Molecular Structure Identification Using Machine Learning

Kristian Tuv

Thesis submitted for the degree of
Master in Computational Physics
60 credits

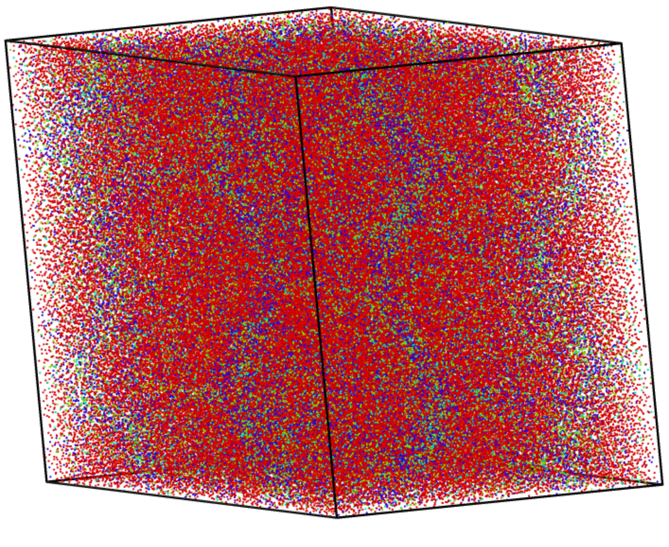
Department of Physics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2020

Molecular Structure Identification Using Machine Learning

Kristian Tuv



Contents

Acknowledgements

In my last semester of high school, my classmate Niels Bonten told me he was going to study physics at the University of Oslo. I did not really know what physics was, but after he gave a passionate explanation of the wonders of cosmology and quantum physics, I was determined that physics was the path I needed to pursue. Without that conversation, this thesis would probably never have happened. Thank you for opening my eyes.

The last six years at Blindern have been fantastic, and I want to thank everyone who has been a part of it. Especially, the communities at Lillefy and Computational Physics have been a reason for coming to Blindern even when it is raining and tempting to stay in bed.

I would like to thank friends and family for their support, and especially Ellen for your continuous deliveries of energy drinks the last two months.

Writing this thesis has been frustrating and hard work, and I would especially like to thank my supervisor, Henrik Sveinsson, for your knowledge, eye for detail and support throughout this process. For you, a problem is always an opportunity and I thank you for the continuous encouragement.

Lastly, Line, thank you for supporting me through this process and not going crazy from my mood swings. You always manage to make me feel better when everything is most bleak, and your role as sugar-mama the last three years has been paramount for keeping me well fed on steak and wine on a student budget.

Abstract

In studies of crystalline solids, identifying the intrinsic molecular structure of a material is paramount for understanding its mechanical properties. Manually evaluating the structures in a material can be an insurmountable task. Because of this, the development of automatic methods for structure identification is of great interest. Such automatic methods are often very specialized, developed for identifying a set of specific predetermined structures. This specialization is a problem if we do not know which structures exist in a material in advance, a problem we would not encounter with a generalized identification algorithm. Machine learning methods have shown impressive abilities in classification problems and have the potential for developing such generalized methods of structure identification.

The bottleneck for developing supervised machine learning methods for structure identification is the labeling of datasets required to train and evaluate such methods. In the present work, we develop an automatic method for creating a labeled dataset using unit structures transformed into adjacency matrices, representing the neighborhood topology of particles. These matrices are used as machine learning features for training and validating machine learning algorithms.

We demonstrate how the trained machine learning methods can be used for identifying structures in a dataset of self-assembling colloidal crystals [14]. Furthermore, we show that these methods are capable of competing with specialized algorithms for structure identification by applying them to simulations of methane hydrate polycrystals under stress, and comparing the results with the CHILL+ algorithm [52].

In addition to utilizing supervised learning algorithms, we test unsupervised clustering methods on datasets of methane hydrate polycrystals. We utilize adjacency matrices as machine learning features, where the matrices have first been reduced in dimensionality using principal component analysis and autoencoders. We show that these methods are viable alternatives for identifying structures without the requirement of a labeled dataset.

Chapter 1

Introduction

Crystalline solids are materials where the atom arrangements exhibit a high degree of order and periodic patterns. The physical properties of a material are largely connected to its crystal structure. The best example of this is illustrated by comparing diamond and graphite. These crystals are both composed exclusively of carbon atoms but because of the structural differences we can use graphite to write with a pencil, while diamond was long considered the hardest material on earth [79].

Identifying the structure of particle collections is important when analyzing molecular dynamics simulations. This is especially true when alyzing simulations of material failure because the failure process of materials changes the atomic stucture of the system. Thus, finding indications of structural changes in simulations of materials under stress aids the understanding of the physical properties of said material.

Identification of molecular structures can be challenging both in terms of mathematical description and computational efficiency. An additional layer of complexity is added by the thermal fluctuations of particles from their ideal crystal structure positions which sligthly alters the structure, making it harder to identify.

In this thesis, we address the challenges of finding automatic methods of classifying individual atoms in crystalline structures and develop new methods for doing so. Through machine learning we develop general methods for structure identification capable of competing with specialized algorithms. We are developing these methods with a particluar class of substances in mind: Clathrates, or more specifically, methane hydrates. Hydrates are interesting for a number of reasons, perhaps most importantly their abundance on earth [4] and their potential importance for the climate [2]. We will be using our newly developed methods to identify grain boundaries in methane hydrate polycrystals without any prior knowledge of the crystal.

In this chapter we give a brief introduction and motivation on the topics of structure identification, methane hydrates and machine learning, as well as ethical considerations regarding these subjects and the goals for this thesis.

1.1 Structure Identification

To understand the mechanical properties of cystals in general, and methane hydrates in particular, we use modern mathematical and computational techniques like *molecular dynamics*. In molecular dynamics we use known physical and mathematical properties of a system of particles to run simulations of said system to advance our understanding of its properties on a microscopic level. During the dissociation of a material, its atomic structure is necessarily going to change, altering the material properties in the process. Because of these structural changes we would ideally like to use automatic structure identification algorithms to quickly identify modifications and distortions in the material during molecular simulations. Writing specialized algorithms for structure identification requires deep insight into the physical and mathematical properties of the molecular system. Traditional methods for structure identification, such as the ones developed by Steinhardt et al. [71] and Ten Wolde et al. [78], evaluate order parameters to identify particles of a solid-like nature. An example of a method utlizing the order parameters is the CHILL+ algorithm by Nguyen and Molinero [52], which is capable of distinguishing the solid phases of water from clathrate hydrate structures. Major problems with traditional structure identification algorithms is that they are very specialized, difficult to develop, and are normally only able to recognize a few predefined structure types. Because of this specialization we have to develop new and unique algorithms for different structural compositions. The mentioned CHILL+ algorithm is for example not designed to distinguish between graphite and diamond, and to do this we would need another specialized algorithm. Ideally we would like to develop a general alogorithm capable of distinguishing between all the known molecular structures. Developing such an algorithm by hand is virtually impossible but with the help of machine learning we hope to come one step closer in doing so.

1.2 Methane Hydrates

Even though we want to develop a general algorithm for structure identification, we need to evaluate the algorithm on specific molecular structures and an interesting crystalline solid to study more closely is the *clathrate hydrate*. In addition to being scientifically interesting compounds, clathrate hydrates are very well suited for the structure identification algorithm we are going to develop. We will be utilizing the topological neighborhoods of particles in the development of the algorithm, and the very specific cage-like structure of clathrate hydrates make them topologically very distict from other molecular structures.

Clathrate hydrates are ice-like hydrogen bonded molecular structures forming cages trapping guests inside, making them so-called clathrate compounds. Because the guest molecules are mostly in gaseous states at ambient conditions, clathrate hydrates are often refered to as *gas hydrates*, or simply as *hydrates*. Clathrate hydrate research interest was propelled by the oil and gas industry after Hammerschmidt [24] discovered that hydrates were blocking transmission lines in the oil extraction process. In later years the discovery of large hydrate deposits in nature has further increased their relevance both as a potential energy source but also for their possible environmental impacts if released into the atmosphere.

When the guest molecules trapped in the cages are methane, we refer to the compound as a *methane hydrate*. Methane hydrates are typically found in the arctic seafloor and permafrost [65]. A concern regarding the abundance of methane hydrates in nature is the potential release of this highly potent greenhouse gas into the atmosphere from decomposing hydrates. There has been no scientific evidence that the slow release of methane below 200 meters of water or land reaches the atmosphere to any appreciable degree because most of the methane is oxidized or dissolved before reaching the surface [65]. Abrupt blow-outs of methane from craters of methane hydrates, on the other hand, are far more likely to reach the atmosphere. Such events have been reported both in marine and land sediments [2]. To understand the formation and release of such craters, a deeper understanding of the mechanical properties of methane hydrates on a molecular level is paramount.

1.3 Machine Learning

Artificial intelligence and machine learning has experienced tremendous growth in research and applications in recent years. As the demands for collecting and processing information keeps increasing in society, methods for structuring, extracting, and evaluating relevant data becomes essential for facilitating growth in technologial and scienfic fields. Machine learning is an application of artificial intelligence where computer systems perform specific tasks without explicit instructions on how to do them. Such algorithms have a wide array of applications and have for example been used to beat humans at chess [10], for automatic cancer detection [38] and recently for remotely finding areas of archaelogical interest from satellite images in Syria [48].

A particularly successful branch of machine learning is the *artificial neural network*, which is a flexible framework that can be applied to a series of applications, from image analysis to speech recognition. The strength of these models are that they do not need prior knowledge of the connections between observations in a dataset. Provided enough data, we just let the algorithm figure it out. The problem with this method is that a machine learning network is essentially a black box algorithm and we have very little knowledge of exactly how the algorithm evaluates the problem internally: In a famous article on skin cancer classification by Esteva et al. [17], the authors noticed after the publication that the algorithm was biased towards predicting skin cancer in images containing rulers. When dermatologists are particularly concerned about a lesion, they often add a ruler to the image to measure the lesions size. Can we then say for sure that the algorithm is actually predicting skin cancer, or has it learned to look for rulers in the image as this indicates a higher risk of cancer?

When developing machine learning algorithms for structure identification we encounter another big challenge with machine learning. The most effective algorithms require a labeling of the datasets in advance. This labeling might be done manually requiring days, maybe even months of work before the machine learning algorithm can even be tested. This is a challenge for applying these types of algorithms to molecular structure identification problems. Going through datasets of possibly millions of particles, labeling each of them by their affiliated structure is a very labor intensive task. Because of this callenge, one of the main aspects of the algorithms developed in this thesis is to develop a method for automatic labling of a dataset using predefined unit structures.

Despite the challenge of labeling datasets there have still been several efforts in applying machine learning algorithms to these kinds of problems. Spellings and Glotzer [69] used a machine learning technique called gaussian mixture models, which does not require a labeled dataset at all. This technique was not used to classify individual particles of a dataset however, but rather to differentiate between several datasets, each containing different kinds of structures. This has very useful applications, especially when studying systems of self-assembling structures, but it is not that useful in the study of dissociation as we need to classify individual particles of a molecular system, not the system as a whole.

Our own work is inspired by another algorithm utilizing machine learning, developed by Reinhart et al. [62]. This algorithm uses the local neighborhood topology of atoms in crystals to classify the particles by comparing their neighborhood graphs. This algorithm was very computationally expensive and by utilized machine learning to choose which neighborhood graphs in a crystal to compare, they where able to increase the speed of the algorithm. However, they did not use machine learning for the actual classification of particles. In our own development we will be using the neighborhood graphs of particles but, as opposed to Reinhart et al., we will be using machine learning directly to classify molecular structures.

1.4 Ethical Considerations

In 1956 John McCarthy was one of the initiators of the Dartmouth college artificial intelligence conference, at which he coined the expression *artificial intelligence*. This conference is widely regarded as the event that initiated AI as a research disipline [50]. In the proposal for the conference McCarthy et al. [45] wrote:

> The study is to proceed on the basis of the conjecture that every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it.

This proposal immediately raises philosophical questions of human-machine interactions. Even if no machine existing today comes close to human intelligence, we do not have to venture into the realm of science fiction to encounter human-machine interactions. We still have to make moral decisions regarding programming and usage of existing technologies. If you are out driving your car and suddenly a trailer is drifting into your lane, you would probably instinctively turn away, even if that involves entering the sidewalk potentially injuring pedestrians in the process. If a self-driving car is in the same situation we can actually make the moral decision in advance: should the car protect the driver at all costs, potentially injuring several pedestrians by doing so, or should the car protect the pedestrians by crashing the car into the trailer? This might seem like an hypothetical, but as self-driving technology keeps advancing it becomes a very real conundrum.

We already have artificial intelligence interacting with our every day life in multiple ways. Machine learning algorithms for creating targeted advertisement campaigns have become so powerful and precise that it is easy to feel surveilled. This interaction between human lives and machine learning algorithms means we need to be wary of how we influence the algorithms through our own biases. Any bias we as humans inject into the dataset, consiously or not, the algorithm will eventually learn. An example of this was when Microsoft tested a chatbot learning from Twitter. It quickly started using racist and sexist language based on the vocabulary used on the social media platform[60]. Even though the algorithms developed in this thesis are only used on the classification of particles in molecular dynamics simulation, the same machine learning theory can easily be transfered to more morally questionable objectives, like facial recognizion used for survaillance [42].

The overarching moral dilemmas of machine learning might not be directly pertinent to the goals of this thesis. However we are applying machine learning algorithms to studies of methane hydrates, and as mentioned, hydrates are of great interest in the petroleum industry. As global temperatures keeps increasing one can question the morality of research which could further perpetuate the increase in CO_2-emissions by aiding the extraction of fossil fuels. Simultaneously the same research might assist in the understanding, and hopefully prevention, of methane release into the atmosphere from methane hydrate blowouts. As researchers we have to be concious of the possible consequenses of new discoveries but generally scientific curiosity should be encouraged, as we never know exactly what uses new discoveries will have. Good or bad.

1.5 Goals

The goal of this master project is to develop machine learning algorithms capable of distinguishing molecular stuctures of individual particles in simulated systems of crystals. The development of supervised machine learning algorithms for structure identification sufferes from the requirement of labeling every particle in a molecular system by their corresponding structure. We will develop and use two methods for overcoming this problem; an automatic method for creating labeled datasets and unsupervised clustering methods, which avoids to problem of labeled dataset altogether.

The trained models will be applied to two non-labeled molecular dynamics datasets. The first dataset contains multiple molecular structures, where we want to distinguish between all the structures of the dataset. The second dataset is of a simulated methane hydrate polycrystal with applied external stress. Here we are especially interested in finding the so-called grain boundaries of the crystal.

To reach the overarching goal we can divide the project into several subgoals:

1. **Implement and verify the CHILL+ algorithm [52]:**
   CHILL+ is an algorithm especially designed for differentiating methane hydrate structures from ice. We will use this algorithm as a benchmark for our supervised and unsupervised machine learning implementations.

2. **Reproduce the dataset created by Engel et al. [14]:**
   Engel et al. created datasets of crystals which self-assembles in to a variety of structures during cooling of the system. We will use the manual classification of these structures, done by Engel et al., as another benchmark for our supervised machine learning algorithms.

3. **Automatic supervised feature creation:**
   We propose an automatic method for creating labeled datasets of molecular structures, making them useable as training sets for supervised machine learning frameworks. The automation of feature creation can be divided into additional subgoals.

   - **Aquire unit structures:**
     We will aquire a dataset of molecular unit structures by implementing a web crawler in Python, which finds and downloads relevant structures.
   - **Induce irregularites:**
     We will induce thermal noise into the molecular unit structures by implementing a harmonic potential in Atomic Simulation Environment (ASE).
   - **Create machine learning features:**
     Inspired by Reinhart et al. we will create an implementation in Python for evaluating the local neighborhood of particles in a dataset. Machine learning features will be created by calculating adjacency matrices for each particle in the temperated and non-temperated unit structures. Each particle will then be labeled by the unit structure it was situated in before temperation.

4. **Implement supervised machine learning methods:**
   We will implement and test two supervised machine learning frameworks for structure identification: Convolutional neural networks and fully-connected networks. Both these frameworks will be trained on the automatically created labeled dataset where we use the same molecular structures as was found in the dataset of Engel et al. The best performing networks will then be applied to the two benchmarking datasets, and compared with the results of CHILL+ and Engel et al.

5. **Implement methods for dimensionality reduction:**
   The number of elements in the adjacency matrices we will implement are a quadratic function of the number of neighbors we include in the topology calculation of the particles. Each of these element is considered as a separate dimension in the machine learning framework which makes the dimensionality of the classifiaction problem potentially very high. To reduce the high dimensionality of the adjacency matrices we will implement two methods for dimensionality reduction before using unsupervised clustering: The popular technique of principal component analysis (PCA) where we remove dimensions of low variance, and an autoencoder where we cluster on the latent space of the autoencoder.

6. **Implement unsupervised machine learning methods:**
   Spellings and Glotzer utilized spherical harmonics as inputs to the gaussian mixture models clustering algorithm and applied this to the datasets of Engel et al. Inspired by this we will try clustering the individual particles of a single dataset of methane hydrates using adjacency matrices as inputs. We will test four unsupervised machine learning algorithms for clustering; Gaussian mixture models, agglomerative clustering, OPTICS and DBSCAN.

1.6 Structure of the thesis

This thesis is divided into three parts. In part I: Background, we start by building a basic understanding of relevant molecular dynamics concepts. This is followed by a thorough breakdown of relevant machine learning theory, where we go through two of the main machine learning branches; supervised learning and unsupervised learning. Part II: Implementation and Results, goes through the methods developed for creating datasets and machine learning algorithms, as well as the results of these algorithms when applied to data from molecular dynamics simulations. In Part III: Conclusions, we summarize and conclude the results and evaluate future prospects.

The code developed in this thesis will be made available at *github.com/kristtuv*.

Part I

Background

Chapter 2

Molecular Dynamics

Molecular dynamics (MD) is a computer simulation method that can be used to study the movement and interactions between atoms and molecules. Performing real experiments on particles is likely to be extremely time consuming and expensive. Computing resources are, on the other hand, relatively cheap and simple to implement. When a mathematical model of a physical system is developed through physical experiments, a simulation using numerical methods can be used to perform thousands of experiments testing the mathematical model. If the simulation is able to reproduce results from the physical model, it might be able to produce new insight into features not tested in the physical experiment, and in the very least help guide the direction of new experiments.

In MD simulations, we assume the particles obey Newton's second law of motion $\sum \boldsymbol{F} = m\ddot{\boldsymbol{r}}$. The forces acting on a particle are defined by an interaction potential $\boldsymbol{U}$, and updated positions and velocities are calculated using standard numerical integration techniques. In molecular dynamics it is common to use the velocity verlet algorithm [80] because of its simplicity, but also because of the low local positional error of $\mathcal{O}(\Delta t^4)$[18]. The algorithm is defined by

$$\begin{aligned} \boldsymbol{r}_{t+1} &= \boldsymbol{r}_t + \Delta t \dot{\boldsymbol{r}}_t + \frac{1}{2}\Delta t^2 \ddot{\boldsymbol{r}}_t, \\ \dot{\boldsymbol{r}}_{t+1} &= \dot{\boldsymbol{r}}_t + \frac{1}{2}\Delta t \left(\ddot{\boldsymbol{r}}_t + \ddot{\boldsymbol{r}}_{t+1}\right), \end{aligned} \tag{2.1}$$

where Δt is the timestep, $\boldsymbol{r}_t$ is the position at time t, $\dot{\boldsymbol{r}}_t$ is the time derivative of the position, and $\ddot{\boldsymbol{r}}_t$ is the double time derivative. The forces acting on a particle are calculated by

$$\boldsymbol{F}_i = m\ddot{\boldsymbol{r}} = \frac{\partial \boldsymbol{U}}{\partial \boldsymbol{r}_i}, \tag{2.2}$$

given a predefined interaction potential $\boldsymbol{U}$.

2.1 Potentials

The equations above are in principle all we need for setting up an MD simulation. There are, however, a flurry of intricate details we need to keep in mind for setting up a realistic simulation. One of these details is which interaction potentials to use. In this thesis, three interaction potentials have been used for different purposes with varying degrees of physical realism.

2.1.1 Stilling-Weber Potential

The Stillinger-Weber potential [74] combines two- and three-body interactions where the bond energy is related to the distances and angles between the atoms.

$$\begin{aligned} U_{SW}(\boldsymbol{r}) &= \sum_i \sum_{j>i} \phi_2(r_{ij}) + \sum_i \sum_{j\neq i} \sum_{k>j} \phi_3(r_{ij}, r_{ik}, \theta_{ijk}), \\ \phi_2(r_{ij}) &= A_{ij}\varepsilon_{ij}\left[B_{ij}\left(\frac{\sigma_{ij}}{r_{ij}}\right)^{p_{ij}} - \left(\frac{\sigma_{ij}}{r_{ij}}^{q_{ij}}\right)\right]\exp\left(\frac{\sigma_{ij}}{r_{ij}-a_{ij}\sigma_{ij}}\right), \\ \phi_3(r_{ij}, r_{ik}, \theta_{ij}) &= \lambda_{ijk}\varepsilon_{ijk}\left[\cos\theta_{ijk} - \cos\theta_{0ijk}\right]^2 \exp\left(\frac{\gamma_{ij}\sigma_{ij}}{r_{ij}-a_{ij}\sigma_{ij}}\right)\exp\left(\frac{\gamma_{ik}\sigma_{ik}}{r_{ik}-a_{ik}\sigma_{ik}}\right), \end{aligned} \tag{2.3}$$

where ϕ_2 is the two-body interaction term and ϕ_3 is the three-body interaction term. Stillinger-Weber is typically used for silicon simulations but can also be used for other systems, like methane hydrates.

We will use this potential on small simulations of methane hydrate crystals.

2.1.2 Oscillating Pair Potential

Mihalkovič and Henley [49] developed a family of oscillating pair potentials given by

$$U(r) = C_1 r^{\nu_1} + C_2 r^{\nu_2} cos(kr + \phi). \tag{2.4}$$

A specific, simplified potential from this family was used by Engel et al. [14] and later Spellings and Glotzer [69] to study systems of self-assembing colloidal crystals. The simplified potential has three wells and mimics the interactions of many metallic systems.

$$\boldsymbol{U}_{OPP}(\boldsymbol{r}) = \frac{1}{\boldsymbol{r}^{15}} + \frac{1}{\boldsymbol{r}^{3}}\cos\left(k(\boldsymbol{r} - 1.25) - \phi\right). \tag{2.5}$$

This potential combines a short-range repulsion with damped oscillation of freqency k and phase shift ϕ. The potential is terminated at the third maximum and smoothly shifted to zero.

We will use this potential for recreating the dataset of Engel et al.

2.1.3 Harmonic Potential

The harmonic bond potential is a simple oscillating potential which simulates particles moving in accordance with Hooke's law in a spring-like fashion.

$$\boldsymbol{U}_H(\boldsymbol{r})c = \frac{k}{2}(\boldsymbol{r} - \boldsymbol{r}_0)^2, \tag{2.6}$$

where k is a constant of proportionality, the division by two is just a simple trick for making the derivative look cleaner, and $\boldsymbol{r}_0$ is the particle positions at equilibrium.

The way we will use this potential there is no interaction between the particles, each particle will only oscillate around its equilibrium position. This is not a realistic potential and will only be used as a means for introducing irregularities in unit crystals, imitating temperature variations in realistic molecular dynamics simulations.

2.2 Cutoff

According to the potentials given in the previous section, we need to calculate the forces between every atom in the MD simulation. If the number of particles in the system is N, this means we have a time complexity of $\mathcal{O}(N^2)$. For large scale MD-simulations, calculating the interactions between all particles is not feasible as the number of particles might be in the millions. To overcome this time complexity, it is typical to introduce a cutoff radius r_{cut}. Only interactions happening between particles closer together than this radius are calculated. Most potentials drop relatively quickly to zero with increased radial distance, making the error from this cutoff distance negligible.

2.3 Thermostats

We define the temperature in a system in terms of the sum over individual kinetic energies of particles. This is what is known as the equipartition theorem,

$$E_k = \frac{f}{2} k_b T, \tag{2.7}$$

where f is the kinetic degrees of freedom, T is the temperature, and k_b is the boltzmann constant. Rearranging the equipartition theorem we get an expression for the temperature,

$$T = \frac{2E_k}{f k_B} = \frac{1}{f k_b} \sum_{i=1}^{N} m_i v_i^2. \tag{2.8}$$

2.3.1 The Langevin Thermostat

In an MD simulation, we will most likely want to control the temperature of the system. This is done by coupling the system to an external heat bath kept at constant temperature and allowing the two systems to exchange energy. This is what is called the canonical- or NVT ensemble. The Langevin thermostat implicitly simulates this heat bath by using

$$\ddot{\boldsymbol{r}}_i = \frac{\dot{\boldsymbol{p}}_i}{m_i} = \frac{\boldsymbol{F}_i}{m_i} - \gamma_i \frac{\boldsymbol{p}_i}{m_i} + \frac{\boldsymbol{f}_i}{m_i}, \tag{2.9}$$

where $\boldsymbol{F}_i$ is the force acting on atom i set up by the interaction potential, γ_i is the viscocity, and $\boldsymbol{f}_i$ is a random force simulating the damping of particles between each other due to friction. The random force is drawn from a Gaussian distribution with variance

$$\sigma_i^2 = 2 m_i \gamma_i k_B T / \Delta t, \tag{2.10}$$

where Δt is the timestep used to integrate the equations of motion.

2.3.2 The Nosé-Hoover Thermostat

The Langevin thermostat simulated the heat bath implicitly by using random friction and a random force to act like a solvent in the system. Nosé-Hoover [54, 32], on the other hand, defines the heat bath explicitly by adding an additional degree of freedom to the system. This extension of the original "real" system will be referred to as the extended system. We have used the original articles and the presentation of Hünenberger [35] for the derivations in this section.

We start by defining the timescale of the extended system as stretched by a factor $\tilde{s}$

$$d\tilde{t} = \tilde{s}dt, \tag{2.11}$$

where the tilde signifies the extended system coordinates. By requiring the particle coordinates of both systems to be equal and using equation 2.11. We end up with the coordinate transformation

$$\begin{aligned}
\tilde{\boldsymbol{r}} &= \boldsymbol{r}, \\
\dot{\tilde{\boldsymbol{r}}} &= \frac{d\tilde{\boldsymbol{r}}}{d\tilde{t}} = \tilde{s}^{-1}\dot{\boldsymbol{r}}, \\
\tilde{s} &= s, \\
\dot{\tilde{s}} &= \frac{d\tilde{\boldsymbol{s}}}{d\tilde{t}} = \tilde{s}^{-1}\dot{s}.
\end{aligned} \tag{2.12}$$

Because of these coordinate definitions, the velocities in the extended system is altered by a factor $\tilde{s}^{-1}$ compared to the initial system.

The Lagrangian of the real system is defined as the kinetic energy minus the potential energy.

$$\mathcal{L}_r(\boldsymbol{r}, \dot{\boldsymbol{r}}) = \frac{1}{2}\sum_{i=1}^{N} m_i \dot{\boldsymbol{r}}^2 - U(\boldsymbol{r}). \tag{2.13}$$

The Lagrangian of the extended system as defined by Nosé [55] is

$$\mathcal{L}_e(\tilde{\boldsymbol{r}}, \dot{\tilde{\boldsymbol{r}}}, \tilde{s}, \dot{\tilde{s}}) = \frac{1}{2}\sum_{i=1}^{N} m_i \tilde{s}^2 \dot{\tilde{\boldsymbol{r}}}_i^2 - U(\tilde{\boldsymbol{r}}) + \frac{1}{2}Q\dot{\tilde{s}}^2 - gk_B T_0 \ln(\tilde{s}). \tag{2.14}$$

We recognize the first two terms as the Lagrangian defined in equation 2.13 with the velocities multiplied by $\tilde{s}^2$ to recover the velocities of the real system. The third and fourth term represent the kinetic and potential energy associated with the $\tilde{s}$ variable. The shape of the last term (the potential energy of the $\tilde{s}$ variable) is chosen as such to ensure that the canonical ensemble averages are recovered, as shown in the original article by Nosé [55].

The equations of motion are derived from the Lagrangian equation

$$\frac{d}{dt}\left(\frac{\partial \mathcal{L}}{\partial \dot{\boldsymbol{q}}_j}\right) = \frac{\partial \mathcal{L}}{\partial \boldsymbol{q}_j}, \tag{2.15}$$

where $\boldsymbol{q}_j$ is one of the variables in the Lagrangian.

This yields the equations of motion

$$\begin{aligned}
\frac{d}{dt}(m_i \tilde{s}^2 \dot{\tilde{\boldsymbol{r}}}_i) &= -\frac{\partial \boldsymbol{U}}{\partial \tilde{\boldsymbol{r}}}_i \\
\rightarrow \ddot{\tilde{\boldsymbol{r}}}_i &= -m_i^{-1}\tilde{s}^{-2}\frac{\partial \boldsymbol{U}}{\partial \tilde{\boldsymbol{r}}_i} - 2\dot{\tilde{s}}\tilde{s}^{-1}\dot{\tilde{\boldsymbol{r}}}_i \\
&= m_i^{-1}\tilde{s}^{-2}\tilde{\boldsymbol{F}}_i - 2\dot{\tilde{s}}\tilde{s}^{-1}\dot{\tilde{\boldsymbol{r}}}_i,
\end{aligned} \tag{2.16}$$

and

$$\begin{aligned} \frac{d}{dt}(Q\dot{\tilde{s}}) &= \sum_{i=1}^{N} m_i \tilde{s}\dot{\tilde{\boldsymbol{r}}}^2 - gkT_{bath}\tilde{s}^{-1} \\ \rightarrow \ddot{\tilde{s}} &= Q^{-1}\tilde{s}^{-1}\left(\sum_{i-1}^{N} m_i \tilde{s}^2 \dot{\tilde{\boldsymbol{r}}}_i - gk_B T_{bath}\right). \end{aligned} \tag{2.17}$$

The velocity scaling between the real and extended system leads to uneven time intervals in the real system as these intervals depend on the factor $\tilde{s}$. Hoover used the extended system equations of Nosé and reformulated them in the real system coordinates effectively removing the dependence on $\tilde{s}$, which removes the problem of uneven time intervals.

The equations formulated by Hoover, which is known as the Nosé-Hoover thermostat, uses a coordinate transformation from extended-system to real-system variables. This transformation is defined in the original article [32] as

$$\begin{aligned} & s = \tilde{s}, \quad \dot{s} = \tilde{s}\dot{\tilde{s}}, \quad \ddot{s} = \tilde{s}^2\ddot{\tilde{s}} + \tilde{s}\dot{\tilde{s}}^2, \\ & \boldsymbol{r} = \tilde{\boldsymbol{r}}, \quad \dot{\boldsymbol{r}} = \tilde{s}\dot{\tilde{\boldsymbol{r}}}, \quad \ddot{\boldsymbol{r}} = \tilde{s}^2\ddot{\tilde{\boldsymbol{r}}} + \tilde{s}\dot{\tilde{s}}\dot{\tilde{\boldsymbol{r}}}, \\ & p_s = \tilde{s}^{-1}\tilde{p}_s, \quad \dot{p}_s = \dot{\tilde{p}}_s - Q^{-1}\tilde{s}^{-1}\tilde{p}_s^2, \\ & \boldsymbol{p} = \tilde{s}^{-1}\tilde{\boldsymbol{p}}, \quad \dot{\boldsymbol{p}} = \dot{\tilde{\boldsymbol{p}}} - Q^{-1}\tilde{s}^{-1}\tilde{p}_s\tilde{\boldsymbol{p}}, \\ & \boldsymbol{F} = \tilde{\boldsymbol{F}}, \end{aligned} \tag{2.18}$$

which in turn gives the equations of motion

$$\begin{aligned} \ddot{\boldsymbol{r}}_i &= m_i^{-1}\boldsymbol{F}_i - \gamma\dot{\boldsymbol{r}}_i, \\ \dot{\gamma} &= -k_B N_{df} Q^{-1} T\left(\frac{g}{N_{df}}\frac{T_{bath}}{T-1}\right). \end{aligned} \tag{2.19}$$

2.4 Radial Distribution Function

As the name suggests, the radial distribution function $g(r)$ describes how the density of particles vary as a function of radial distance from a reference particle. The general algorithm for finding $g(r)$ is simply to determine how many particles are within a radial shell with distance r and $r + dr$ from the reference particle. In practice, we do this by calculating the distance between all particle pairs and calculating a histogram of the distribution, and normalizing the distribution with respect to an ideal gas which in three dimensions is the number density of the system multiplied by the volume of the spherical shell $\frac{N}{V}4\pi r^2 dr$. Where N is the number of particles and V is the volume of the system. We can now calculate the RDF as

$$g(r) = \frac{dn(r)}{\frac{N}{V}4\pi r^2 dr}, \tag{2.20}$$

where $dn(r)$ is the number of particles found within the spherical shell.

2.5 Crystal Structure

The evaluation and description of the ordered arrangement of atoms and molecules in a crystal is described by the *crystal structure*. Defining the crystal structure is important because crystals may have very different properties depending on how the atoms are stacked within the crystal. For instance, graphite and diamond are both made purely of carbon atoms, however the graphite crystal structure forms loosely bonded sheets, which easily releases from the crystal, making it possible to write with a pencil. Diamond, on the other hand, is very thightly stacked making it one of the hardest materials on earth.

We can build a theoretical crystal by combining two concepts; the *lattice* and the *basis*. The lattice points are mathematical points in space which, if we were standing on a point and moving from this point to another, the crystal will look the same. At each of these lattice points a periodic arrangement of atoms is placed, called the basis. The combination of these two concepts describes the entirety of the crystal structure. The smallest group of atoms in a crystal system which constitutes a repeating pattern is called the unit cell of the structure. The unit cell in three dimensions is described by the axes between neighboring lattice points, called the principal axes (a, b, c), and the angles between them (α, β, γ). This unit cell and the particle placements (x_i, y_i, z_i) within this cell is a complete description of the system and a crystal can be constructed by replicating the unit cell along its principal axes.

The lattice systems mentioned in Table 2.1 describe the geometry of the unit cells in three dimensions. These systems combined with the so-called centering types gives us what is known as the Bravais lattices shown in Figure 2.1. The centering types are defined as follows:

- Primitive (P): Lattice points at cell corners.
- Base-centered (C): Lattice points at cell corners and one additional point at the center of two of the faces parallel to each other.
- Body-centered (I): Lattice points at cell corners and one additional point in the center of the cell.
- Face-centered (F): Lattice points at cell corners and one additional point at the center of each of the faces.

The *Pearson symbol* is used to classify and describe crystal structures and was originally proposed by W.B Pearson in 1958 [57]. Our description of the Pearson symbol system is described as given by Hubbard and Calvert [34]. The Pearson symbol system consists of three symbols in the order given below. The letters are described in Table 2.1.

1. A lower case letter describing the crystal system (a, m, o, t, h, c)
2. A capital letter describing the lattice centering (P, C, F, I, R):
3. A number describing the number of atoms in the conventional unit cell

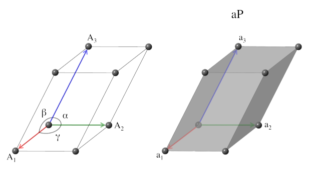
The Pearson symbol only considers translational and rotational symmetry in the crystal and because of this, the Pearson symbol does not uniquely define the symmetry group of a configuration. The 14 Bravais lattices are only 14 symmetry groups of a possible 230 if all symmetry operations are considered.

Table 2.1: Pearson characters and their meaning. Table reproduced from Hubbard and Calvert [34]

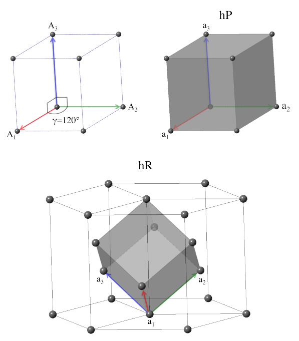
Crystal System (Lattice System)	Code
Anorthic	a
Monoclinic	m
Orthorombic	o
Tetragonal	t
Hexagonal/Trigonal (Hexagonal, Rhombohedral)	h
Cubic	c

Lattice Centering	Code
Primitive	P
All-faces-centered	F
Body centered (Innenzentriert)	I
One-face-centered	C
Rhombohedral	R

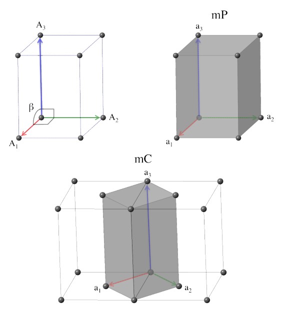
Note: The trigonal crystal system and hexagonal crystal system is part of the same family and are grouped as the hexagonal crystal family. The trigonal crystal system is split into two lattice systems, the rhombohedral system and the hexagonal system. Where the rhombohedral system can be transformed into the hexagonal system only with 3-fold rotational symmetry, as opposed to the 6-fold rotational symmetry of a hexagonal lattice. The reason for the lowered symmetry happens during the coordinate transformation as two atoms are placed on one of the body diagonals of the hexagonal unit cell. Because of this, we also have the R lattice centering which is specific for the hexagonal crystal family.



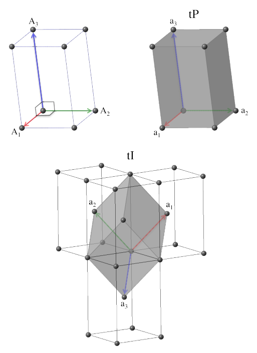
(a) Triclinic



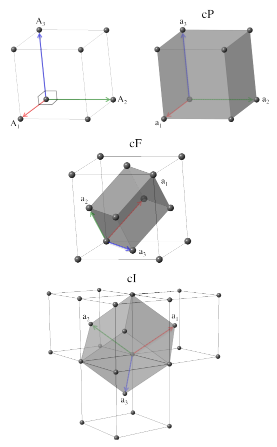
(b) Trigonal



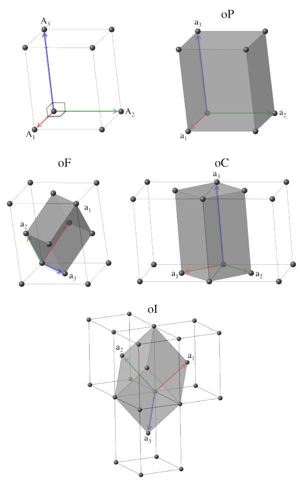
(c) Monoclinic



(d) Tetragonal



(e) Cubic



(f) Orthorhombic

Figure 2.1: The 14 unique Bravais lattices. Figure reprinted from aflow.org [46, 29]

2.6 Methane hydrates

Natural gas hydrates, or clathrate hydrates, are crystalline solids with gas molecules (guests) trapped in frozen water cavities. Multiple types of gas molecules can be trapped in these water cavities, but typical examples are methane, ethane, propane, and carbon dioxide [68]. Hydrates were unquestionably discovered as a scientific curiosity by Humphry Davy in 1810 but might have been discovered as early as 1778 by Joseph Priestley [59, 44]. In the 1934 Hammerschmidt discovered that gas hydrates were blocking gas transmission lines [24], which propelled the interest in gas hydrates from a scientific curiosity into a key scientific research area in the natural gas industry. In the 90s and 00s, there has been an increasing interest in gas hydrates as climate change has increased the earths temperature, and some scientists are worried the methane hydrate stored in the permafrost regions will be released and further perpetuate the rising temperature. Even though this might seem plausible, it is still an open scientific question due to uncertainty in the climate models [53, 28].

