones potential is used in this report it is possible to use it to find the exact local energies. In fact using the energy information is the most effective strategy using the *Lennard-Jones potential*, but it is not a realistic potential and hence not applicable to find structures in the real world. Here one has to apply "Density Functional Theory" (DFT) calculations where the real Hamiltonian is solved. These calculations are extremely costly and it can take several hours to calculate the *total* energy of one structure [29]. Furthermore, DFT doesn't give local energies.

The basic structure of the basin-hopping algorithm is identical to the stochastic case seen in figure 2.13 except for a different rattling strategy. The basic idea comes from the article [1]. During iteration the energy information and the Behler-Parrinello features of the structure are saved so they can be used to train the machine learning model. Hence, the model is trained at each iteration or *on the fly* and becomes better and better at predicting the local energies with each iteration. Pseudocode of the machine learning rattling can be seen in algorithm 2 and the complete code can be seen in appendix A.3.4 and A.3.6. Training amounts to dividing the feature space into the clusters using k-means and assigning energies to the clusters using ridge regression. Then each atom in the current structure is assigned a cluster and the predicted energies of each atom can then be used to choose the molecule(s) with the highest energy. As shown in the pseudocode the current structure should not be used for training the model. This is because in general one shouldn't use the data one is trying to predict for training the model. This might lead to overfitting and hence decrease the predictive power of the model [30]. However, because of the numerous data points (many atoms and energies) it *was* used in the training. Another reason that this is acceptable is that the goal is not to predict new total energies but the local energies themselves.



(a) Before rattling. (b) After rattling. (c) Global minimum found.

Figure 3.3: An example of the last step of the machine learning basin-hopping algorithm before finding the global minimum for the hexamer structure. Figures a)-c) show the molecular energies as calculated by the Lennard-Jones potential marked by colour.

In figure 3.4 one can see a typical graph of how well the local energies

Algorithm 2 This algorithm describes coarsly how the machine learning algorithm trains and chooses which molecule to rattle at every iteration.
s_i is the relaxed current structure.

```
1: Transform Cartesian coordinates of the atoms of $s_i$ to local feature vector as explained in section 3.2.1 and add to feature matrix of all encountered structures.
2: Perform k-means on the feature matrix excluding $s_i$.
3: Transform the clustering to the global feature vector $\mathbf{f_G}$ for each structure and gather them in a matrix $\mathbf{X}$ as in equation (3.10).
4: Add energy of $s_i$ to a vector $\mathbf{y}$ as in equation (3.10).
5: Solve $\mathbf{Xw} = \mathbf{y}$ using ridge regression excluding data from $s_i$.
6: Use centroids of k-means clusters to assign each atom in $s_i$ to the cluster they are closest to.
7: Make list of atom energies using $\mathbf{w}$ and the clustering.
8: Transform it into a list of molecule energies by summing the energies of atoms in a molecule.
9: Choose the molecules with the largest energy to rattle.
10: Rattle as usual.
```

were predicted. The local energies were calculated using the Lennard-Jones energy atom-wise as in appendix A.1.3 and were then summed to obtain the local molecule energies. If the energy prediction was perfect the data would lie on top of the red identity line. So the prediction is not perfect but decent. On the figure one can see that the Lennard-Jones energies of the molecules gather mostly at 4-5 different groups with a spread on the predicted energy. This could possibly indicate that our model is overfitting and is predicting too many different energy groups. Even though not perfect this precision turns out to be enough to significantly improve the basin-hopping algorithm as one can see in figure 3.5. The figure shows the success rate of the machine learning basin-hopping algorithm using different features and number of cluster. As one can see the best strategy seems to be with 10 clusters and using the $\mathbf{f_2}$ feature from 3.1.

A comparison of the success rate of different rattling strategies can be seen on figure 3.6. While the stochastic rattling finds the global minimum by pure chance as in figure 2.14, the machine learning model *sometimes* rattles a stable molecule but mostly unstable molecules as in figures 3.3. The degree to which it performs the correct choice of rattling the most unstable molecules depends on how well it can predict the local energies. If the Lennard-Jones energy information itself is utilized to choose the molecules, the choice will always be the most unstable molecules which is reflected in the success rate in figure 3.6. As one can see the machine learning on the fly improves the algorithm significantly but not overwhelmingly. It seems to only start to make a difference around iteration 25 where the difference between the machine learning and the stochastic rattling choice increases. This is probably because much data is needed to make a good model as is further corroborated by the graph of training before starting iterations. Here the model was trained on 100 structures before entering the loop and the resulting improvement of the success rate is dramatic.

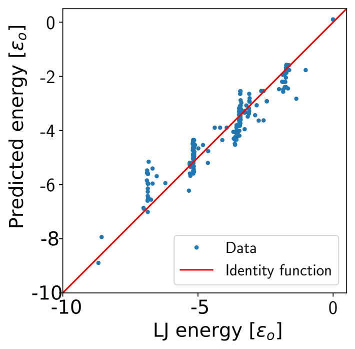


Figure 3.4: Prediction of the energy of molecules vs Lennard-Jones energies using 10 clusters and the $\mathbf{f_2}$ feature from table 3.1. The graph of the identity function is shown for comparison. Created using 200 structures for training and 50 different structures for testing.

Overall the algorithm seems to be significantly improved by using machine learning. Although in the case of the Lennard-Jones potential this is not impressive it is still a big improvement if the potential was calculated us-

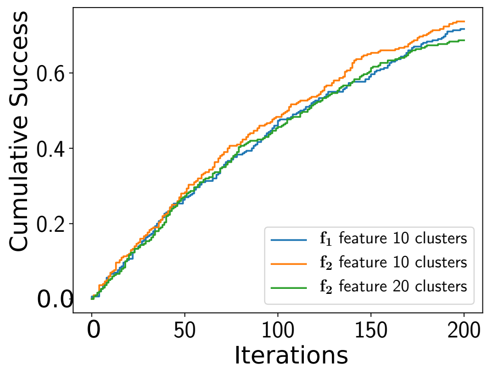


Figure 3.5: Comparison of the success rate of the machine learning basin-hopping algorithm using different features and clusters. The parameters of $\mathbf{f_1}$ and $\mathbf{f_2}$ can be seen in table 3.1.