In this thesis, we will keep our attention on the structure classification of three different hydrate structures. For describing the cages of the clathrate structures we use the notation x^y where x is the number of edges of a particular face type and y is the number of face types with x amount of edges. Clathrate hydrates usually form two types of cubic structures called sI and sII, or a hexagonal structure called sH or just H. The smallest cubic structure, sI, consists of 46 water molecules forming two types of cages. The sI clathrate unit cell has two 5^{12} cages (twelve pentagons) and six cages of the shape $5^{12}6^2$ (two hexagons and 12 pentagons). This clathrate has a cubic primitive lattice, which makes the Pearson symbol $cP46$. The sII clathrate unit cell is much larger, consisting of 136 water molecules. This type is also structured into two cage types. Sixteen small cages of shape 5^{12} and eight larger ones with shape $5^{12}6^2$. This is also a cubic primitive structure, which is represented by the Pearson symbol $cP136$. A third structure, the H-structure, is less prevalent but may also be observed. It consists of 34 water molecules forming three cages of shape 5^{12}, two cages of shape $4^35^66^3$ and one huge cage of shape $5^{12}6^8$. This is a primitive hexagonal structure giving the Pearson symbol $hP34$. These structures are shown in Figure 2.2.

When the guest molecules trapped in the frozen water cavities are methane, we refer the gas hydrate as a methane hydrate. A concern about methane hydrates found in nature is that the dissociation of hydrates may trigger slope failure in marine sediments, which in worst-case scenarios can cause massive tsunamis [8]. To outline the risks of events like these happening, we need to understand the properties of methane hydrates during mechanical failure. In molecular dynamics simulations of methane hydrate failure, the bulk of cracks developed is situated in the grain boundaries of the material. Because the grain boundaries are of particular interest, it is especially important to locate them in a simulation and developing pattern recognition algorithms that automatically find these boundaries alleviates the manual process of doing so.

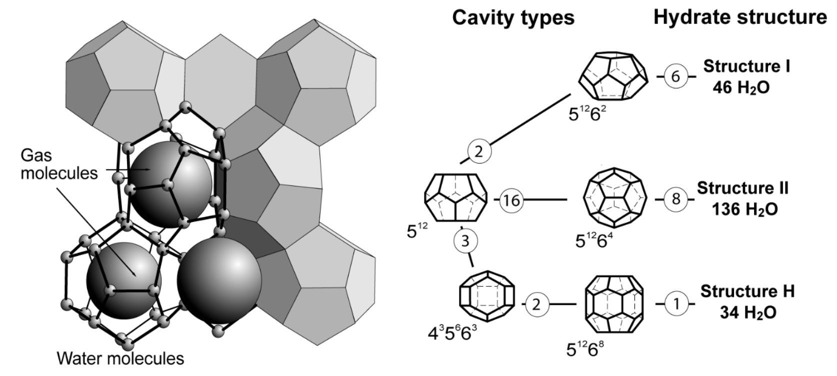


Figure 2.2: Three clathrate hydrate structures, sI, sII and sH. Figure reprinted from Bohrmann and Torres [7].

Chapter 3

Machine Learning

In traditional algorithms, we employ a predetermined and complete set of rules to a problem, taking an input and producing an output. The process of creating such an algorithm might be complicated and if the input data changes, one might have to completely rewrite the algorithm. If the task at hand is to recognize a cat in an image, we would have to predetermine what features constitute a cat and figure out how the algorithm can recognize these features in an image. If we decide the most prominent features of a cat to be e.g. the eyes and ears, problems quickly arise if we look for a cat in an image where the cat is partially hidden and only the tail is sticking out. To a human, there is obviously still a cat in the image, but our algorithm will never be able to find it. If we now instead want to find boats in an image, we would have to completely rewrite our algorithm and again figure out in advance what features constitute a boat. This is a tedious and near impossible task.

Machine learning also takes an input and produces an output but we now leave it up to the machine to implement the inner workings of the algorithm. The machine is, of course, not producing the algorithm entirely on its own. We do impose some more general rules, or more precisely, a framework for how the machine will produce the algorithm. These different frameworks are what we will be discussing in depth in this chapter.

The theory in this chapter is mostly based on the introduction to machine learning for physicists, by Mehta et al. [47].

3.1 Categories of Machine Learning

Machine learning is traditionally grouped into three categories; supervised learning, unsupervised learning, and reinforcement learning.

Supervised learning is defined as training the machine by presenting a correct answer to the problem at hand. We need a dataset which, in some way or another, has labeled each data point, e.g. and image, with the correct answer of what we want the machine to be able to recognize. In the training process, the machine learning algorithm presents what it thinks the answer to the problem is, and if it is wrong, we tell the machine to update and change the algorithm until we are satisfied with the results that are produced.

As opposed to supervised learning, when doing unsupervised learning the machine is never told what the correct answer to the problem is. A typical example of an unsupervised task is to cluster data points into groups based on distance and density metrics applied to the dataset. We do not tell the machine which cluster a point belongs to, and it has to decide solely based on predefined metrics. Another unsupervised task is the autoencoder, where the objective is to reproduce the input given to the machine. This is a powerful framework for e.g. removing noise from the dataset. We will go in depth on supervised learning and unsupervised learning in the following sections.

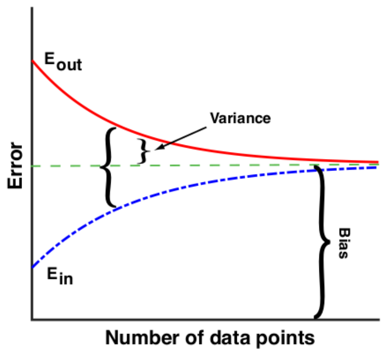
Reinforcement learning exposes the model to an environment where it through trial and error and previous experiences, trains itself to make specific decisions. A typical user case for reinforcement learning is training a machine to play videogames and is only mentioned here for completeness.

3.2 Bias-Variance Tradeoff

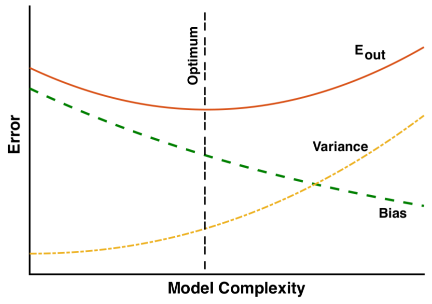
When doing any kind of learning or fitting to a dataset, we need to differentiate between fitting and predicting. We only have access to a limited subset of data stemming from an unknown function, and the objective is not to fit a model arbitrarily well to the known data points. The aim is to construct a model that generalizes and predicts well when given new data points from the unknown function. Because of this, we usually use two different performance measures when fitting the model. The in-sample error (training error) E_{in} which is a measure of how well the model fits the data it has access to, and the out-of-sample error (test error) E_{out} which is a measure of how well the model performs on new data unknown to the model during training time. To imitate a set of unknown data points, it is standard practice to split the subset of data we have access to in a training set and a validation set where we use the models performance on the validation set as a proxy for the out-of-sample error. Assuming the training set to be sufficiently large and a good representation of the true function f, the subsampling into training and validation sets is unbiased.

In Figure 3.1a we see the out-of-sample error and the in-sample error as a function of the amount of training data. The assumption in this graph is that the data is drawn from a complex function, and we are using a too simple model to learn the exact true function. Because the model is too simple the in-sample error increases because it does not have the complexity to represent the true function it is trying to approximate. The out-of-sample error, on the other hand, is decreasing as the number of data points increases. This is because increasing the amount of data reduces the sampling noise making the training data a progressively better representation of the true function. With an infinite amount of data, the out-of-sample error and the in-sample error converges to the same value. The bias is the best error our simple model can produce with an infinite amount of data.

The reason we had a bias in our model was that we did not create a sufficiently complex model to represent the data we are seeing. The most intuitive thing to do is simply increase the complexity of the model. This would be a good idea if we indeed had an infinite amount of data, but that is never the case. The second quantity in Figure 3.1a, which we have not discussed, is the variance. The variance represents the difference between the out-of-sample error and the in-sample error. Or in other words, how well the in-sample error reflects the out-of-sample error, and by extension, how much worse we would expect the model to do new unseen data from the same complex true function. In Figure 3.1b, we see what is known as the bias-variance tradeoff. When using a complex model, we tend to decrease the bias of the error as we are able to approximate the complex true function to a higher degree when the complexity is higher. However, because our data is finite, the model starts to fit too well to the training data, including the sampling noise, and the out-of-sample error starts increasing again. Because of this, we need to use a model that finds a good compromise in the bias-variance tradeoff, using just the right amount of complexity.



(a)



(b)

Figure 3.1: Two illustrations of the bias-variance tradoff. As the amount of training data increases towards infinity, the out-of-sample error and the in-sample error converges (a). With a finite amount of data, a complex model will fit very well to the training data but generalize poorly (high variance), while a too simple model will not be able to represent the true underlying function at all (high bias). The optimal model is a compromise between the two (b). Figures reprinted from Mehta et al. [47].

3.3 Linear Regression

In Section 3.2, we considered the reasoning for why, when fitting a model to a dataset, we are primarily concerned with minimizing some out-of-sample error. We will make this more concrete by presenting an example of a simple machine learning algorithm, *linear regression*. Even though machine learning is mostly associated with neural networks, which we will discuss later in the chapter, machine learning as a whole utilizes several techniques developed in the fields of statistics and statistical learning, and linear regression is one such technique. Linear regression and the method of least squares is usually, although not conclusively [73], attributed to Carl Friedrich Gauss and is a staple of statistical learning. Linear models are stable but often inaccurate due to large assumptions made on the dataset. For a linear model to correctly represent our dataset, we need to assume that the data is indeed linearly distributed, which we have no guarantee of being a proper assessment.

Presume we we have a dataset with n observations $\mathcal{D} = \{(y_i, \boldsymbol{x}_i)\}_i^n$ where $\boldsymbol{x}_i \in \mathbb{R}^{p+1}$ is the i-the observation and y_i is a scalar response to some unknown linear relationship of the observation vector. We add the $+1$ to the dimensionality of the observation vector as a placeholder for the intercept. In the context of machine learning the individual values of the observation vector is often referred to as *features*. We define our model by assuming there exists a true function f that generated the responses y_i through the relationship

$$y_i = f(\boldsymbol{x}_i, \boldsymbol{\beta}_{exact}) + \varepsilon_i, \tag{3.1}$$

where $\boldsymbol{\beta}_{exact} \in \mathbb{R}^{p+1}$ are the unknown parameters or coefficients we want to approximate and ε_i is some independent and identically distributed noise with mean μ and standard deviation σ. We do not know the function f which generated the samples, but for a linear regression model we make the hopefully warranted assumption that f is just the identity function and so the relationship becomes

$$y_i = f(\boldsymbol{x}_i, \boldsymbol{\beta}_{exact}) + \varepsilon_i = \boldsymbol{x}_i^T \boldsymbol{\beta}_{exact} + \varepsilon_i \tag{3.2}$$

To approximate the function parameters $\boldsymbol{\beta}_{exact}$ we minimize a cost function $\mathcal{C}(y_i, h(\boldsymbol{x}_i; \boldsymbol{\beta}))$ for evaluating our models ability to make predictions on the dataset, where $h(\boldsymbol{x}_i; \boldsymbol{\beta}) = \boldsymbol{x}_i^T \boldsymbol{\beta}$ is our approximation to the underlying true function f. For ordinary least squares linear regression the most popular cost function of choice is the residual sum of squares (RSS):

$$\text{RSS} = \min_{\beta \in \mathbb{R}^{p+1}} \sum_{i=1}^{n} (\boldsymbol{x}_i^T \boldsymbol{\beta} - y_i)^2, \tag{3.3}$$

where i runs over all the samples in the dataset. For a more compact representation of the entire dataset we write this equation with matrix notation

$$\text{RSS} = \min_{\beta \in \mathbb{R}^{p+1}} ||\boldsymbol{X}\boldsymbol{\beta} - \boldsymbol{y}||_2^2. \tag{3.4}$$

A linear regression example is shown in Figure 3.2.

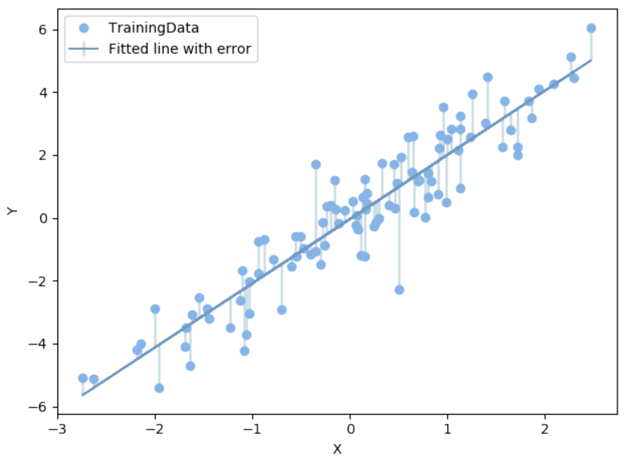


Figure 3.2: Linear regression example. Line fitted to a linear dataset with Gaussian noise and a residual sum of squares cost function.

3.4 Overfitting and Regularization

As discussed when fitting a model to a dataset, what we want is to minimize the out-of-sample error (test error), not the in-sample error (training error). As we increase the complexity of our model, the training error will generally decrease but as we saw from the bias-variance tradeoff, this does not generally decrease the test error. When the complexity of the model reaches a point where the testing and training errors start to diverge, we say the model is *overfitting* to the data. As we see from Figure 3.3, increasing the complexity of the model we are able to fit the training data arbitrarily well. The true function which generated the data is a simple second-degree polynomial function with some added Gaussian noise, and even though a polynomial of degree 25 is a good fit to the dataset, we can not expect to make good predictions on new data outside of the domain of the training set. The model is overfitted. To mitigate the issue of overfitting, we introduce the concept of regularization. Regularization is a method for punishing the model for adding more complexity. We can view this as adding friction to the model, and for the model to increase in complexity, the benefits for doing so must overcome the friction working against it.

The most widely used methods for adding regularization to a model is the so-called L_1 and L_2 regularization, which in linear regression is named Lasso and Ridge regression, respectively.

For ordinary least squares regression, we sought to minimize the residual sum of squares

$$\min_{\beta \in \mathbb{R}^{p+1}} ||\boldsymbol{X\beta} - \boldsymbol{y}||_2^2. \tag{3.5}$$

To add a ridge penalty to this equation, we simply add the square of the proposed weights β

$$\min_{\beta \in \mathbb{R}^{p+1}} ||\boldsymbol{X\beta} - \boldsymbol{y}||_2^2 + \lambda\boldsymbol{\beta}^2, \tag{3.6}$$

where λ is a tuning parameter regulating the strength of the regularization. As

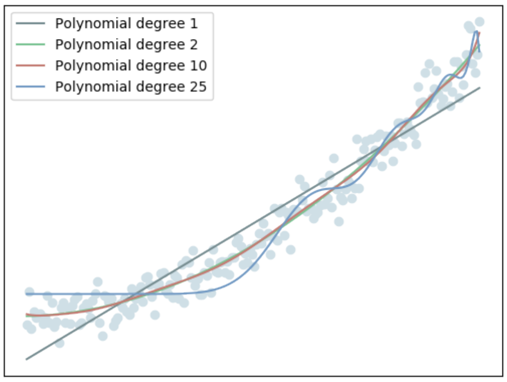


Figure 3.3: Polynomial regression on a dataset generated from a second-degree polynomial. As the complexity of the model increases the model starts overfitting to the dataset which would make worse predictions on new data.

with ordinary least squares, we are trying to minimize the cost function but the added term $\lambda\boldsymbol{\beta}^2$ is only small if the coefficients are kept small. As we increase λ, we will essentially push the coefficients towards zero. The reason this gives better out-of-sample results is because of the bias-variance tradeoff. Increasing λ corresponds to decreasing the variance and increasing the bias. Finding the perfect size of λ is typically done through testing different values on a validation set and monitoring the RSS error.

As discussed, ridge regression decreases all coefficients towards zero. The disadvantage of this approach is that generally, none of the coefficients are actually set to zero unless $\lambda \to \infty$. This essentially means that even though the variance of a complex model is greatly reduced, the complexity is never completely removed from the model. Lasso regression, on the other hand, seeks to minimize the quantity

$$\min_{\beta \in \mathbb{R}^{p+1}} ||\boldsymbol{X}\boldsymbol{\beta} - \boldsymbol{y}||_2^2 + \lambda|\boldsymbol{\beta}|. \tag{3.7}$$

Lasso regression has the effect of forcing some of the coefficients to be exactly zero. This effect is perhaps not completely clear from equation 3.7, so we use Figure 3.4 to build an intuitive understanding. Increasing and decreasing the tuning parameter λ corresponds to the decrease and increase of the blue areas of Figure 3.4, and by doing so we also increase and decrease the valid regions of the residual sum of squares. We can see that because ridge regression corresponds to a constraining region of a circle, the point at which the RSS contours and the constraint meets is generally not at one of the axes and hence the coefficients β_1 and β_2 will not be zero. For lasso regression, on the other hand, the corners of the constraint follow the axes and so the points at which the contours and the constraint meet is likely to be at a point where one of the coefficients is zero. In higher dimensions, many of the coefficients can be set to zero at the same time.

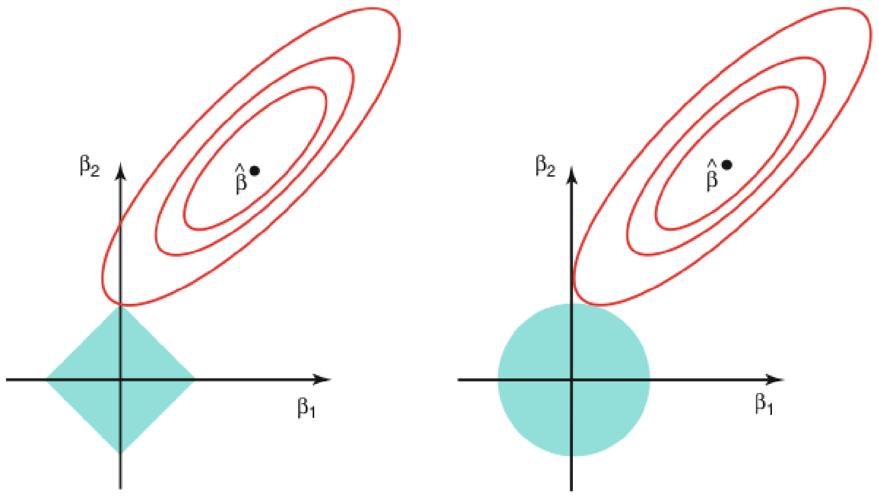


Figure 3.4: Red circles represents regions of constant residual sum of squares. The blue are constraint regions set up by lasso regression (left) and ridge regression (right). Figure from Hastie [25]

3.5 Neural Networks

Artificial Neural Networks (ANN) are vaguely inspired by the brain. The brain is of course far more complicated than any neural network but the brain is, in a sense, the most powerful computer we know of and trying to replicate some of the properties of the brain in a computer seems like a good starting point for creating artificial intelligence. The brain consists of billions of nerve cells called neurons. These neurons are complicated biological structures, but their general purpose is simple. The charged chemicals outside and within the neuron can move through channels in the cell membrane, which raises or lowers the electrical potential between the outside and inside of the cell. This is called the membrane potential. If the membrane potential is raised to some threshold, the neuron discharges by firing an electrical pulse down what is called the axon. This axon is connected to many other neurons in its vicinity and transmits some of the electrical charge to these neurons, which in turn raises their membrane potential. Now, what we are really interested in is not the mechanical and biological properties of the neurons but how they actually learn anything new. This is known as neural-plasticity and is described by Hebb's rule. Hebb's rule states that the strength of connections between neurons is proportional to the correlation of firing between them. This means that if two neurons have repeatedly fired at the same time, both of them will most likely fire if just one of them is stimulated.

In 1943 a mathematical model of a neuron was first presented by McCulloch and Pitts. This model was simple:

1. A set of inputs x_i to the neuron, where x_i is the electrical charge transferred from other neurons.

2. A set of weights w_i, which corresponds to the connection strength between

the other neurons and the one we are looking at.

3. An adder $\sum_{i=1}^{N}$ that sums the weighted inputs $x_i w_i$, which equates to the membrane potential collecting the electrical charge.

4. An activation function f, which decides whether the neuron fires for the current membrane potential.

We summarize this model with the equation

$$y_i = f\left(\sum_{j=1}^{n} w_j x_j + b_j\right) = f(z), \tag{3.8}$$

where y_i is the output from a single neuron, x_j are the inputs, w_j are the weights, z is the weighted sum of the inputs and f is the activation function. Of course a lot has happened in the field of neuroscience since this model was developed and it is unfortunately not very realistic. However, as we shall see later in this chapter, this is precisely the mathematical model we are going to use for developing an artificial neural network.

3.5.1 Multilayer Perceptron

We have mentioned one of the major tasks of supervised learning, regression. Another widely used objective for a supervised algorithm is classification. In this task, we have a set of inputs that can be categorized into two or more classes, e.g. pictures of cats and dogs. The main difference between regression and classification is the output variable. In regression, we output a continuous variable where our model is trying to fit an underlying true function. Classification, on the other hand, outputs a discrete number representing a class of which the input belongs. We are still in practice fitting some function to the dataset, only now we are not concerned with the function's fit to the dataset itself, we instead want to find a function which separates the data in the best possible way.

Classifiers computing a linear combination of the input vectors and returning a binary response were called perceptrons in the engineering literature in the 1950s [25]. Rosenblatt's perceptron learning algorithm [63] is the foundation for all neural network algorithms developed in the machine learning revolution. The Rosenblatt perceptron algorithm is a simple binary classifier that minimizes the distance of misclassified points to the linear decision boundary. We define the cost function

$$\mathcal{C}(\boldsymbol{\theta}, \boldsymbol{\theta}_0) = \sum_{i\in\mathcal{M}} (-1)^{1-y_i} (\boldsymbol{x}_i^T \boldsymbol{\theta} + \boldsymbol{\theta}_0), \tag{3.9}$$

where the sum is over all instances i in the set of misclassified points $\mathcal{M}$, y_i is the correct response being either 0 or 1, $\boldsymbol{\theta}$ is the parameters and $\boldsymbol{\theta}_0$ is the bias[1]. To construct the more powerful multilayered perceptron we, as the name suggests, simply stack these perceptrons into layers, as shown in Figure 3.5b. The layers between the input and output layers are referred to as hidden layers. When generalizing the perceptrons to use any kind of activation function, we

[1]It is standard practice to use a different notation of the parameters when referencing standard regression techniques (β), and neural network techniques (θ).

usually refer to them as neurons or nodes. Every circle in this figure represents one neuron that takes as input the output from the previous layer. The neuron calculates the weighted sum of the inputs before using a designated activation function to introduce non-linearities to the network, as shown in Figure 3.5a. This figure is, of course, exactly the mathematical model defined in the introduction to this chapter, Equation 3.8. This representation of a multilayered perceptron where every neuron in a layer is connected to every neuron in the next layer is a so-called fully-connected neural network. When the network architecture is as presented in this section the terms multilayered perceptrons and neural networks can be used synonymously, however the term neural networks is a lot broader and includes more complicated network architectures.

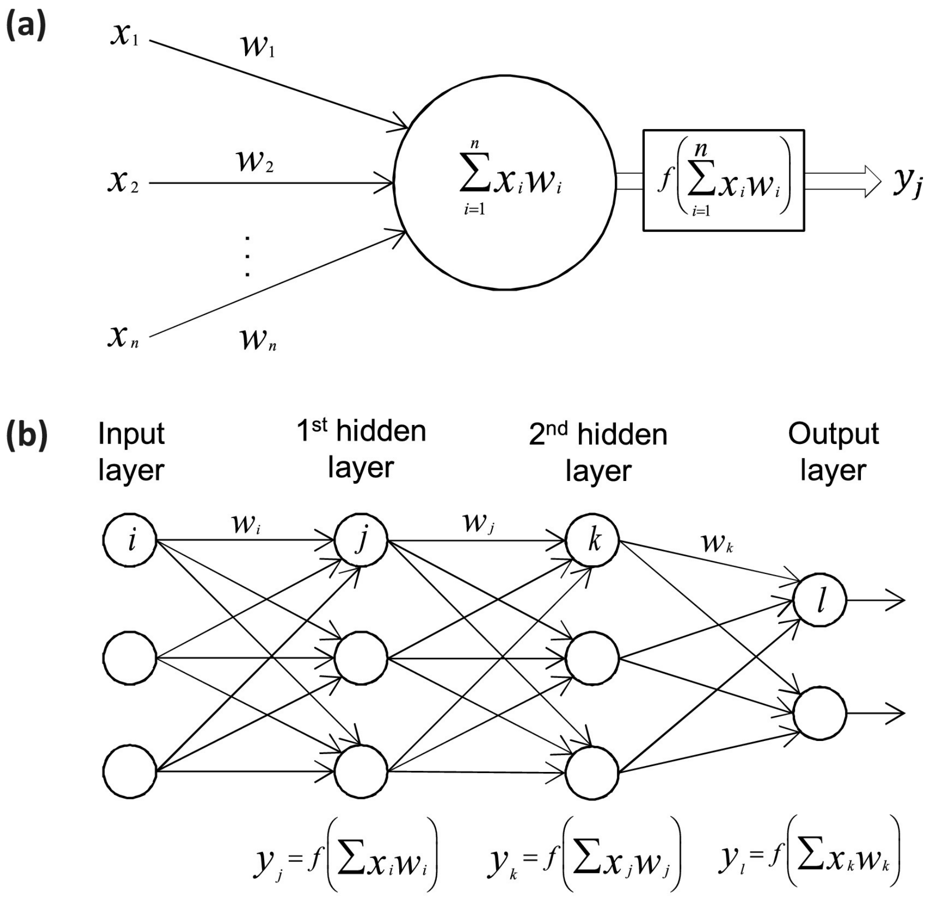


Figure 3.5: Visual representation of tha calculations done by a single neuron (a) and the architecture of a full-connected neural network. Figure reprinted from Vieira et al. [81]

It is not obvious that stacking neurons into layers is automatically going to improve the prediction results of a model. To explain why this is, in fact, a good idea, we need to refer to the universal approximation theorem. The universal approximation theorem states that given an activation function that is continuous, non-constant and bounded, any continuous function can be approximated by a neural network within an arbitrarily small error using only a single hidden layer containing a finite number of neurons.[33]. This entails that in practice, if we are approximating a continuous function, only one hidden layer is needed. The theorem does not, however, state how many neurons are needed for the approximation and for complex machine learning problems, it has been empirically proven to be more efficient to increase the depth rather than the width of the network [66]. Why increasing the depth is beneficial becomes more clear if we view each layer in a network as compressing and extracting the most relevant features of the previous layer and in this way improving the networks ability to generalize.

3.5.2 Network Topology

The first thing we need to decide when building a neural network is the *architecture* of the network. This means deciding how many layers and neurons we want the network to utilize. We can use an arbitrary number of hidden layers in our neural network depending on the nature of the problem. Even though be may decide to use as deep of a network as our computational capacity allows, it is likely not necessary as we know from the universal approximation theorem. The number of layers and nodes are examples of what we call *hyperparameters*. These parameters are set before any training has taken place, contrary to the weights and biases, which are updated and adjusted during training time. There is no obvious way of deciding which hyperparameters and architectures will work best for a given problem, and we usually need to brute force the problem by running the network with several combinations of hyperparameters. A standard way of deciding on hyperparameters is to test the network with several random values within a probable span of the parameters. If any of the parameters show promise and predicts well on the training and validation set, we may choose to narrow down our span of the parameter around this value and retrain the model in this parameter range.

3.5.3 The Cost Function

The cost function is a function that maps a response to a real number quantifying how the algorithm is performing. The RSS error mentioned in Section 3.3 is one example of a cost function used for regression, which evaluates the distance of a set of data points to a proposed model. We seek to minimize this cost function to get an optimal result for the error of the model. In classification problems, we are trying to find a function that separates the data into a given number of classes. The perceptron defined in Section 3.5.1 is one such cost function.The perceptron is an example of a so-called hard classifier. It uses a linear boundary to separate the classes, and the cost function is a step function that classifies the output as either correct or wrong with respect to this linear boundary. A data point that is classified correctly and far away from the decision boundary is viewed as just as correct as a correctly classified point right next to the boundary. Because of this, there is no sense of how wrong or how correct the classification actually is. In most cases, it is more advantageous to use a so-called soft classifier. A soft classifier maps the linear output to a probability by using an activation function. One way to do this is to use the sigmoid function

$$\begin{aligned} P(y_i = 1|\boldsymbol{x}_i, \boldsymbol{\theta}) &= \frac{1}{1+e^{-\boldsymbol{x}_i^T\boldsymbol{\theta}}} = f(\boldsymbol{x}_i^T\theta), \\ P(y_i = 0|\boldsymbol{x}_i, \boldsymbol{\theta}) &= 1 - P(y_i = 1|\boldsymbol{x}_i, \boldsymbol{\theta}), \end{aligned} \tag{3.10}$$

where $\boldsymbol{x}_i$ is a data point and $\boldsymbol{\theta}$ are the weights we wish to learn from the data. The sigmoid function compresses the linear combination between 0 and 1, which we can view as a probability. If the probability is above 0.5, we classify a point as part of class 1 and as class 0 otherwise. We still set a hard boundary between the classes, only now we have a sense of how confident the model is of the classification result. We can combine these two expressions into a single cost function evaluated on the dataset $\mathcal{D}$

$$L = P(\mathcal{D}|\theta) = \prod_{i=1}^{N} \left[f(\boldsymbol{x}_i^T\boldsymbol{\theta}\right]^{y_i} \left[1 - f(\boldsymbol{x}_i^T\boldsymbol{\theta})\right]^{1-y_i}, \tag{3.11}$$

where y_i is the correct class of a data point x_i. This expression is called the *maximum likelihood estimation* and is the likelihood of seeing the data under our current model. By maximizing it, we find a combination of model parameter values that maximizes the probabilities. In practice, we normally take the negative logarithm of this expression, also called to cross-entropy, as this is more computationally convenient. The probabilities are never higher than one, and by taking the repeated product of the probabilities, we will most likely run into underflow errors for a large dataset. Instead, we minimize the negative sum of logarithms to avoid this problem,

$$l = -\sum_{i=1}^{N} y_i log[f(\boldsymbol{x}_i^T\boldsymbol{\theta}J)] + (1-y_i)log[1 - f(\boldsymbol{x}_i^T\boldsymbol{\theta})]. \tag{3.12}$$

When dealing with a binary response, we just used the scaler values 0 and 1 for the possible classes. To label the responses of a multiclass classification problem we use so-called one-hot encoded vectors to represent the responses

$$\boldsymbol{y}_i = (0, 1, \cdots, 0), \tag{3.13}$$

where the placement of the 1 indicates which class a data point $\boldsymbol{x}_i$ belongs to, and the rest of the values are zeros. This particular example then belongs to class 2. To evaluate datasets containing more than one class we usually use the softmax function which is a generalization of the sigmoid function. The probability of a data point $\boldsymbol{x}_i$ belonging to a particular class c' when evaluated with the softmax function is given by

$$P(y_{ic'}) = 1|\boldsymbol{x}_i, \boldsymbol{\theta}) = \frac{e^{-\boldsymbol{x}_i^T\boldsymbol{\theta}_{c'}}}{\sum_{c=0}^{C-1} e^{-\boldsymbol{x}_i^T\boldsymbol{\theta}_c}}, \tag{3.14}$$

where $y_{ic'}$ refers the the c'-th component of the one-hot encoded vector $\boldsymbol{y}_i$. This gives us the likelihood function

$$L = P(\mathcal{D}|\boldsymbol{\theta}) = \prod_{i=1}^{N}\prod_{c'=0}^{C-1} \left[P(y_{ic'} = 1|\boldsymbol{x}_i, \boldsymbol{\theta}_{c'})\right]^{y_{ic'}} \left[1 - P(y_{ic'} = 1|\boldsymbol{x}_i, \boldsymbol{\theta}_{c'})\right]^{1-y_{ic'}}. \tag{3.15}$$

Again we do not wish to maximize this function, but rather minimize the negative log-likelihood

$$l = -\sum_{i=1}^{N}\sum_{c'=0}^{C-1} y_{ic'} log\left[P(y_{ic'}=1|\boldsymbol{x}_i,\boldsymbol{\theta}_{c'})\right] + (1-y_{ic'}) log\left[1-P(y_{ic'}=1|\boldsymbol{x}_i,\boldsymbol{\theta}_{c'})\right]. \tag{3.16}$$

We can see that for $C = 1$ this reproduces the cross-entropy for the sigmoid function.

3.5.4 Accuracy, Precision and Recall

The cost function evaluates how well the model is performing, and by minimizing this function, we can find the optimal combination of parameters. The cost is not a very intuitive measure of how well the model is doing, though. We need to use a cost function for training the model, but we can create other measures to make it easier to interpret the model results. One such measure is the *accuracy* of the model. For classification tasks, we can simply calculate the percentage of correctly classified data points, which is easy for humans to interpret. The accuracy in itself is not a perfect measure of the performance as the cost may still be improving even if the accuracy stays the same. This is because with a soft classifier we might not always be improving the cost by moving the decision boundary in a way that reclassifies misclassified data points. Instead, an improvement might be a movement of the boundary in such a way that the algorithm is more certain of the points which are correctly classified. If the cost has improved, the model has become a better classifier even though the accuracy stays the same.

The accuracy is an easy way to interpret the classification results, however, for a classification problem with few classes, it is not always to be trusted. Imagine we have a heavily skewed binary dataset with responses of 90 percent 0s and 10 percent 1s. If our algorithm guesses everything to be zeros, we get an accuracy of 90 percent. This looks like a good result, but in reality, the algorithm is useless as no prediction has taken place at all. Let us say that for this type of dataset, we really care about the classifiers ability to predict the least prevalent class. A better measure of the performance would then be the *recall.* The recall is defined as the ratio of true positive classifications of a certain class over the possible correct classifications of this class.

$$\text{Recall} = \frac{\sum TP}{\sum TP + FN}, \tag{3.17}$$

where TP are true positives and FN are false negatives. One would typically calculate the recall of the least represented class in the dataset. In the previous example, where all samples were classified into class 0, the recall of class 1 would be 0, which is a terrible classifier if this is a class we want to be classified correctly.

If we imagine the same dataset as before, only this time we are not as concerned with the classifiers ability to predict the 1s; we mostly care that the data points actually classified as 1s are indeed 1s. In this case, we would use the *precision* performance measure. The precision is defined as the ratio of true positive classifications of a certain class over the total classifications into this class.

$$\text{Precision} = \frac{\sum TP}{\sum TP + FP}, \tag{3.18}$$

where TP are true positives and FP are false positives. If we have a very high precision of a certain class, we can be reasonably certain that the classifier will not misclassify any points of other classes into this one. Using the skewed dataset as before and evaluating the 0s class, this would give a precision of 90 percent.

In Figure 3.6, we can see the relationship between the predicted labels and correct labels of the data points.

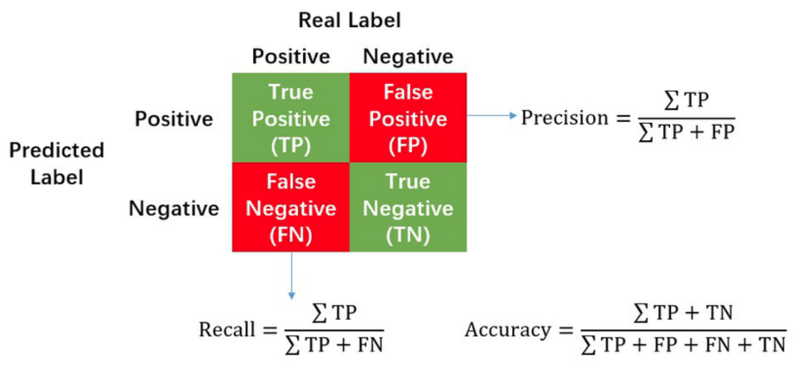


Figure 3.6: Illustration of accuracy, precision and recall. Figure reprinted from Ma et al. [43].

3.5.5 Gradient Descent

As mentioned in the previous sections, no matter what algorithm we are using we always want to find a set of parameters which minimizes some cost function $\mathcal{C}(\boldsymbol{X}, g(\boldsymbol{\theta}))$. For complicated models this is not necessarily an easy task. Especially for neural networks with several hidden layers, this becomes very complicated as the cost function depends on the parameters of the entire network. For a non-convex cost function, there might also be several local minima which makes it even more difficult to find the optimal global minimum of the function.

A powerful method for finding the minimum of the cost function is to use so-called *gradient descent*. The final input to the cost function will be the output from the final layer, the final layer depends on the output of the previous layer, and so on. Hence, when minimizing the cost function, we are minimizing it as a function of all the previous layers. This is a complicated function in potentially high dimensional space, which makes it hard to optimize. In gradient descent, we take advantage of the fact that all the layers are nested, and so the gradient will be a simple chain-rule calculated from the cost-function and all the way back to the first layer.

Gradient descent is an iterative approach to finding the minimum of the cost function. We find the derivative of the cost function with respect to the parameters and iteratively move in the direction of the negative gradient

$$\boldsymbol{\theta}^l_{t+1} = \boldsymbol{\theta}^l_{\boldsymbol{t}} - \mu \nabla_{\theta^l_t} \mathcal{C}(\boldsymbol{X}, g(\boldsymbol{\theta}^l_t)) \tag{3.19}$$

where $\nabla_\theta \mathcal{C}(\boldsymbol{X}, g(\boldsymbol{\theta}))$ is the gradient of the cost function at iteration t and μ is a hyperparameter called the *learning rate.* The learning rate controls how fast we move in the direction of the negative gradient, towards the minimum. Choosing an appropriate learning rate is paramount for converging towards the correct minimum. Choosing the learning rate to small, Figure 3.7a, increases the training time and for non-convex multidimensional space increases the chances of getting stuck in local minima. With too large learning rate, Figure 3.7c and d, we bounce around in the data space and can even start moving away from the minimum. For just the right training rate, we move more or less directly towards the global minimum. In practice, the learning rate is rarely a fixed constant. Several ways of updating the learning rate exist, but we will focus on one of the most widely used updating algorithm called ADAM [37].

Several algorithms for updating the learning rate utilizes momentum in their calculations. The idea behind this is that by combining the gradient at a current point with a constructed "velocity" depending on the gradient from the previous steps, we can make a smoother descent to the minimum as the direction of the gradient used when updating the weights are also dependent on the direction of the velocity, and as an extension, on the direction of the previous gradients. When using momentum we might also be able to avoid getting stuck in local minima because the size of the learning rate also depends on the previous gradients. The ADAM optimizer adapts the learning rate by storing an exponentially decaying running average of the gradients and squared gradients. This is often referred to as the first and second momenta, m_t and v_t, respectively. These parameters are initialized to zero, which makes them biased towards zero, especially during the first iterations. To counteract this bias, the running averages are divided by a bias-correcting term. The update rule is defined by

$$\begin{aligned}
m_0 &= 0 && \leftarrow \text{Initialize,} \\
v_0 &= 0 && \leftarrow \text{Initialize,} \\
m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t && \leftarrow \text{First momentum,} \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 && \leftarrow \text{Second momentum,} \\
m'_t &= \frac{m_t}{1-\beta_1^t} && \leftarrow \text{Bias correction,} \\
v'_t &= \frac{v_t}{1-\beta_2^t} && \leftarrow \text{Bias correction,} \\
\theta_{t+1} &= \theta_t - \frac{\mu}{\sqrt{v'_t} + \varepsilon} m'_t && \leftarrow \text{Update weights,}
\end{aligned} \tag{3.20}$$

where g_t is the gradient with respect to the parameters at iteration t. The decay parameters β_1 and β_2 should both be close to, but less than 1. Because the bias is greatest shortly after initialization the bias correction is constructed to decay for each iteration. The ε in the denominator of the final update of the weights is just a small number to avoid division by zero. The authors of ADAM proposed default values for the decay rates $\beta_1 = 0.9$ and $\beta_2 = 0.999$ which in most cases will give good learning outcomes, and because the learning

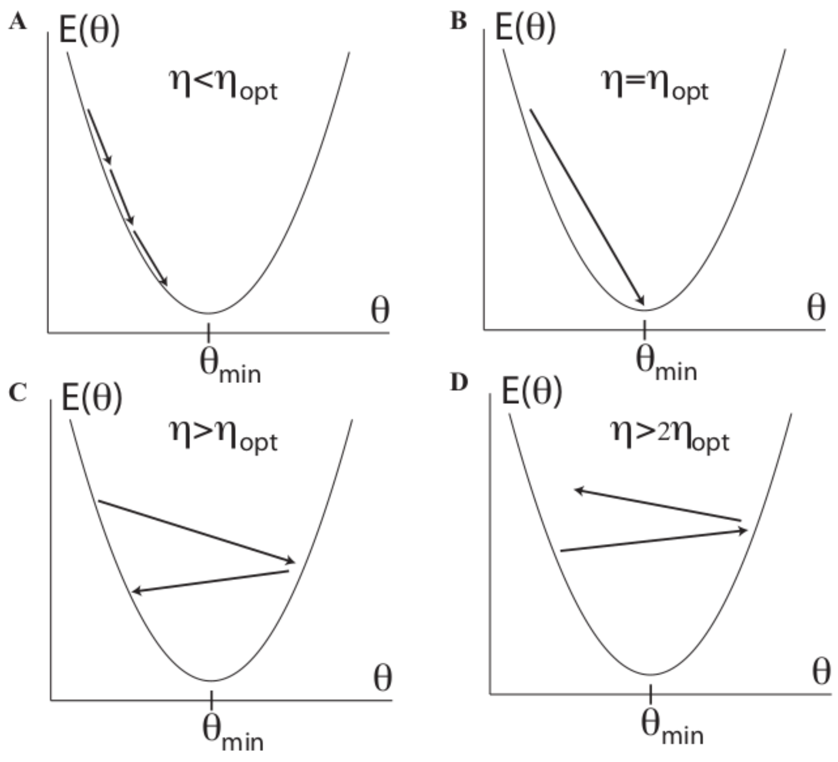


Figure 3.7: Illustration of the importance of choosing the correct learning rate for efficient learning. Too small learning rate gives slow convergence (A), and too high learning rate causes the descent to bounce around and can even move away from the minimum (C and D). With a correct learning rate we move directly towards the minimum(B). Figure reprinted from Mehta et al. [47].

rate automatically adapts throughout the training process we most likely do not need to do a parameter search for the learning rate at all.

Calculating the gradient for a large machine learning network can be quite computationally expensive and in practice we use what is called stochastic gradient descent with mini-batches. A minibatch is a subset of the training data used to approximate the gradient for the entire dataset. Instead of only updating the weights once for every complete run through the dataset, we now update the weights for every minibatch. This approach greatly decreases convergence time for high dimensional datasets. This stochastic approach also makes the gradient descent a bit more erratic, which is not necessarily a negative consequence as the stochasticity will make it less susceptible to getting stuck in isolated local minima.

3.5.6 Activation functions

The cost functions defined in the Section 3.5.3 depended on an activation function in the output layer which transformed the input to a probability. For a neural network, we are using several hidden layers in the network architecture, which all have activation functions applied to them. The activation functions applied to hidden layers do not have to be transformed into probabilities, and in principle, we can use any continuous function we want. There is, however, a few functions which have been proven to perform particularly well in neural networks, and we will go through them in this section.

The most important purpose of an activation function is to induce non-linearities into the network. Without these non-linearities, there is no point in using several layers in the network, as we could simply rewrite the weights to a single layer

$$\begin{aligned} y &= W_1(W_0x + b) + b_1 \\ &= W_1W_0x + W_1b_0 + b_1 \\ &= Wx + b, \end{aligned} \tag{3.21}$$

where $W = W_1W_0$ and $b = W_1b_0 + b_1$. By using a non linear activation function this transition is no longer possible and the network can learn different things in different layers.

There is not a single best activation function, and in theory, any non-linear function will suffice. There are however a few things to keep in mind when choosing an activation function, which will be illustrated through the most popular choices of activation functions.

Sigmoid

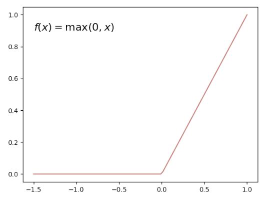
The sigmoid function was mentioned in Section 3.5.3 as an activation function applied to the output layer. This function can be used anywhere in the network, not just in the output layer. The sigmoid constrains the output between 0 and 1, which was a desirable trait when we wanted to transform the input into probabilities. There is a problem with this behavior, though. We want to minimize the cost function with respect to all the weights of the network and the gradient of the sigmoid function becomes saturated as it is close to flat in the asymptotic domain. The sigmoid is also not zero centered, which is not desirable because the output will always be positive. Because of this, the gradient of the output in a layer will all be either positive or negative. This could introduce a zig-zag pattern in the gradient descent, which is not optimal. This is not a significant complication, only an inconvenience as it makes training less accurate.

Hyperbolic tangent

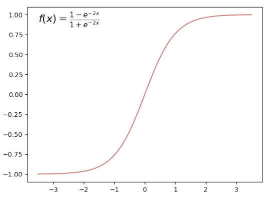
The hyperbolic tangent (tanh) constrains the output between -1 and 1 which solves the problem of non-zero centering we encounter with the sigmoid function. However, we still have the problem of saturated gradients as the gradient still becomes approximately zero as we approach the asymptote.

Rectifier Linear Unit

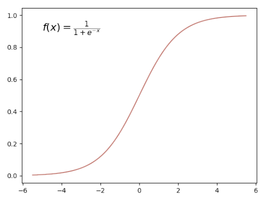
Rectifier linear unit (ReLU) has become the go-to activation function for deep neural networks as it has been shown to greatly accelerate the convergence of the neural network [39], which is argued to be because of the non-saturating shape. It is, however, not without issues. If, during training, the gradient of a neuron reaches the negative domain, the neuron will “die” and never activate again as the gradient from this point on will always be zero and never update. This issue can largely be mitigated by using a proper learning-rate during backpropagation.



(a) Relu



(b) Hyperbolic tangent



(c) Sigmoid

Figure 3.8: Figure of the Relu (a), Hyperbolic tangent (b) and Sigmoid (c) activation functions.

3.5.7 Initialization

Even though Rosenblatt introduced the perceptron already in 1958, the depth of the neural networks was, for a long time, quite shallow. A reason for this was a common problem of unstable gradients. Gradients for randomly initialized weights tended to vanish as the network got deeper, thus the training process would halt. In 2010 Glorot and Bengio [20] examined the output variance from layers using sigmoid and tanh activation functions and found that initializing the weights using the standard normal or uniform distribution resulted in the variance of the output within a layer to increase from layer to layer resulting in saturated learning. They proposed using a normalization scheme for initializing the weights, often called Xavier initialization, which lets the variance of the random weights drawn during initialization depend on the size of the incoming and outgoing layer sizes. This keeps the variance approximately constant throughout the network, and as a result, the chance of saturation is greatly reduced. Their reasoning in proposing the normalization scheme was to make sure the variance of the input and output of a layer would be as close as possible, and the same

Table 3.1: Initialization schemes

Init type	Uniform [-r, r]	Normal
Xavier	$r = \sqrt{\frac{6}{n_{inputs} + n_{outputs}}}$	$\sigma = \sqrt{\frac{2}{n_{inputs} + n_{outputs}}}$
He	$r = \sqrt{2}\sqrt{\frac{6}{n_{inputs} + n_{outputs}}}$	$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{inputs} + n_{outputs}}}$

should be true for the variance of the gradient during backpropagation. They found it is not possible to guarantee both, unless the number of nodes for the input and output are the same, and using an average between the outgoing and incoming layer sizes was the best compromise. In 2015 He et al. [26] proposed a similar scheme which extended the work of Glorot and Bengio to include the relu activation function. Both schemes can be seen in Table 3.1.

3.5.8 Running the Neural Network

The input to a neural network consist of a set of data point with each data point consiting of N features $\boldsymbol{x}^T = [x_0, x_1, \cdots, x_N]$. The output from the first layer will be

$$a_j^1 = f^1(z_j^1) = f^1\left(\sum_{i=1}^{N^0} \theta_{ij}^1 x_i + b_j^1\right), \tag{3.22}$$

where the j refers to individual neurons in current layer, i to the neurons in the previous layer, the superscript refers to the layer as a whole and f is some activation function The input to the second layer will now be

$$\begin{aligned} a_j^2 = f^2(z_j^2) &= f^2\left(\sum_{i=1}^{N^1} \theta_{ij}^2 a_i^1 + b_j^2\right) \\ &= f^2\left(\sum_{i=1}^{N^1} \theta_{ij}^2 f^1\left(\sum_{k=1}^{N^0} \theta_{ik}^1 x_k + b_i^1\right) + b_j^2\right). \end{aligned} \tag{3.23}$$

This can be generalized for any layer l

$$a_j^l = f^l(z_j^l) = f^l\left(\sum_{i=1}^{N^{l-1}} \theta_{ij}^l a_i^{l-1} + b_j^l\right), \tag{3.24}$$

where z_j^l is the linear weighted sum

$$z_j^l = \sum_i \theta_{ij}^l a_i^{l-1} + b_j^l. \tag{3.25}$$

3.5.9 Backpropagation

The actual implementation of gradient descent is done through what is known as backpropagation. Computing the derivative of the cost function with respect to all the parameters is an immense computational task. Fortunately, we can take advantage of the structure of the network defined by Equation 3.24. The cost function $\mathcal{C}$ depends directly on the output from the last layer $\boldsymbol{a}^L$, which in turn is dependent on the previous layers, making the cost function indirectly dependent on all the layers of the network. Utilizing this fact by using the chain rule, we can implement a set of equations for the backpropagation algorithm. We assume the network consists of L layers where we use $l = 1, \cdots, L$ for indexing the individual layers. We also denote θ^l_{ij} as the weights connecting the i-th neuron in layer $l-1$ to the j-th neuron in layer l. The output of a layer l is defined as in Equation 3.24. We start by defining what is referred to as the error of the j-th neuron in the l-th layer, denoted as δ^l_j. This error is simply the change in the cost function with respect to the input z^l_j.

$$\delta^l_j = \frac{\partial \mathcal{C}}{\partial z^l_j} = \frac{\partial \mathcal{C}}{\partial a^l_j}\frac{\partial a^l_j}{\partial z^l_j} = \frac{\partial \mathcal{C}}{\partial a^l_j} f'(z^l_j) \tag{3.26}$$

where $f'(z^l_j)$ is the derivative of the activation function f with respect to the weighted input sum.

The error function can also be interpreted as the derivative of the cost function with respect to the bias

$$\delta^l_j = \frac{\partial \mathcal{C}}{\partial z^l_j} = \frac{\partial \mathcal{C}}{\partial b^l_j}\frac{\partial b^l_j}{\partial z^l_j} = \frac{\partial \mathcal{C}}{\partial b^l_j} \tag{3.27}$$

where the last term is 1, as is evident from the definition of z^l_j in equation 3.25. With these definitions in place we can define the backpropagation equations using the chain rule.

$$\begin{aligned}\delta^l_j &= \frac{\partial \mathcal{C}}{\partial z^l_j} \\ &= \sum_i \frac{\partial \mathcal{C}}{\partial z^{l+1}_i}\frac{\partial z^{l+1}_i}{\partial z^l_j} \\ &= \left(\sum_i \delta^{l+1}_i \theta^{l+1}_{ij}\right) f'(z^l_j)\end{aligned} \tag{3.28}$$

and finally we define the gradient of the cost function with respect to the weights $\boldsymbol{\theta}^l$

$$\frac{\partial \mathcal{C}}{\partial \theta^l_{jk}} = \frac{\partial \mathcal{C}}{\partial z^l_j}\frac{\partial z^l_j}{\partial \theta^l_{jk}} = \delta^l_j a^{l-1}_k \tag{3.29}$$

By applying these equations we say that the gradient is flowing down the network as we are starting from the last layer and utilizing the gradient of the previous layer to calculate the gradient in the next. By combining the gradient with a learning rate μ we now know how much to change each and every weight in the network according to the gradient descent algorithm.

One complete forward and backward pass through the entire dataset is called an *epoch*.

3.5.10 Regularization in Neural Networks

Deep neural networks are prone to overfitting, and because of this we want to add regularization to the network like we did with linear regression. We can regularize neural networks in the same way by adding an L_1 or L_2 term to the cost function. Another option is to use so-called *dropout* in the network. Dropout was introduced by Srivastava et al. [70] in 2014, and the technique is to randomly ignore some neurons during training time. That is, for each training sample, randomly choose some neurons which are not included in the forward and backward pass, shown in Figure 3.9. The idea is that fully-connected layers develop a high co-dependency between each other which leads to overfitting, and by randomly dropping some of the neurons each iteration we are forcing the layers to not depend as highly on the previous layers of the network. This can be interpreted as a sampling of the neural network within the complete neural network where only the parameters of the sampled network is updated. The probability of a neuron being dropped is known as the dropout rate. When we drop some neuron during training time, the output from this layer will be smaller than if the complete layer was run. During testing of the network, the full network is run without the dropout. Because of this, it is essential to scale the output from the layers where dropout was applied with the dropout rate. This scaling can either be done during testing time by multiplying the output with the dropout rate which scales down the output or, more popularly, during training time by dividing the output by the dropout rate which scales up the output. Dropout has become a very popular technique due to its effectiveness but also because it reduces the computation time of the network as opposed to L_1 and L_2 regularization which adds an additional term to the cost function, effectively increasing the computation time.

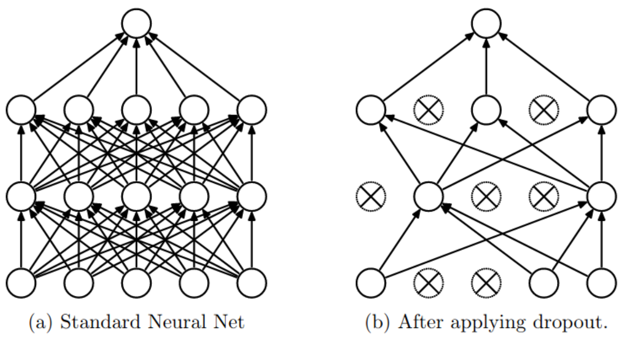
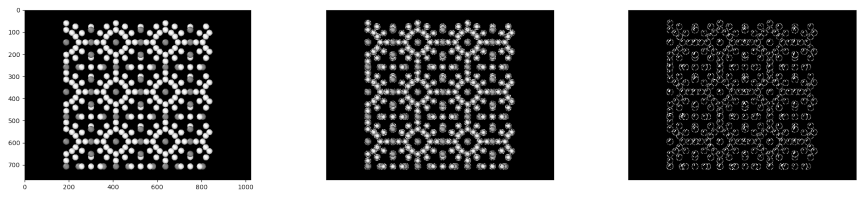


Figure 3.9: Fully-connected network without dropout (a) and with dropout (b). The crossed out neurons of (b) are the dropped units. Figure reprinted from Srivastava et al. [70].

3.6 Convolutional Neural Networks

Convolutional Neural Networks (CNN) are closely related to fully-connected neural networks. Nevertheless, there are some clear differences in the two architectures. CNNs use small weight matrices, often called kernels, to act as filters on the input data. This approach takes advantage of highly structural input data, like images, effectively finding local areas of interest in the input. If we imagine the input being an image, we can manually construct a filter that looks for edges (Figure 3.10) in the input, and by convolving the image with the filter, reveal hidden information in the data. This is essentially what a con-



(a) (b) (c)

0	-1	0
-1	5	-1
0	-1	0

0	-1	0
-1	4	-1
0	-1	0

Figure 3.10: a: Gray scale picture of methane hydrate sI.
b: Sharpened version of figure (a) convolved with the matrix below.
c: Edge filter applied to figure (a), using the matrix below. Only the edges of the atoms remain in the picture.

volutional network is doing, only we do not predefine what the filter is going to look for, this is done through the training of the network weights. The most important layers in a CNN are these convolutional layers, with small weight matrices acting like filters. These matrices are usually small, with standard sizes being 3×3, 5×5, and 7×7. Unlike fully-connected neural networks which only use one weight matrix per layer, a convolutional layer can stack several of the filter's outputs, often called feature maps, in a third dimension, as shown in Figure 3.11. Because of this, the filters must also have a third dimension corresponding to the depth dimension of the input.

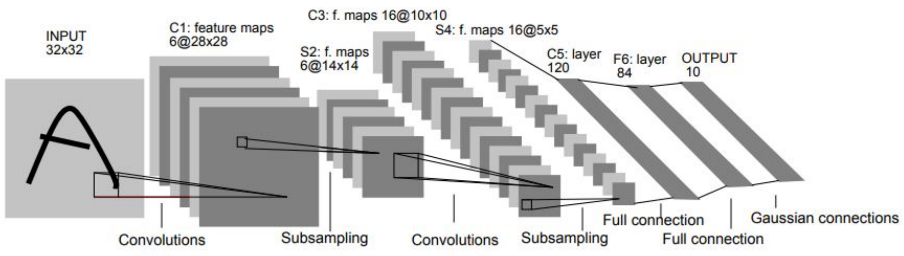


Figure 3.11: Example of a convolutional network. Reprinted from LeCun et al. [41].

The rationale for stacking the feature maps is that different filters within a layer may find different types of information in an image, like blobs, corners, edges, etc., and by stacking the output from these filters, we can collect more information in each layer.

The mathematical definition of discrete convolution is

$$O(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{3.30}$$

where O is the output from the convolution, often called the feature map, I is the image and K is the kernel. This function is commutable and can equivalently be written as

$$O(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n) \tag{3.31}$$

When viewing I and K as two-dimensional matrices, we see that we may encounter negative indices. We will discuss how this is handled in practice but because of this, the latter equation is easier to implement in a machine learning algorithm as there is less variation in the range of valid values of the kernel indices. When doing convolution, we are essentially sliding a filter over the image, taking the dot product between the overlapping image pixels and filter values. In the correct mathematical definition of convolution above, we are essentially flipping the kernel along both axes before sliding it over the image. In reality, many CNNs utilize a function called the cross-correlation which is defined as

$$O(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n) \tag{3.32}$$

We can view this equation as the same as convolution, only without flipping the kernel, and in machine learning we rarely differentiate between the two methods. We are not predefining the filter weights as they are something the network should learn through training. Because of this, it does not really matter if we flip the kernel in the convolution. If the kernel is flipped, the network will just learn the mirror opposite.

When performing a convolution, shown in Figure 3.12a, the dimensions of the output is smaller than the original image. This might not be desirable, and we solve this by adding zero-padding to the edges of the input image. To calculate the size of the output after a convolution, we use the formula

$$\frac{W - K + 2P}{S}, \tag{3.33}$$

where W is the width or height of the image. If they are different, the width and height of the output will also be different and must be independently calculated. K is the size of the kernel, P is the padding applied to the image, and S is the stride. The stride is defined as how many pixels we are moving for each convolution. One important thing to note is that the output size must be an integer. If we are using a combination of kernel, padding and stride that does not "fit" across the input image, the output will be skewed, which is not desirable. In Figure 3.12a, the stride $S = 1$ is used with no padding $P = 0$. We see this combination fits across the image, but with no padding, the output size is reduced by one in each dimension. If we had used a stride of $S = 2$, we would skip the second output in the x-dimension, reducing the output size by two. The kernel would not fit in the y-dimension, however, which we do not want to do. Hence, this is a combination we would not use. We are generally not concerned with the depth-axis of the input as the filters will always traverse the entire volume of the depth-axis resulting in a two-dimensional feature map. To calculate the padding needed to keep the input dimension, we require equation 3.33 to be the same size as the input. Solving for the padding yields

$$P = (W - 1)S - W + K. \tag{3.34}$$

We need to keep in mind that the padding adds a loss in resolution of the output at the edges. If the image is of a sufficiently large resolution, this is rarely a problem, though.

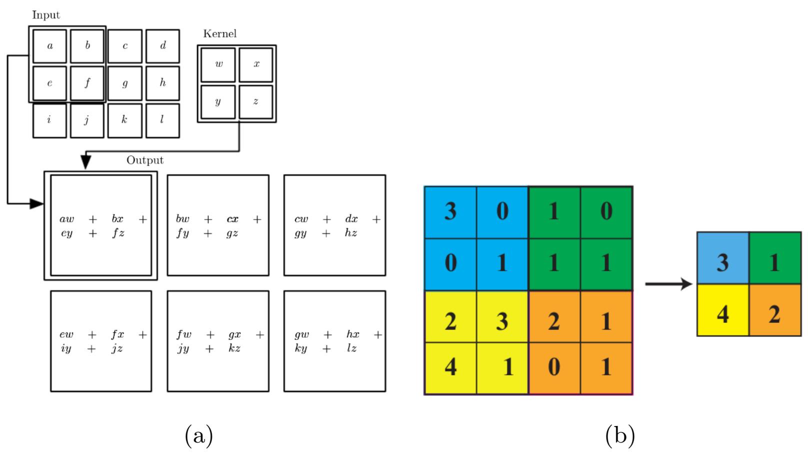


Figure 3.12: The process of cross-correlating an image with a filter (a). Reprinted from Ian Goodfellow, Yoshua Bengio [36]. Example of a 2×2 max-pooling layer. Reprinted from Mehta et al. [47].

In fully-connected networks, we are generally trying to reduce the size of the output as we are moving through the network, and the same is true for a convolutional network. A way to do this might be to not add any zero-padding to the input or increase the stride, and in this way reduce the output size layer by layer. A more common way to do this is to use pooling layers. There are several types of pooling layers, but the most common ones are max-pooling and average pooling. Pooling layers are most often a 2×2 filter which slides over the image without any overlapping between the filter computations, this means the stride equals the size of the pooling filter. A 2×2 filter is most common because this reduces the input size by half in each dimension, and unless the input image is extremely large, the size will quickly decrease for every pooling layer added. A max-pooling filter, illustrated in Figure 3.12b, finds the maximum value of the input image for each computation. The idea behind this is that we can view the maximum value as the most important feature in the input and by using this value for further computations, we are removing the noise and keeping the most relevant information. The drawback of this pooling method is that if the inputs are very close in values, we are removing a lot of relevant information. The average-pooling method is essentially the same idea as max-pooling, only now we take the average over the filter instead of the maximum value. This ensures all inputs are included in the pooling operation, but this will also include the noise, unlike with max-pooling.

3.7 PCA

Principal Component Analysis (PCA) was proposed by Karl Pearson in 1901 [56] and is used in data analysis as a data transformation and reduction technique. PCA performs a linear projection of the data points into a lower-dimensional space. A common observation on inspection of datasets is that the bulk of relevant information is often contained in the directions of largest variance. This is illustrated in Figure 3.13 where the axis containing the least amount of variance is interpreted as noise. We consider a dataset in $N \times D$ dimensional space

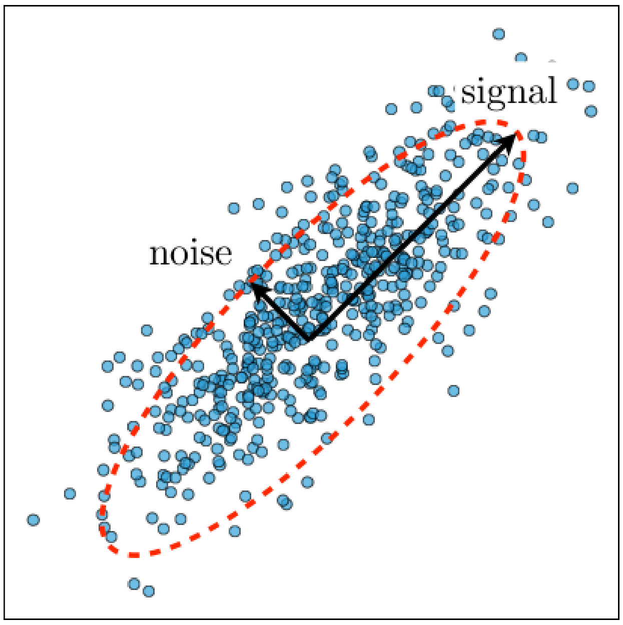


Figure 3.13: Principal component analysis seeks to find a set of orthogonal axes where the first axis carries the most amount of variance, the second axis the second most variance, and so on. This is illustrated in this figure as the most variance is in the direction of the axis marked as "signal" and the least variance is in the direction marked as "noise". We can reduce the number of dimensions by projecting the data from the noise axis to the signal axis and hopefully, not lose much of the relevant information. This figure only shows two dimensions, but the same argument holds for multiple dimensions. Figure reprinted from Mehta et al. [47].

$\boldsymbol{X} = [\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots, \boldsymbol{x}_N]^T$ where $\boldsymbol{x}_i \in \mathbb{R}^D$ is a single data point in D-dimensional space centered around the empirical mean. The covariance is calculated by

$$\Sigma(\boldsymbol{X}) = \frac{1}{N-1}\boldsymbol{X}^T\boldsymbol{X}. \tag{3.35}$$

We are looking for a linear transformation which maximizes the covariance between the feature vectors $\boldsymbol{x}_i$. To do this we make use of the Singular Value Decomposition [72] which states that a rectangular matrix can be decomposed into three matrices: A $N \times D$ diagonal matrix $\boldsymbol{S}$ containing the singular values s_i, that is, the square roots of the eigenvalues of $\boldsymbol{X}^T\boldsymbol{X}$ and $\boldsymbol{X}\boldsymbol{X}^T$, an orthonormal $N \times N$ matrix $\boldsymbol{U}$ containing the eigenvectors of $\boldsymbol{X}\boldsymbol{X}^T$ called the left singular vectors and an orthonormal $D \times D$ matrix $\boldsymbol{V}$ containing the eigenvectors of $\boldsymbol{X}^T\boldsymbol{X}$ called the right singular values

$$\boldsymbol{X} = \boldsymbol{U}\boldsymbol{S}\boldsymbol{V}^T. \tag{3.36}$$

We now insert this into equation 3.35

$$\begin{aligned}
\Sigma(\boldsymbol{X}) &= \frac{1}{N-1}\boldsymbol{X}^T\boldsymbol{X} \\
&= \frac{1}{N-1}\boldsymbol{V}\boldsymbol{S}\boldsymbol{U}^T\boldsymbol{U}\boldsymbol{S}\boldsymbol{V}^T \\
&= \boldsymbol{V}\frac{\boldsymbol{S}^2}{N-1}\boldsymbol{V}^T \\
&\equiv \boldsymbol{V}\boldsymbol{\Lambda}\boldsymbol{V}^T,
\end{aligned} \tag{3.37}$$

where $\boldsymbol{\Lambda}$ is a diagonal matrix with eigenvalues in decreasing order along the diagonal and the columns of $\boldsymbol{V}$ being the principal directions of $\boldsymbol{\Sigma}(\boldsymbol{X})$. To reduce the dimensionality of $\boldsymbol{X}$ while keeping the maximum amount of variance we simply multiply $\boldsymbol{X}$ with the columns of $\boldsymbol{V}$ corresponding to the number of principal axes we want to keep.

3.8 Clustering

Supervised learning is a powerful tool in classification, but it has a large caveat; we depend on acquiring a labeled dataset before any training is possible. This is often done manually, which is a tedious and repetitive task. Unsupervised learning is self-organizing and is of great use in finding unknown patterns in the data without pre-existing labels. One type of unsupervised learning is called *clustering*. The goal of clustering is not unlike classification, where we are trying to group the data into different classes. The difference is we do not predefine what the correct class or cluster of a data point is in advance. Instead, we are evaluating the data using distance and density measures to infer if any of the data points seem to be clustered together in the feature space.

3.8.1 K-means Clustering

The most readily understood clustering algorithm is so-called *k-means clustering*. It is not a particularly powerful algorithm, but it is a nice way of building intuition about clustering as it is relatively simple. Even though k-means clustering in practice is mostly used for initializing other, more powerful algorithms, it also has its uses as a stand-alone algorithm as the computational speed scales linearly.

We assume a dataset of N unlabelled data points $\{\boldsymbol{x}\}_{n=1}^{N}$ where $\boldsymbol{x}_n \in \mathbb{R}^p$. We also assume the dataset can be partitioned into a set of clusters with cluster centers $\boldsymbol{\mu}_k$, where $\boldsymbol{\mu}_k \in \mathbb{R}^p$. In the simplest form of the algorithm, we randomly place the cluster centers in the data domain and we seek to minimize the following function

$$\mathcal{C}\left(\boldsymbol{x}, \boldsymbol{\mu}\right) = \sum_{k=1}^{K} \sum_{n=1}^{N} r_{nk} \left(\boldsymbol{x}_n - \boldsymbol{\mu}_k\right)^2, \tag{3.38}$$

where $r_{nk} \in 0, 1$ is a binary variable defining if a data point belongs to a given cluster k. The algorithm is as follows:

1. Randomly place cluster centers $\boldsymbol{\mu}_k$.

2. Find r_{nk} which minimizes $\mathcal{C}$. This is done by assigning each data point to the nearest cluster-mean.

$$r_{nk} = \begin{cases} 1 \text{ if } k = \text{argmin}_{k'} (\boldsymbol{x}_n - \boldsymbol{\mu}_{k'})^2 \\ 0 \text{ else} \end{cases}. \tag{3.39}$$

3. Update $\boldsymbol{\mu}_k$ by calculating the center of mass of the assigned data points

$$\boldsymbol{\mu}_k = \frac{1}{N_k} \sum_n r_{nk} \boldsymbol{x}_n. \tag{3.40}$$

4. Repeat 2. and 3. until the change in the cost function is less than a predefined threshold.

An example of the k-means algorithms is shown in Figure 3.14

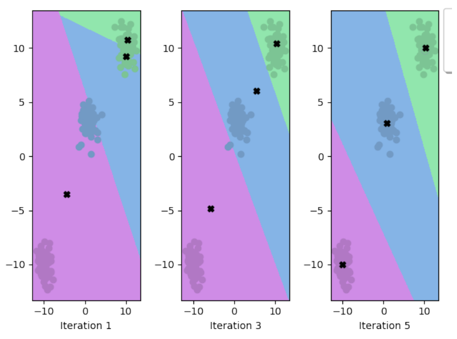


Figure 3.14: Perfoming k-means clustering on three random data clusters over five iterations. The black crosses are the cluster centers. In iteration one they are randomly placed, and through the iterations they are gradually moved towards the idividual clusters.

3.8.2 Gaussian Mixture Models

In unsupervised learning, we often speak of the concept of a latent variable, which is a hidden variable inferred from the dataset. Just like in many physical models, we might not be able to directly observe the construct we are trying to understand. Instead, we observe the effect of the latent variable through observed variables and try to build models explaining the behavior of the latent variable through mathematical and numerical concepts. Finding exoplanets is an example of this in physics. Scientists are not able to directly observe planets around distant stars, however they are able to infer their existence and their properties by observing the influence the planets have on the star they are orbiting.

The estimation of latent variables is done by analyzing the statistical properties of the observed variables. In clustering, the latent variable is often thought of as the cluster identity of which a data point initially stems. In this context, we think of clustering as a probability algorithm for learning the most likely value of a latent variable associated with every data point. To calculate this probability, we need to make broad assumptions on the dataset, like what kind of distributions generated the data. In Gaussian mixture models, we assume these distributions are Gaussians, and we predefine how many clusters we think the dataset contains.

K-means clustering is actually just a special case of Gaussian mixture models (GMM) where a hard boundary is set between the clusters, and no variance or covariance is taken into account. The general GMM algorithm, on the other hand, is referred to as a soft clustering method as there might be overlapping boundaries between clusters, and a probability is used to predict to which cluster a data point belongs. We assume each cluster in the dataset is characterized by a specific probability distribution with a specific mean $\boldsymbol{\mu}_k$ and covariance $\boldsymbol{\Sigma}_k$ where we try to maximize the probability of retrieving the observed dataset under a generative model. Points are drawn from one of K Gaussian distributions, each defined by

$$\mathcal{N}\left(\boldsymbol{x}, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k\right) \equiv \exp\left(-(\boldsymbol{x}-\boldsymbol{\mu}_k)\boldsymbol{\Sigma}_k^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_k^T/2\right). \tag{3.41}$$

The probability for a point to be drawn from a distribution k is normal to denote by π_k and is often referred to as mixing components. The probability of generating a point $\boldsymbol{x}_i$ from the Gaussian distributions are given by the sum of the K distribution probabilities

$$p\left(\boldsymbol{x}_i; \{\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k, \pi_k\}\right) = \sum_{k=1}^{K} \mathcal{N}\left(\boldsymbol{x}_i|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k\right)\pi_k, \tag{3.42}$$

with a dataset of i.i.d data points $\boldsymbol{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots, \boldsymbol{x}_N\}$ we can write the complete probability, or likelihood, for generating the dataset as the product of the individual probabilities of each data point

$$p\left(\boldsymbol{X}; \{\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k, \pi_k\}\right) = \prod_{i=1}^{N} p\left(\boldsymbol{x}_i|\left\{\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k, \pi_k\right\}\right). \tag{3.43}$$

We now introduce a K-dimensional latent variable $\boldsymbol{z}$ for each data point $\boldsymbol{x}_i$. We construct $\boldsymbol{z}$ to be a so-called one-hot-encoded variable, which is a vector of binary values with a 1 if a point $\boldsymbol{x}_i$ was generated from the k-th Gaussian distribution and zero otherwise. In this way, we get distinct representations of K number of clusters. We denote the elements of $\boldsymbol{z}$ as z_k and there are K possible states of $\boldsymbol{z}$ according to which elements are non-zero, as shown in Equation 3.44.

$$\begin{aligned} \boldsymbol{z} &= (1, 0, \cdots, 0) \\ \boldsymbol{z} &= (0, 1, \cdots, 0) \\ &\vdots \\ \boldsymbol{z} &= (0, 0, \cdots, 1). \end{aligned} \tag{3.44}$$

We can collect all the latent variables in a single one-hot-encoded matrix corresponding to the N data points of $\boldsymbol{X}$ denoted by $\boldsymbol{Z} \in \mathbb{R}^{N \times K}$.

The marginal distribution of the latent variables are given by the mixing components $p(z_k = 1) = \pi_k$, where π_k must satisfy

$$\begin{aligned} 0 \leq \pi_k \leq 1, \\ \sum_{k=1}^{K} \pi_k = 1. \end{aligned} \tag{3.45}$$

Because $\boldsymbol{z}$ can be in one of K states, we can write the distribution of all the possible states of the latent variables as

$$p(\boldsymbol{z}; \{\pi_k\}) = \prod_{k=1}^{K} \pi_k^{z_k}, \tag{3.46}$$

where z_k is the k-th component of the given $\boldsymbol{z}$, being one if $\boldsymbol{x}_i$ belongs to cluster k end zero otherwise.

Following the same logic the conditional probability of $\boldsymbol{x}_i$ given $\boldsymbol{z}$,

$$p(\boldsymbol{x}_i | z_k = 1) = \mathcal{N}(\boldsymbol{x}_i; \{\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k\}), \tag{3.47}$$

can be written as a product over the components of the latent variable

$$p(\boldsymbol{x}_i | \boldsymbol{z}; \{\boldsymbol{\mu_k}, \boldsymbol{\Sigma}_k\}) = \prod_{k=1}^{K} \mathcal{N}(\boldsymbol{x}_i; \{\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k\})^{z_k}. \tag{3.48}$$

As we do not know the values of the latent variables, what we are really trying to maximize is the likelihood of the latent variables under the given dataset,

$$p(\boldsymbol{Z}) = p(\boldsymbol{Z} | \boldsymbol{X}; \{\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k, \pi_k\}), \tag{3.49}$$

by finding the best values of $\{\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k, \pi_k\})$. We do this by using an iterative approach. From Bayes rule we have

$$p(\boldsymbol{z} | \boldsymbol{x}) = \frac{p(\boldsymbol{x} | \boldsymbol{z}) p(\boldsymbol{z})}{p(\boldsymbol{x})}, \tag{3.50}$$

which, by using the previous results gives

$$\gamma(z_k) \equiv p(z_k = 1 | \boldsymbol{x}_i; \{\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k, \pi_k\}) = \frac{\pi_k \mathcal{N}(\boldsymbol{x}_i | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} \mathcal{N}(\boldsymbol{x}_i | \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)}, \tag{3.51}$$

where γ is referred to as the responsibility a mixture k takes for explaining $\boldsymbol{x}_i$. The problem of calculating the GMM is that we do not know the values of the latent variables $\boldsymbol{z}$ and we also do not know the underlying parameters $\{\boldsymbol{\mu}_k, \boldsymbol{\Sigma_k}, \pi_k\}$ for any of the Gaussians. We have to infer all these variables from the dataset. To do this, we are now going to use the so-called expectation maximation (EM) algorithm, which iteratively calculates the maximum expectation value of the likelihood.