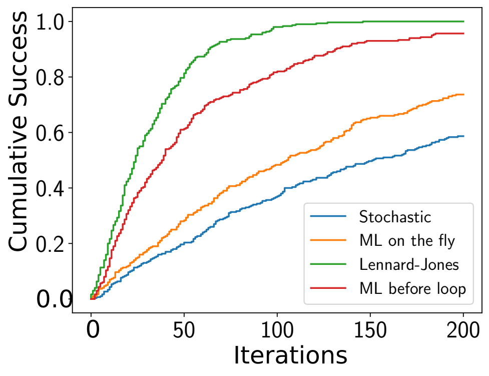


Figure 3.6: Comparison of the success rate of different rattling strategies. ML stands for machine learning and is shown for training "on the fly" and before the basin-hopping algorithm respectively. The machine learning feature used was $\mathbf{f_2}$ from 3.1 with 10 clusters.

ing DFT calculations. For further improvement of the algorithm one could change the number of clusters or the local feature vectors used.

3.4 Conclusion

In this report a global description of molecules was constructed by clustering local atom-wise feature vectors. From this description fictitious local energies were extracted using ridge regression and utilized in a basin-hopping algorithm to choose the least stable molecules to perturb. Using these techniques resulted in an improvement of the success rate of the algorithm from 59% to 73%. These numbers show that machine learning techniques are capable of enhancing global minimum search.

In order to test the techniques in the real world one would have to use DFT calculations instead of the Lennard-Jones potential. Even though the structures here are only 2D the techniques used could be extended to real surface or even bulk structures. In principle any molecule in 3D could be easily build by the techniques used here as long as they are held fixed with respect to a center.

For further studies a more in-depth parameter search could be performed. This includes among other things the rattling method, feature vectors, number of clusters in k-means and regularization used in ridge regression. Especially finding an expression for an analytical gradient of the potential would greatly increase the speed of the algorithm. The feature vectors used were atom-wise and it would be interesting to develop a special molecule feature utilizing some of the special characteristics of molecules as opposed to atoms.

Bibliography

[1]S. A. Meldgaard et al, "An introductory example of machine learning enhanced global optimization", unpublished.

[2]Wikipedia contributors, *Broyden–fletcher–goldfarb–shanno algorithm*, `https://en.wikipedia.org/wiki/Broyden-Fletcher-Goldfarb-Shanno_algorithm`, [Online; accessed 2-June-2018], 2018.

[3]D. Schroeder, *An introduction to thermal physics* (Pearson Education Limited, Harlow, 2014).

[4]Wikipedia contributors, *Absolute zero — Wikipedia, the free encyclopedia*, `https://en.wikipedia.org/w/index.php?title=Absolute_zero&oldid=843303063`, [Online; accessed 2-June-2018], 2018.

[5]Wikipedia contributors, *Metal–organic framework — Wikipedia, the free encyclopedia*, `https://en.wikipedia.org/wiki/Metalorganicframework`, [Online; accessed 10-June-2018], 2018.

[6]K. Krämer, *Mofs: metal–organic frameworks*, `https://www.chemistryworld.com/podcasts/mofs-metalorganic-frameworks-/3007204.article`, [Online; accessed 10-June-2018], 2017.

[7]Wikipedia, *Mof-5 — wikipedia, die freie enzyklopädie*, [Online; accessed 11. Juni 2018], 2018.

[8]F. Bebensee, K. Svane, C. Bombis, F. Masini, S. Klyatskaya, F. Besenbacher, M. Ruben, B. Hammer, and T. R. Linderoth, "A surface coordination network based on copper adatom trimers", Angewandte Chemie International Edition **53**, 12955–12959 (2014).

[9]Wikipedia contributors, *Oligomer — Wikipedia, the free encyclopedia*, `https://en.wikipedia.org/wiki/Oligomer`, [Online; accessed 29-May-2018], 2018.

[10]Wikipedia contributors, *Potential energy surface — Wikipedia, the free encyclopedia*, `https://en.wikipedia.org/w/index.php?title=Potential_energy_surface&oldid=826803282`, [Online; accessed 14-June-2018], 2018.

[11]E. Jones, T. Oliphant, P. Peterson, et al., *SciPy: open source scientific tools for Python*, 2001–.

[12]D. J. Wales and J. P. K. Doye, "Global optimization by basin-hopping and the lowest energy structures of lennard-jones clusters containing up to 110 atoms", The Journal of Physical Chemistry A **101**, 5111–5116 (1997).

[13]N. J. Nilsson, *The quest for artificial intelligence: a history of ideas and achievements* (Cambridge University Press, New York, 2010).

[14]Y. S. Abu-Mostafa, M. Magdon-Ismail, and H.-T. Lin, *Learning from data - a short course* (AMLBook, 2012).

[15]A. Ng, *Sparse autoencoder*, Stanford course CS294A Lecture notes, 2011.

[16]Wikipedia contributors, *Sampling (statistics) — Wikipedia, the free encyclopedia*, `https://en.wikipedia.org/w/index.php?title=Sampling_(statistics)&oldid=840595602`, [Online; accessed 25-May-2018], 2018.

[17]A. Boggess and F. J. Narcowich, *A first course in wavelets with fourier analysis* (John Wiley & Sons, New Jersey, 2009).

[18]D. V. Fedorov, *Yet another introduction to numerical methods, version 14.04* (GNU General Public License 3.0, 2013).

[19]J. Behler and M. Parrinello, "Generalized neural-network representation of high-dimensional potential-energy surfaces", Phys. Rev. Lett. **98** (2007).