The joint probability likelihood is given by

$$\begin{aligned} p(\boldsymbol{X}, \boldsymbol{Z}; \{\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k, \pi_k\}) &= \prod_{i=1}^{N} p(\boldsymbol{x}_i | \boldsymbol{z}_i; \{\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k\}) p(\boldsymbol{z}_i | \{\pi_k\}) \\ &= \prod_{i=1}^{N} \prod_{k=1}^{K} \mathcal{N}(\boldsymbol{x}_i | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)^{z_k} \pi_k^{z_k}. \end{aligned} \tag{3.52}$$

Taking the expectation value of the log-likelihood gives

$$E\left[log(p(\boldsymbol{X}, \boldsymbol{Z}; \{\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k, \pi_k\}))\right] = \sum_{i=1}^{N} \sum_{k=1}^{K} \gamma_{ik}^{(t)} [log\mathcal{N}(\boldsymbol{x}_i | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) + log\pi_k], \tag{3.53}$$

where $\gamma_{ik}^{(t)} = p(z_{ik} | \boldsymbol{X}; \{\boldsymbol{\mu}_k^{(t)}, \boldsymbol{\Sigma}_k^{(t)}, \pi_k^{(t)}\})$ and (t) is the iteration step. This is the expectation step in the EM algorithm. We now take the derivative of this equation with respect to the parameters and setting this to zero yields the equations

$$\begin{aligned} \boldsymbol{\mu}_k^{(t+1)} &= \frac{\sum_i^N \gamma_{ik}^{(t)} x_i}{\sum_i \gamma_{ik}^{(t)}}, \\ \boldsymbol{\Sigma}_k^{(t+1)} &= \frac{\sum_i^N \gamma_{ik}^{(t)} (\boldsymbol{x}_i - \boldsymbol{\mu}_k)(\boldsymbol{x}_i - \boldsymbol{\mu}_k)^T}{\sum_i \gamma_{ik}^{(t)}}, \\ \boldsymbol{\pi}_k^{(t+1)} &= \frac{1}{N} \sum_k \gamma_{ik}^{(t)}, \end{aligned} \tag{3.54}$$

which is the maximation step.

By alternating the expectation and maximation step, we iteratively move towards the maximum of the expected log-likelihood.

3.8.3 Agglomerative Clustering

Agglomerative clustering is known as a bottom up hierarchical clustering method. This means we start from small initial clusters that are being merged until a desirable number of clusters remain. Two important parameters to decide when doing agglomerative clustering is the distance metric and the linkage criterion. The distance metric influences the shape of the clusters as using different metrics might yield different results for which clusters are closest together. In each iteration, the clusters that are closest in distance will be merged. In Table 3.2, we define some possible distance metrics.

The linkage criterion defines between what points we calculate the distance, e.g. between the center of the clusters, the edges of the clusters, etc. There are several possible linkage criterions. If we define the linkage criterion as $D(\boldsymbol{X}_i, \boldsymbol{X}_j)$ where $\boldsymbol{X}_i$ and $\boldsymbol{X}_j$ is two different clusters, and the distance metric as $d(\boldsymbol{x}_i, \boldsymbol{x}_j)$ with $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ as specific points situated in these clusters, we can define the most common linkage criterions as:

Table 3.2: Possible distance metrics used for agglomerative clustering.

Metric	Equation
Euclidian distance	$d(a,b) = \|\|a-b\|\|_2 = \sqrt{\sum_i (a_i - b_i)^2}$
Manhatten distance	$d(a,b) = \|\|a-b\|\|_1 = \sum_i \|a_i - b_i\|$
Cosine distance	$d(a,b) = \cos(\theta) = \frac{\boldsymbol{a}\cdot\boldsymbol{b}}{\|\|\boldsymbol{a}\|\|\|\|\boldsymbol{b}\|\|} = \frac{\sum_{i=1}^n a_i b_i}{\sqrt{\sum_{i=1}^n a_i^2}\sqrt{\sum_{i=1}^n b_i^2}}$

- Single-linkage: The minimum distance between two elements of different clusters.

$$D(\boldsymbol{X}_i, \boldsymbol{X}_j) = \min_{\boldsymbol{x}_i \in \boldsymbol{X}_i, \boldsymbol{x}_j \in \boldsymbol{X}_j} d(\boldsymbol{x}_i - \boldsymbol{x}_j) \tag{3.55}$$

- Complete linkage: The maximum distance between two elements in different clusters.

$$D(\boldsymbol{X}_i, \boldsymbol{X}_j) = \max_{\boldsymbol{x}_i \in \boldsymbol{X}_i, \boldsymbol{x}_j \in \boldsymbol{X}_j} d(\boldsymbol{x}_i - \boldsymbol{x}_j) \tag{3.56}$$

- Average linkage: Average distance between points in two different clusters.

$$D(\boldsymbol{X}_i, \boldsymbol{X}_j) = \frac{1}{|\boldsymbol{X}_i||\boldsymbol{X}_j|} \sum_{\boldsymbol{x}_i \in \boldsymbol{X}_i, \boldsymbol{x}_j \in \boldsymbol{X}_j} d(\boldsymbol{x}_i - \boldsymbol{x}_j) \tag{3.57}$$

- Ward: Minimizes the variance of the clusters. This is analagous to the K-means clustering algorithm. That is, we are minimizing the center of mass between the merged clusters and the individual clusters. Ward's method requires a euclidean distance to be used.

$$\begin{aligned} D(X_i, X_j) &= \sum_{k \in X_i \cup X_j}^{N_i + N_j} ||\boldsymbol{x}_k - \mu_{X_i \cup X_j}||^2 - \left(\sum_{i \in X_i}^{N_i} ||\boldsymbol{x}_i - \mu_{X_i}||^2 + \sum_{j \in X_j}^{N_j} ||\boldsymbol{x}_j - \mu_{X_j}||^2 \right) \\ &= \frac{|X_i||X_j|}{|X_i \cup X_j|} (\boldsymbol{\mu}_i - \boldsymbol{\mu}_j)^2 \end{aligned} \tag{3.58}$$

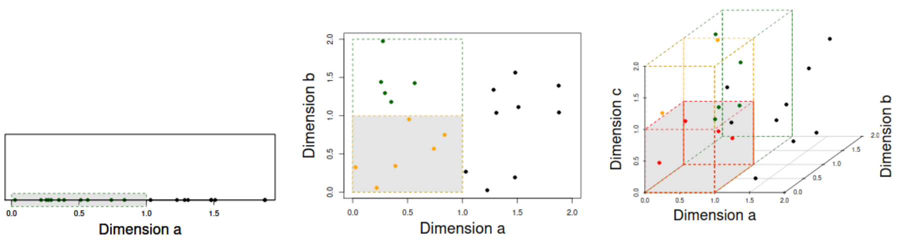
Unfortunately agglomerative clustering does not scale well computationally, as a distance matrix between all the clusters need to be computed at every iteration. A common way to circumvent this problem is to initialize the clusters with K-means using a large number of clusters (but still small compared to the number of points), and then use agglomerative clustering on the initialized clusters.

3.8.4 DBSCAN

Density-based spatial clustering of applications with noise (DBSCAN)[16] is, as the name implies, a density-based clustering method. Density clustering assumes clusters to be defined by regions of space with a higher density of data points. A significant advantage of density-based methods is that they are not concerned with the shape of the clusters and is also able to find outliers in the dataset. These methods assume a local density estimation is possible, which is probably a reasonable assumption in lower dimensions. If the number of dimensions is high, we encounter what is known as *The curse of dimensionality* [82]. The curse of dimensionality is an important aspect in all machine learning problems, including supervised frameworks and the previously defined clustering algorithms, but becomes very apparent in density-based clustering as the density of points is the only metric we consider. The curse states that as we increase the number of dimensions, the sparsity of the data will increase, as illustrated in Figure 3.15. To counter the curse, we are reliant on increasing the training data to keep the density high. To illustrate the problem mathematically, we consider a set uniformly distributed set of points in a d-dimensional unit hypercube. If we expand a hypercube about a given point to capture a fraction ρ of the points, we can calculate the fraction of points captured as a function of the edges L of the hypercube

$$\rho(L) = L^d \tag{3.59}$$

If we expand the hypercube to have edges 50% of the unit hypercube edge length in 2 dimensions we see that this will encompass $\rho = 0.5^2 = 25\%$ of the data points. Increasing the dimensionality to 10 we will, with the same edge length, only encompass 0.1% of the data points. If we still want to include 25% of the points, we now need to expand the hypercube to have edges 87% the length of the unit hypercube. We defined the density-based clustering methods as a method searching for local density, however, 87% of the cube length can no longer be viewed as a local density, and we can see why the density methods are so affected by high dimensionalities.



(a) 11 Objects in One Unit Bin (b) 6 Objects in One Unit Bin (c) 4 Objects in One Unit Bin

Figure 3.15: Illustration of the curse of dimensionality. As the dimensions increase, less datapoints are included within a unit bin when using constant edge lengths. Figure reprinted from Parsons et al. [40].

We start by reiterating the definitions given by the authors of DBSCAN as these are the foundation for understanding both the current method and the next density-based method we will look at; OPTICS.

Definition 1. ε-neighborhood of a point.

$$N_\varepsilon(\boldsymbol{x}_i) = \{\boldsymbol{x}_j \in \boldsymbol{X} | d(\boldsymbol{x}_j, \boldsymbol{x}_i < \varepsilon\}, \tag{3.60}$$

where the ε-neighborhood is the points situated within a distance d of a central point $\boldsymbol{x}_i \in \boldsymbol{X}$ where $\boldsymbol{X}$ is the complete set of data points.

Definition 2. Directly density-reachable.

1) $\boldsymbol{x}_j \in N_\varepsilon(\boldsymbol{x}_i)$,
2) $|N_\varepsilon(\boldsymbol{x}_i)| \geq MinPts$,

where condition 2 is referred to as the core-point condition. A core point is defined as a point where the ε-neighborhood contains at least $MinPts$ number of points.

Definition 3. Density-reachable.
A point $\boldsymbol{x}_n$ is density-reachable from point $\boldsymbol{x}_i$ if, given parameters ε and $MinPts$ there is a chain of points $\boldsymbol{x}_i, \boldsymbol{x}_{i+1} \cdots \boldsymbol{x}_n$ such that each point in the chain is directly density-reachable to the next.

Definition 4. Density-connected.
A point $\boldsymbol{x}_i$ is density-connected to a point $\boldsymbol{x}_j$, given parameters ε and $MinPts$, if both $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ are density-reachable from a point $\boldsymbol{x}_k$.

Definition 5. Clusters and noise.
We define a cluster C as a non-empty subset of all data points $\boldsymbol{X}$ given parameters ε and $MinPts$.
1) If $\boldsymbol{x}_i \in C$ and $\boldsymbol{x}_j$ is density-reachable from $\boldsymbol{x}_i$, then $\boldsymbol{x}_j \in C$ $\forall \boldsymbol{x}_i, \boldsymbol{x}_j$
2) $\forall \boldsymbol{x}_i, \boldsymbol{x}_j \in C$: $\boldsymbol{x}_i$ is density connected to $\boldsymbol{x}_j$
3) Any point not in a cluster is classified as noise

DBSCAN starts by defining the ε-neighborhood of a point $\boldsymbol{x}_i \in \boldsymbol{X}$ in definition 1, where d is some distance measure as defined in Table 3.2 in Section 3.8.3 on agglomerative clustering. N_ε can be viewed as an estimate of the local density. $\boldsymbol{x}_i$ is called a *core point* if at least $MinPts$ number of points are within its ε-neighborhood. Both ε and $MinPts$ are hyperparameters where $MinPts$ sets the scale of the smallest cluster we expect to find and ε can be viewed as a density-parameter. When a point $\boldsymbol{x}_j$ is within the ε-neighborhood of a core point, it is said to be density-reachable. This core point and all density-reachable points are considered to be a cluster, and if any of the density-reachable points are also core points, their density-reachable points will be part of the cluster and so on. The algorithm is defined as

Algorithm 1 DBSCAN

```
function DBSCAN(points)
    clusters = empty list
    for point in points not visited do
        point=visited
        N = list of neighbors of point
        if $MinPts \leq |N|$ then
            if No points in N are core points then
                cluster = N
                clusters.append(cluster)
            else
                DBSCAN(N)
            end if
        end if
    end for
    return clusters
end function
```

3.8.5 OPTICS

Ordering Points To Identify the Clustering Structure (OPTICS) [3] is similar to DBSCAN, with the ε parameter relaxed from a single value to a value range. OPTICS builds on the definitions of DBSCAN by adding two new concepts; core-distance and reachability distance. Because of the similarities between the methods, definition 1-5 is defined in Section 3.8.4 and will not be repeated here.

Definition 6. Core-distance.
The core-distance d_c of a point $\boldsymbol{x}_i \in \boldsymbol{X}$ is defined as

$$d_c(\boldsymbol{x}_i) = \begin{cases} \text{UNDEFINED}, & |N_\varepsilon(\boldsymbol{x}_i)| < MinPts \\ MinPts\text{-distance}(\boldsymbol{x}_i), & else \end{cases} \tag{3.61}$$

Where $MinPts$-distance is definied to be the minimum distance $\varepsilon' < \varepsilon$ surrounding $\boldsymbol{x}_i$ where $\boldsymbol{x}_i$ is still a core point.

Definition 7. Reachability-distance
The reachability-distance d_r between two points $\boldsymbol{x}_i \in \boldsymbol{X}$ and $\boldsymbol{x}_j \in \boldsymbol{X}$ is defined as

$$d_r(\boldsymbol{x}_i, \boldsymbol{x}_j) = \begin{cases} \text{UNDEFINED}, & |N_\varepsilon(\boldsymbol{x}_j)| < MinPts \\ \max(d_c(\boldsymbol{x}_j), d(\boldsymbol{x}_i, \boldsymbol{x}_j)), & else \end{cases} \tag{3.62}$$

In other words, this means that for a point $\boldsymbol{x}_j$ in the ε-neighborhood of a point $\boldsymbol{x}_i$ situated within the core-distance radius ε', the reachability-distance will be the core-distance itself, and for a neighboring point outside the core-distance the reachability distance will be the distance between $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$

Using these added definitions, OPTICS orders the points in the dataset using a priority queue. In the way the algorithm is built up, points that are close together will be close together in the ordering.

The first point visited $\boldsymbol{x}_0$, which we call the first central point, is always set to UNDEFINED in the ordered list. We find the $N_\varepsilon(\boldsymbol{x}_0)$-neighbors of the central points and its core distance. If the point is a core point, that is, the core distance is not UNDEFINED, we move on to update the priority queue in the function UpdateSeeds. We look at each neighbor of the central point $\boldsymbol{x}_0$ and find the reachability distances. If the neighbor is not already added to the priority queue (which is always the case for the first central point), we just insert them to the queue one by one. We go back to the OPTICS function and go through each neighboring point in the priority queue by popping them off the queue, starting with the highest priority. For each neighbor, we check if it is itself a core point and if so, we repeat the process using each neighbor as the new central point iteratively. Now we may encounter neighboring points $N_\varepsilon(N_\varepsilon(\boldsymbol{x}_0))$, which are already in the priority queue. If this is the case, we do not insert them again. We instead check if the reachability distances of these points are smaller than the ones already stored in the queue. If it is, we update the reachability distance to the smallest and increase the priority of the neighbor. If it is not smaller, we just leave the point in the queue as is. In this way, we are progressively ordering points that are close together towards each other in the final ordered list. If we plot this ordering against the reachability

Algorithm 2 OPTICS: Main loop

```
function OPTICS(pointObjects, $\varepsilon$, $MinPts$)
    ordered_list = empty list
    for point in pointsObjects do
        if point.processed == False then
            neighbors = pointObjects.neighbors(point, $\varepsilon$)
            point.processed = True
            point.reachability_distance = UNDEFINED
            point.setCoreDistance(neighbors, $\varepsilon$, $MinPts$)
            ordered_list.append(point)
            seeds = empty priority queue
            if point.core_distance != UNDEFINED then
                UpdateSeeds(neighbors, point, seeds)
                while seeds not empty do
                    currentPoint = seeds.popHighestPriority
                    neighbors = pointObjects.neighbors(currentPoint, $\varepsilon$)
                    currentPoint.processed = True
                    currentPoint.setCoreDistance(neighbors, $\varepsilon$, $MinPts$)
                    ordered_list.append(currentPoint)
                    if currentPoint.core_distance != UNDEFINED then
                        UpdateSeeds(neighbors, currentPoint, seeds)
                    end if
                end while
            end if
        end if
    end for
end function
```

distance of each point, we can now find regions of higher density, as shown in Figure 3.16. The minima are the regions with small reachability distance, hence high density, which implies these regions are distinct clusters.

Algorithm 3 OPTICS: Priority Queue Update

```
function UpdateSeeds(neighborObjects, pointObject, seeds)
    core_dist = pointObject.core_distance
    for neighbor in neighborObjects do
        if neighbor.processed == False then
            dist = distance(pointObject, neighbor)
            r_dist = max(core_dist, dist)
            if neighbor.r_dist == UNDEFINED then
                neighbor.reachability_distance = r_dist
                seeds.insert(neighbor, r_dist)
            else
                if r_dist < neighbor.reachability_distance then
                    neighbor.reachability_distance = r_dist
                    seeds.increasePriority(neighbor, r_dist)
                end if
            end if
        end if
    end for
end function
```

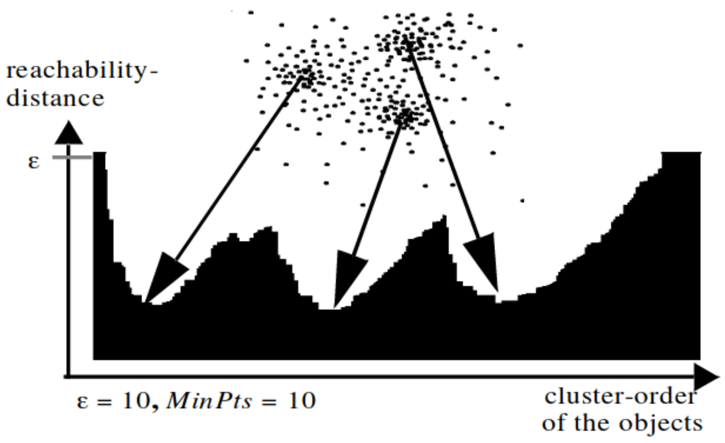


Figure 3.16: Figure showing the result of the OPTICS algorithm, where the y-axis represents the reachability and x-axis the ordering. Neighboring points in the plot are also neighbors in the dataset, and when the reachability distance is low this indicates regions of high density. Using this observation, we can infer that there are three clusters in this dataset, defined by the minima of the plot. Figure reprinted from Ankerst et al. [3].

3.9 Unsupervised Performance Metrics

To perform clustering, we would ideally have a labeled dataset so we can actually evaluate how well the algorithm is doing. When we have a dataset that is not labeled, we need to use some other metrics for evaluating the performance. These metrics use the clustering results themselves to do the evaluation by calculating the density of the clusters and the distances to other clusters. This is not perfect methods, but intuitively, if we have dense, well-separated clusters, this is likely to be an indication of a good performance by the algorithm. We are going to go through three such metrics in this section.

3.9.1 Calinski-Harbasz

The Calinski-Harabasz metric [9] is defined as the ratio of the between-clusters dispersion mean $\boldsymbol{B}$ and the within-cluster dispersion $\boldsymbol{W}$. Or in other words, how well defined the clusters are in space.

$$\begin{aligned} \boldsymbol{W} &= \sum_{q=1}^{k} \sum_{\boldsymbol{x} \in C_q} (\boldsymbol{x} - \boldsymbol{c}_q)(\boldsymbol{x} - \boldsymbol{c}_q)^T, \\ \boldsymbol{B} &= \sum_{q=1}^{k} n_q (\boldsymbol{c}_q - \boldsymbol{c}_E)(\boldsymbol{c}_q - \boldsymbol{c}_E)^T, \end{aligned} \tag{3.63}$$

where C_q is the set of points in cluster q, $\boldsymbol{c}_q$ is the center of cluster q, $\boldsymbol{c}_D$ is the center of the complete dataset D and n_q is the number of points in cluster q. We sum up the variances of each dispersion mean by taking the trace tr and calculate the ratio

$$s = \frac{tr(\boldsymbol{B}_k)}{tr(\boldsymbol{W}_k)} \times \frac{n_E - k}{k-1}, \tag{3.64}$$

where $k-1$ are the degrees of freedom of $\boldsymbol{B}$ and $n-k$ the degrees of freedom for $\boldsymbol{W}$.

3.9.2 Davies-Bouldin

Davies-Bouldin (DB) index [12] is a separation measure that utilizes the ratio between an internal dispersion measure of two clusters and the distance between their centroids. The index is given by an average over the worst separated clusters in the dataset. The dispersion measure is defined as

$$\sigma_i = \left(\frac{1}{N_i} \sum_{\boldsymbol{x} \in C_i} |\boldsymbol{x} - \boldsymbol{\mu}_i|^p \right)^{1/p}, \tag{3.65}$$

where i is the cluster index, N_i is the size of cluster C_i and $\boldsymbol{\mu}_i$ is the centroid of the cluster. Usually p is taken to be 2 which makes this the standard deviation. The distance between the centroids of cluster i and cluster j is defined to be

$$d_{i,j} = \left(\sum_{k=1}^{K} |\mu_{k,i} - \mu_{k,j}|^p \right)^{1/p}, \tag{3.66}$$

where the sum runs over the dimensions of the centroids. Again with $p = 2$ this is just the euclidean distance. The measure of separation between cluster C_i and cluster C_j is now defined as

$$R_{i,j} = \frac{\sigma_i + \sigma_j}{d_{i,j}}. \tag{3.67}$$

For each cluster C_i we calculate the separation measure $R_{i,j}$ to all the other clusters and find the the worst comparable cluster, that is the cluster where the sum of standard deviations are high and the distance between the clusters are low. We then calculate the Davies-Bouldin index by taking the average of the worst separation measure for each cluster,

$$DB = \frac{1}{N} \sum_{i=1}^{N} \max_{j \neq i} R_{i,j}, \tag{3.68}$$

where N is the number of clusters. Because we are taking an average over the worst separated clusters, the separation between individual clusters might be quite good even though the Davies-Bouldin index is high.

3.9.3 Silhouette

The silhouette score [64] is calculated individually for every point in the dataset giving a measure of how similar a point is to its assigned cluster compared to the other clusters. For a complete silhouette score the average of every individual score is taken.

We define a cluster C_i which contains a data point $\boldsymbol{x}_i$. We now define the average distance between the data point $\boldsymbol{x}_i$ and every other data point $\boldsymbol{x}_j$ in the same cluster as

$$a(\boldsymbol{x}_i) = \frac{1}{N_i - 1} \sum_{\boldsymbol{x}_j \in C_i, \boldsymbol{x}_i \neq \boldsymbol{x}_j} d(\boldsymbol{x}_i, \boldsymbol{x}_j), \tag{3.69}$$

where N_i is the number of data points in C_i and d is some distance measure. We can interpret this average distance as to how well the point is assigned to its cluster, as a data point on the edge of the cluster will have a larger average distance to every other point than a point in the center of the cluster. We now calculate the mean distance from point $\boldsymbol{x}_i$ to cluster $C_k \neq C_i$ for all k. The cluster C_k that has the smallest mean distance to the data point, is said to be the neighboring cluster of this point. We define this as

$$b(x_i) = \min_{k \neq i} \frac{1}{N_k} \sum_{j \in C_k} d(\boldsymbol{x}_i, \boldsymbol{x}_j), \tag{3.70}$$

where N_k is the number of data points in cluster C_k and d is some distance measure. We can view the neighboring cluster as the next best cluster fit for data point $\boldsymbol{x}_i$. The silhouette value is now defined as

$$s(\boldsymbol{x}_i) = \frac{b(\boldsymbol{x}_i) - a(\boldsymbol{x}_i)}{\max a(\boldsymbol{x}_i), b(\boldsymbol{x}_i)}. \tag{3.71}$$

From this we see that $-1 < s(\boldsymbol{x}_i) < 1$ and for the score to be close to 1, $a(\boldsymbol{x}_i) << b(\boldsymbol{x}_i)$ which would mean that the data point is assigned to an appropriate cluster. If, on the other hand, the value is close to -1 this means that the data point should be assigned to the neighboring cluster. To calculate a single number for the silhouette score, we use the average of all the data points

$$\bar{s} = \frac{1}{N} \sum_{i}^{N} s(\boldsymbol{x}_i), \tag{3.72}$$

where N is the total number of points in the dataset.

3.10 Autoencoder

An autoencoder is a special kind of neural network that seeks to recreate its inputs, e.g. images. The whole idea is to have an *encoder* part of the network, which progressively reduces the size of the input down to a predetermined number of features called the *latent space*. Our goal is that this latent space should hold all the information necessary to recreate the input. In a way, we are compressing the input. The output from the latent space is used as input to a mirror opposite of the encoder, called the *decoder*, which outputs a result of the same size as the input coming into the encoder. These two parts together are called an autoencoder. We can now minimize a cost function simply given by the squared difference between the input and the output of the autoencoder

$$\begin{aligned} &\phi : \mathcal{X} \to \mathcal{Z}, \\ &\omega : \mathcal{Z} \to \mathcal{X}, \\ &\phi, \omega = \arg\min_{\phi,\omega} \sum_i ||\boldsymbol{x}_i - \phi(\omega(\boldsymbol{x}_i))||^2, \end{aligned} \tag{3.73}$$

where ϕ and ω are the encoder and decoder, respectively, $\mathcal{X}$ is the original space of the dataset and $\mathcal{Z}$ is the latent space. An example architecture of an autoencoder is shown in Figure 3.18. A typical task for an autoencoder is denoising. As shown in Figure 3.17, we can add white noise to the training samples before they are processed by the autoencoder. If we now calculate the output loss by comparing the output with the original images we are teaching the autoencoder to remove white noise from images. Another important use of the autoencoder, which is how we will use it in this thesis, is for dimensionality reduction. If the autoencoder can recreate the input prefectly, the latent space is a perfect representation of the input in a lower dimensional space. This latent space can then be used for different computational tasks, like clustering.

In this thesis, we do not expect the latent space to be a perfect representation of our dataset, however, we are still hoping that the latent space will be removing some of the least prominent features making it easier to cluster on the latent space rather than the original input.

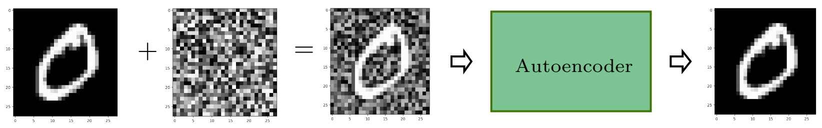


Figure 3.17: A denoising autoencoder. We add noise to the training samples before running them through the autoencoder. The output is compared to the input without noise, teaching it to recognize and remove noise from the picture

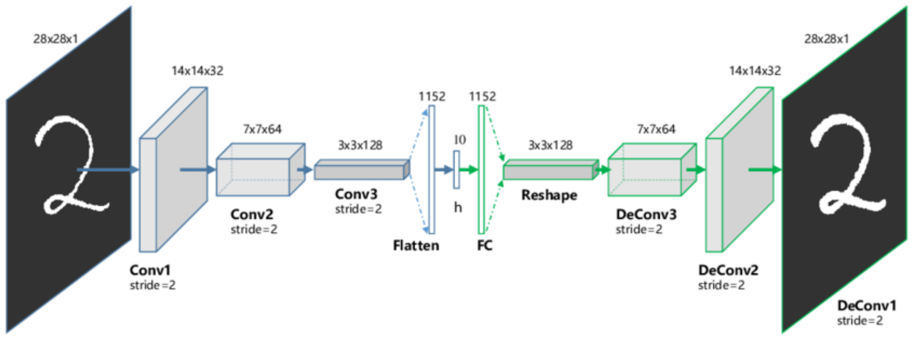


Figure 3.18: An example of a convolutional autoencoder architecture, reducing a picture of 28×28 pixels down to a latent space h of 10 features. Picture is reprinted from Guo et.al [22]

3.11 Machine Learning in Practice

In implementations of machine learning tasks, we rarely write the algorithms from scratch. There are extensive libraries for several programming languages available which deliver either complete algorithms or easy to use APIs for setting up and running network architecture. The methods mentioned in this chapter is a mathematical foundation for the implementations developed in the next part of this thesis.

In this thesis, we will use two popular machine learning libraries. For building neural networks, we will use Keras [11]. Keras is a high-level Python API that can be run on top of several more low-level frameworks. As a low-level backend, we are using the Tensorflow GPU implementation [21]. For out-of-the-box machine learning tasks, we are using the Scikit-learn Python framework [58]. Scikit-learn offers less flexibility than Keras and can only be run on CPU. The advantages of Scikit-learn is the rich library of optimized machine learning algorithms it provides, and running a complicated algorithm can often be done in just a couple of short lines of Python code.

Moving on, we are going to classify molecular structures on data from molecular dynamics simulations with supervised learning, using fully-connected networks and convolutional networks. In addition, we are going to use the Gaussian mixture models, agglomerative clustering, OPTICS and DBSCAN algorithms for clustering a dataset of methane hydrates subject to stress hoping to find the grain boundaries in the system. In the clustering algorithms, we are dependent on reducing the dimensionality of the data and are going to test two techniques for this; the principal component analysis technique, where we keep a certain percentage of the variance of the dataset, and an autoencoder where we cluster on the latent space.

Part II

Implementations and Results

Chapter 4

Conventional Structure Identification Algorithms

To benchmark our machine learning implementations we are using established structure identification algorithms. The order parameters of Steinhardt et al. [71] and Ten Wolde et al. [78] are the basis for many modern identification algorithms and we will go through the most important order parameter concepts in this chapter. In addition, we have taken a closer look at the CHILL+ algorithm of Nguyen and Molinero [52] which utilizes the order parameter correlation developed by Ten Wolde et al. and have been used successfully in distinguishing water, cubic ice, hexagonal ice, and clathrate structures.

4.1 Order parameters

Order parameters [23] have been used successfully in several automated studies of two dimensional system [6, 15]. The parameters are defined as

$$\psi_n = \frac{1}{N_j} \sum_j \exp(in\theta_{ij}), \tag{4.1}$$

where n is the symmetry of the order parameter which typically matches the number of neighbors N_j of the particle i, the sum is over neighbors of particle i and θ_{ij} is the angle of the bond between particle j and k. The non-subscript i indicates the imaginary unit, not a particle index. Due to the n-symmetry of Equation 4.1 it is often referred to as the n-atic order parameters.

As a basis for most structure identification algorithms in three dimensions lies the Steinhardt bond oriental order parameters, developed by Steinhardt et al. in 1982. These order parameters are often referred to as natural three-dimensional extensions of the n-atic order parameters and utilizes spherical harmonics, as these functions are a complete set of orthogonal functions on the sphere. Thus, they are a natural choice for representing functions defined on a sphere in the same way the n-atic order parameters represent functions on a two-dimensional circle.

The neighbors of a central particle i are defined as the particles j that are within a given radius r_q of the central particle. We call the vectors connecting the central particle to the neighbors r_{ij}. The unit vector of r_{ij} is uniquely determined by the polar and azimuthal angles θ_{ij} and ϕ_{ij}. To represent the neighborhood of the central particle i, we utilize the spherical harmonics $Y_{lm}(\theta_{ij}, \varphi_{ij}) = Y_{lm}(\hat{r}_{ij})$. The local order parameter as defined by Steinhardt et al. is formulated as follows

$$\bar{q}_{lm}(i) \equiv \frac{1}{N_b(i)} \sum_{j=1}^{N_b(i)} Y_{lm}(\hat{r}_{ij}), \tag{4.2}$$

where $\bar{q}_{lm}(i)$ as an average over the $N_b(i)$ bonds to the nearest neighbors of particle i. As $\bar{q}_{lm}(i)$ only considered the local neighborhood of particle i they are known as local order parameters. They are, however, sensitive to the global reference frame of the system and thus we need to construct local rotational invariants of these parameters for a given l

$$q_l(i) = \left[\frac{4\pi}{2l+1} \sum_{m=-l}^{l} |\bar{q_{lm}}(i)|^2 \right]^{1/2}. \tag{4.3}$$

To calculate the degree of crystallinity of a system Steinhardt et al. and later Ten Wolde et al. made use of the global version of these parameters which averages over all particles in the system

$$\bar{Q}_{lm} \equiv \frac{\sum_{i=1}^{N} N_b(i)\bar{q}_{lm}(i)}{\sum_{i=1}^{N} N_b(i)}, \tag{4.4}$$

where $\bar{Q}_{lm}$ depends on the choice of reference frame. We construct rotational invariants for the global order parameters as we did for the local parameters

$$Q_l \equiv \left(\frac{4\pi}{2l+1} \sum_{m=-l}^{l} |\bar{Q}_{lm}|^2 \right)^{1/2}. \tag{4.5}$$

As the order parameters are based on spherical harmonics they are sensitive to the degree of spatial correlation between vectors joined by neighboring particles. In a liquid there is little to none correlation between these vectors and all bond-order parameters are small or zero. In a crystal, there are repetitions of the local structure throughout the crystal making the bond-order parameters large. Another choice of order parameter are the third order invariants used by both Steinhardt et al. and Ten Wolde et al. in their work

$$\hat{W}_l \equiv W_l / \left(\sum_{m=-l}^{l} |\bar{Q}_{lm}|^2 \right)^{3/2}, \tag{4.6}$$

with W_l given by

$$W_l \equiv \sum_{\substack{m_1, m_2, m_3 \\ m_1+m_2+m_3=0}} \begin{pmatrix} l & l & l \\ m_1 & m_2 & m_3 \end{pmatrix} \bar{Q}_{lm_1}\bar{Q}_{lm_2}\bar{Q}_{lm_3}. \tag{4.7}$$

The term in parantheses is a Wigner-3j symbol.

The bond-order parameters of choice for defining the crystallinity of a system should be large for crystal structures and small for liquids. We can see from

Table 4.1: Table reproduced from Ten Wolde et al. [78] showing some simple crystal structures and their bond-parameter values. We can see the Q_6 parameter having the desired quality of having a high parameter value for solids and beeing zero for liquids

Structure	Q_4	Q_6	$\hat{W}_4$	$\hat{W}_6$
fcc	0.191	0.575	-0.159	-0.013
hcp	0.097	0.485	0.134	-0.012
bcc	0.036	0.511	0.159	0.013
sc	0.764	0.354	0.159	0.013
Icosahedral	0	0.663	0	-0.170
Liquid	0	0	0	0

Table 4.1 that the parameter Q_6 fulfill both these conditions and would be a natural choice for evaluating the crystallinity of a system.

Due to the randomness of a liquid, the global order parameters add up incoherently, and vanishes in the liquid, while it stays large for the solids. The local bond-order parameters, on the other hand, might have large values for both liquids and solids and on their own is not capable of distinguishing between liquid-like and solid-like particles. Instead, we use the correlations between the local bond order of neighboring particle as a measure of the crystallinity of individual particles

$$c(i,j) = \frac{\bar{q}_l(i)\bar{q}_l(j)}{|\bar{q}_l(i)||\bar{q}_l(j)|} = \frac{\sum_{m=-l}^{l} \bar{q}_{lm}(i)\bar{q}^*_{lm}(j)}{(\sum_{m=-l}^{l} \bar{q}_{lm}(i)\bar{q}^*_{lm}(i))^{1/2}(\sum_{m=-l}^{l} \bar{q}_{lm}(j)\bar{q}^*_{lm}(j))^{1/2}}. \tag{4.8}$$

4.2 CHILL+

As a benchmark for methane hydrate structure identification, we have used the CHILL+ algorithm developed by Nguyen and Molinero [52]. The CHILL+ algorithm is an extension of the CHILL algorithm developed by Moore et al. [51], which distinguishes ice from liquids. CHILL+ extends this by using the number of staggard, and eclipsed water-water bonds, shown in Figure 4.2, to distinguish cubic ice, hexagonal ice, and clathrate hydrates from liquid water. Each of these structures has distinct configurations of the number of staggard and eclipsed bonds, as shown in Table 4.2.

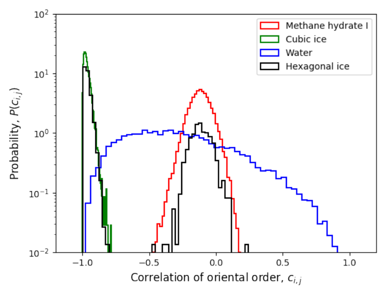
CHILL+ uses the correlation of local bond orientational order of a water molecule's four nearest neighbors, defined in Equation 4.8, with $l = 3$ or $l = 4$, to identify staggered and eclipsed bonds of each atom, thus indicating to which structure the atom belongs. CHILL+ identifies an eclipsed bond in the correlation range $(-0.35 \geq c(i,j) \geq 0.25)$ and staggered bonds for $c(i,j) \leq -0.8$.

We have made a Python implementation of CHILL+ using the Ovito Python interface [75] to handle particle positions, calculating the azimuthal and polar angles between neighboring atoms, and spherical harmonics using SciPy [67]. Ovito is a powerful visualization tool with an easy to use Python interface,

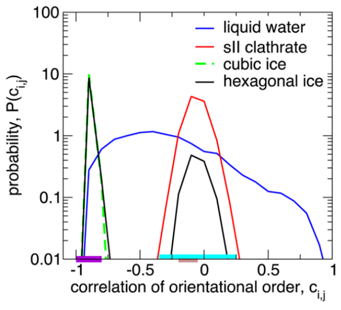
Table 4.2: Table of staggered and eclipsed bond definitions.