[20]A. Khorshidi and A. A. Peterson, "Amp: a modular approach to machine learning in atomistic simulations", Computer Physics Communications **207**, 310 –324 (2016).

[21]M. J. Zaki and W. Meira Jr., *Data mining and analysis: fundamental concepts and algorithms* (Cambridge University Press, New York, 2014).

[22]F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: machine learning in Python", Journal of Machine Learning Research **12**, 2825–2830 (2011).

[23]D. Arthur and S. Vassilvitskii, "K-means++: the advantages of careful seeding", **8**, 1027–1035 (2007).

[24]D. J. Griffith, *Introduction to quantum mechanics* (Pearson Education, Inc., Harlow, 2014).

[25]Wikipedia contributors, *Hamiltonian (quantum mechanics) — Wikipedia, the free encyclopedia*, `https://en.wikipedia.org/w/index.php?title=Hamiltonian_(quantum_mechanics)&oldid=837300265`, [Online; accessed 25-May-2018], 2018.

[26]E. W. Weisstein, *"singular matrix." from mathworld–a wolfram web resource.* `http://mathworld.wolfram.com/SingularMatrix.html`, [Online; accessed 25-may-2018].

[27]A. Jain, *A complete tutorial on ridge and lasso regression in python*, `https://www.analyticsvidhya.com/blog/2016/01/complete-tutorial-ridge-lasso-regression-python/`, [Online; accessed 25-may-2018].

[28]T. P. S. University, *5.1 - ridge regression*, `https://onlinecourses.science.psu.edu/stat857/node/155/`, [Online; accessed 25-may-2018].

[29]E. L. Kolsbjerg, A. A. Peterson, and B. Hammer, "Neural-network-enhanced evolutionary algorithm applied to supported metal nanoparticles", Phys. Rev. B **97**, 195424 (2018).

[30]Brownlee, Jason, *A simple intuition for overfitting, or why testing on training data is a bad idea*, `https://machinelearningmastery.com/a-simple-intuition-for-overfitting/`, [Online; accessed 7-June-2018], 2014.

Appendices

A
Python Codes

Here are the essential parts of the numeric code used in this report. It is by no means exhaustive but contains enough to perform the basin-hopping algorithm for Lennard-Jones, machine learning and stochastic rattling strategies. To understand this part fully one has to have a basic knowledge of python and the NumPy syntax, but an undergraduate in physics should be sufficient to understand most of it as the code is completely commented.

The codes will be presented in a specific order: Important general functions (and classes), helper-functions and finally the basin-hopping algorithm code for machine learning, stochastic and Lennard-Jones rattling respectively. The code should be fully functioning with minor tweaks. If large chunks of code are repeated in another function the code will just be referred to with the relevant changes. The interesting parts are the basin-hopping code and the important functions although to a lesser degree. The helper functions have been included for completeness but feel free to skip them.

Throughout the code some conventions are used:

- The individual atoms are "contained" within the specific molecule object from the respective molecule class.
- A structure is a python list containing molecule objects.
- A "structures" object is a python list containing "structure" objects.
- CM means center of mass.

- The center of mass and angle coordinates of a structure is called its coordinates or variables.
- ”typelet” is an integer representing the number of atoms in the respective molecule. It comes from duplet, triplet, etc.

The packages used and their abbreviations in the code are:

- numpy as np.
- math.
- scipy.optimize.minimize as minimize.
- sklearn.cluster.KMeans as KMeans.
- random.
- matplotlib.pyplot as plt.

Aside from packages a few field variables were defined in the beginning of the code and used throughout the code unless changed in specific sections. These were with their typical values.

-
  ```
  number_of_structures = 200
  ```

-
  ```
  atoms_per_structure = 42
  ```

-
  ```
  number_of_clusters = 10
  ```

-
  ```
  basin_hop_iterations = 200
  ```

-
  ```
  test_size = 100
  ```

A.1 General Functions

A.1.1 Molecule Hexamer Class

The other molecule classes were made with identical code except the typelet constant.

```python
'''
Class Hexamer representing a cyclic molecule of six atoms with radius
0.5. It holds just the x and y coordinates of the molecule, the angle
from the horizontal and a label.
'''
class Hexamer(object):
    # radius from center of mass
    radius = 0.5
    typelet = 6

    # initialization of molecule with center of mass coordinates,
    # angle and optionally an integer as a label
    def __init__(self, CM_x, CM_y, angle, label=0):
        self.label = label
        self.CM_x = CM_x
        self.CM_y = CM_y
        self.angle = angle

    # Function for Getting the cartesian coordinates of all atoms in a
    # matrix
    def get_coordinates(self):
        # define center of mass
        CM = np.array([self.CM_x, self.CM_y])
        # return in a numpy array as x/y columns. This is done by
        # divididing 2pi into the angles that equally spaced atoms of
        # the type, typelet, would have.
        coordinates = np.zeros((self. typelet,2))
        for a in range(self.typelet):
            a_coord = np.array(
                [self.radius*np.cos(self.angle+a*2*np.pi/self.typelet),
                    self.radius*np.sin(self.angle+a*2*np.pi/self.typelet)])
                coordinates[a] = CM + a_coord
            return coordinates

    # Set new coordinates
    def set_coordinates(self, CM_x, CM_y, angle):
        self.CM_x = CM_x
        self.CM_y = CM_y
        self.angle = angle

    # Rotate molecule by updating degree parameter.
    def rotate(self, new_angle):
        self.angle = angle + new_angle

    # Plot itself using pyplot
    def plot_molecule(self):
        coordinates = self.get_coordinates()
        plt.plot(coordinates[:, 0], coordinates[:, 1],'b.')
        plt.plot(self.center_of_mass[0], self.center_of_mass[1], 'bx')
        plt.xlim(-5, 5)
        plt.ylim(-5, 5)
        plt.axis('equal')
        plt.show()
```

A.1.2 Lennard-Jones Total Potential

```
'''
Input: list of coordinates/variables in the form of [CMx1,...,CMxn,CMy1,
...,CMyn,angle1,...,angle_n] where n is the number of molecules.
Output: Value of the total potential using Lennard-Jones potential of
individual atoms and excluding the contribution from pairs of atoms in
the same molecule.
'''
def potential(variables, typelet):
    number_molecules = int(len(variables)/3)
    # splitting the variables in the three types of parameters per
    # molecule
    CM_x = np.array(variables[0:number_molecules])
    CM_y = np.array(variables[number_molecules:2 * number_molecules])
    angles = variables[number_molecules * 2:number_molecules * 3]
    # radius of the molecules used
    radius = 0.5
    # Putting the CM coordinates of the molecules into one matrix like:
    #  x|y
    CM = np.transpose(np.array([CM_x, CM_y]))
    #finding the contribution of all pairs of atoms in the structure
    # excluding the ones on the same molecule.
    energy = 0
    # for each molecule
    for i in range(number_molecules):
        # get the type of molecule i
        i_type = typelet[i]
        # with respect to each other molecule j
        for j in range(number_molecules):
            # check that molecule i and j are different
            if (i != j):
                # get the type of molecule j
                j_type = typelet[j]
                # within molecule i look at atom a
                for a in range(0, i_type):
                    # the coordinate of a in the molecule frame
                    a_coord = np.array([radius*np.cos(angles[i] +
                                                  a*2*np.pi/i_type),
                                       radius*np.sin(angles[i] +
                                                  a*2*np.pi/i_type)])
                    # the coordinate of atom a in the normal frame
                    # adding the CM coordinates of molecule i
                    R_a = CM[i] + a_coord
                    # within molecule j look at atom b
                    for b in range(j_type):
                        # coordinates of atom b in the frame of
                        # molecule j.
                        b_coord = np.array([radius*np.cos(angles[j] +
48                                                      b*2*np.pi/j_type),
49                                              radius*np.sin(angles[j] +
50                                                      b*2*np.pi/j_type)])
51                          # coordinate of atom b in the normal frame
52                          R_b = CM[j] + b_coord
53                          # vector between atom a and b
54                          R_ab = R_a - R_b
55                          # finding the L2 norm/distance between a and b
56                          r_ab = np.linalg.norm(R_ab, ord=2)
57                          # add Lennard-Jones potential between them
58                          # with the minimum of the potential at r_m=0.5
59                          rm=0.5
60                          energy += (rm/ r_ab)**12 - 2*(rm/r_ab)**6
61      # Because every atom was counted twice.
62      energy = energy/2
63      return energy
```

A.1.3 Lennard-Jones Local Potential

This function is very similar to the one of function A.1.2. Now the energy contributions of each atom is saved in a NumPy array. Line 22 is replaced by:

```
    local_energies = np.zeros(np.sum(typelet))
```

Line 60 by

```
                        local_energies[i*typelet + a] += (rm/r_ab)**12\
                                                 - 2*(rm/r_ab)**6
```

Line 62-63 by

```
    local_energies = local_energies/2
    return local_energies
```

The output is now a NumPy array of the local energies of each atom. Note that this function only works for hexamer molecules, but this could easily be remedied.

A.1.4 Relaxation of structure

```
'''
Input: a list of molecules and a list of integers representing the
type of each molecule
Output: the "relaxed" structure, which means the minimization of the
potential adjusting the coordinates of each molecule using the BFGS
algorithm. It is a list of molecules.
'''
def relax(structure, typelet):
    number_molecules = len(typelet)
    # get the coordinates of all molecules using the appropriate function
    coord = molecules_get_coordinates(structure)
    # Use scipy.optimize.minize function without providing a
    # jacobian. The current coordinates are supplied as startvalues
    # and the types as parameters
    result = minimize(potential, coord, typelet, method='BFGS', jac=None)
    # the coordinates are in the x parameters of the result object
    relaxed_coord = result.x
    # generalized set coordinates to each molecule applying the
    # relaxed coordinates
    for i in range(number_molecules):
        structure[i].set_coordinates(relaxed_coord[i],
                                     relaxed_coord[i+number_molecules],
                                     relaxed_coord[i+2*number_molecules])
    return structure
```