Structure	Eclipsed bonds	Staggered bonds	Neighbors
Cubic ice	0	4	4
Hexagonal ice	1	3	4
Interfacial ice	any	2	4
Clathrate	4	0	4
Interfacial clathrate	3	any	4
Liquid	N/A	N/A	any

Note: Table reproduced from Nguyen and Molinero [52]



(a)



(b)

Figure 4.1: Probability distribution for the correlation of oriental order parameters for spherical harmonics with $l = 3$. Figure (a) is our own implementation produced with hexagonal ice, cubic ice and liquid water but with sI clathrate as opposed to Nguyen and Molinero [52], shown in Figure (b), which used sII clathrate in the calculation.

which recently has been added to the Python Package Index (PyPi) as a standalone package. Ovito has a useful input/output module capable of handling several file formats.

```python
from ovito.io import import_file
from ovito.data import NearestNeighborFinder
import numpy as np
from scipy.special import sph_harm
from collections import defaultdict

class ChillPlus:
    def __init__(self, l, nn, filename):
        self.used_frames = []
        self.correlations_allframes = []
        self.correlation_dict = dict()
        self.__pipeline = import_file(filename,
    ↪ multiple_frames=True)
        self.n_frames = self.__pipeline.source.num_frames
        self.l = l
        self.nn = nn
```

The Ovito processing pipeline is initiated through the import_file method,

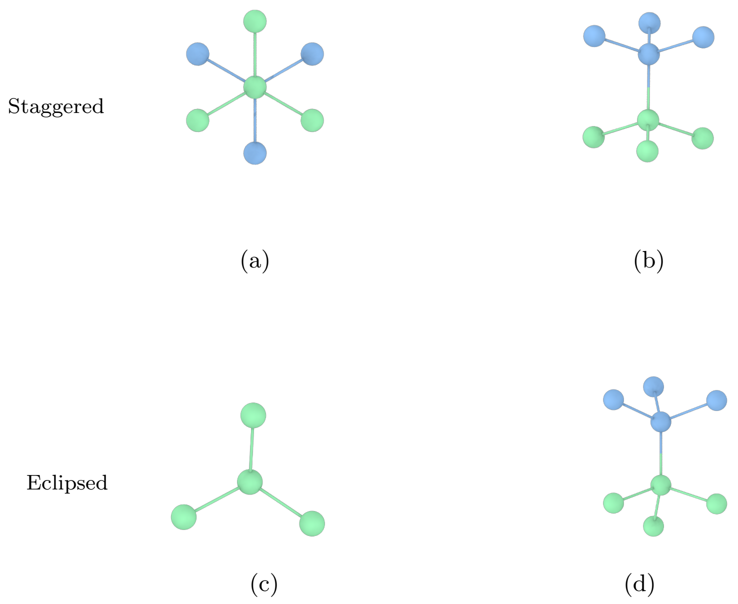


Figure 4.2: Depiction of staggered bonds (a) and (b) and eclipsed bonds (c) and (d). Figure (a) shows the top view of a staggered bond. We can see there is no overlap between the neighboring bonds of the central atoms. They are rotated 60 degrees with respect to each other. Figure (b) shows the side view of the same configuration. Figure (c) shows an eclipsed configuration. There is a total overlap between the neighboring bonds of the central atoms. Figure (d) shows the side view of Figure (c)

which reads the input data from an external data file. When initiated, the pipeline contains global information on the dataset as a whole, which should not change from frame to frame, like the number of particles. The purpose of the pipeline is to enable a non-destructive and repeatable workflow. The pipeline also handles modifiers, which are function objects made to dynamically modify, filter and analyze the data.

```
    def __call__(self, frame):
        if frame in self.used_frames:
            print(f'Frame {frame} as already run')
            pass
        else:
            self.used_frames.append(frame)
            data = self.__pipeline.source.compute(frame)
            nearest_neighbors, local_order =
    ↪ self.neighbors(self.nn, data)
            c = self.correlation(nearest_neighbors, local_order)
            self.correlation_dict[frame] = c
```

To process each data frame in the pipeline, we call the compute method with a given frame as an argument, returning a DataCollection object. This DataCollection contains the particle information of the frame, or in other words, the relevant information of a specific moment in time of the simulation. Particle positions, velocities, and values calculated by modifiers added to the pipeline can now be used externally by the Python script.

```
    @staticmethod
    def spherical_harmonics(l, angles):
        return np.array([sph_harm(m, l, *angles) for m in
    ↪ range(-l, l+1)])

    @staticmethod
    def cartesian_to_spherical(positions):
        x, y, z = np.float64(positions)
        r = np.linalg.norm(positions)
        theta = np.arctan2(y, x, dtype=np.float64) + np.pi
        phi = np.arccos(z/r, dtype=np.float64)
        return theta, phi

    @staticmethod
    def count_eclipsed(correlation_list):
        return sum(-0.34<e<0.25 for e in correlation_list)
```

To use the CHILL+ algorithm, we need to find the four nearest neighbors of every atom. This is done by using the NearestNeighborFinder class of Ovito. Another option would be to use the CutoffNeighborFinder, which uses a defined cutoff to find the indices of the closest neighbors of a central atom within this cutoff. The spherical harmonics are calculated by finding the azimuthal and polar angles from the positional data of a central particle and its neighbors.

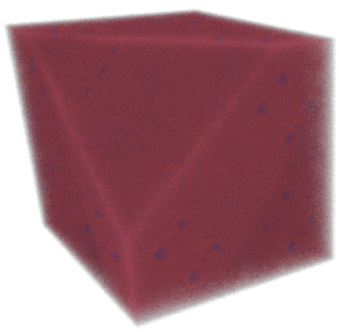
```
    def neighbors(self, nn, data):
        nn_finder = NearestNeighborFinder(nn, data)
        identifiers = data.particles.identifiers.array
        local_order = dict()
        nearest_neighbors = defaultdict(list)
        for idx, identifier in enumerate(identifiers):
            qlm = complex()
            for neigh in nn_finder.find(idx):
                neighbor_id = identifiers[neigh.index]
                nearest_neighbors[identifier].append(neighbor_id)
                positions = neigh.delta
                angles = self.cartesian_to_spherical(positions)
                qlm += self.spherical_harmonics(self.l, angles)
            qlm = qlm/float(self.nn)

            local_order[identifier] = qlm
        return nearest_neighbors, local_order
```

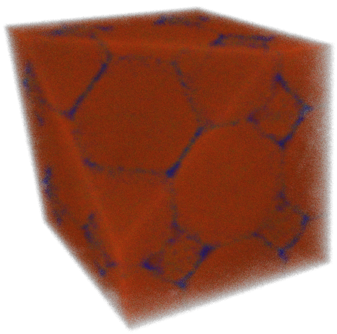
After spherical harmonics for each atom has been calculated we can find the correlation by using Equation 4.8. By evaluating each correlation result, we can use the ranges of eclipsed and staggered bonds defined in Table 4.2 to infer the structure a particle belongs to.

```
    def correlation(self, nearest_neighbors, local_order):
        chill_dict = {}
        for atom, neighbor_list in nearest_neighbors.items():
            atom_i = local_order[atom]
            atom_i_len = np.linalg.norm(atom_i)
            neighbor_correlations = []
            for nn in neighbor_list:
                atom_j = local_order[nn]
                atom_j_len = np.linalg.norm(atom_j)
                c = np.real(
                    np.vdot(atom_j, atom_i)/(atom_j_len*atom_i_len)
                    )
                neighbor_correlations.append(c)
            self.correlations_allframes.extend(
                neighbor_correlations
                )
            eclipsed_neighbors = self.count_eclipsed(
                neighbor_correlations
                )
            chill_dict[atom] = eclipsed_neighbors
        return chill_dict
```

An important thing to note is that the CHILL+ algorithm ignores the guest molecules in the clathrate structures. Because of this, the data provided must first remove the guests from the data collection to have any meaningful results.

A C++ implementation of CHILL+ as recently been implemented in Ovito, which we have used for large scale datasets due to its superior speed compared to Python. A practical example of the algorithm is shown in Figure 4.3 applied to a methane hydrate simulation containing 16 million particles that was recently made in our research group [76, 77].


(a)


(b)

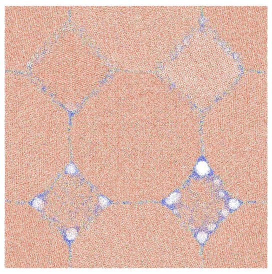

(c)

Figure 4.3: A 16 million particle simulation of methane hydrates under external stress. Particles colored by type (a) with methane in blue and water in red. By using the CHILL+ algorithm implemented in Ovito (b) we see the grain boundaries of the system with methane hydrates in orange, interfacial hydrate in green and undefined in blue. A slice of this dataset (c) shows how cracks are starting to form in the grain boundaries.

Chapter 5

Creating Datasets

For training and evaluating our machine learning algorithms we need three kinds of datasets; *training* data, *validation* data and *testing* data. Often these datasets are created by splitting a single labeled dataset into three pieces where the machine develops the algorithm by processing the training data, and during this training process the validation data is evaluated after each epoch, creating an assessment of how the algorithm will perform on actual unseen testing data. After training, the model is evaluated on the last split, the unseen testing data, to give a final measure of the performance. This way of splitting the data is a nice method if we know that new data always stems from the same source. We are, on the other hand, going to create a training and validation set that imitates realistic molecular dynamics simulations since we do not have access to labeled data of real simulations. Because of this, we can not trust the performance calculated on a test set that stems from the same dataset as our training data. Instead, we are going to evaluate the models on realistic unlabeled molecular dynamics data and compare the results with established methods of structure identification. This way we can see trends in how the models are performing on realistic models by comparing plots, we can not, however, create quantitative measures of the performance without labeled test data.

We will create a training dataset by implementing a web crawler, which downloads unit structures of crystals from aflowlib.org [46, 29]. We evaluate the models' performances on this dataset by splitting it into a training set and a validation set, and test the results on two types of molecular dynamics data; one dataset generated from a methane hydrate simulation and the other created from the oscillating pair potential.

5.1 Test Data

Ideally, we would like to have a labeled testing set to accurately assess the performance of a machine learning algorithm. We do not have access to such a dataset for crystal identification. Instead, we are going to use two kinds of datasets where it is possible to evaluate the algorithm. The first dataset is shown in Figure 5.1. The dataset is not labeled, and so we can not accurately assess how to algorithm is doing. However, we can use the CHILL+ algorithm as a benchmark to see if any of our machine learning models are able to recreate the grain boundaries seen in the figure. We will refer to this dataset as the *methane hydrate* dataset. The second dataset is a recreation of data used by Spellings and Glotzer [69]. This dataset was originally created by Engel et al. [14] and our implementations of the oscillating pair potential is a direct copy from the supplementary material of Engel et al. This dataset was manually evaluated by Engel et al. to create a phase diagram over different Pearson structures found in the phase space, seen in Figure 5.2. By comparing our machine learning results with this phase diagram, we can evaluate how the algorithm is performing. We will refer to this dataset as the *phase* dataset.

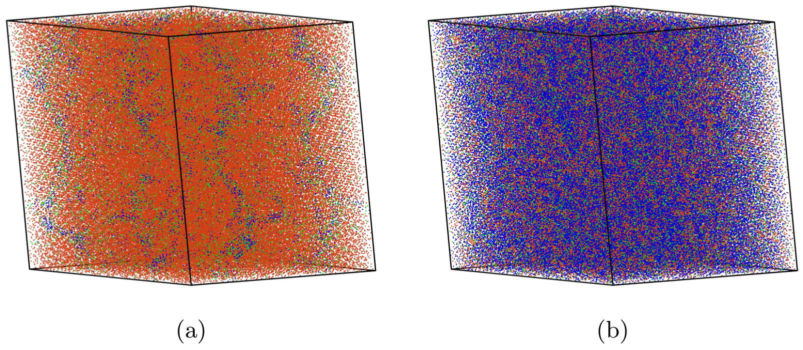


Figure 5.1: Figures of the CHILL+ algorithm evaluated on a dataset of methane hydrates under stress. The simulation was initiated with grain boundaries in the data, which we see clearly in Figure (a). Orange is classified as sI methane hydrate, green is interfacial hydrate which is hydrates with a missing bond, and blue particles are unclassified particles. The algorithm only takes into account the water molecules of the hydrate structures and all guest molecules need to be removed before applying the algorithm. In Figure (b) we can see the result when the methane guest molecules are not removed.

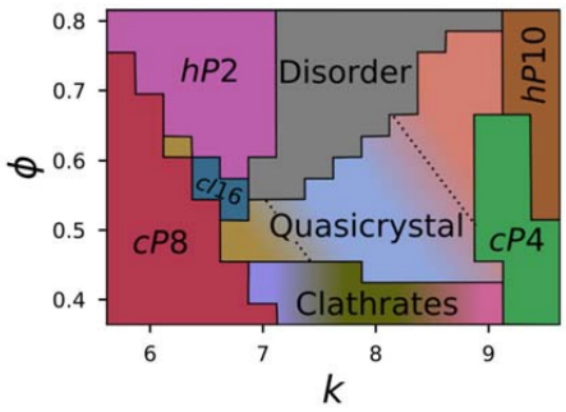


Figure 5.2: Phase diagram over the structures produces by the oscillating pair potential. The drawn boundaries and classifications was manually produces by Engel et al. [14] and the figure is reprinted from Spellings and Glotzer [69]

This code is a direct copy of the oscillating pair potential implemented by Engel et al. with a function OPP calculating and returning the force using a precalculated derivative of the potential and energy, and a separate function determineRange finding the third maximum of the potential and smoothing the function down to zero at this point.

```
# A simple molecular dynamics simulation script for the HOOMD-blue
    ↪ package
# (available for download at:
    ↪ http://codeblue.umich.edu/hoomd-blue/)
# Purpose: Self-assembles an icosahedral quasicrystal.
#
# This script is part of the Supplementary Information of:
# M. Engel, P.F. Damasceno, C.L. Phillips, S.C. Glotzer
# "Computational self-assembly of a one-component icosahedral
    ↪ quasicrystal"
# Nature Materials, doi: 10.1038/nmat4152

import math
import hoomd
import hoomd.md
import hoomd.deprecated
import numpy as np

# Define the OPP
def OPP(r, rmin, rmax, k, phi):
    cos = math.cos(k * (r - 1.25) - phi)
    sin = math.sin(k * (r - 1.25) - phi)
    V = pow(r, -15) + cos * pow(r, -3)
    F = 15.0 * pow(r, -16) + 3.0 * cos * pow(r, -4) + k * sin *
    ↪ pow(r, -3)
    return (V, F)


# Determine the potential range by searching for extrema
def determineRange(k, phi):
    r = 0.5
    extremaNum = 0
    force1 = OPP(r, 0, 0, k, phi)[1]
    while (extremaNum < 6 and r < 5.0):
        r += 1e-5
        force2 = OPP(r, 0, 0, k, phi)[1]
        if (force1 * force2 < 0.0):
            extremaNum += 1
            force1 = force2
    return r
```

The run implementation uses the HOOMD-blue [1, 19] molecular dynamics package for the simulations. A big strength of HOOMD is the user-friendly interface, which provides integrations for most standard MD-calculations. Through the hoomd.md package we are in addition able to set custom pair potentials through Python functions, not having to implement them through the source code.

The central interface of the Hoomd-blue Python API is the hoomd object. By setting the initialize method of the hoomd object, we can easily switch between running our simulations on CPU and GPU.

Our run function initializes 4096 random particles as in the work of Engel et al. We did preliminary tests initializing the system using unit cells, and then breaking the structures by heating the system before cooling, but this method had a tendency to preserve too much information about the initial structure, leading to contaminated results. If the system is first heated sufficiently this will not be a problem, although it will prolong the runtime of the simulation. Instead, we randomly place the particles in the simulation box at initialization. Through the hoomd.md.pair.table object the code utilizes the OPP and determineRange Python functions for the potential calculations and the Nosé-Hoover thermostat for integrating the equations of motion.

```python
def run(T_start, T_stop, timeSteps, potential_k, potential_phi):
    hoomd.context.initialize("--gpu 0")
    system = hoomd.deprecated.init.create_random(
        N = 4096, phi_p = 0.03
        )
    nl = hoomd.md.nlist.cell()
    all = hoomd.group.all()
    range = determineRange(potential_k, potential_phi)
    table = hoomd.md.pair.table(width=1000, nlist=nl)
    table.pair_coeff.set(
        'A', 'A', func = OPP, rmin = 0.5, rmax = range,
        coeff = dict(k = potential_k, phi = potential_phi)
        )

    filename = (
        "400quasi/quasicrystal_k" + str(potential_k)
        + "_phi" + str(potential_phi)
        )
    filename += "_T" + str(T_start) + '-' + str(T_stop)
    hoomd.dump.gsd(
        filename, period=timeSteps*1e-3, group=all, overwrite=True
        )
    temperature_ramp = hoomd.variant.linear_interp(
        [(0, T_start), (timeSteps, T_stop)]
        )
    hoomd.md.integrate.nvt(
        group = all,  kT = temperature_ramp , tau = 1.0
        )
    hoomd.md.integrate.mode_standard(dt = 0.01)
    hoomd.run(timeSteps + 1)
```

In our simulation the complete dataset was created from 20 evenly spaced values between $\phi_{start} = 0.38$, $\phi_{stop} = 0.8$ and 20 evenly spaced values between $k_{start} = 5.8$, $k_{stop} = 9.5$. Each combination of ϕ and k gives a distinct pair potential, which when cooled to form crystalline states yields 400 datasets of crystal structures. Each of the 400 simulations was run over 7×10^7 timesteps and the particles system was linearly cooled from $0.4T$ to $0.1T$ in reduced temperature units using the Nosé-Hoover thermostat.

Because the original dataset of Engel et al. was cooled more slowly, using 10^8 timesteps, there might be differences between the sets in the exact structures created in the crystals. Some selected combinations of k and ϕ, and their corresponding potential shapes are shown in Figure 5.3. As the crystal is cooling, the particles will become trapped in the potential wells, and with a faster cooling of the system we leave less opportunity for the particles to fluctuate and cross the potential tops. This will probably mean that, for a given potential, a larger amount of particles will be trapped in the deepest minima than would be if the system is cooled slower, and by extension, this will slightly alter the intrinsic crystal structure. We chose to cool the structure faster as a compromise due to time and resource constraints.

One of the 400 datasets is shown in Figure 5.4 which shows the evolution of a simulation using parameters $k = 8.137$ and $\phi = 0.38$. This results in structures situated in the clathrate domain as shown in Figure 5.2.

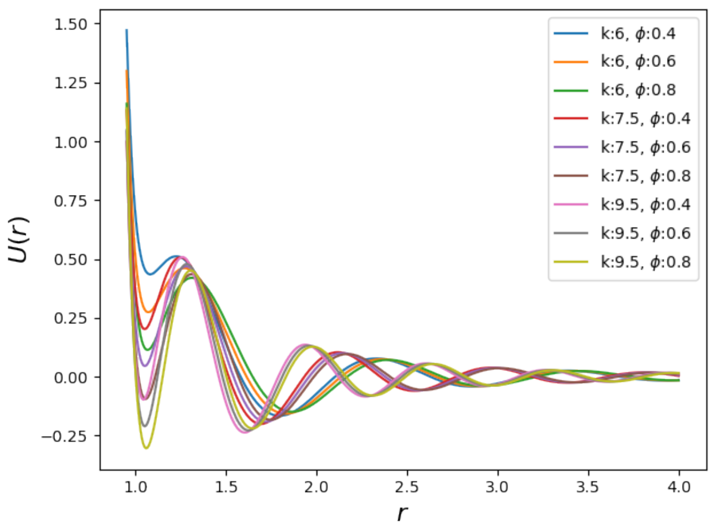


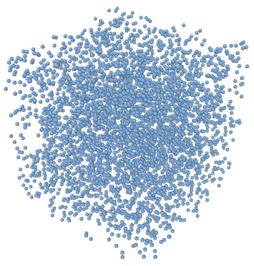
Figure 5.3: Some selected combinations of k and ϕ for the oscillating pair potential used to recreate the dataset of Engel et al. [14].

5.2 Training Data

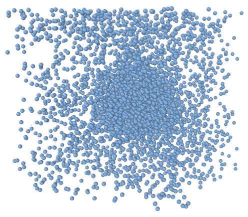
The biggest problem to overcome when doing structure classification with machine learning on molecular dynamics simulations is obtaining a labeled dataset for training. To automate this process we have used structures found on aflowlib.org, which contains several hundred unit structures in POSCAR files. All these structures are labeled in several ways, but we will use the corresponding Pearson groups discussed in Section 2.5 on crystal structures. By using the Pearson structures, our labels are directly comparable with the manual classification of Engel et al. on the phase dataset. To download these files, we have made a web crawler in Python. This crawler downloads all POSCAR files and information of which Pearson group the structure belongs to.

As this is a specific crawler for the AFLOW website, we start by defining the hyperlinks we wish to search. We are only interested in the Pearson classification and set the relevant links hardcoded in the script.

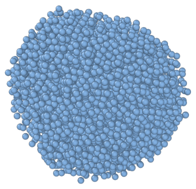
```
import requests
from bs4 import BeautifulSoup
import re
import numpy as np

tlds = [
```



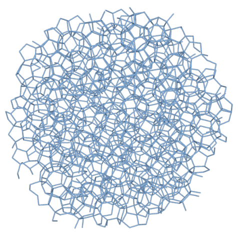
(a) Timestep 0.



(b) Timestep 3.773×10^7.



(c) Timestep 7×10^7.



(d) Timestep 7×10^7. Slice of structure with added bonds.

Figure 5.4: Time evolution for simulation with $k = 8.137$ and $\phi = 0.38$. We can see from Figure d that these parameters yields clathrate structures.

```
'triclinic_pearson.html',
'monoclinic_pearson.html',
'orthorhombic_pearson.html',
'tetragonal_pearson.html',
'trig_hex_pearson.html',
'cubic_pearson.html',
        ]
struk_domain = 'http://aflowlib.org/CrystalDatabase/'
```

We then find all relevant sublinks and from these create the POSCAR hyperlinks.

```
def get_strukt_links(url, domain):
    r = requests.get(domain+url)
    html = r.text
    soup = BeautifulSoup(html, 'html.parser')
    all_imgs = list(soup.find_all("img"))

    poscar_links = []
    names = []
    pearson = {}
    for img in all_imgs:
        img = str(img)
        find_strukture = re.compile('alt="([^_]*_([^_]*)_.*).html')
        match = find_strukture.search(img)
        link_name = match.group(1)
        pearson_symbol = match.group(2)

        names.append(link_name)
        poscar_links.append(
            struk_domain+'POSCAR/'+link_name+'.poscar'
            )
        pearson[link_name] = pearson_symbol
    return names, poscar_links, pearson

def get_all_struk_links(urls, domain):
    all_names = []
    all_poscar_links = []
    all_pearson = {}
    for url in urls:
        names, poscar_links, pearson = get_strukt_links(
            url, domain
            )
        all_names.extend(names)
        all_poscar_links.extend(poscar_links)
        all_pearson.update(pearson)
    return all_names, all_poscar_links, all_pearson
```

After all the relevant links are found we collect the POSCAR data by evaluating the HTML code and dumping all valid information to standard text files.

```
def write_poscar_files(dump_dir, names, links):
    strange = []
    for name, link in zip(names, links):
        print(name, link)
        r = requests.get(link)
        if r.status_code != 200:
            strange.append(link)
            continue
        else:
            text = r.text
            with open(dumpdir+name, 'w') as f:
                f.write(text)
    return strange
```

In addition to the unit structures downloaded with the crawler we can use the Atomic Simulation Environment (ASE)[31] spacegroup module to create the unit structures of methane hydrate I (sI), methane hydrate II (sII) and methane hydrate H (H), as these structures are of special interest in this thesis.

```
import ase.spacegroup
import ase
import numpy as np
from ase.io import write
from ovito.io import export_file
from ovito.io.ase import ase_to_ovito
my_crystals = {
    "methane_hydrate_I" :
    {
        "symbols" : ("O", "O", "O", "C", "C"),
        "spacegroup" : 223,
        "cellpar" : [12.03, 12.03, 12.03, 90, 90, 90],
        "basis" : [ (.1841,.1841,.1841),
                    (0,.3100,.1154),
                    (0,.5,.25),
                    (0,0,0),
                    (.25,.5,0)
                    ],
    },
    "methane_hydrate_II" :
    {
        "symbols" : ["O", "O", "O", "C"],
        "spacegroup" : 227,
        "cellpar" : [17.092, 17.092, 17.092, 90, 90, 90],
       "basis" : [  (.1822, .1822, .3719),
                    (.2196, .2196, .2196),
                    (.125,  .125,  .125),
                    (.375,  .375, .375)
                    ],
        "setting" : 2,
    },
    "methane_hydrate_H" :
    {
        "symbols" : ["O", "O", "O", "O",  "C", "C", "C"],
        "spacegroup" : 191,
        "cellpar" : [11.9100 , 11.9100, 9.8940, 90, 90, 120],
        "basis" : [ (0., 0.38, 0.125),
                    (0.22,0.44,0.25),
                    (0.125,0.25,0.5),
                    (0.333,-0.333,-0.375 ),
                    (0.5,0.5,0.5),
                    (0.3333, -0.3333, 0.0),
                    (0,0,0)
                    ],
        'onduplicates': 'replace',
        'symprec': 0.002,
    },
}
def create_crystal(crystal_info, **kwargs):
    return ase.spacegroup.crystal(**crystal_info, **kwargs)
```

The POSCAR files are idealized unit structures and need a bit of handling before use.

Firstly, we need to replicate the unit cell until a desired number of atoms exists in the structure. We do this by using the ReplicateModifier in the Ovito Python interface and testing for a minimum number of particles threshold. We set the minimum amount of particles to 500 and stop the replication when we are above this limit. However, there is no upper limit and because of this, the number of particles may vary between 500 and at most 4000.

```
import numpy as np
from ovito.io import export_file, import_file
from ovito.modifiers import ReplicateModifier
from util import is_dir
def replicate_cell(
    file_name, dump_name, replicate=None, n_minimum_particles=500
):
    pipe = import_file(file_name)
    data = pipe.compute()
    n_particles = data.number_of_particles
    if not replicate:
        min_particles_after_replicate = n_minimum_particles
        rep = int(np.ceil(
            (min_particles_after_replicate / n_particles)**(1/3)
                    )
                )
        replicate = {'num_x': rep, 'num_y': rep, 'num_z': rep}
    data.apply(ReplicateModifier(**replicate, adjust_box=True))
    new_n_particles = data.number_of_particles
    unique = len(np.unique(data.particles.positions[...], axis=0))
    if unique != new_n_particles:
        print(new_n_particles, unique)
        print(file_name)
    export_file(data, f'{dump_name}.data', 'lammps/data')
```

Secondly, we still only have access to idealized unit structures, which we are unlikely to encounter in realistic MD simulations as temperature and pressure fluctuations might slightly alter the crystals without breaking the structures. For realistic MD simulations, specialized potentials are used depending on the system simulated. As we need to be able to temperate hundreds of structures to make the conditions as close to real as possible, individualized potentials for each structure is not feasible. To imitate this temperation, we have implemented a harmonic potential in the ASE molecular dynamics module. The harmonic potential can be implemented by downloading the latest ASE version from GitLab, copying the code into a file called harmonic.py under ase/calculators, and compiling the code.

```
import numpy as np
from ase.neighborlist import NeighborList
from ase.calculators.calculator import Calculator, all_changes
from ase.calculators.calculator import PropertyNotImplementedError

# Determine the potential range by searching for extrema
class Harmonic(Calculator):
    implemented_properties = ['energy', 'forces', 'stress']
    default_parameters = {'k': 10.0,
                          'rc': None}
    nolabel = True

    def __init__(self, r0, **kwargs):
        Calculator.__init__(self, **kwargs)
        self.r0 = r0

    def calculate(self, atoms=None,
                  properties=['energy'],
                  system_changes=all_changes):
        Calculator.calculate(self, atoms,
                             properties, system_changes)
        natoms = len(self.atoms)
        k = self.parameters.k
        rc = self.parameters.rc
        if rc is None:
            rc = 3.0
        if 'numbers' in system_changes:
            self.nl = NeighborList([rc / 2] * natoms,
                                   self_interaction=False)

        self.nl.update(self.atoms)
        positions = self.atoms.positions
        cell = self.atoms.cell
        energy = 0.0
        forces = np.zeros((natoms, 3))
        stress = np.zeros((3, 3))

        energy += self.harmonic(positions, self.r0, k)[0]
        forces -= self.harmonic(positions, self.r0, k)[1]

        self.results['energy'] = energy
        self.results['free_energy'] = energy
        self.results['forces'] = forces

    def harmonic(self, r, r0, k):
        V =  k/2*(r - r0)**2
        F = k*(r-r0)
        return (V, F)
```

Recently a harmonic potential has been implemented in the ASE source code and our harmonic potential is no longer strictly required but is included here as this is the potential we have used in our simulations.

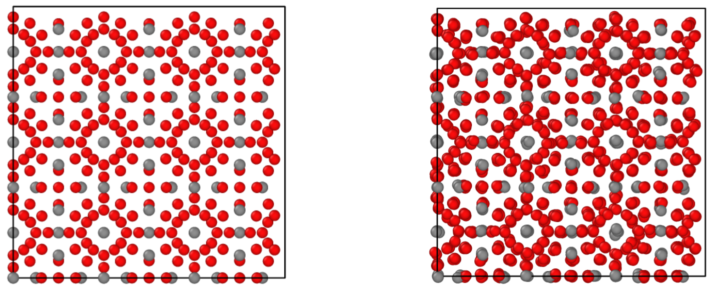
The potential can now be implemented by utilizing the ASE Langevin thermostat for heating the crystal. Because there are big differences between the densities of the unit structures, and so the impact of the fluctuations added by the potential will vary, we have implemented a break condition to make sure the crystals does not heat to a point where the structures completely break. As a break condition we have used: If any atoms have moved more than 25 percent of the distance to the first minimum of the radial distribution function, we will stop the simulation. An example of a replicated unit structure is shown in Figure 5.5a and the same structure temperated with the harmonic potential is shown in Figure 5.5b .

```
import numpy as np
from ase.md.langevin import Langevin
from ovito.io import import_file, export_file
```

```python
from ovito.io.ase import ase_to_ovito
from ase.calculators.harmonic import Harmonic
from ovito.modifiers import CoordinationAnalysisModifier
from util import cutoff_finder

coordination = CoordinationAnalysisModifier(
    cutoff=10, number_of_bins=100
    )

def run_md(
    file_name=None, dump_name=None, steps=None, crystal=None
    ):
    crystal = import_file(file_name).compute().to_ase_atoms()
    data = ase_to_ovito(crystal)
    data.apply(coordination)
    cut = cutoff_finder(data)
    positions = crystal.get_positions()
    calc = Harmonic(positions)
    crystal.set_calculator(calc)
    start_pos = crystal.get_positions()
    dyn = Langevin(crystal, 0.1, 0.4, 0.02)
    prev_step=0
    for step in steps:
        prev_step += step
        try:
            dyn.run(step)
            atoms = dyn.atoms
            pos = atoms.get_positions()
            if np.all(
                np.linalg.norm(pos - start_pos, axis=1) < 0.25*cut
                ):
                data = ase_to_ovito(atoms)
                export_file(
                    data,
                    f"{dump_name}_step{prev_step}.data",
    ↪ 'lammps/data'
                    )
            else:
                break
        except NotImplementedError as e:
            print(e)
            break
```

5.3 Creating features

We now have access to a flurry of structures, both in idealized form and tempered. Before we can do any machine learning on this data we need to define a set of features which can properly describe the structure of the dataset. The datasets we created in Sections 5.1 and 5.2 consists of positional data of every atom and we need to transform this data to include information of the neighborhood of each atom to say anything meaningful of what structure an atom is situated in. Inspired by Reinhart et al. [62] and Reinhart and Panagiotopoulos [61], we have used the local topology of individual atoms to create features for structure identification. Reinhart used common neighborhood analysis (CNA) to classify particles by evaluating the topology of each atom. We describe the local topology by so-called adjacency matrices. These are matrices of binary



(a) (b)

Figure 5.5: The methane hydrate unit structure created by ASE (a), and the same structure temperated with an harmonic potential (b). Both structures are three-dimensional crystals seen from the side.

values where 1 indicates a nearest-neighbor relationship between two atoms and 0 indicates no such relationship. The topology of a central atom is defined by its nearest neighbors. In Figure 5.6a, we have a 2D system of particles, and we are focusing on a single central particle marked in blue. We now find a set of neighboring particles either within a cutoff radius around the central particle or by choosing a given number of nearest neighbors. We denote these neighboring particles as the *outer* neighbors. In Figure 5.6b, we have focused in on nine outer neighbors, where we count the central particle as a neighbor of itself. For each of the outer neighbors, we then need to find which are nearest neighbors of each other within a given cutoff radius. This, we denote as the *inner* neighbors. In Figures 5.6c and 5.6d, we have marked the outer neighbor we are looking at in dark blue and the corresponding inner neighbors in light blue. To describe this relationship in a matrix, we start by ordering the atoms by distance from the central particle, as shown in Figure 5.7a. We can now go through each of the outer neighbors and see which particles are its inner nearest neighbors and denote the relationship by a binary value, as shown in Figure 5.7b. Because we chose to define particles as neighbors of itself, we will always have a series of ones on the diagonal. By doing this process for each particle in a dataset, we get a set of features for every particle describing the local topology.

To create the adjacency matrices we need to define the cutoff used to calculate the inner neighbor relationships. For this task, we use the shape of the radial distribution function.

We use the radial distribution function (RDF) to calculate how the particles are situated in the crystal. A top in the RDF indicates order in the crystal. In an ideal crystal, there are very sharply defined peaks as the particles can only be situated in very defined distances from each other as given by the unit structure. We can view these peaks as shells around the particles where we find the nearest neighbors. We can see from Figure 5.8a that for the unit structure of sI methane hydrates shown in Figure 5.5a, there are very sharply defined peaks as all particles are perfectly placed to form the methane hydrate structure. In Figure 5.8b the structure has been temperated using a harmonic potential and the particles are no longer situated in a perfect unit structure. This results in a less sharply defined RDF. In our structure evaluation, we have chosen to use the

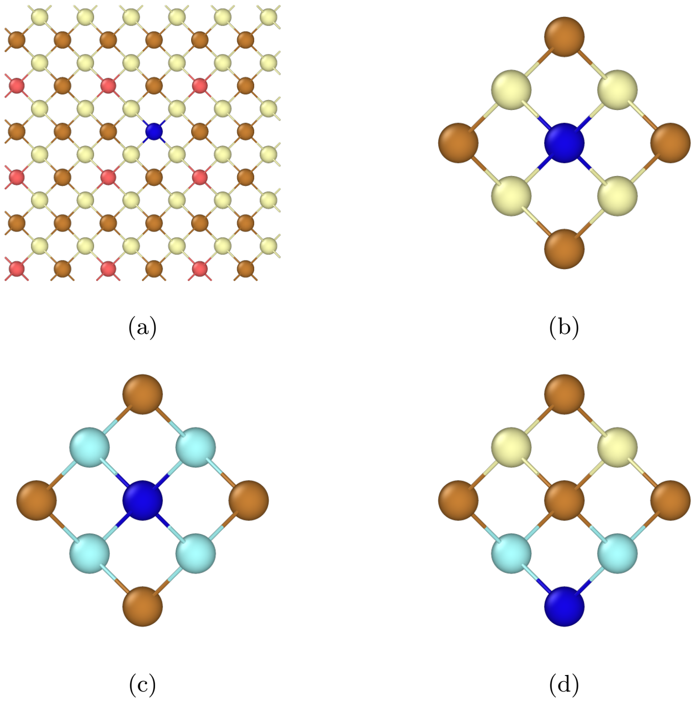
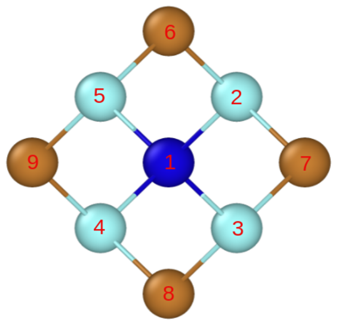


Figure 5.6: Two dimensional figures showing the creation of adjacency matrices. We iterate through all the particles, considering each of them as the central particle once. Figure (a) shows a central particle marked in blue. We decide to make an adjacency matrix out of the nine nearest neighbors of the central particle, including itself, shown in Figure (b). For each of the nearest neighbors, we check which are nearest neighbors of each other, shown in Figures (c) and (d). The particle in dark blue is the nearest neighbor we are considering, and particles in light blue are its neighbors within a cutoff distance defined by the radial distribution function.



(a)

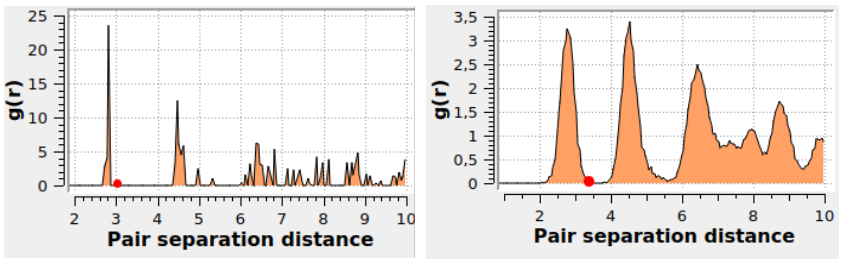
$\Longrightarrow$

	1	2	3	4	5	6	7	8	9
1	1	1	1	1	1	0	0	0	0
2	1	1	0	0	0	1	1	0	0
3	1	0	1	0	0	0	1	1	0
4	1	0	0	1	0	0	0	1	1
5	1	0	0	0	1	1	0	0	1
6	0	1	0	0	1	1	0	0	0
7	0	1	1	0	0	0	1	0	0
8	0	0	1	1	0	0	0	1	0
9	0	0	0	1	1	0	0	0	1

(b)

Figure 5.7: A numbering by distance of the nearest neighbors of a central particle marked in dark blue (a). We iterate through the nearest neighbors in the order shown and calculate which nearest neighbors are neighbors of each other. If they are nearest neighbors this is denoted with a 1 in the adjacency matrix and 0 otherwise, as shown in Figure (b).

first minimum of the RDF as the cutoff distance for the inner nearest neighbor calculations. For some temperated crystals there might, however, be some small tops in the RDF before we reach a very distinct top. Because of this we have used the condition that what is considered as the first top in the RDF has to be over a certain height. The arbitrary height we have chosen is 40 percent of the global maximum of the RDF.