A.1.5 Behler-Parrinello Feature

```
'''
Input: numpy array with atom coordicates of size (n,2) where n is the
number of atoms. Column 0 is the x-coordinate and 1 the y-coordinate.
Optional Input: np arrays containing the different parameters
Output: numpy array with Behler-Parinello fetures of size (n,s) where
s is the dimension of the feature. The final feature is consists of
the radial and angular feature of all possible permutations of the
parameters.So in the example underneath is a 12-dimensional feature.
```

```
Although this is an atomwise feature as the normal Behler-Parinello
feature there is a slight difference in that the feature an atom
doesn't "see" other atoms in the same molecule.
'''
def behler_feature(xy,
                   rc_list=np.array([2]),
                   rs_list=np.array([0]),
                   eta_list=np.array([0.05, 2, 8]),
                   zeta_list=np.array([0.05, 0.005]),
                   lambdda_list=np.array([1])):
    # current number of structures
    n = int(len(xy) / atoms_per_structure)

    # number of different permutations of paramters and resulting
    # dimension of feature
    number_of_combinations = len(eta_list) * len(zeta_list) *    \
                             len(lambdda_list) * len(rc_list) *  \
                             len(rs_list)
    number_of_features = number_of_combinations * 2

    # Scenario matrix of type (rc|rs|eta|zeta|lambda) for different
    # combinations
    scenarios = np.zeros((number_of_combinations, 5))
    i = 0
    for a in range(len(rc_list)):
        for b in range(len(rs_list)):
            for c in range(len(eta_list)):
                for d in range(len(zeta_list)):
                    for e in range(len(lambdda_list)):
                        scenarios[i] = np.array([rc_list[a], rs_list[b]\
                        ,eta_list[c], zeta_list[d],lambdda_list[e]])
                        i += 1

    # initialize huge feature matrix like (f1_1|f2_1|f1_2|f2_2|....)
    # for each atom and scenario where f1_1 means
    # radial feature scenario 1, f2_2 means angular feature scenario 2
    feature = np.zeros((len(xy), number_of_features))

    # for all structures
    for x in range(n):
        # for all different scenarios
        for s in range(number_of_combinations):
            # for all atoms in the structure x
            for i in range(atoms_per_structure):
                G_i_rad = 0
                G_i_ang = 0
                molecule_start = i - (i % 6)
                # for all atoms in structure x
                for j in range(atoms_per_structure):
                    # if i and j are different atoms continue
                    if (j<molecule_start or j>molecule_start+5):
                        # find the Euclidean distance between the points
                        r_ij =np.linalg.norm(xy[x*atoms_per_structure+i]\
                              - xy[x * atoms_per_structure + j])
                        # Add to the sum of the radial feature.  The
                        # cutoff function f_c is placed seperately below
                        G_i_rad +=np.exp(-scenarios[s, 2] * (r_ij -
                        scenarios[s,1])**2)*f_c(r_ij,scenarios[s, 0])
                        # for all other atoms in structure x
                        for k in range(atoms_per_structure):
                            # check furthermore that atom k is not part
                            #  of the same molecule as i and is not j
                            if ((k<molecule_start or k>molecule_start+5)
                                    and k != j):
                                # find the vectors between the
                                # different  atoms
                                R_ij = xy[x * atoms_per_structure + i] \
                                       -  xy[x * atoms_per_structure + j]
                                R_ik = xy[x * atoms_per_structure + i] \
                                       -  [x * atoms_per_structure + k]
                                R_jk = xy[x * atoms_per_structure + j] \
                                       -  [x * atoms_per_structure + k]
                                # find the angle between the atoms
                                # centered on atom i
                                theta = np.dot(R_ij, R_ik) /  (
                                np.linalg.norm(R_ij)*np.linalg.norm(R_ik))
                                # add this to the angular feature for
                                # atom i.
                                G_i_ang += (1 + scenarios[s,4]*
                                np.cos(theta)) ** scenarios[s, 3] * \
                                np.exp(-scenarios[s, 3] *    \
                                (np.linalg.norm(R_ij) ** 2 +
                                 np.linalg.norm(R_ik) ** 2 +
                                 np.linalg.norm(R_jk) ** 2)) * \
                                f_c(np.linalg.norm(R_ij),scenarios[s,0])*\
                                f_c(np.linalg.norm(R_ik),scenarios[s,0])*\
                                f_c(np.linalg.norm(R_jk),scenarios[s,0])
                # add the radial and the angular feature to the feature
                # array
                feature[x * atoms_per_structure + i, s * 2 + 0] = G_i_rad
                feature[x * atoms_per_structure + i, s * 2 + 1] = G_i_ang

    return feature
'''
Input: distance between atoms r and j, cuttoff radius.
Output: cuttoff function value
'''
def f_c(r_ij, rc):
    if (r_ij <= rc):
        result = 0.5 * (np.cos(np.pi * r_ij / rc) + 1)
        return result
    else:
        return 0
```

A.1.6 Ridge Regression

```
'''
Input: data numpy array X of shape (n,m) where n is the number of
structures and m is the dimension of the feature vector and
corresponding energy numpy array of shape n
```

```
Output: numpy array of shape m containing the energy of each cluster
(the weights)
'''
def ridge_regression(X, y):
    n, m = X.shape
    I = np.identity(m)
    # The regularization parameter
    lambdda = 0.3
    # prevent singular matrix by using ridge regression
    X_cross = np.matmul(np.linalg.inv(np.matmul(np.transpose(X), X)
                                   + I * lambdda), np.transpose(X))
    # finding the weights/energies by using numpy's matrix multiplication
    w_lin = np.matmul(X_cross, y)
    return w_lin
```

A.1.7 Initialize Structure

Works for dimer, trimer, tetramer and hexamer molecules. Could easily be extended.

```
'''
Input: integer list representing the molecules, 2=Dimer, 3=trimer, etc.
Output: list of randomly initialized molecules within a 4x4 square
centered at the origin
'''
def initialize(typelet):
    number_molecules = len(typelet)
    # initialize the center of mass at random values random values from
    # within the 4*4 square centered at the origin
    CM_x = np.zeros(number_molecules)
    CM_y = np.zeros(number_molecules)
    CM = np.zeros((number_molecules, 2))
    # for every molecule
    for i in range(number_molecules):
        # repeat until a location and orientation is found where none
        # of the center of masses are within 1.2 distance of the others.
        # this prevents overlapping molecules
        a = True
        while (a):
            a = False
            CM[i] = np.random.rand(2) * 8 - 4
            # check if it overlaps with some of the atoms so far
            for j in range(i):
                if (np.linalg.norm(CM[i] - CM[j]) < 1.2):
                    a = True
        CM_x[i] = CM[i, 0]
        CM_y[i] = CM[i, 1]
    # initialize the angle of each molecule
    angles = np.transpose((np.random.rand(number_molecules) * np.pi*2))
    # fill up the list of molecules by initializing each molecule with
    #  it's type, parameters and a particular label i
    molecules = []
    for i in range(number_molecules):
        if (typelet[i] == 2):
            molecules.append(Dimer(CM_x[i], CM_y[i], angles[i], i))
        if (typelet[i] == 3):
            molecules.append(Trimer(CM_x[i], CM_y[i], angles[i], i))
        if (typelet[i] == 4):
            molecules.append(Tetramer(CM_x[i], CM_y[i], angles[i], i))
        if (typelet[i] == 6):
            molecules.append(Hexamer(CM_x[i], CM_y[i], angles[i], i))
    return molecules
```

A.1.8 Plot Structure

```
'''
Input: list of molecules (a structure), string to give the plot and
file a title.
This function uses dictionaries to make colours for the molecules and
moves it's axis to zoom in on the structure wherever it might be in
the plane.
'''
def plot_molecules(molecules, title):
    number_molecules = len(molecules)
    # translate the label of each molecule which is and integer to a
    # color code to distinguish different molecules using a
    # dictionary structure
    color_dict = {0: 'b', 1: 'g', 2: 'r', 3: 'c', 4: 'm', 5: 'y',
                  6: 'k', 7: 'w'}
    # iterate over each molecule and plot each. Define fig here so
    # that all the molecules get plottet in the same window
    fig = plt.figure()
    # for molecules in structure
    for m in molecules:
        # get m's coordinates
        coordinates = m.get_coordinates()
        # find it's colour from the dictionary
        color = color_dict[m.label]
        # plot it using the coordinates of the atoms and the colour.
        # The CM is marked with an 'x'.
        plt.plot(coordinates[:, 0], coordinates[:, 1], color + '.')
        plt.plot(m.CM_x, m.CM_y, color + 'x')
    # find the center of mass of the whole structure
    coord = molecules_get_coordinates(molecules)
    CMx = 0
    CMy = 0
    for i in range(number_molecules):
        CMx += coord[i]
        CMy += coord[number_molecules + i]
    CMx /= number_molecules
    CMy /= number_molecules
    # plot and save the structure with axis that are equal and follow
    # the center of mass of the structure.
    plt.axis('equal')
    plt.axis([CMx - 10, CMx + 10, CMy - 10, CMy + 10])
    plt.title(title)
    plt.show()
```

A.2 Helper Functions

A.2.1 Get Molecule Coordinates of Structure

```python
'''
Input: list of molecules
Output: list of coordinates of the molecules as [CMx1,...,CMxn,CMy1,
...,CMyn,angle1,...,angle_n] where n is the number of molecules.
'''
def molecules_get_coordinates(molecules):
    number_molecules = len(molecules)
    # Add the cordinates by using the field variable values of the
    # molecule class
    variables = [a * 0 for a in range(3 * number_molecules)]
    for i in range(number_molecules):
        variables[i] = molecules[i].CM_x
        variables[i + number_molecules] = molecules[i].CM_y
        variables[i + number_molecules * 2] = molecules[i].angle
    return variables
```

A.2.2 Get Molecule Coordinates of a List of Structures

```python
'''
Input: list of structures (each element being a list of molecules),
integer list representing types of molecules in structure
Output: numpy array of size (n,m) where n is the number of structures
and m is the number of coordinates for each structure. The array
contains the molecule coordinates of all structures.
'''
def structures_get_molecule_coord(structures, typelet):
    number_structures = len(structures)
    number_molecules = len(typelet)
    number_coord = number_molecules * 3
    # for each structure get it's molecules coordinates
    coord_np = np.zeros((number_structures, number_coord))
    for i in range(number_structures):
        coord_np[i] = molecules_get_coordinates(structures[i])
    return coord_np
```

A.2.3 Get Atom Coordinates of a List of Structures

```
'''
Input: list of structures (each element being a list of molecules),
integer list representing types of molecules in structure
Output: numpy array of size (n,2) where n is the total number of
atoms in all the structures. column 0 is the x-coordinate, column 1
is the y-coordinate.
'''
def get_atom_coord_structures(structures, typelet):
    number_structures = len(structures)
    number_molecules = len(typelet)
    # get the coordinates of all the molecules of every structure
    coord_molecules = structures_get_molecule_coord(structures, typelet)
    # np array for all the atoms of the structures. x/y for each atom
    xy = np.zeros((number_structures * np.sum(typelet), 2))
    current_atom = 0
    # for all structures
    for m in range(number_structures):
        # the coordinates of structure m
        variables = coord_molecules[m]
        # splitting the variables in the three types of parameters per
        # molecule
        CM_x = np.array(variables[0:number_molecules])
        CM_y = np.array(variables[number_molecules:2 * number_molecules])
        angles = variables[number_molecules * 2:number_molecules * 3]
        # the radius from the center of mass to the atoms used
        radius = 0.5
        # Putting the CM coordinates of structure m into one matrix like:
        #  x/y
        CM = np.transpose(np.array([CM_x, CM_y]))
        # for each molecule in m
        for i in range(number_molecules):
            # which type is molecule i?
            i_type = typelet[i]
            # within molecule i look at atom a
            for a in range(i_type):
                # coordinates of atom a in molecule i's frame
                a_coord = np.array([radius*np.cos(angles[i] + a*2*np.pi
                                                  /i_type),
                                    radius*np.sin(angles[i] + a*2*np.pi
                                                  /i_type)])
                # coordinates in the normal frame
                R_a = CM[i] + a_coord
                # add the coordinate to xy
                xy[current_atom] = R_a
                current_atom += 1
    return xy
```

A.2.4 Initialize List of Structures

```
'''
Input: list of integers describing the structures for initialization
Output: list of initialized structures. Each element of the list is a
list of initialized molecules using the list of integers. So the list
of structures in the end contain the same structures all initialized
randomly using the initialize function.
'''
def initialize_structures(typelet):
    list_of_structures = []
    for i in range(number_of_structures):
        # append new structure to the list
        list_of_structures.append(initialize(typelet))
    return list_of_structures
```

A.2.5 Relax List of Structures

```
'''
Input: list of structures (each element being a list of molecules),
integer list representing types of molecules in structure
```

```
Output: list of relaxed structures.
'''
def relax_structures(structures, typelet):
    number_structures = len(structures)
    relaxed_structures = []
    for i in range(number_structures):
        relaxed_structures.append(relax(structures[i], typelet))
    return relaxed_structures
```