(a) (b)

Figure 5.8: Radial distribution function of water molecules in methane hydrate sI crystal as a unit structure (a) and temperated with an harmonic potential (b). The red markings show where the cutoff would be set for each function.

```
from operator import itemgetter
from collections import Counter
from ovito.io import import_file
import os
import numpy as np
import freud
from util import is_dir, cutoff_finder, check_frame
def adjacency(
    data, num_neighbors, inner_cutoff,
    frame, dump_filename, recompute=False
    ):
    if inner_cutoff==None:
        cutoff = cutoff_finder(data)
    else:
        cutoff=inner_cutoff
    dump_filename = (
            dump_filename
            + (f'_frame{frame}_nneigh{num_neighbors}'
              + f'_icut{np.round(cutoff, 3)}.npy'
              )
            )
    if os.path.isfile(dump_filename) and not recompute:
        adjacency_matrix = np.load(dump_filename)
        return adjacency_matrix
    else:
        N_particles = data.particles.count
        sim_cell = data.cell.matrix
        box = freud.box.Box.from_matrix(sim_cell)
        positions = data.particles.positions
```

We make use of the Freud-Analysis [13] Python library for nearest neighbor queries. The freud.locality module is used for finding a given number of outer nearest neighbors. The returned list from Freud is sorted by atom index which is not ideal for creating adjacency matrices, and has to be sorted by their distances to the central atom to be consistent for each adjacency matrix.

```
            query_args = { 'mode': 'nearest',
                           'num_neighbors': num_neighbors,
                           'exclude_ii':False
                       }
            aabb = freud.locality.AABBQuery(box, positions)
            nearest_neighbors = aabb.query(
                positions, query_args=query_args
                ).toNeighborList()
            neighbor_counts = nearest_neighbors.neighbor_counts
            outer_distances = nearest_neighbors.distances.reshape(
                -1, num_neighbors
                )
            outer_distances = np.argsort(outer_distances, axis=1)
            outer_list = nearest_neighbors.point_indices.reshape(
                -1, num_neighbors
                )
            outer_list = outer_list[np.arange(
                len(outer_distances)
                ), outer_distances.T].T
            outer_repeat = np.repeat(
                outer_list, neighbor_counts, axis=0
                ).astype(np.int64)
```

To finish the adjacency matrix, we need to find which of the outer nearest neighbors of a central atom are inner nearest neighbors of each other. We filter the list found in the outer neighbor query by a cutoff distance defined using the first minimum after the first clearly defined maximum in the radial distribution function.

```
            nearest_neighbors = nearest_neighbors.filter_r(
                r_max=cutoff
                )
            inner_segments = nearest_neighbors.segments
            point_indices = nearest_neighbors.point_indices
            inner_list = np.split(
                point_indices, inner_segments
                )[1:]
            inner_list_getter =
    ↪  itemgetter(*outer_list.ravel())(inner_list)

            adjacency_matrix = np.asarray(
                [np.isin(x, y) for x, y in zip(
                                    outer_repeat,
                                    inner_list_getter)
                               ], dtype=np.int8
                ).reshape(-1, num_neighbors, num_neighbors)
            np.save(dump_filename, adjacency_matrix)
            return adjacency_matrix
```

The nearest neighbor query is done entirely in Freud, which utilizes parallelized C++, while the construction of the adjacency matrices is done with Numpy and a Python loop. This process in Python is slow and memory intensive and should ideally be implemented in C++ at the same time as finding the nearest neighbors.

Depending on the structure we are trying to classify, different sizes of adjacency matrices might be needed. For small, simple structures, only a few nearest neighbors are needed to completely describe the local topology. For large, complicated structures, like methane hydrates, we need quite a lot of nearest neighbors just to include one complete structure. Including additional neighbors should not lead to worse classification of the small structures, although the computation time will increase, so we will generally expect many neighbors to perform better than few. There might be a limit to this, however, as extra neighbors do mean increased dimensionality, and as discussed in Section 3.8.4 adding more dimensions does not necessarily give a better learning outcome because of the curse of dimensionality.

We have created two different training datasets for supervised learning. The first set was created from the AFLOW library with structures matching the ones defined by Engel et al. found in Figure 5.2, except for the quasicrystal and disordered regions. We have included the methane hydrate structures sI, sII and H which we created ourselves and all cP8, cP4, cI16, hP10 and hP2 Pearson structures found at aflowlib.org. We will call this the *aflow* dataset. The second dataset was created from only the clathrate hydrate structures, which we call the *clathrate* dataset. All the structures have been temperated with the harmonic potential for 300 timesteps, unless the break condition takes effect before the timestep limit is reached, and we made adjacency matrices for the particle configurations every 10 timesteps still labeling each particle as part of the same structure it was situated in the non-temperated configuration. Hopefully these small displacements in the crystal structures make the dataset more similar to how particles are crystallized in a realistic MD simulation without breaking the crystal structure.

Chapter 6

Supervised learning

Creating powerful structure identification algorithms like CHILL+ is not an easy task and the authors of the algorithm needs to have deep insight into the structures they are trying to identify. These kinds of algorithms are also likely to be very specialized, forcing us to keep developing new algorithms for every identification problem we encounter. As mentioned in the machine learning theory, we are not developing a specialized algorithm in machine learning. We only implement a framework defining what we value as good or bad results with a few additional rules on how to improve towards the best result possible. The task of actually developing the algorithm is left to the machine. The immediate advantage of this is that the machine can develop an algorithm as general as need be, depending on the data and framework we provide. Theoretically, the machine can develop an algorithm capable of differentiating between every known molecular structure if the data provided is ideal and the framework is perfectly set up. This is, of course, a dream scenario and in actual implementations, it is often hard to find a machine learning algorithm that can compete with a well thought out, specialized algorithm. In this chapter, we will use the aflow and clathrate datasets created in the previous chapter to train a fully-connected neural network and a convolutional network, as well as testing the best network architectures on the phase dataset and the methane hydrate dataset.

6.1 Neural network

In Section 5.3, we transformed the datasets into adjacency matrices which we can use as input features for developing machine learning models for classification. In this section, we will construct two kinds of machine learning architectures. A fully-connected network with all nodes in each layer connected to all nodes in the previous layer and a convolution neural network. Convolutional neural networks are traditionally used for image classification, however, we can view the adjacency matrices as black and white images with binary values instead of pixel values, as illustrated in Figure 6.1 using the adjacency matrix defined in Figure 5.7b.

Each machine learning architecture as been tested for adjacency matrices of 10-80 neighbors with 10 neighbor increments. This gives us between 100 and 6400 features per atom as input to the networks, as shown in Table 6.1. The

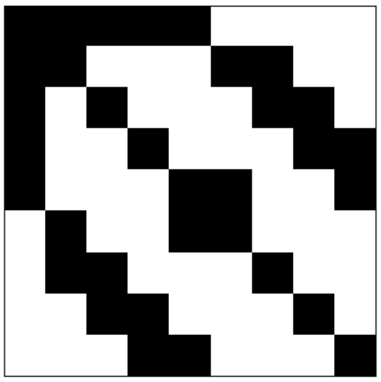


Figure 6.1: An adjacency matrix converted into a black and white image.

Table 6.1: Outer neighbors of adjacency matrices and the number of input features to the machine learning algorithm

Neighbors	Input Dimensions
10	100
20	400
30	900
40	1600
50	2500
60	3600
70	4900
80	6400

adjacency matrices are symmetric, and for a fully-connected network we could decide to just use the upper triagonal as inputs to the network as the lower triagonal does not add any new information. This would, on the other hand, not work for a convolutional network as we need to keep the input in an image-like form. Because we want to compare the two networks on the same input, we have chosen to keep the adjacency matrices in its original square shape.

For implementing the two network types, we are going to use Keras. Keras is a high-level neural network API which can utilize TensorFlow, CNTK and Theano as backends. In our implementations, we have chosen to use the Tensorflow 2.0 GPU version as the backend, compatible with Python 3.6 and Cuda 10.0.

The Sequential Keras object stacks layers linearly, meaning each layer is only connected to the previous layer. Keras also provides a functional API for more flexibility if a more complex architecture with communication between several layers is required. Layers to the sequential model are added to the code in a top-down approach, as shown in the code snippet below of a network with one 10 node layer and one 5 node layer.

```python
from keras.models import Sequential, Dense
model = Sequential()
model.add(Dense(10))
model.add(Dense(5))
```

In addition to the Seqential object, we make use of the Keras History object for storing the training and validation results for each epoch during training time. As both network types need some of the same methods, we implement these common methods in a parent class called NeuralNet. This class defines how we save the models after training and a fit method that trains the model.

```python
import keras
import numpy as np
from keras.callbacks import EarlyStopping, History
from keras.layers import Dense, Dropout, Conv2D, AveragePooling2D,
    ↪ Flatten
from keras.models import Sequential

class NeuralNet:
    def __init__(self, patience=10):
        self.history = History()
        self.callbacks = [
                EarlyStopping(
                monitor='val_loss',
                patience=patience,
                verbose=1), self.history
                ]
    def save_model(self, model_dumpname, history_dumpname):
        model_json = self.model.to_json()
        with open(model_dumpname+'.json', "w") as json_file:
                json_file.write(model_json)
                self.model.save_weights(f"{model_dumpname}.h5")
                print("Saved model to disk")
        np.save(history_dumpname, self.history.history)
        print("Saved history to disk")

    def fit(
        self, X, y, epochs=100,
        batch_size=512, validation_split=0.4,
        ):
        self.model.fit(
            X, y, epochs=epochs,
            validation_split=validation_split,
            batch_size=batch_size, shuffle=True,
            callbacks=self.callbacks,
            )
```

6.2 Fully-Connected Network

To implement a fully-connected network, we have made a DenseNet class which utilizes the Keras Dense object with the option of adding a dropout layer every other layer. It inherits from the NeuralNet class and takes parameters defining the output size and activation for each layer.

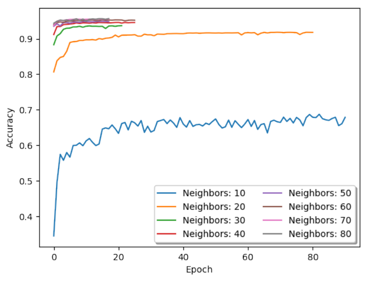
```python
class DenseNet(NeuralNet):
    def __init__(
        self, num_classes=255, dropout_rate = 0.1,
        dropout=True, layer_params=[]
        ):
        super().__init__()
        model = Sequential()
        for layer in layer_params:
            model.add(Dense(**layer))
            if dropout:
                model.add(Dropout(dropout_rate))
            print(model)
        model.add(Dense(num_classes, activation='softmax'))
        model.summary()
        model.compile(
            optimizer='adam', loss='categorical_crossentropy',
            metrics=['accuracy']
            )
        self.model=model
```

We have trained networks consisting of four hidden layers on the aflow dataset, where the output sizes of each layer depend on the input size to the network. The output size of each layer is defined by the formula $N/(2+2l)$ where N is the size of the input and $l = [0, 1, 2, 3]$ are the hidden layers. We have implemented this scheme because we want the output size to decrease throughout the network, but because the input has vastly different dimensionalities depending on the number of neighbors used in the adjacency matrix, we can not set the output sizes as fixed numbers. The network was tested for the relu, tanh and sigmoid activation functions as well as dropout rates of $0, 0.1, 0.3$ and 0.5. The complete set of hyperparameters tested is shown in Appendix A. The architectures and hyperparameters of the best testing networks are shown in Table 6.2, where we see that all the best architectures use relu as an activation function throughout the network and a high dropout rate gives the best performance. A high dropout rate in a fully-connected network is not that surprising as the number of parameters per layer is very high. Using many parameters equates to using a complex model, and as we discussed in the theory on machine learning, a complex model is prone to overfitting. A high dropout rate counteracts this problem.

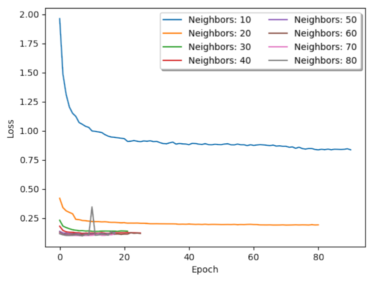
In Figure 6.2 we can see the best validation accuracy and validation loss for each of the nearest neighbor combinations. We see that there is a dramatic difference in accuracy between 10 and 20 neighbors, and for 40 to 80 neighbors the accuracy difference is less prominent. Even though the dataset of 80 neighbors performs best during training, we have to keep in mind that the dataset we have created does use a harmonic potential for temperating the crystals, which is not the case for a realistic MD simulation, and because of this a small increase in accuracy on the validation set does not necessarily translate perfectly to a testing set.

The Phase Dataset

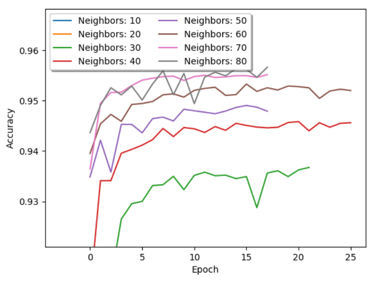
In Figure 6.3 we have tested the best networks on the phase dataset at the end of simulations for each combination of ϕ and k. We count the occurrence of different structures found in the dataset and plot the most prevalent structure



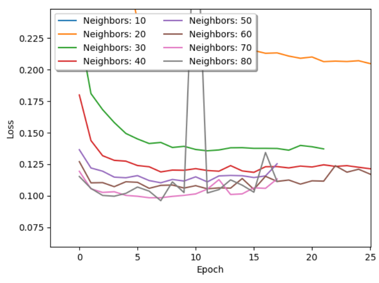
(a)



(b)



(c)

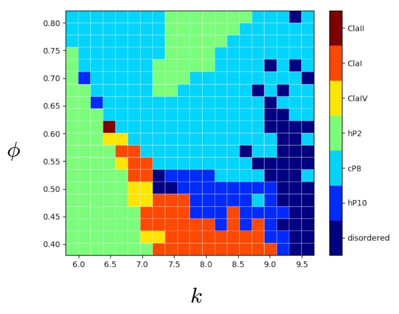


(d)

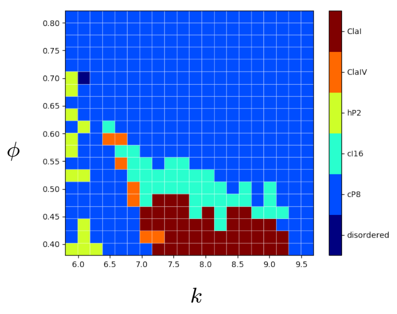
Figure 6.2: Valdidation accuracy and validation loss for the best fully-connected network for each nearest neighbor configuration. Figures (a) and (b) shows the complete results, while (c) and (d) is zoomed in on the best testing networks.

Table 6.2: Best performing fully-connected networks trained and validated on the aflow dataset.

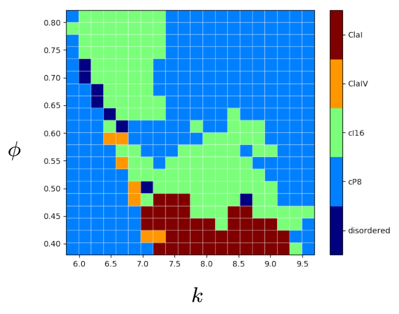
Neighbors	Layers	Activations	Dropout	Accuracy
10	[66, 40, 28, 22]	[Relu, Relu, Relu, Relu]	0.5	0.6876
20	[266, 160, 114, 88]	[Relu, Relu, Relu, Relu]	0.5	0.9182
30	[600, 360, 257, 200]	[Relu, Relu, Relu, Relu]	0.5	0.9367
40	[1066, 640, 457, 355]	[Relu, Relu, Relu, Relu]	0.5	0.9458
50	[1666, 1000, 714, 555]	[Relu, Relu, Relu, Relu]	0.5	0.9491
60	[2400, 1440, 1028, 800]	[Relu, Relu, Relu, Relu]	0.5	0.9533
70	[3266, 1960, 1400, 1088]	[Relu, Relu, Relu, Relu]	0.5	0.9552
80	[4266, 2560, 1828, 1422]	[Relu, Relu, Relu, Relu]	0.5	0.9566



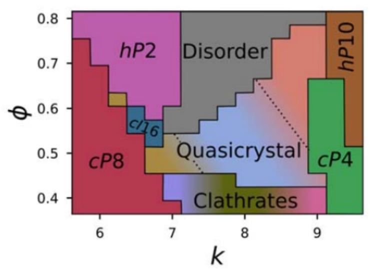
(a) 10 Neighbors



(b) 60 Neighbors

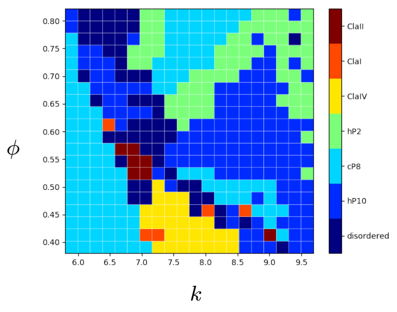


(c) 80 Neighbors

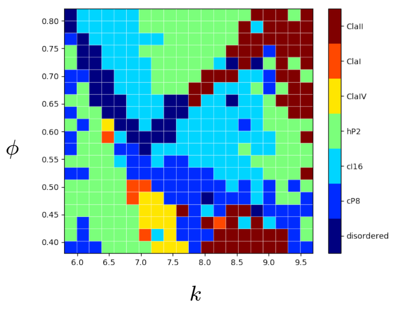


(d) Engel et al. manual classification

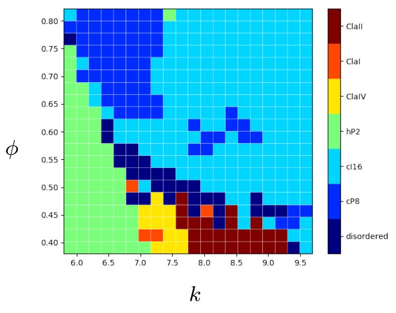
Figure 6.3: A phase representation of the structures found in the datasets created from the oscillating pair potential. We have plotted the most prevalent structure found in each dataset using fully-connected machine learning models trained on adjacency matrices of 10, 60 and 80 nearest neighbors.



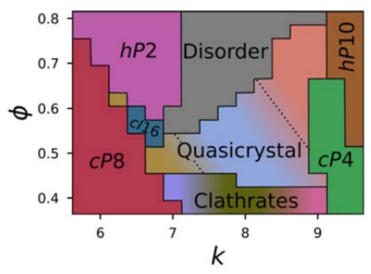
(a) 10 Neighbors



(b) 60 Neighbors



(c) 80 Neighbors



(d) Engel et al. manual classification

Figure 6.4: A phase representation of the structures found in the datasets created from the oscillating pair potential. We have plotted the second most prevalent structure found in each dataset using fully-connected machine learning models trained on adjacency matrices of 10, 60 and 80 nearest neighbors.

at every grid point. Counting the occurence and using the most prevalent structures is the same technique used by Engel et al. in their manual classification. The manual classification was done very rigorously by comparing five different simulations at each grid point.

The figures shown are for 10, 60 and 80 neighbors. We chose this set of neighbors to evaluate the performance between the worst and best performing network, 10 and 80 neighbors respectively, as well as 60 neighbors to see if a slightly worse performance on the training set have any impact on the results on the test set. The complete set of results is found in Appendix B.2. In all figures some of the structures are classified as disordered. We did not include a disordered classification in the training process of the networks, but to simulate a disordered dataset we set a limit on the probability of the classification of individual structures. If a structure is classified with a probability of less than 30 %, it is reclassified as disordered.

Starting with the 10-neighbors result, we see that albeit completely misclassified, the model does separate the cP8 region quite well, when comparing with the manual classification of Engel et al. in Figure 6.3d. We could also argue that the hP2 and cP4 regions er somewhat separated though none of them are correctly classified. What is a bit surprising is that it does find the clathrate sI structure quite well even though its unit cell contains 46 particles and we are only evaluating the topology of the 10 nearest neighbors. When increasing the neighbor count to 60 and 80 we see the same tendencies of finding separate regions, only with a better classification of the cP8 region. Unfortunately, the cP8 structure completely dominates the phase space making the classification useless. Because of this, we have tried in, Figure 6.4, plotting the second most prevalent structure if more than one structure is found in the data, and the most prevalent structure if only one structure is found. We try this approach because the classifier might find a high number of the correct structure in the dataset though the most prevalent structure is not the correct one. If the second most classified structure is the correct one, we can at least infer that the model is close to finding the correct structures after all.

When plotting the second most prevalent structure, we remove the dominance of cP8 in the phase space and even though we see better separation of the regions, the classification is still quite useless in all cases. Especially we can note that the cP4 structure has not been present in any of the plots. Another thing to notice is that when comparing the clathrate regions of 60 and 80 neighbors with the previous plots of the most prevalent structures, we actually get a good separation between the individual clathrate structures if the dominance of clathrate sI is reduced.

The cP8 structure which dominates the phase space is the Cr_3Si structure, shown in Figure 6.5. It is not obvious why exactly this structure is the dominant one, but when comparing the bonds of this structure for timestep 0 and timestep 300 during temperation, in Figure 6.5a and Figure 6.5b, we can see that the temperation has actually changed the structure to a large degree by adding several bonds in the structure. The goal of temperation is to add irregularities in the unit structures without breaking them. It is clear that the preliminary test of the break condition during temperation was not sufficient, and in this case, the temperation should have stopped earlier. As we are training our dataset on 30 sampling points of the temperation where the adding of bonds will get progressively worse throughout, this might make this structure look similar to several other structures in the dataset, making it harder to separate them in the phase space.

Another possibility for this dominance might be that we cooled the particle systems faster when creating the datasets than the original authors did, altering the intrinsic structures too much in the process. The fact that the network is largely able to find the manually classified regions of Engel et al. but not classify them correctly might suggest that this is actually a significant contributing factor to the results.

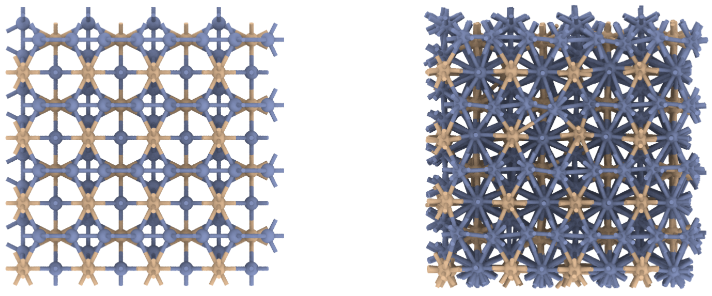
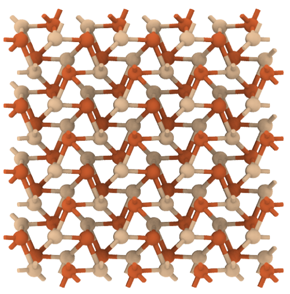

(a) Cr_3Si (cP8): Timestep 0 (b) Cr_3Si (cP8): Timestep 300

Figure 6.5: Figure (a) and (b) shows a side view of the most prevalent structure found in the phase dataset. In Figure (a) we see the idealized unit structure and in Figure (b) the same structure temperated for 300 timesteps using a harmonic potential. We can see that the temperation of the crystal has been too excessive, and there have been added several new bonds to the structure. The goal of temperation is to add some irregularities to the unit structure, however, the structure should still be largely intact and this figure shows the structure has been altered to a large extent. This suggests that the criterion for stopping the temperation should have been more rigorous.

The Methane Hydrate Dataset

In Figure 6.7 we have tested the best network for 10, 60 and 80 neighbors on the methane hydrate dataset. The complete set of results is found in Appendix C.2.

All the networks shown in this section find several structures in the dataset, which makes the cluster colors blend together. Because of this, we have kept the three most prevalent structures as separate colors and merged the rest into an unlabeled cluster, which is shown in blue. Comparing with the result of CHILL+ in Figure 6.7d, we see that the same grain boundaries appear for all neighbor configurations, although far more clearly for a higher number of neighbors. For a high number of neighbors, we can argue that visually the grain boundaries are just as defined, if not more so, as the result of CHILL+. A thing to notice is that the grain boundaries are almost entirely classified as unlabeled, which is a big difference between these results and the result of CHILL+, where the grain boundaries are a mixture of unlabeled particles and interfacial hydrate. The CHILL+ algorithm is, however, completely dependant on the removal of the methane particles before the algorithm is applied, as


(a) $FeSi$ (cP8): Timestep 0

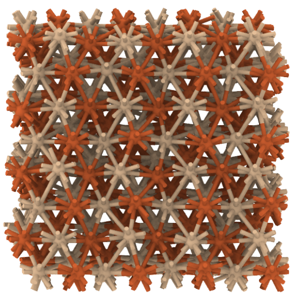
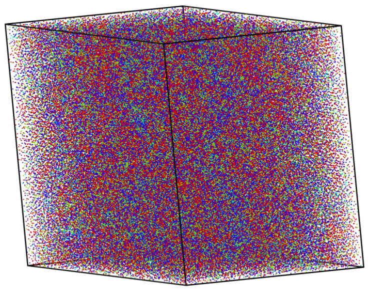
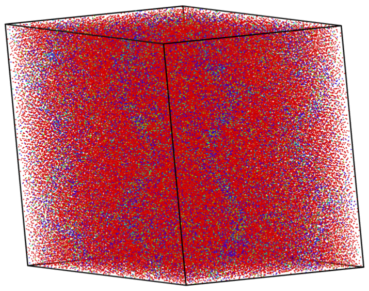

(b) $FeSi$ (cP8): Timestep 300

Figure 6.6: When testing the network on the methane hydrate dataset and only keeping the three most prevalent structures, we find an abundance of the structure shown in (c) and (d). This structures exhibits a cage-like nature after the temperation, which might make them look like deformed methane hydrate structures to the machine learning network.

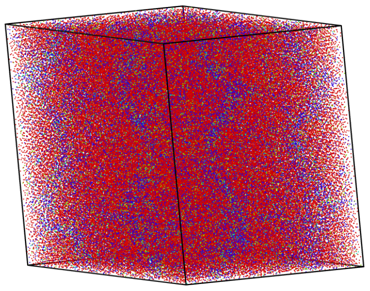
we have seen in Figure 5.1b, while the machine learning approach makes no such assumptions and is able to find the grain boundaries while the methane particles are still present. The three most prevalent structures and their cluster sizes are shown in Table 6.3. The structure found by the 60 and 80 neighbor networks, in addition to the unlabeled and methane hydrate structures, can be seen in Figure 6.6. After temperation, this structure exhibits a cage-like nature and we can argue that again the temperation has been too excessive. Because the temperation is not to be entirely trusted, we can not be certain that the structure found is indeed the correct structure in the dataset, but to represent the interfacial (deformed) hydrates found in the CHILL+ algorithm we do need some structures which exhibits similarity to the clathrate hydrates, only with some deformation. Because of the shape of this structure after temperation, it is not unreasonable that it is classified as a deformed hydrate by the algorithm.



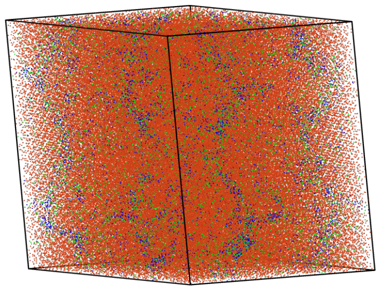
(a) **10 Neighbors:**
Red: sI, Yellow: sII,
Green: Er_3Ru_2 (hP10),
Blue: Unlabeled



(b) **60 Neighbors:**
Red: sI, Yellow: $FeSi$ (cP8),
Green: sII, Blue: Unlabeled



(c) **80 Neighbors:**
Red: sI, Yellow: sII,
Green: $FeSi$ (cP8), Blue: Unlabeled



(d) CHILL+

Figure 6.7: Fully-connected network trained on the aflow dataset applied to simulation of methane hydrate under stress. Figures shown are for adjacency matrices of 10 (a), 60 (b) and 80 (c) neighbors. Only the three most prevalent structures are kept and the rest are merged into an unlabeled cluster. Comparing with the results of CHILL+ (d) we see that both 60 and 80 neighbors are able to find the grain boundaries of the system.

Table 6.3: Structures found in methane hydrate dataset.

Structure	Cluster Size
10 Neighbors	
Methane hydrate I (sI)	82113
Methane hydrate II (sII)	52229
$Er_r Ru_2$ (hP10)	19819
Unlabeled	92859
60 Neighbors	
Methane hydrate I (sI)	180738
$FeSi$ (cP8)	14348
Methane hydrate II (sII)	11318
Unlabeled	40616
80 Neighbors	
Methane hydrate I (sI)	162544
Methane hydrate II (sII)	16527
$FeSI$ (cP8)	7792
Unlabeled	60157

6.3 Convolutional Network

The particle datasets are transformed into adjacency matrices which can be interpreted as black and white images. Because the most widely used neural network for classifying images are convolutional networks, utilizing this seems like a reasonable approach. To construct a convolutional network, we create a ConvNet class which inherits from the NeuralNet class.

```python
class ConvNet(NeuralNet):
    def __init__(
        self, num_classes=255, layers='caccaca',
        dropout=0.1, layer_params=None
        ):
        super().__init__()
        layer_names = {'c': Conv2D, 'a':AveragePooling2D}
        model = Sequential()
        for i, layer in enumerate(layers):
            model.add(layer_names[layer](**layer_params[i]))
            if dropout and layer=='a':
                model.add(Dropout(dropout))
        model.add(Flatten())
        if dropout:
            model.add(Dropout(dropout))
        model.add(Dense(num_classes, activation='softmax'))
        model.compile(
            optimizer='adam', loss='categorical_crossentropy',
            metrics=['accuracy']
            )
        model.summary()
        self.model=model
```

Table 6.4: Architectures Tested

Input Dim $< 30 \times 30$	Input Dim $\geq 30 \times 30$
Conv[16]	Conv[16]
Pool[2]	Pool[2]
Conv[32]	Conv[32]
Conv[64]	Conv[64]
Conv[128]	Pool[2]
	Conv[128]

The architecture of the convolutional network is given in the layers argument of the constructor. In this implementation, we only allow two layer types, the convolutional two-dimensional layer and an averaging 2×2 pooling layer, which reduces the size of the image by averaging a pixel area. Before calculating the loss function, the output from the last convolutional layer is flattened and run through a fully-connected layer with softmax activation. If the dropout argument is given, dropout layers are also added after every pooling layer and after the flattening of the convolutional output. To prevent overfitting, we also use the EarlyStopping callback to stop the training process when no improvement is detected on the validation set for 10 epochs. In Appendix A, an overview of the hyperparameters tested are given.

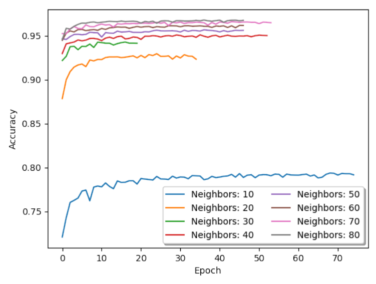
In Table 6.4 we see the two types of architectures tested, where we have used one less pooling layer for adjacency matrices with less than 30 neighbors to keep the output feature map at a reasonable size.

The best results for each of the nearest neighbor configurations of the adjacency matrices trained on the aflow dataset are shown in table 6.5. We can see the accuracy does increase as we increase the number of neighbors. However, the size of the added improvement declines for each neighbor configuration, as was the case for the fully-connected network. One could argue there are two reasons for this. Firstly, even though adding more neighbors adds more information about the local topology of each atom, there is probably a limit to how much extra information is added as we increase the number of neighbors beyond the number of particles in the unit cell of the largest structure. Secondly, we are adding quite a lot of dimensions to the classification problem as we increase the number of neighbors which does make it harder for the network to find the relevant information for a good classification.

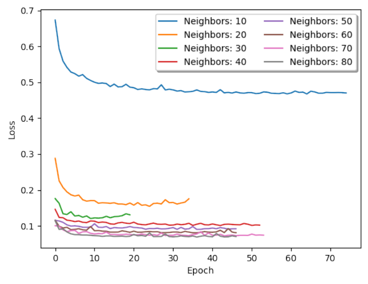
From Figure 6.8, we see the same trend as with the fully-connected network, a higher number of neighbors tends to increase the accuracy on the validation data and there is a large jump in accuracy from 10 to 20 neighbors. Just like with the fully-connected network there is also not that big of a difference between the accuracy for 40-80 neighbors. However, there is a more distinct separation in the accuracy between each neighbor increment. The convolutional network also performs better for each of the neighbor configurations than the fully-connected networks. However, the difference is so small we can not infer just from this result that the convolutional networks will perform better on testing data than the fully-connected ones.

Table 6.5: Best Architectures

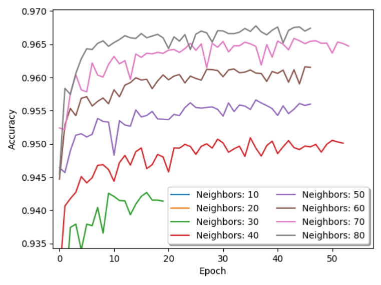
Neighbors	Filter	Activation	Dropout	Accuracy
10	5×5	[Relu, Tanh, Tanh, Relu]	0.3	0.7937
20	5×5	[Relu, Relu, Relu, Relu]	0.5	0.9295
30	5×5	[Relu, Relu, Relu, Tanh]	0.3	0.9427
40	5×5	[Tanh, Relu, Tanh, Relu]	0.5	0.9509
50	5×5	[Relu, Relu, Tanh, Relu]	0.5	0.9566
60	5×5	[Relu, Relu, Tanh, Relu]	0.5	0.9616
70	5×5	[Relu, Tanh, Tanh, Relu]	0.5	0.9659
80	5×5	[Relu, Relu, Relu, Relu]	0.5	0.9678



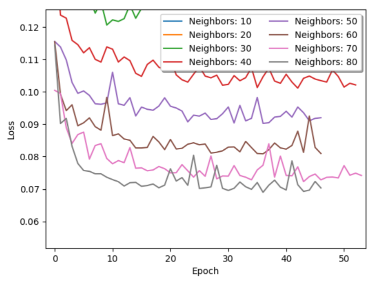
(a)



(b)



(c)



(d)

Figure 6.8: The best convolutional networks for each nearest neighbor size used in the adjaceny matrix calculation. Figures a and c shows the validation accuracy of the networks and figures b and d shows, the loss for the same networks. We clearly see that increasing the number of neighbors does improve the accuracy and loss of the network.

Table 6.6: Structures found with convolutional network.

Structure	Cluster Size
10 Neighbors	
Methane hydrate I (sI)	81237
Methane hydrate II (sII)	47141
CoU (cI16)	34064
Unlabeled	84578
60 Neighbors	
Methane hydrate I (sI)	142124
High pressure Li (cI16)	36903
$FeSi$ (cP8)	21305
Unlabeled	46688
80 Neighbors	
Methane hydrate I (sI)	161231
$BC8$ (cI16)	29086
High pressure Li (cI16)	16026
Unlabeled	40677

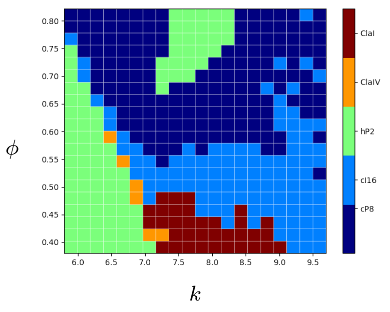
The Phase Dataset

We tested the network on the phase dataset in the same fashion as for the fully-connected network, using the same probability tolerance of 30%. The results for all neighbor combinations can be seen in Appendix B.1.

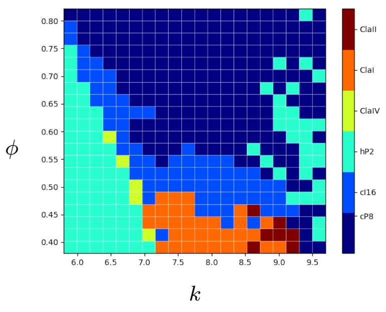
The most prevalent structure of neighbors 10, 60 and 80, for each combination of ϕ and k, are shown in Figure 6.9. In this figure, we can see the same trends as with the fully-connected network. The 10 neighbor network seems to be somewhat able to separate interesting regions. However, the classification itself is very inaccurate except arguably for the clathrate sI structure. Increasing the neighbors also shows the same trends with separation of regions and cP8 being the dominant structure.

When plotting the second most prevalent structure in Figure 6.10 we can perhaps argue we get a better result than for the fully-connected case. Especially we can see that the hP10 region is mostly correctly classified for both 60 and 80 neighbors, which was not the case for the fully-connected network. In addition, we see that the cP4 structure is present in the phase space, although it is not correctly classified. Again the separation of clathrates is quite good, which we will examine more closely on the methane hydrate dataset.

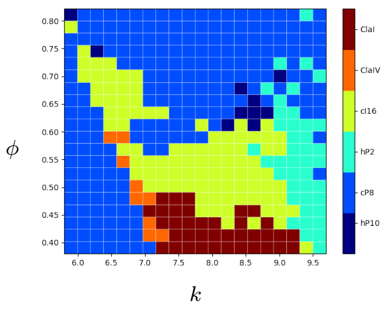
If we accept the argumentation put forth in the discussion of the fully-connected networks, and assume the misclassification is, to a large degree, a result of the cooling process in the dataset creation, then these results are actually not that bad. Ignoring the regions of disorder and quasicrystals, we can see that at least for the 60 and 80 neighbor networks, the regions are all quite well separated. This means that the networks understand that there are separate structure types in these regions. Iit is just not able to find the correct structures with the current dataset.



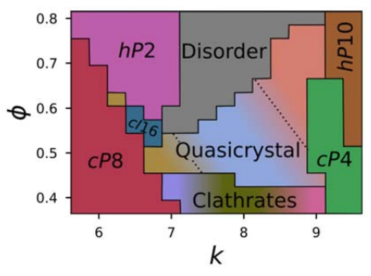
(a) 10 Neighbors



(b) 60 Neighbors

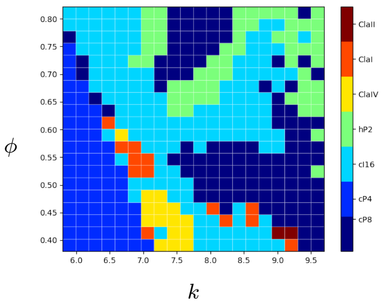


(c) 80 Neighbors

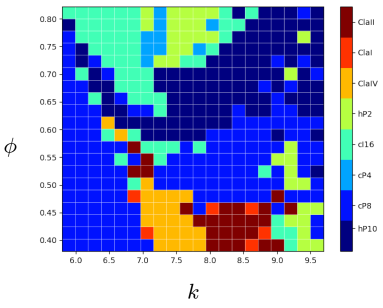


(d) Engel et al. manual classification

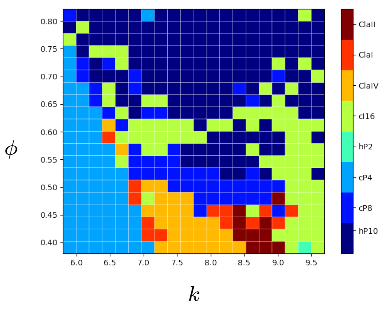
Figure 6.9: A phase representation of the structures found in the datasets created from the oscillating pair potential. We have plotted the second most prevalent structure found in each dataset using a convolutional neural network trained on adjacency matrices of 10, 60 and 80 nearest neighbors.