A.2.6 Energy of Each Structure in a List of Structures

```
'''
Input: list of structures (each element being a list of molecules),
integer list representing types of molecules in structure
Output: numpy array of total Lennard-Jones energy of each structure
excluding atoms in the same molecule of the atom.
'''
def structures_energy(structures, typelet):
    number_structures = len(structures)
    # get the coordinates of each structure using the appropriate
    # function
    coord_structures = structures_get_molecule_coord(structures, typelet)
    # find the potential of each structure
    energy_structures = np.zeros(number_structures)
    for i in range(number_structures):
        energy_structures[i] = potential(coord_structures[i], typelet)
    return energy_structures
```

A.3 Basin-Hopping algorithm

A.3.1 General Rattle Function

Explanation for the machine learning rattle-function. The stochastic and Lennard-Jones version don't have the arguments ”w_lin_behler_train”, the predicted energies of each cluster, and the Kmeans-object ”kmeans_behler_.

```python
'''
Input: list of molecules (a structure), integer list representing
types of molecules in structure, the weights/energies as learned by
the ridge regression, the kmeans-result of the feature vector.
Output: rattled structure where the number of molecules rattled is
decided by the distribution below and the molecule chosen for rattling
is the most unstable (highest energy) according to the energy prediction.
'''
def rattle(structure,typelet,w_lin_behler_train,kmeans_behler_train):
    number_molecules = len(structure)
    number_atoms = np.sum(typelet)
    # random real number from 0 to 1 using the random package
    choose_rattle = random.random()
    # choosing the number of molecules to rattle using the random number
    # if the number is between 1 and 0.5^3 number to rattle is 1,
    # between 0.5^3 to 0.5^5 it is 2 etc.
    number_rattle=1
    if (choose_rattle > 0.5 ** 3):
        number_rattle = 1
    if (choose_rattle <= 0.5 ** 3 and choose_rattle >= 0.5 ** 5):
        number_rattle = 2
    if (choose_rattle <= 0.5 ** 5 and choose_rattle >= 0.5 ** 7):
        number_rattle = 3
    if (choose_rattle <= 0.5 ** 7 and choose_rattle >= 0.5 ** 9):
        number_rattle = 4
    if (choose_rattle <= 0.5 ** 9 and choose_rattle >= 0.5 ** 11):
        number_rattle = 5
    if (choose_rattle <= 0.5 ** 11 and choose_rattle >= 0.5**13):
        number_rattle = 6
```

This middle part is specific to the different rattling strategies and decides, which molecules will be rattled and hence appear as integers in the "rattle_integers" list.

```python
    # change to only include the number of molecules to rattle
    rattle_integers = rank[:number_rattle]
    # find the center of mass of the whole structure
    CMx = 0
    CMy = 0
    for i in range(number_molecules):
        CMx += coord[i]
        CMy += coord[number_molecules + i]
    CMx /= number_molecules
    CMy /= number_molecules
    # Rattle to a safe space with proper distance to neighboring atoms
    for i in rattle_integers:
        # test that the distance from this molecule to all others is
        # at least 1.3.
        a = True
        while (a):
            a = False
            # the center of mass of the rattling molecule will be
            # passed within a disk of the radius
            r0 = random.random() * 3.5
            # it will have the random angle
            theta = random.random() * 2 * np.pi
            # find the x and the y values of the center of mass of the
            # molecule and put them in an numpy array like x/y
            CMmmx = CMx + r0 * np.cos(theta)
            CMmmy = CMy + r0 * np.sin(theta)
            CM = np.array([CMmmx, CMmmy])
            # compare to all other molecules that molecule i's center
            # of mass is not within 1.3 from their center of mass
            for j in range(number_molecules):
                CM2 = np.array([structure[j].CM_x, structure[j].CM_y])
                if (np.linalg.norm(CM - CM2) < 1.3):
                    a = True
        # add molecule to the structure
        structure[int(i)].set_coordinates(CMmmx, CMmmy, theta)
    return structure
```

A.3.2 Stochastic Rattling Choice

```python
# make list of random integers of length number_rattle where no
# number is used twice
rattle_integers = []
# for number of molecules to be rattled
for i in range(number_rattle):
    # make sure it's not in the list already
    a = False
    while (not a):
        # use randint function to get a random integer
        random_int = random.randint(0, number_molecules - 1)
        if (random_int not in rattle_integers and random_int < 6):
            rattle_integers.append(random_int)
            a = True
```

A.3.3 Lennard-Jones Rattling Choice

```python
# find the lennard-jones energies of every molecule. First find
# coordinates of the molecules
coord = molecules_get_coordinates(structure)
# find the local energies of each atom
local_energies = local_potential(coord, typelet)
# transform to the total energy of the molecules
energy_molecules=np.zeros(number_molecules)
# for each molecule
for i in range(number_molecules):
    # type of molecules
    itype=typelet[i]
    # add the contribution from all atoms in the molecule
    for j in range(itype):
        energy_molecules[i]+=local_energies[i*6 + j]
# find the integers that would sort the molecules from biggest to
# smallest
rank = np.argsort(-1*energy_molecules)
```

A.3.4 Machine Learning Rattling Choice

```
# get the molecule coordinates of the structure and put it in a
# list of one structure. Then use it to find the coordinates fo
# all the atoms in the structure.
coord = molecules_get_coordinates(structure)
structures = [structure]
coord_atom = get_atom_coord_structures(structures, typelet)
# transform the structure to the behler feature
feature_pred = behler_feature(coord_atom)
# use the kmeans object to predict to which cluster each atom belongs
clustering_pred = kmeans_behler_train.predict(feature_pred)
# first reshape clustering so that each row consists of the
# assigned cluster of each atom in the structure
clustering_reshaped_pred = \
    clustering_pred.reshape(atoms_per_structure)
# changing each cluster number with their predicted local energy
atoms_energy = np.zeros(clustering_reshaped_pred.shape)
for j in range(atoms_per_structure):
    atoms_energy[j] = w_lin_behler_train[clustering_reshaped_pred[j]]
# convert the array of atom energies to an array of the energies
# of each molecule
energy_molecules = np.zeros(number_molecules)
# for each molecule
for i in range(number_molecules):
    # the type of molecule i
    itype = typelet[i]
    # for each atom in molecule i
    for j in range(itype):
        energy_molecules[i] += \
            clustering_reshaped_pred[np.sum(typelet[0:i]) + j]
# find the integers that would sort the molecules in order from
# highest to lowest energy
rank = np.argsort(-1*energy_molecules)
```

A.3.5 General Basin-Hopping Algorithm

Code used in all three parts with the extra machine learning training in line 35. This code can be seen in A.3.6. For the Lennard-Jones and stochastic rattling; line 40-41 only has the two first arguments.

```
1  '''
2  Initialize and put energy of global minimum. This global minimum is
3  found by repeating the search many times and finding the minimum
4  found in the most cases and from symmetry considerations.
5  '''
6  energy_limit = -40.5
7  typelet = np.array([6]*7)
8  #typelet = np.array([6]*2)
9  #atoms_per_structure=12
10 structure = initialize(typelet)
11 # relax the structure
12 relaxed_structure = relax(structure, typelet)
13 # iterate until the structure has reached an energy less than the
14 # limit and break if it is found within 200 iterations.
15 # variables and numpy arrays needed to enter the loop
16 integer=0
17 final_energy=0
18 coord1=np.zeros(3*len(structure))
19 coord2=np.zeros(3*len(structure))
20 coord3=np.zeros(3*len(structure))
21 coord_atoms=np.zeros((atoms_per_structure,2))
22 coord_encountered=np.zeros((basin_hop_iterations,3*len(structure)))
23 relaxed_structures_train = []
24 energies_train=np.zeros(basin_hop_iterations)
25 # finding the energy of the relaxed structure before entering the loop
26 coord1 = molecules_get_coordinates(relaxed_structure)
27 energy = potential(coord1, typelet)
28 # This is the dimension of the feature vector used. It has to be the
29 # same as the one used, here the one from the "behler_feature()" function
30 dimension_feature=12
31 feature_train=np.zeros((basin_hop_iterations*atoms_per_structure,dimension_feature))
```

```
# maximum number of iterations is 200
for i in range(basin_hop_iterations):

    ### Machine Learning Trainig code here###

    # The number of iterations (1 indexed)
    integer+=1
    # rattle the structure using the rattle function
    structure=rattle(relaxed_structure,
                        typelet,w_lin_train,kmeans_train)
    # find the coordinates after rattling
    coord2=molecules_get_coordinates(structure)
    # relax the structure
    relaxed_structure = relax(structure,typelet)
    # get the coordinates of the structure
    coord3=molecules_get_coordinates(relaxed_structure)
    # find the total Lennard-Jones energy of the structure
    energy = potential(coord3, typelet)
    # check wether the energy is below the limit, if it is the global
    # minimum is most likely found and iteration stops.
    relaxed_structures_train.append(relaxed_structure)
    if(energy<energy_limit):
        final_energy=energy
        break
#save the encountered energies and molecule coordinates only in machine
# learning algorithm
np.savetxt('structure_coordinates',coord_encountered)
np.savetxt('energies',energies_train)
# print when the global minimum was found, the coordinates before the
# final rattle, after the final rattle and after relaxing and print
# the energy of the found structure.
print(integer)
print(coord1)
print(coord2)
print(coord3)
```

A.3.6 Machine Learning Training

Training of the machine learning model resulting in "w_lin_behler_train", the predicted energies of each cluster, and the Kmeans-object "kmeans_behler_train" both used in the rattling. This training is inserted in line 35 of the general basin-hopping algorithm.

```
'''
Implementation of machine learning training code performed each
iteration in the basin hopping loop. The result is the w_lin_train
vector of the predicted energies of the clusters
'''
### Now begins the training the machine learning model part
# get the coordinates of the structure
coord1 = molecules_get_coordinates(relaxed_structure)
# Add the Lennard-Jones energy just found to a energy database
energies_train[i] = energy
# find the coordinates of the atoms of current structure
coord_atoms = get_atom_coord_structures([relaxed_structure], typelet)
# Add to a database of all encountered molecules in structure (for
#  documenting the path to the global minimum)
coord_encountered[i] = coord1
# find behler feature vector of atoms of current structure and add
#  it to a feature database for training
feature_train[
i * atoms_per_structure:atoms_per_structure * (i + 1)] = \
    behler_feature(coord_atoms)
# use the k-means++ Scikit-Learn function to cluster all of feature
# space. clustering_train now contains the assigned cluster as an
# integer for each atom
# Out model will only be updated each 5th iteration to save time

kmeans_train = KMeans(n_clusters=number_of_clusters,
                      init='random',
                      max_iter=300).fit(feature_train[
                                       0:atoms_per_structure
                                       *(i + 1)])
```

```python
clustering_train = kmeans_train.labels_
# Now we can try to predict using ridge regression on the variable of
# how many times a cluster has an atom in a certain cluster. First we
# reshape clustering so that each row is the clusters of each atom in a
# structure. The -1 option means that it adapts to whatever
# atoms_per_structure is.
clustering_reshaped_train = \
    clustering_train.reshape(-1, atoms_per_structure)
# now, the global feature vector is how many atoms belong to each
# cluster we have per structure. This is found by fill out a histogram
# row wise
cluster_histogram_train = \
    np.zeros((i + 1, number_of_clusters))
# for each structure
for j in range(i + 1):
    # find the histogram of that row resulting in the global feature
    cluster_histogram_train[j] = np.histogram(
        clustering_reshaped_train[j],
        bins=[a - 0.5 for a in range(number_of_clusters + 1)])[0]
# this is our data matrix, it has structures in each row, and number
# of atoms for that structure on each cluster in the columns. The
# energy is the labels from kmeans. We can now find the energies
# of the clusters by using the ridge_regression function
w_lin_train = ridge_regression(cluster_histogram_train,
                               energies_train[0:(i + 1)])
```