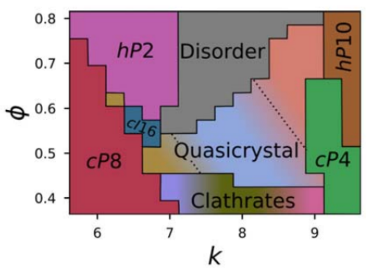
(a) 10 Neighbors



(b) 60 Neighbors

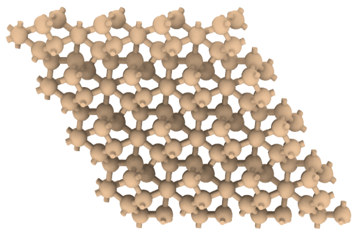


(c) 80 Neighbors

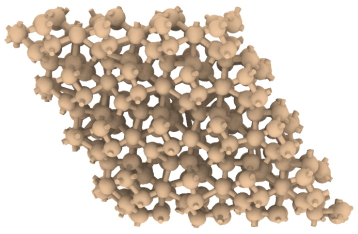


(d) Engel et al. manual classification

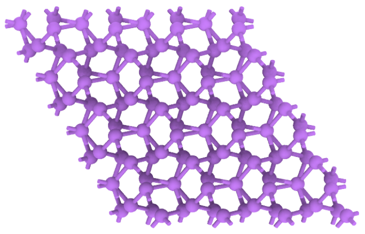
Figure 6.10: A phase representation of the structures found in the datasets created from the oscillating pair potential. We have plotted the second most prevalent structure found in each dataset using a convolutional neural network trained on adjacency matrices of 10, 60 and 80 nearest neighbors.



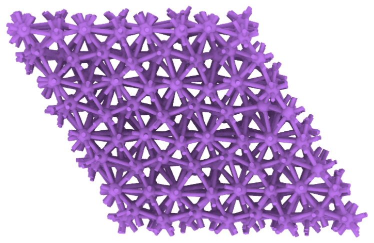
(a) BC8 (cI16): Timestep 0.



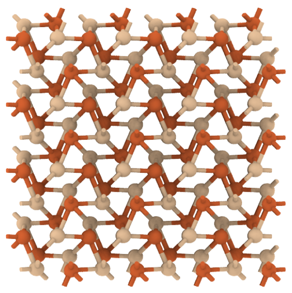
(b) BC8 (cI16): Timestep 300.



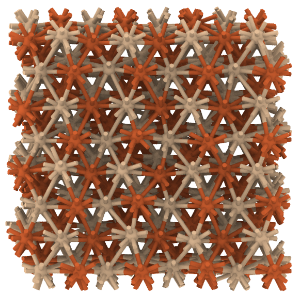
(c) High-pressure Li (cI16): Timestep 0.



(d) High-pressure Li (cI16): Timestep 300.



(e) $FeSi$ (cP8): Timestep 0



(f) $FeSi$ (cP8): Timestep 300

Figure 6.11: Structures found in methane hydrate simulation in adition to methane hydrate structures, using 60 and 80 neighbors. All structures exhibit a cage-like nature after temperation.

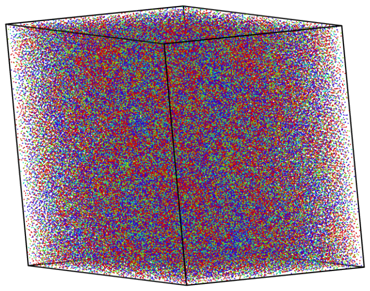
The Methane Hydrate Dataset

For the methane hydrate dataset, we again focus on the results of 10, 60 and 80 neighbors. The complete set of results for all neighbors can be seen in Appendix C.1. We color the three most prevalent structures individually and merge the rest into a single cluster of unlabeled structures colored in blue. The results are shown in Figure 6.12.

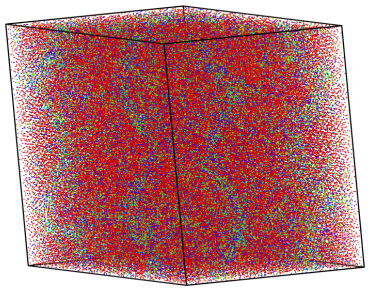
The results look very similar to the fully-connected case and the grain-boundaries found in the CHILL+ algorithm is unmistakable once again. There is a slight difference compared to the fully-connected result, however. For 60 and 80 neighbors, we see a much larger number of labeled particles in the grain boundaries, meaning the network is far more confident in the clustering results of the three largest clusters. In Table 6.6 we see the structures found for each nearest neighbor model, and the structures found by the 60 and 80 neighbor networks are shown in Figure 6.11. The $BC8$ structure in Figures 6.11a and 6.11b largely keeps its initial structure during temperation only with some added irregularities. Because the structure has not been completely altered, we can have more confidence that this is actually a structure which is present in the dataset. This is only to some extent though, as we have only trained the network to recognize a small subset of Pearson structures, and the confidence the network has in the $BC8$ structure might drop drastically if other structures were to be introduced in the training set.

The two other structures, in Figures 6.11d and 6.11f, has been excessively temperated leading to altered structures. When temperated they both look cage-like, and because of this, they might look like deformed hydrates to the algorithm. Even though this helps in finding the grain boundaries, because of the structure alteration, we can not have any confidence in the actual classification of these structures in the grain boundaries.

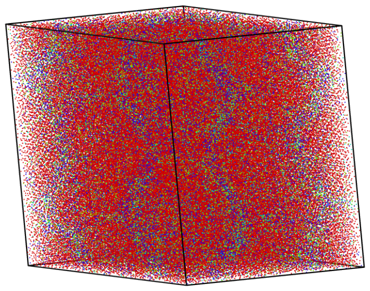
Finally, we have trained the best models on the clathrate dataset only containing the three clathrate structures and tested on the methane hydrate dataset. The results are shown in Figure 6.13. We see that even though the grain boundaries are still present, they are far less prevalent than the results using the phase dataset. This is not that surprising considering the previous results did not find a lot of sII and H structures in the grain boundaries, and to represent the interfacial hydrate structures of CHILL+ we need some structures in the training dataset that are quite similar to the sI structure only with some bonds added or removed. So, to find hydrate grain boundaries, it is more advantageous for the network to have seen both hydrates and other structures. Not just the sI, sII and H hydrates.



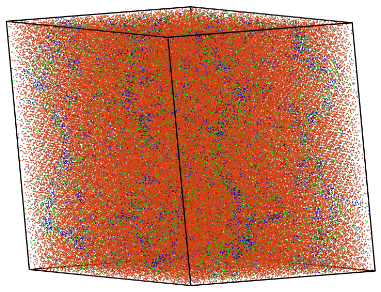
(a) **10 Neighbors:**
Red: sI, Yellow: sII,
Green: CoU (cI16), Blue: Unlabeled.



(b) **60 Neighbors:**
Red: sI, Yellow: BC8 (cI16),
Green: *FeSi* (cP8), Blue: Unlabeled.

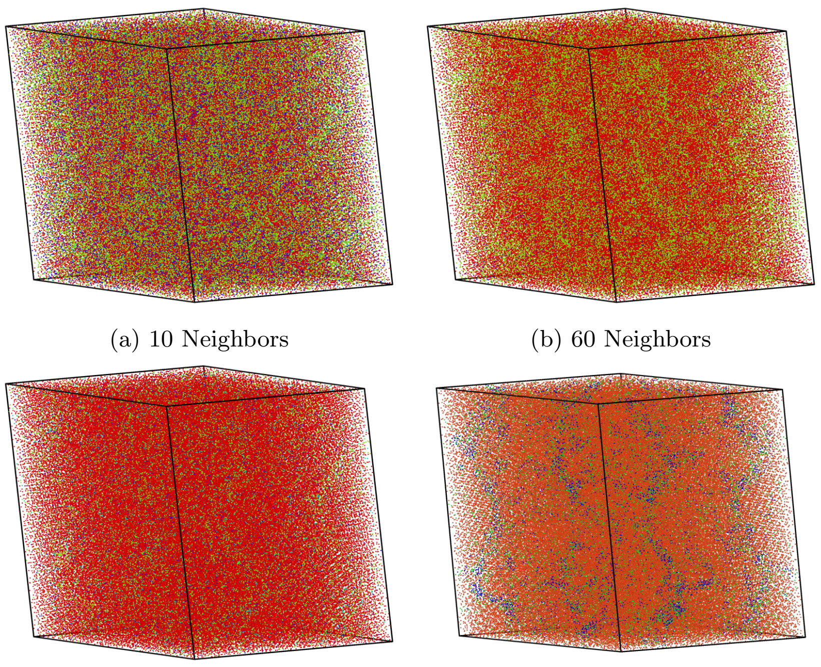


(c) **80 Neighbors:**
Red: sI, Yellow: BC8 (cI16),
Green: High pressure *Li* (cI16), Blue: Unlabeled.



(d) CHILL+

Figure 6.12: Convolutional neural network trained on the aflow dataset applied to simulation of methane hydrate under stress. Figures shown are for adjacency matrices of 10 (a), 60 (b) and 80 (c) neighbors. Only the three most prevalent structures are kept and the rest are merged into an unlabeled cluster. Comparing with the results of CHILL+ (d) we see that both 60 and 80 neighbors are able to find the grain boundaries of the system.



(a) 10 Neighbors

(b) 60 Neighbors

(c) 80 Neighbors

(d) CHILL+

Figure 6.13: Convolutional neural network trained on the clathrate hydrate dataset containing sI, sII and H clathrate structures. The trained network is applied to a simulation of sI methane hydrates under stress. Figures shown are for 10 (a), 60 (b) and 80 (c) neighbors in the adjacency matrices. Comparing with the results of CHILL+ (d) we see that both 60 and 80 neighbors are able to find the grain boundaries of the system, however, they are not very prominent suggesting there are not many sII and H structures developing in the boundaries.

Chapter 7

Unsupervised Learning

Unsupervised learning removes the difficulty of acquiring labeled data for machine learning tasks. Because of this significant advantage, finding unsupervised models which perform as well as supervised models are of great interest. The most common unsupervised learning task is the clustering of data. With clustering the machine is trying to find underlying structures in the dataset by evaluating such metrics as the density and the distances within the dataset.

The dataset we have used for unsupervised clustering is a methane hydrate simulation under stress which has grain boundaries in the crystal. A side view of the dataset is shown in Figure 7.1a, where the blue atoms are methane and the red are water molecules. In Figure 7.1b, we have applied the CHILL+ algorithm to this dataset, which is able to find the grain boundaries. The CHILL+ result will be our benchmark for the unsupervised clustering results.

The dataset was first transformed into adjacency matrices for each of the particles in the dataset. We have used a range of nearest neighbors for the adjacency calculations, starting from 20 and up to 70 with 10 neighbor increments. This gives in total six different datasets ranging from 400 to 4900 features as shown in Table 7.1.

7.1 Dimensionality Reduction

There are two main reason, for performing dimensionality reduction on a dataset.

Table 7.1: Outer neighbors of adjacency matrices and the number of input features to the clustering algorithm

Neighbors	Input Dimensions
20	400
30	900
40	1600
50	2500
60	3600
70	4900

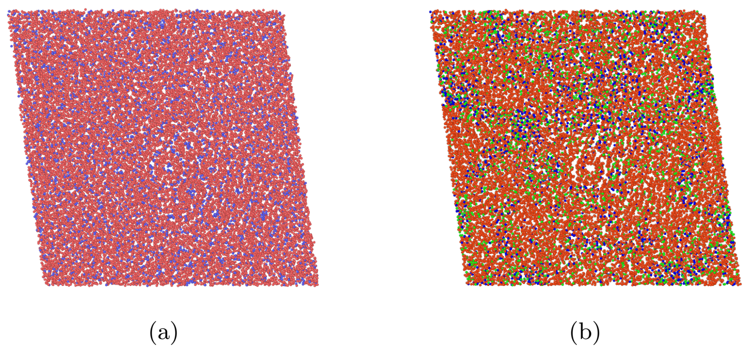


Figure 7.1: Side view of the three-dimensional 30716 particle methane hydrate simulation used for testing and training. The blue atoms are methane and the red are water molecules (a).The same simulation with the CHILL+ algorithm of Ovito coloring the atoms is shown in (b). The orange atoms are classified as sI hydrate, the green as interfacial hydrate and the blue as unlabeled.

Firstly, we are working with datasets containing thousands of features, and without a reduction in the number of features it is not feasible to expect the machine to find any meaningful patterns in the data. One reason for this is the time complexity of unsupervised learning, but the most important is the curse of dimensionality, as we discussed in Section 3.8.4 on density-based clustering. Because we are very dependent on the density and the distance between data points in clustering algorithms, adding too many dimensions is likely to influence the prediction outcome negatively.

Secondly, most datasets contain a certain amount of noise. It might not be inherently obvious from the dataset itself and a dimensionality reduction technique, like principal component analysis, might help improve the learning accuracy by removing the dimensions containing the least amount of information.

We have used two techniques for reducing the dimensionality of the data. Principal component analysis, where we have kept 90% of the explained variance, and an autoencoder mapping the features into a latent space of lower dimensionality.

7.1.1 Autoencoder

As a means of dimensionality reduction in unsupervised learning, we have implemented an autoencoder. If the autoencoder perfectly recreates the input, the latent space of the autoencoder is a perfect representation of the input in lower dimensions. We have tried to utilize this when clustering by clustering directly on the latent space after the autoencoder has been trained.

We have used Keras to implement an AutoEncoder class, which handles the splitting of input data into training and validation data defined using a percentage argument defining the amount of training data we want, as well as the setup of the autoencoder architecture.

```
from keras.layers import Input, Dense, Dropout
from keras.models import Model
from keras.callbacks import EarlyStopping
from keras import regularizers
# from sklearn.metrics import roc_auc_score
# import tensorflow as tf
import numpy as np
import os

class AutoEncoder:
    def __init__(self, epoch, frame, dump_filename):
        self.epoch = epoch
        self.data_train = None
        self.data_test = None
        self.dump_filename = dump_filename + f'_frame{frame}'

    def train_test(self, data, percent_training_data=0.6):
        data = data.reshape(len(data), np.prod(data.shape[1:]))
        N_training_samples = int(
            np.floor(percent_training_data*data.shape[0])
            )
        data_train = data[:N_training_samples]
        data_test = data[N_training_samples:]
        data_train = data_train.reshape(
            len(data_train), np.prod(data_train.shape[1:])
            )
        data_test = data_test.reshape(
            len(data_test), np.prod(data_test.shape[1:])
            )
        self.data_train = data_train
        self.data_test = data_test
        self.data = data
        return data_train, data_test
```

For the autoencoder architecture, we have implemented two options. The autoconstruct method adds fully-connected layers with output sizes corresponding to powers of two, with the smallest possible power defined in the argument and the largest power being the largest possible without being larger than the input data. So with 100 input features and a power_start of 4 we get an encoder architecture as described in Table 7.2.

Table 7.2: Example layer structure of autoencoder.

Input Size	Hidden Layers Size		Latent Space Size
100	2^6	2^5	2^4

The encoder and decoder are mirror images of each other, not considering the latent space, with the output of the decoder being the same size as the input, as is required for an autoencoder. Each layer in the autoconstruct method uses relu as the activation function.

```
    def autoconstruct_layers(
        self, N_input_features, power_start=4
        ):
        bases = np.arange(15)
        powers = np.power(2, bases)[power_start:]
        startunit = np.argmax(powers > N_input_features) - 1
        input_img= Input(shape=(N_input_features,))
        encoded = Dense(
            units=powers[startunit], activation='relu'
            )(input_img)
        for i in reversed( range(startunit)):
            encoded = Dense(
                units=powers[i], activation='relu'
                )(encoded)
        decoded = Dense(
            units=powers[1], activation='relu'
            )(encoded)
        for i in range(startunit+1):
            decoded = Dense(
                units=powers[i], activation='relu'
                )(decoded)
        decoded = Dense(
            units=N_input_features, activation='relu'
            )(decoded)
        return input_img, encoded, decoded
```

The construct method gives more options to the user as the parameters of each layer must be defined explicitly with size and activation function. Only the encoder layers are given since the decoder is just the reverse implementation of the encoder layers.

```
    def construct_layers(
        self, N_input_features,
        layers:list, activations=None
        ):
        if activations==None:
            activations=['relu']*len(layers)
        input_img = Input(shape=(N_input_features,))
        encoded = Dense(
                units=layers[0], activation=activations[0],
                kernel_regularizer=regularizers.l2(0.01),
                activity_regularizer=regularizers.l1(0.01)
                )(input_img)
        for layer, activation in zip(
                    layers[1:], activations[1:]
                    ):
            encoded = Dense(
                    units=layer, activation=activation,
                    kernel_regularizer=regularizers.l2(0.01),
                    activity_regularizer=regularizers.l1(0.01)
                     )(encoded)
        decoded = Dense(
                units=layers[-2], activation=activations[-2],
                kernel_regularizer=regularizers.l2(0.01),
                activity_regularizer=regularizers.l1(0.01)
                )(encoded)
        for layer, activation in zip(
                    layers[-3::-1],
                    activations[-3::-1]
                    ):
            decoded = Dense(
                    units=layer, activation=activation,
                    kernel_regularizer=regularizers.l2(0.01),
                    activity_regularizer=regularizers.l1(0.01)
                    )(decoded)
        decoded = Dense(
            units=N_input_features, activation='relu'
            )(decoded)
        return input_img, encoded, decoded
```

The autoencoder method trains the autoencoder with the training data given. What we are really interested in is the latent space of the autoencoder. Because of this, what is returned from the autoencoder method is the encoded inputs of the entire dataset after the autoencoder has been trained. The input has now been mapped to the same dimension as the latent space.

```
    def autoencoder(
                self, data_train=None, data_test=None,
                autoconstruct_layers=True, power_start=4,
                input_layers=None, activations=None,
                recompute=False,
                ):
            if input_layers is None:
                self.dump_filename = (
                    self.dump_filename
                    + f'_autoconstructpowerstart{power_start}.npy'
                    )
            else:
                layers = '|'.join(map(str, input_layers))
                act = '|'.join(activations)
                self.dump_filename = (
                        self.dump_filename
                        + f'_layers{layers}_act{act}.npy'
                        )

            if (os.path.isfile(self.dump_filename)
                and not recompute
                ):
                print(
                    f'Precalculated: Loading {self.dump_filename}'
                    )
                encoded_imgs = np.load(self.dump_filename)
                return encoded_imgs, self.dump_filename
            else:
                if (self.data_train is not None
                    and self.data_test is not None
                    ):
                    data_train, data_test = (self.data_train,
                                             self.data_test
                                             )
                N_input_features = data_train.shape[1]
                if input_layers==None:
                    input_img, encoded, decoded = (
                        self.autoconstruct_layers(
                            N_input_features, power_start
                            )
                        )
                else:
                    input_img, encoded, decoded = (
                        self.construct_layers(
                            N_input_features,
                            input_layers,
                            activations
                            )
                        )
                autoencoder=Model(input_img, decoded)
                encoder = Model(input_img, encoded)
                autoencoder.summary()
```

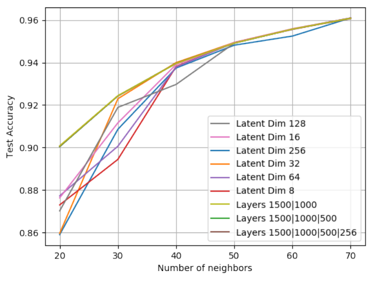
Table 7.3: Autoencoder architectures tested.

Activations	Layers
Relu	$[2^N, 2^{N-1}, \cdots, 2^3]$
Relu	$[2^N, 2^{N-1}, \cdots, 2^4]$
Relu	$[2^N, 2^{N-1}, \cdots, 2^5]$
Relu	$[2^N, 2^{N-1}, \cdots, 2^6]$
Relu	$[2^N, 2^{N-1}, \cdots, 2^7]$
Relu	$[2^N, 2^{N-1}, \cdots, 2^8]$
Relu	$[1500, 1000]$
Relu	$[1500, 1000, 500]$
Relu	$[1500, 1000, 500, 256]$

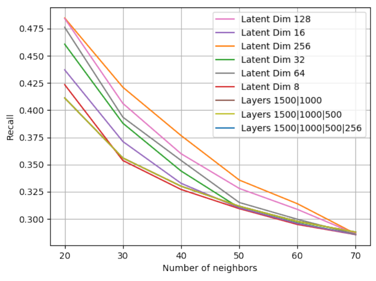
```
autoencoder.compile(
    optimizer='adam',
    loss='binary_crossentropy',
    metrics=['accuracy']
    )
callbacks = [
    EarlyStopping(monitor='val_loss', patience=10)
    ]
autoencoder.fit(data_train, data_train,
                epochs=self.epoch,
                callbacks=callbacks,
                batch_size=256,
                shuffle=True,
                validation_data=(
                        data_test,
                        data_test)
                )
evaluation = autoencoder.evaluate(
    data_train, data_train
    )
np.save(
    self.dump_filename+'_evaluation',
    evaluation
    )
encoded_imgs = encoder.predict(
    self.data, verbose=1
    )
print(f'Creating file {self.dump_filename}')
np.save(self.dump_filename, encoded_imgs)
return encoded_imgs, self.dump_filename
```

In Table 7.3 we see all the autoencoder architectures tested. Each of these architectures was tested on adjacency matrices of the methane hydrate dataset, shown in Figure 7.1, using between 20 and 70 nearest neighbors with 10 neighbor increments, as defined in Table 7.1.

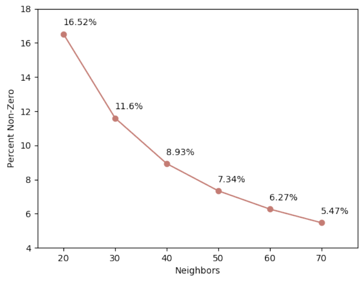
In Figure 7.2a, we see the validation accuracy of the autoencoder as a function of the number of neighbors in the adjacency matrices. This gives the impression that increasing the number of neighbors yields a better result. This is, however, not a correct assessment. As we increase the number of neighbors we are also increasing the percentage of 0s in the adjacency matrix because adding more neighbors in the matrix does not, in general, mean that more neighbors



(a)



(b)



(c)

Figure 7.2: The accuracy (a) and recall (b) of the autoencoders tested. Because the percentage of 0s in the dataset is far greater than the 1s (c), we have to use the recall as a performance measure.

become neighbors of each other. In Figure 7.2c we see the percentage of 1s in the matrices as a function of the number of neighbors. We can see that for 70 neighbors, only 5.47% of the 4900 binary numbers in the matrix are 1s. In practice, this means that if the machine predicts all the numbers to be 0s, it will have an accuracy of 94.53%. Because of this, we need another metric to define how well the algorithm is doing. In Figure 7.2b we see the recall of the class of 1s as a function of neighbors. We can now see that reducing the number of neighbors actually yields better results as the recall is higher. We can also see that a higher latent dimensionality generally gives a better recall score. None of the networks gives a very impressive recall score, however, as it ideally should have been close to 1. The best recall score of 0.476 means that only about half of the 1s in the adjacency matrix is actually recreated. The rest are set to zero. We can also see that the autogenerated hidden layers of powers of two give the best result of the networks tested. The general trend for the latent dimension is that higher dimensions give better recreation results, which is not surprising as we can view this as less compression of the image, and less compression makes it easier to recreate.

7.1.2 PCA

For the principal component analysis dimensionality reduction we have used the Scikit-learn built-in PCA class. This object as a method for finding the explained variance of each component in the new eigenvector space, which as we explained in the theory chapter is ordered from highest variance to lowest. To ensure we do not remove too much of the relevant information, we take the cumulative sum of the explained variance and keep the first n components equating to 90 percent of the variance. There is no rule of thumb explaining how much variance and how many components should be kept, other than it should be “high enough”, and so the choice of 90 percent is a bit arbitrary. Our reasoning has been that if we are working in a low dimensional space, each component is more likely to account for a larger percentage of the variance, making each component more “valuable” and thus keeping a large percentage in the high 90s might be reasonable. In addition, it is not as relevant to remove many dimensions in a low dimensional space as the dimensionality may already be quite manageable. We are, on the other hand, working in a space of between 400 and 4900 dimensions and we are likely to have many components with low variance. These components may add up to a large variance but individually their variance may be very low, and hence expendable.

```
def compute_pca(self, data=None):
        if data is None:
            data = self.data

        if data.ndim > 2:
            print(
              f'Data shape is {data.ndim}, raveling down to 2 dims'
              )
            data = data.reshape(len(data), np.prod(data.shape[1:]))

        pca = PCA().fit(data)
        explained_v = pca.explained_variance_ratio_
        cum = np.cumsum(explained_v)
        idx = np.argmax(cum > 0.9)
        return pca.transform(data)[:, :idx]
```

With the dimensionality reduction techniques applied, we now move on to clustering the data.

7.2 Clustering

The unsupervised clustering task is particularly difficult as we do not have a labeled training set to validate the performance of the clustering method. We instead used the unsupervised performance metrics defined in Section 3.9 to evaluate the models. These methods measure the density and separation of the clusters but are not guaranteed to be good measures of how well the clustering method is doing. There is also not a given which of the evaluation methods are best, and they often do not agree on which model is the best. Because of this, we have used the metrics separately to order the models from best to worst for each evaluation metric , and manually check the results for each of them. All the results can be seen in Appendix D.

A big difference between the supervised and unsupervised implementations is that the clustering results say nothing about what kinds of structures exist in the dataset. Clustering can only help reveal clusters of similar structures by coloring the particles in a simulation by their respective cluster values, but the actual labeling of what kinds of structures exist must be done manually.

In this section, we are only going to focus on some selected results. Most of the results only found one or two clusters in the dataset, and we will focus on the results that found three or more. A thing to notice is that none of the selected results use PCA as a dimensionality reduction technique. We have not kept track of the number of dimensions retained after the PCA reduction, but with 90% variance kept it is likely that quite a lot of dimensions prevail. The bad results for PCA are then likely to be because of the curse of dimensionality, though we have not tested this explicitly.

We did not use separate testing and training sets for the clustering implementations. We tested and trained the four clustering algorithms defined in Section 3.8; agglomerative clustering, Gaussian mixture models (GMM), OPTICS and DBSCAN on the methane hydrate data shown in Figure 7.1. The dataset was first transformed into adjacency matrices with nearest neighbors $N \in [20, 30, 40, 50, 60, 70]$. The dimensionality of these matrices was then reduced by using both PCA and the latent spaces of the autoencoder networks shown in Table 7.3. This gives a total of 60 different sets of features describing the methane hydrate dataset. These 60 feature sets were then clustered with the different clustering algorithms. Furthermore, the results were evaluated with the unsupervised performance metrics; Calinski-Harbasz, Davies-Bouldin and silhouette score.

The Cluster class we have implemented uses Scikit-learn for all clustering and evaluation tasks as all the algorithms are implemented in the framework. Scikit-learn has no GPU implementation available as it is trying to keep its library as lightweight as possible. Advantages of Scikit-learn is that we can keep the code very compact for machine learning methods implemented in the library but we are sacrificing flexibility and so other machine learning frameworks might be more suitable for more complex tasks, like implementing a new clustering algorithm from scratch.

The unsupervised performance metrics are all implemented in Scikit-learn, and we use these functions directly in our own implementations.

```
Class Cluster:
    def __init__(self, data):
    self.data = data

    def evaluate(self, data, labels):
        calinski = metrics.calinski_harabasz_score(data, labels)
        silhouette = metrics.silhouette_score(data, labels)
        davies = metrics.davies_bouldin_score(data, labels)
        return {
            'calinski': calinski,
            'silhouette': silhouette,
            'davies': davies
            }
```

We have chosen to separate the clustering results into density-based clustering, referencing the OPTICS and DBSCAN methods which both bases their algorithms on density measures, and non-density-based clustering, referencing Gaussian mixture models and agglomerative clustering as these methods base their algorithms purely on distance measures.

7.2.1 Density Based Clustering

With density-based clustering, we are referring to the DBSCAN and OPTICS algorithms. We have created separate methods in the Cluster class for each of the clustering methods, both using the implementations imported from Scikit-learn.

```
    def optics(self, params):
        for eps, sample in params:
            optics_model = OPTICS(
                eps=eps,min_samples=sample
                ).fit(self.data)
            labels = optics_model.labels_
            try:
                evaluate = (
                    self.evaluate(self.data, labels),
                    Counter(labels)
                    )
            except ValueError:
                evaluate = Counter(labels)
            yield labels, evaluate

    def dbscan(self, params):
        for eps, sample in params:
            dbscan_model = DBSCAN(
                eps=eps, min_samples=sample
                ).fit(self.data)
            labels = dbscan_model.labels_
            try:
                evaluate = (
                    self.evaluate(self.data, labels),
                    Counter(labels)
                    )
```

Table 7.4: Clustering parameters tested

DBSCAN		OPTICS	
ε	$MinPts$	ε	$MinPts$
0.0001	2	0.0001	2
0.001	5	0.001	5
0.01	10	0.01	10
0.1	20	0.1	20
1	40	1	40
10	100	10	50
	200	20	

```
            except ValueError:
                evaluate = Counter(labels)
            yield labels, evaluate
```

The parameters tested for each algorithm is shown in Table 7.4 and by analyzing the datasets of Appendix D.3 and Appendix D.4 we have chosen to look closer at one model for each algorithm that seems the most promising. In Table 7.5 we can see the parameters and evaluation results for these selected models.

An important thing to keep in mind when reading the evaluation scores is that they are not comparable between the algorithms. If they where there would be no doubt which model gives the best result as all the scores for DBSCAN is better than the ones for OPTICS.

As we can see from Table 7.5 the neighbor count for both models are 60, which is not the highest number of neighbors we tested for. As a higher neighbor count gives a lower recall score for the autoencoder reconstruction it makes sense that the best testing network is a compromise between the information gained by adding more neighbors to the adjacency matrix and the lowered recall value for doing so. What is a bit surprising is that the best results use a latent space of only eight dimensions for both methods. The reason for this might be the curse of dimensionality. Even though using a higher dimensionality for the latent space gives a better recall value, this might make the clustering a lot harder as the data points might be moved further apart in hyperspace. A very distinct difference between the two models is that DBSCAN finds three clusters in the dataset while OPTICS finds 10000. In Figures 7.3 and 7.4, we have colored the particles of the methane hydrate dataset using the OPTICS model and the DBSCAN model, respectively. The DBSCAN model we chose to look closer at was the only top-scoring model that found more than two clusters in the dataset. As we can see in Figure 7.4a, the model is able to separate the methane from the water particles, but that is about it. The blue cluster is just particles classified as noise and this result is not useable for finding any grain boundaries in the dataset.

Because the OPTICS model finds 10000 clusters in the dataset, the colors of the different clusters are blended together in the color spectrum. To see if there is any pattern in the data, we have tried merging the 3000 biggest clusters into one cluster of red particles and the 4000 smallest clusters into one cluster of red

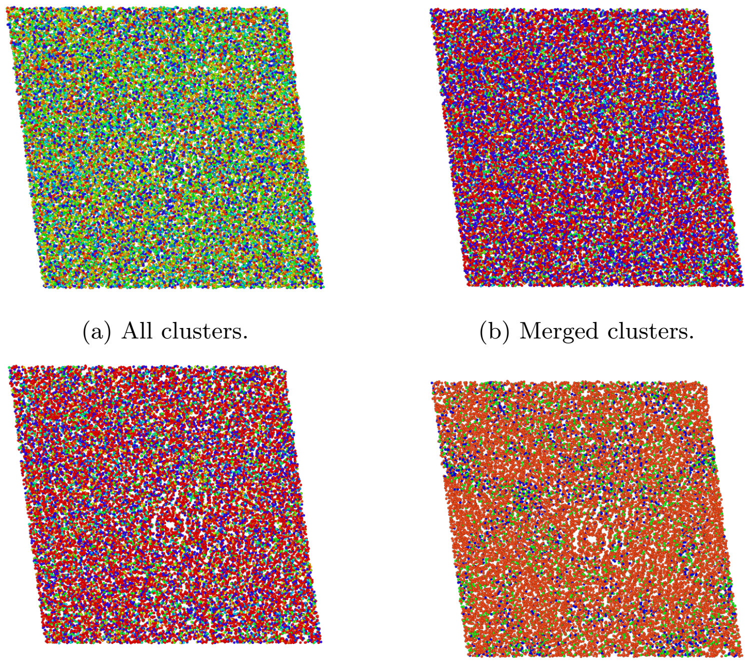
Table 7.5: Best models

OPTICS		DBSCAN	
Evaluation	Score	Evaluation	Score
Calinski	21.28	Calinski	130624.75
Silhouette	0.54	Silhouette	0.77
Davies	8.31	Davies	0.96
Parameters		Parameters	
Neighbors	60	Neighbors	60
Clusters	10050	Clusters	3
$MinPts$	2	$MinPts$	40
ε	0.0001	ε	0.1
Latent Dim	8	Latent Dim	8

Note: Clustered on latent space of autoencoder. Best Calinski-Harabasz score for DBSCAN and silhouette score for OPTICS

particles, as we can see in Figure 7.3a. This looks mostly like noise but there seems to be a bit higher density of blue and green particles around the grain boundaries when comparing with the CHILL+ algorithm in Figure 7.3d. By removing the methane particles the apparent grain boundaries become slightly more explicit, but this is too vague to make the model useable, and with 10000 clusters found in a dataset of just over 30000 particles, we can not conclude that the model has any merit. The vague appearance of the grain boundaries does, however, suggest that the algorithm might be useable for some other combination of parameters.

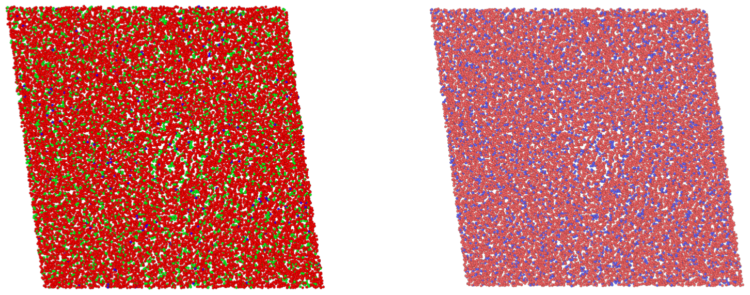
One thing to keep in mind is that the performance metrics use the clustering results itself to evaluate the model. We have no guarantee that the best results from these metrics are indeed the best models of the ones we have tested, and by using a labeled dataset for comparison, one of the other models might actually have been shown to perform better.



(a) All clusters. (b) Merged clusters.

(c) Merged clusters. Methane removed. (d) CHILL+ results.

Figure 7.3: Selected result for the OPTICS algoritm. The algorithm finds 10000 clusters in the dataset (a). To see any trends in the color spectrum we merge the 4000 least prevalent structures into a single blue cluster, and the 3000 most prevalent structures into a red cluster (b). After doing this merge we can vaguely see the grain boundaries when comparing with the CHILL+ results (d). This becomes more prominent when removing the methane atoms from the system (c).



(a) All clusters. (b) Colored by particle type.

Figure 7.4: Selected result for the DBSCAN algorithm. Figure shows the particles colored by cluster value (a) and the particles clustered by type (b). This algorithm is only able to separate the blue methane and red water particles as we can see when comparing with the system colored by particle type.

7.2.2 Non-Density-Based Clustering

The non-density-based clustering methods we tested was the agglomerative clustering algorithm and the Gaussian mixture models. We group these two algorithms together because they give similar results and we have used the same parameters for the algorithms. In the Cluster class, we have implemented these two algorithms as separate methods using Scikit-learn.

```
    def agglomerative(self, params):
        for c in params:
            agglomerative_model = AgglomerativeClustering(
                n_clusters=c
                ).fit(self.data)
            labels = agglomerative_model.labels_
            try:
                evaluate = (
                    self.evaluate(self.data, labels),
                    Counter(labels)
                    )
            except ValueError:
                evaluate = Counter(labels)
            yield labels, evaluate

    def gauss(self, params):
        for c in params:
            gauss_model = GaussianMixture(c).fit(self.data)
            labels = gauss_model.predict(self.data)
            try:
                evaluate = (
                    self.evaluate(self.data, labels),
                    Counter(labels)
                    )
            except ValueError:
                evaluate = Counter(labels)
            yield labels, evaluate
```

The only parameter we have varied for agglomerative clustering and Gaussian mixture models is the predefined number of clusters. The parameters we have tested for agglomerative clustering and GMM can be seen in Table 7.6.

By viewing the sorted results of the different evaluation metrics, which can be seen in Appendix D.1 and Appendix D.2, we have chosen one result for each of the clustering methods which found more than two clusters in the dataset. The evaluation score and parameters used for these "best" models is shown in Table 7.7 and Table 7.8 for agglomerative clustering and GMM, respectively.

Again we see that the best testing models are using a neighbor count of 60 and only eight latent dimensions, suggesting that for any meaningful clustering result, the dimensionality has to be kept to a minimum. Both algorithms use the maximum number of 10 clusters tested for on this dataset. Increasing the number of clusters might yield an even better result, but we know that this

Table 7.6: Parameters agglomerative clustering and GMM

Clustering Parameters									
Clusters	2	3	4	5	6	7	8	9	10

Table 7.7: Agglomerative

Evaluation	Score
Calinski	368796.71
Silhouette	0.49
Davies	0.55
Parameters	
Neighbors	60
Clusters	10
Latent Dim	8

Table 7.8: GMM

Evaluation	Score
Calinski	417665.15
Silhouette	0.52
Davies	0.53
Parameters	
Neighbors	60
Clusters	10
Latent Dim	8

Note: Clustered on latent space of autoencoder. Sorted by best Calinski-Harabasz score

dataset should mostly consist of one cluster of sI hydrates and only a few other clusters in the grain boundaries. Even though increasing the number of clusters might give better scores in the performance metrics, it would probably not give us any better separation of the grain boundaries.

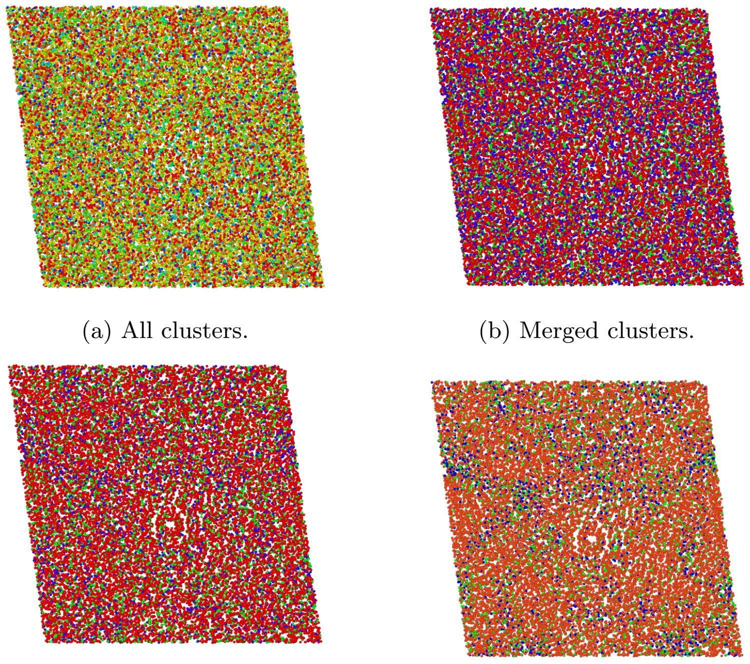
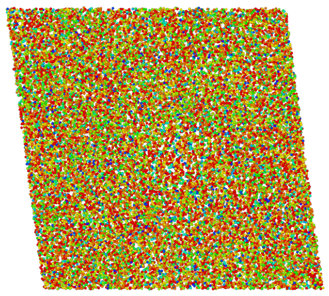
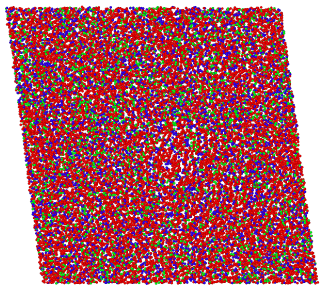

(a) All clusters. (b) Merged clusters. (c) Merged clusters. Methane removed. (d) CHILL+ results.

Figure 7.5: Selected agglomerative clustering results. The best model finds 10 clusters in the dataset (a). By merging the three most prevalent clusters and the six least prevalent clusters we can see the grain boundaries appearing (b) when comparing with the CHILL+ result (d). When removing the methane particles from the system this becomes more apparent (c).

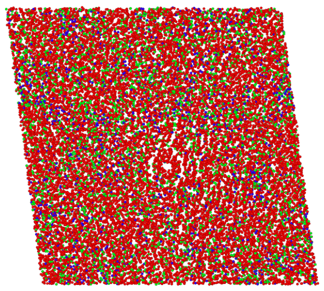
The clusters found in each of the algorithms were sorted by size of the clusters, with the most prevalent clusters using the highest index and colored by their respective cluster value. The result of this can be seen in Figures 7.5a and 7.6a, which does not seem very promising when comparing to the CHILL+ result in Figure 7.5d. We know there should probably not be as many as 10 clusters in the dataset, and because of this, we have merged the three most prevalent clusters into one cluster of red particles and the six least prevalent clusters into one cluster of blue particles. The result of this is shown in Figures 7.5b and 7.5c, and Figures 7.6b and 7.6c, which are plotted with and without the methane particles, respectively. We can see from these results that the grain boundaries found by CHILL+ are starting to appear but just like with CHILL+ we need to remove the methane for any meaningful result. This is not an inherent trait of the algorithm but simply because the methane particles are themselves a separate cluster. This is a quite small cluster, and because of this, they are merged together with the least prevalent clusters, cluttering the figure. It is not obvious which of the two algorithms produces the best results but at least with some manual merging of clusters, they are both able to find some disparity between the methane hydrates and the grain boundaries in the dataset. This suggests that with some improvements, these methods might be viable alternatives to supervised methods.



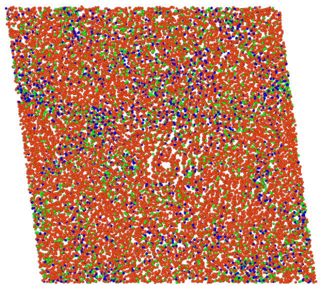
(a) All clusters.



(b) Merged clusters.



(c) Merged clusters.
Methane removed.



(d) CHILL+ result.

Figure 7.6: Selected GMM clustering results. The best model finds 10 clusters in the dataset (a). By merging the three most prevalent clusters and the six least prevalent clusters we can see the grain boundaries appearing (b) when comparing with the CHILL+ result (d). When removing the methane particles from the system this becomes more apparent (c).

Part III

Conclusions

Chapter 8

Summary and Conclusions

In this thesis, we have developed machine learning algorithms for the identification of molecular structures in simulations of crystals. Conventional structure identification algorithms are very specialized and only able to recognize a small set of predefined structures. The development of generalized identification algorithms is of particular interest for the understanding of mechanical properties in systems of mechanical failure. This is because the dissociation of a crystal will necessarily alter its intrinsic structure, and several structures might develop in the system in this process.

The development of machine learning algorithms for structure identification is to a large degree halted by the requirement of labeled datasets for the training process. We have explored two options for overcoming this problem in the present work; developing an automatic method for creating labeled datasets and using unsupervised clustering methods, which requires no such labeling.

In this chapter, we summarize and discuss the results of the previous chapters, emphasizing the methods' strengths and weaknesses.

8.1 Summary

In this thesis, we have created a method for automatically labeling training data for the development of supervised learning algorithms. We utilized this training data for the development of supervised algorithms for structure identification. In addition, we explored unsupervised clustering algorithms for the same purpose.

Conventional structure identification: We implemented the CHILL+ algorithm of Nguyen and Molinero in Python and were able to reproduce the results from the original article. With this verification, we could use CHILL+ as a benchmark for the development of new algorithms, and chose to use the newly implemented CHILL+ Ovito modification for creating these benchmarks on large systems of methane hydrates.

For testing our supervised machine learning algorithms on a dataset of multiple structures, we recreated the manually classified dataset of Engel et al., which utilized an oscillating pair potential for the self-assembly process of colloidal crystals. The dataset was largely created with the same code as used by the original authors, but due to time and computational resource constraints we used 7×10^7 timesteps as opposed to the original authors 10^8 timesteps.

For benchmarking our own algorithms we used the manual classifications by the original authors and their separation of the pair potential parameter phase space into distinct regions.

Automatic feature creation: To acquire a set of unit structures, we created a web crawler in Python which was applied to aflowlib.org, downloading all molecular unit structures and marking them by their respective Pearson symbols. We induced irregularities into the unit structures by temperating them using a harmonic potential with a break condition to prevent the irregularities from becoming too excessive. We still found the temperation to have been too aggressive for several of the unit structures, which taints the results.

The positional data of the unit- and temperated structures was then transformed into feature vectors by utilizing adjacency matrices, defined by the local neighborhood topology of the particles.

Supervised learning: We implemented two supervised machine learning frameworks; convolutional neural networks and fully-connected neural networks and trained them using the automatically labeled datasets from aflowlib.org. To make the training process faster and simpler, we only used the Pearson structures existing in the phase space defined by Engel et al. We found that using a topological neighborhood of 40 neighbors or more was sufficient for both frameworks to achieve an accuracy in the high 90s.

We applied the best performing models of both frameworks, trained on adjacency matrices of 10, 60 and 80 neighbors to the manually classified dataset of Engel et al. When plotting the most prevalent structures in the phase space we found that increasing the number of neighbors generally did increase the accuracy of the classification when comparing to Engel et al. However, the phase space became dominated by one cP8 structure. To counter this domination, we chose to plot the second most prevalent structures in the phase space which gave much better separation between the structure regions in the space, however the actual classification of the regions was ambiguous. We concluded that the convolutional neural network performed the best because it was able to separate and correctly classify the cP8 region and the hP10 region as well as the separation between the individual clathrate structures in the clathrate region.

We applied the same trained models to a simulation of methane hydrates and benchmarked against the CHILL+ results on the same dataset. After merging the least prevalent clusters in the dataset and keeping the three most prevalent as separate labeled clusters, the three models of both frameworks was able to find the grain boundaries in the dataset. The definition of these boundaries increased when using a higher number of nearest neighbors in the adjacency matrices. We concluded that again, the convolutional neural network performed the best because it found a higher number of labeled structures in the grain boundaries, while the fully-connected models mostly found the merged unlabeled structures in the boundaries.

Dimensionality reduction: Two dimensionality reduction techniques were implemented; principal component analysis, keeping 90 % of the explained variance, and an autoencoder. The autoencoder was trained on adjacency matrices of 20 to 70 neighbors, created from a methane hydrate simulation. We found the autoencoder to struggle with recreating the input as we increased the number of neighbors in the binary adjacency matrices. We found this to be caused by the increased skewness of the adjacency matrices as the neighborhood topology increased, and the best recall value was achieved using 20 nearest neighbors.

The bad recreation of the input was also aggravated when reducing the latent space size of the autoencoder.

Clustering: We applied four clustering algorithms to a methane hydrate simulation of 30000 particles, which had been transformed to adjacency matrices and reduced in dimensionality. The clustering algorithms; agglomerative clustering, Gaussian mixture models, OPTICS and DBSCAN, were evaluated using unsupervised performance metrics, all calculating the degree of cluster compactness as well as cluster separation.

We found that all the best results used adjacency matrices of 60 neighbors as well as an encoded latent space of 8 dimensions. The best performing OPTICS model produced 10000, clusters but through manual merging of the clusters, we could vaguely see the grain boundaries of the simulation when benchmarked against the CHILL+ algorithm. The best agglomerative clustering and Gaussian mixture results both used 10 clusters in the calculation. Again by manually merging the clusters, we could see the grain boundaries in the system.

8.2 Discussion

The most profound limitation of supervised machine learning algorithms is the development of labeled training data. The manual labeling of individual particles of molecular systems is not a feasible task, hence developing automatic ways of labeling data would aid in further development of machine learning applications in the field of molecular dynamics. In the present work, we have proposed a way of automating this process by utilizing various predefined unit structures as labeled data.

Realistic molecular dynamics simulations use specialized potentials for the specific material simulated in the system. During the creation or temperation of such materials, the intrinsic structure of the crystals will be altered slightly compared with the unit structures, due to the fluctuations of atoms caused by differences in pressure and temperature. The biggest difficulty regarding the automatic approach we have proposed is to imitate these slightly altered structures found in realistic molecular dynamics simulations.

We found that our approach of temperating the crystals using a harmonic potential was too aggressive, which altered the unit structures excessively. The supervised machine learning algorithms were still able to separate several of the structures in the dataset of Engel et al. which suggests that even though our results were not conclusive, there should still be merit in pursuing this approach further.

On the dataset of Engel et al. we found that both the fully-connected network and the convolutional network was to a large extent able to reproduce the boundaries of the manually classified regions. However, they both performed poorly in the actual classification of these regions. There might be two reasons for explaining this behavior.

Firstly, due to time and resource constraints our recreation of the dataset used fewer timesteps in the cooling of the individual crystals. This might have altered the crystals and in which case, we would not be able to reproduce the manual results at all. The fact that the networks were largely able to separate the different regions but not classify them correctly can be argued to support this hypothesis because the networks understand that there are in fact different structures in these regions, they just do not agree with the manual classification on which structures the regions contain.

Secondly, we found in the results that the temperation we used in the creation of the labeled dataset was too excessive and many of the unit structures were altered to a large degree. This alteration had the effect of adding too many bonds between the individual atoms, becoming progressively worse throughout the temperation. This might have made the features of the altered crystal similar to several other crystals in the dataset, hence making them difficult to classify correctly.

On the methane hydrate dataset, we found that both frameworks were able to identify the grain boundaries of the crystal, recreating a similar result as the CHILL+ algorithm. CHILL+ is specialized for finding ice and clathrate structures in simulations and the fact that we can compete with this algorithm using machine learning is very promising. As opposed to CHILL+, the machine learning algorithms have not been explicitly programmed to recognize clathrate structures and because of this we can simply expand the networks ability to recognize structures by adding other structures to the training set.

The most significant difference between the two supervised frameworks tested was that the fully-connected network classified most of the grain boundaries as an unlabeled cluster, while the convolutional neural network to a larger degree found separate labeled structures in the boundaries. However, because of the temperation problem we can not be completely confident in the classification of these separate structures. Ideally, we would like to classify precisely which structures exist in the grain boundaries to aid the understanding of their mechanical properties and because the convolutional neural network seems more promising for this purpose we will conclude that there seems to be most merit in pursuing further implementations using the convolutional neural network.

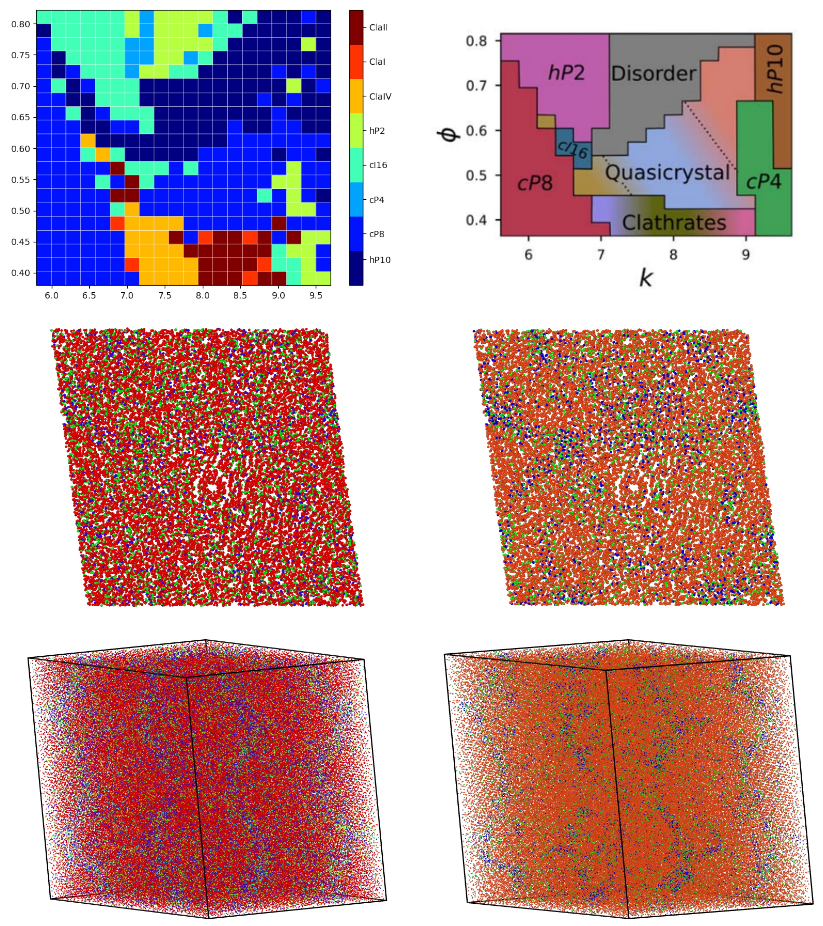
We also tested a convolutional network trained only on the three clathrate structures, which produced less prominent grain boundaries on the test set. When comparing with the previous result, this is not that surprising because the bulk of the grain boundaries were not classified as sII or H clathrate structures but rather as some different cage-like structures.

When applying unsupervised learning to a methane hydrate dataset, we first reduced the dimensionalities of the features. We found that we produced the best clustering results when using an autoencoder for the dimensionality reduction. The reason the autoencoder did so much better might be due to the curse of dimensionality. For principal component analysis, we set a condition of keeping at least 90 % of the explained variance from the original dataset. This is likely to keep the dimensionality high and thus making the clustering task very difficult.

The autoencoder was shown to best recreate the input if we kept the number of neighbors in the adjacency matrices low and the latent dimensionality of the autoencoder high. The autoencoder was very sensitive to the number of nearest neighbors used in the calculation of adjacency matrices. This is because these matrices are binary, and as the number of neighbors increases, the matrices become heavily skewed towards the zero class, making it easier for the network to achieve a high accuracy by simply setting every value in the matrices to zero. When using a higher number of nearest neighbors in the adjacency matrices we are collecting more information about the neighborhood of a particle, but because an increase in neighbors also produces a worse recreation for the autoencoder we saw that the best clustering algorithms all used a compromise between the two by using 60 neighbors, not the maximum neighbor count of 70. We also found that even though the best recreation result of the autoencoder was produced with a high number of dimensions in the latent space, the best clustering results used a latent space of only eight dimensions. This was the lowest number of dimensions we tested and this further confirmes the assumption that we are reliant on keeping the number of dimensions very low for the clustering algorithms to be of any use.

We were able to reproduce the grain boundaries found by CHILL+ when using agglomerative clustering, Gaussian mixture models and to some degree when using OPTICS. However, they all had to be manually evaluated by merging some of the clusters. The fact that we found any grain boundaries was because we already knew what to look for, and without having the result of CHILL+ to compare to, all the results would probably be discarded as noise. Nevertheless, the results do suggest that it is possible to use clustering techniques for identifying the grain boundaries if we find the ideal clustering algorithm and hyperparameters. Unsupervised learning would remove the need for creating labeled datasets altogether and the fact that we have shown that it is possible to use clustering techniques for structure identification is a very promising prospect for future implementations.

Some selected results are shown in Figure 8.1.



Selected Results **Benchmarks**

Figure 8.1: Top: Plot of the second most prevalent structures found by a convolutional neural network in the phase space produced by the oscillating pair potential (left). The manually classified regions of Engel et al. is laregly reproduced by the network, but the classification within the regions differ somewhat from the benchmark(right).
Middle: The result of agglomerative clustering after merging the three most prevalent clusters and the four least prevalent clusters (left), and the result of CHILL+ on the same dataset (right). Both plots are three-dimensional datasets viewed from the side.
Bottom: Result of convolutional neural network when using adjacency matrices of 80 neighbors on a dataset of methane hydrates (left), and the results of CHILL+ (right).

8.3 Outlook

In this section, we will mention some immediate improvements to our algorithms based on what we discovered in the results, as well as some new ideas for methods to test in the future.

Temperation:
We found in the result section that the temperation process tended to alter the molecular structures excessively. The first step to fixing this problem would be to set a more rigorous break condition for the temperation. We tried setting a condition which stops the temperation if one of the particles had moved more than a certain distance from its initial position and unfortunately we set this distance too high, allowing the particles to move too freely. With this approach, we used the same harmonic potential for all structures, and a better solution might be to alter the harmonic potential depending on how thightly packed the structure in question is instead of stopping the temperation based on the particle movements.

Setting a high constant of proportionality would make the potential well deeper, forcing the particles to move less about their initial positions. This would be a problem in sparse structures as there might not be enough forces acting on a particle to actually temperate the crystal at all, and for a thightly packed structure the temperation might still become too excessive. We could solve this by finding some relationship between the first minimum of the radial basis function and the constant of proportionality; if the first minimum is situated at a small radial distance, the structure is tightly packed and we should increase the constant of proportionality before temperation, if the radial distance is high, the structure is sparse and we should have a lower constant of proportionality. The most difficult part of this approach would be to find a suitable relationship between the radial basis function and the constant of proportionality, however, if a working relationship is established this is likely to be a far more rigorous approach than what was done in this thesis.

Recreate the dataset of Engel et al.:
As we suggested in the discussion, the bad classification on the dataset of Engel et al. is likely to be because of our choice of cooling the crystals quicker than what was done in the original simulations. Creating this dataset took several weeks on a high-end GPU but to make a more accurate comparison we would need to recreate the dataset with the same conditions as the original.

Dimensionality reduction:
The dimensionality reduction techniques used in this thesis was only applied to the unsupervised learning algorithms. The curse of dimensionality is more prominent in the unsupervised case, but it is still a problem in supervised learning. Applying these techniques to the supervised algorithms developed in this thesis would be a natural next step for improving the accuracy of the models.

For the unsupervised models, we found that clustering on reduced features using PCA gave unsatisfactory results. We argued that this might be because we used the explained variance as a criterion for how many dimensions should be kept. Because the best result of the autoencoder only used eight dimensions in the reduced space, we could retry PCA with a fixed, low number of retained dimensions to see if this would yield any improvements in the clustering result.

Another technique we could try is to use a convolutional autoencoder instead of the fully-connected autoencoder we implemented in this thesis. We established that convolutional neural networks generally seem to have a slightly improved performance on the adjacency matrices compared to the fully-connected networks. The fact that a convolutional autoencoder would use filters to evaluate small regions in the adjacency matrices might make it less susceptible to the skewness of the adjacency matrices, as the filters finds regions of interest instead of considering the entire matrix at once.

Labeled training set for unsupervised learning:
To evaluate our clustering methods we used unsupervised performance metrics, which is not guaranteed to find the best models in our dataset. To test the clustering algorithms more rigorously we could train them on the labeled dataset we developed in this thesis. This way we could test quantitatively which algorithm works best on a training set, before applying the best ones to a test set.

Adding ice to the training set:
When evaluating the methane hydrate dataset, we used networks trained with some selected Pearson structures. A polycrystal of methane hydrate is very similar to ice, and regular ice is likely to be created in the grain boundaries during the deformation process. Because of this it would be interesting to train a network for recognizing the solid phases of water in addition to the different clathrate structures.

Utilizing other feature vectors:
We chose to use local neighborhood topology as our feature vectors to the neural networks. This is, however, not the only option of features to use. Spellings and Glotzer [69] used the neighborhood average of spherical harmonics over the N nearest neighbors of a particle i

$$\bar{Y}_l^m(i, N) = \frac{1}{N} \left| \sum_{j=1}^{N} Y_l^m(\theta_{ij}, \phi_{ij}) \right| . \tag{8.1}$$

Spellings and Glotzer found these features to work well for supervised learning but due to the combinatorial degeneracy of placing particles inside neighbor shells, they did not work as well for unsupervised learning. We did preliminary tests of using spherical harmonics for unsupervised learning but found the results to be unsatisfying, we did not test this with supervised learning, however.

Another option is to use so-called Smooth Overlap of Atomic Position (SOAP)-vectors [5], which can be calculated using the DScribe Python package [30]. The description of the SOAP vectors as given in the DScribe documentation states

> SOAP vectors encode regions of atomic geometries by using a local expansion of a Gaussian smeared atomic density with orthonormal functions based on spherical harmonics and radial basis functions.

We did preliminary testing with these descriptors in supervised learning but the results could not compete with using adjacency matrices. We did not test this thoroughly, however, and because the vectors are based on comparing the similarity between any two neighborhood environments, they might show promise if investigated properly.

Transfer Learning:
Transfer learning refers to a technique in machine learning of storing knowledge gained from solving one problem an applying it to another. The general benchmark for convolutional neural networks (CNN) is how it performs on the ImageNet Large Scale Visual Recognition Challenge [66] (ILSVRC). ILSVRC uses about 1.4 million images divided into 1000 classes and in 2015 Microsoft developed ResNet [27], the first CNN which was able to outperform human image classification. We could utilize ResNet in our training on adjacency matrices by using transfer learning. We assume that this deep, powerful network is finding very general features in images in the bulk of the network which are applicable to our own classification problem. To adapt the network to our own dataset, we would keep the weights fixed for all layers, except the last few, and train the network on our own dataset. The fixed weights would hopefully be able to find information in the adjacency matrices our own network did not, and by simply training the last few layers on our own dataset we could improve on our results.

3D convolutional network:
In this thesis, we first transformed the positional data of atoms into adjacency matrices and used these as feature vectors. An interesting thing to test would be to skip this step entirely. We could consider the molecular systems as three-dimensional images, and apply 3D convolution to the systems. In this approach, the network would have to infer the topology itself, and we have no guarantee that it would work. The most computationally time consuming and memory intensive part of the techniques we developed in this thesis was the transformation of positional data into adjacency matrices. With the approach of 3D convolution, we would drastically reduce the time complexity when the trained algorithm is applied to testing sets, which would be a vast improvement over our own algorithm.

Bibliography

[1] Joshua A. Anderson, Chris D. Lorenz and A. Travesset. 'General purpose molecular dynamics simulations fully implemented on graphics processing units'. In: *Journal of Computational Physics* (2008). ISSN: 10902716. DOI: 10.1016/j.jcp.2008.01.047.

[2] K. Andreassen et al. 'Massive blow-out craters formed by hydrate-controlled methane expulsion from the Arctic seafloor'. In: *Science* (2017). ISSN: 10959203. DOI: 10.1126/science.aal4500.

[3] Mihael Ankerst et al. 'OPTICS: Ordering Points to Identify the Clustering Structure'. In: *SIGMOD Record (ACM Special Interest Group on Management of Data)* (1999). ISSN: 01635808. DOI: 10.1145/304181.304187.

[4] David Archer, Bruce Buffett and Victor Brovkin. 'Ocean methane hydrates as a slow tipping point in the global carbon cycle'. In: *Proceedings of the National Academy of Sciences of the United States of America* (2009). ISSN: 00278424. DOI: 10.1073/pnas.0800885105.

[5] Albert P. Bartók, Risi Kondor and Gábor Csányi. 'On representing chemical environments'. In: *Physical Review B - Condensed Matter and Materials Physics* (2013). ISSN: 10980121. DOI: 10.1103/PhysRevB.87.184115. arXiv: 1209.3140.

[6] Etienne P. Bernard and Werner Krauth. 'Two-step melting in two dimensions: First-order liquid-hexatic transition'. In: *Physical Review Letters* (2011). ISSN: 00319007. DOI: 10.1103/PhysRevLett.107.155704. arXiv: 1102.4094.

[7] Gerhard Bohrmann and Marta E. Torres. 'Gas hydrates in marine sediments'. In: *Marine Geochemistry.* 2006. ISBN: 3540321438. DOI: 10.1007/3-540-32144-6_14.

[8] Tom Bugge et al. 'A giant three-stage submarine slide off Norway'. In: *Geo-Marine Letters* (1987). ISSN: 02760460. DOI: 10.1007/BF02242771.

[9] T. Calinski and J. Harabasz. 'A Dendrite Method for Cluster Analysis'. In: *Communications in Statistics - Simulation and Computation* (1974). ISSN: 0361-0918. DOI: 10.1080/03610917408548446.

[10] Murray Campbell, A. Joseph Hoane and Feng Hsiung Hsu. 'Deep Blue'. In: *Artificial Intelligence* (2002). ISSN: 00043702. DOI: 10.1016/S0004-3702(01)00129-1.

[11] François Chollet. 'Keras (2015)'. In: *URL http://keras. io* (2017). ISSN: 1550-235X.

[12] David L. Davies and Donald W. Bouldin. 'A Cluster Separation Measure'. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (1979). ISSN: 01628828. DOI: 10.1109/TPAMI.1979.4766909.

[13] Bradley Dice et al. 'Analyzing Particle Systems for Machine Learning and Data Visualization with freud'. In: *Proceedings of the 18th Python in Science Conference.* 2019. DOI: 10.25080/majora-7ddc1dd1-004.

[14] Michael Engel et al. 'Computational self-assembly of a one-component icosahedral quasicrystal'. In: *Nature Materials* (2015). ISSN: 14764660. DOI: 10.1038/nmat4152.

[15] Michael Engel et al. 'Hard-disk equation of state: First-order liquid-hexatic transition in two dimensions with three simulation methods'. In: *Physical Review E - Statistical, Nonlinear, and Soft Matter Physics* (2013). ISSN: 15393755. DOI: 10.1103/PhysRevE.87.042134. arXiv: 1211.1645.

[16] Martin Ester et al. 'A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise'. In: *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining.* 1996.

[17] Andre Esteva et al. 'Dermatologist-level classification of skin cancer with deep neural networks'. In: *Nature* (2017). ISSN: 14764687. DOI: 10.1038/nature21056.

[18] Daan Frenkel and Berend Smit. *Understanding molecular simulation: From algorithms to applications.* 1996. ISBN: 9780122673702. DOI: 10.1063/1.881812.

[19] Jens Glaser et al. 'Strong scaling of general-purpose molecular dynamics simulations on GPUs'. In: *Computer Physics Communications* (2015). ISSN: 00104655. DOI: 10.1016/j.cpc.2015.02.028. arXiv: 1412.3387.

[20] Xavier Glorot and Yoshua Bengio. 'Understanding the difficulty of training deep feedforward neural networks'. In: *Journal of Machine Learning Research.* 2010.

[21] GoogleResearch. 'TensorFlow: Large-scale machine learning on heterogeneous systems'. In: *Google Research* (2015). arXiv: arXiv:1603.04467v2.

[22] Xifeng Guo et al. 'Deep Clustering with Convolutional Autoencoders'. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics).* 2017. ISBN: 9783319700953. DOI: 10.1007/978-3-319-70096-0_39.

[23] B. I. Halperin and David R. Nelson. 'Theory of Two-Dimensional melting'. In: *Physical Review Letters* (1978). ISSN: 00319007. DOI: 10.1103/PhysRevLett.41.121.

[24] E. G. Hammerschmidt. 'Formation of Gas Hydrates in Natural Gas Transmission Lines'. In: *Industrial and Engineering Chemistry* (1934). ISSN: 00197866. DOI: 10.1021/ie50296a010.

[25] T; Tibshirani Hastie. 'The Elements of Statistical Learning Second Edition'. In: *Math. Intell.* (2017). ISSN: 03436993. DOI: 111. arXiv: arXiv:1011.1669v3.

[26] Kaiming He et al. 'Delving deep into rectifiers: Surpassing human-level performance on imagenet classification'. In: *Proceedings of the IEEE International Conference on Computer Vision.* 2015. ISBN: 9781467383912. DOI: `10.1109/ICCV.2015.123`. arXiv: `1502.01852`.

[27] Kaiming He et al. 'ResNet'. In: *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (2016). ISSN: 10636919. DOI: `10.1109/CVPR.2016.90`. arXiv: `1512.03385`.

[28] J. P. Henriet and J. Mienert. 'Gas hydrates: relevance to world margin stability and climatic change'. In: *Geological Society Special Publication* (1998). ISSN: 03058719. DOI: `10.5860/choice.36-5103`.

[29] David Hicks et al. 'The AFLOW Library of Crystallographic Prototypes: Part 2'. In: *Computational Materials Science* (2019). ISSN: 09270256. DOI: `10.1016/j.commatsci.2018.10.043`. arXiv: `1806.07864`.

[30] Lauri Himanen et al. 'DScribe: Library of descriptors for machine learning in materials science'. In: *Computer Physics Communications* (2020). ISSN: 00104655. DOI: `10.1016/j.cpc.2019.106949`. arXiv: `1904.08875`.

[31] Ask Hjorth Larsen et al. *The atomic simulation environment - A Python library for working with atoms.* 2017. DOI: `10.1088/1361-648X/aa680e`.

[32] William G. Hoover. 'Canonical dynamics: Equilibrium phase-space distributions'. In: *Physical Review A* (1985). ISSN: 10502947. DOI: `10.1103/PhysRevA.31.1695`.

[33] Kurt Hornik. 'Approximation capabilities of multilayer feedforward networks'. In: *Neural Networks* (1991). ISSN: 08936080. DOI: `10.1016/0893-6080(91)90009-T`.

[34] Camden R. Hubbard and Lauriston D. Calvert. 'The pearson symbol'. In: *Bulletin of Alloy Phase Diagrams* (1981). ISSN: 01970216. DOI: `10.1007/BF02881453`.

[35] Philippe H. Hünenberger. *Thermostat algorithms for molecular dynamics simulations.* 2005. DOI: `10.1007/b99427`.

[36] Aaron Courville Ian Goodfellow, Yoshua Bengio. 'Deep Learning Book'. In: *Deep Learning* (2015). ISSN: 1437-7780. DOI: `10.1016/B978-0-12-391420-0.09987-X`. arXiv: `arXiv:1011.1669v3`.

[37] Diederik P. Kingma and Jimmy Lei Ba. 'Adam: A method for stochastic optimization'. In: *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings.* 2015. arXiv: `1412.6980`.

[38] Konstantina Kourou et al. *Machine learning applications in cancer prognosis and prediction.* 2015. DOI: `10.1016/j.csbj.2014.11.005`.

[39] Alex Krizhevsky, Ilya Sutskever and Geoffrey E. Hinton. 'ImageNet classification with deep convolutional neural networks'. In: *Communications of the ACM* (2017). ISSN: 15577317. DOI: `10.1145/3065386`.

[40] Parsons L., Haque E. and Liu H. 'Subspace Clustering for High Dimensional Data: A Review'. In: *SIGKDD Explorations, Newsletter of the ACM Special Interest Group on Knowledge Discovery and Data Mining* (2004).

[41] Yann LeCun et al. 'Gradient-based learning applied to document recognition'. In: *Proceedings of the IEEE* (1998). ISSN: 00189219. DOI: 10.1109/5.726791.

[42] James Leibold. 'Surveillance in China's Xinjiang Region: Ethnic Sorting, Coercion, and Inducement'. In: *Journal of Contemporary China* (2020). ISSN: 14699400. DOI: 10.1080/10670564.2019.1621529.

[43] Jun Ma et al. 'Analyzing the Leading Causes of Traffic Fatalities Using XGBoost and Grid-Based Analysis: A City Management Perspective'. In: *IEEE Access* (2019). ISSN: 21693536. DOI: 10.1109/ACCESS.2019.2946401.

[44] Y. F. Makogon, S. A. Holditch and T. Y. Makogon. 'Natural gas-hydrates - A potential energy source for the 21st Century'. In: *Journal of Petroleum Science and Engineering* (2007). ISSN: 09204105. DOI: 10.1016/j.petrol.2005.10.009.

[45] John McCarthy et al. 'A proposal for the Dartmouth summer research project on artificial intelligence'. In: *AI Magazine* (2006). ISSN: 07384602.

[46] Michael J. Mehl et al. 'The AFLOW Library of Crystallographic Prototypes: Part 1'. In: *Computational Materials Science* (2017). ISSN: 09270256. DOI: 10.1016/j.commatsci.2017.01.017. arXiv: 1607.02532.

[47] Pankaj Mehta et al. *A high-bias, low-variance introduction to Machine Learning for physicists*. 2019. DOI: 10.1016/j.physrep.2019.03.001. arXiv: 1803.08823.

[48] Bjoern H. Menze and Jason A. Ur. 'Mapping patterns of long-term settlement in Northern Mesopotamia at a large scale'. In: *Proceedings of the National Academy of Sciences of the United States of America* (2012). ISSN: 00278424. DOI: 10.1073/pnas.1115472109.

[49] Marek Mihalkovič and C. L. Henley. 'Empirical oscillating potentials for alloys from ab initio fits and the prediction of quasicrystal-related structures in the Al-Cu-Sc system'. In: *Physical Review B - Condensed Matter and Materials Physics* (2012). ISSN: 10980121. DOI: 10.1103/PhysRevB.85.092102.

[50] James Moor. 'The Dartmouth College Artificial Intelligence Conference: The next fifty years'. In: *AI Magazine*. 2006.

[51] Emily B. Moore et al. 'Freezing, melting and structure of ice in a hydrophilic nanopore'. In: *Physical Chemistry Chemical Physics* (2010). ISSN: 14639076. DOI: 10.1039/b919724a.

[52] Andrew H. Nguyen and Valeria Molinero. 'Identification of Clathrate Hydrates, Hexagonal Ice, Cubic Ice, and Liquid Water in Simulations: The CHILL+ Algorithm'. In: *Journal of Physical Chemistry B* (2015). ISSN: 15205207. DOI: 10.1021/jp510289t.

[53] Fulong Ning et al. *Mechanical properties of clathrate hydrates: Status and perspectives*. 2012. DOI: 10.1039/c2ee03435b.

[54] Shuichi Nosé. 'A unified formulation of the constant temperature molecular dynamics methods'. In: *The Journal of Chemical Physics* (1984). ISSN: 00219606. DOI: 10.1063/1.447334.

[55] Shüichi Nosé. 'A molecular dynamics method for simulations in the canonical ensemble'. In: *Molecular Physics* (1984). ISSN: 13623028. DOI: `10.1080/00268978400101201`.

[56] Karl Pearson. ' LIII. On lines and planes of closest fit to systems of points in space '. In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* (1901). ISSN: 1941-5982. DOI: `10.1080/14786440109462720`.

[57] W. B. Pearson and George H. Vineyard. 'A Handbook of Lattice Spacings and Structures of Metals and Alloys'. In: *Physics Today* 11.9 (Sept. 1958), pp. 36–36. ISSN: 0031-9228. DOI: `10.1063/1.3062734`. URL: `http://physicstoday.scitation.org/doi/10.1063/1.3062734`.

[58] Fabian Pedregosa et al. 'Scikit-learn: Machine Learning in Python Pedregosa, Varoquaux, Gramfort et al'. In: *Journal of Machine Learning Research* (2011). arXiv: `arXiv:1201.0490v4`.

[59] Joseph Priestley. *Experiments and Observations on Different Kinds of Air.* 2013. DOI: `10.1017/cbo9781139644419`.

[60] Rachel Metz. *The Daily: Why Microsoft Accidentally Unleashed a Neo-Nazi Sexbot.* 2016.

[61] Wesley F. Reinhart and Athanassios Z. Panagiotopoulos. 'Automated crystal characterization with a fast neighborhood graph analysis method'. In: *Soft Matter* (2018). ISSN: 17446848. DOI: `10.1039/c8sm00960k`.

[62] Wesley F. Reinhart et al. 'Machine learning for autonomous crystal structure identification'. In: *Soft Matter* (2017). ISSN: 17446848. DOI: `10.1039/c7sm00957g`.

[63] F. Rosenblatt. 'The perceptron: A probabilistic model for information storage and organization in the brain'. In: *Psychological Review* (1958). ISSN: 0033295X. DOI: `10.1037/h0042519`.

[64] Peter J. Rousseeuw. 'Silhouettes: A graphical aid to the interpretation and validation of cluster analysis'. In: *Journal of Computational and Applied Mathematics* (1987). ISSN: 03770427. DOI: `10.1016/0377-0427(87)90125-7`.

[65] Carolyn D. Ruppel and John D. Kessler. *The interaction of climate change and methane hydrates.* 2017. DOI: `10.1002/2016RG000534`.

[66] Olga Russakovsky et al. 'ImageNet Large Scale Visual Recognition Challenge'. In: *International Journal of Computer Vision* (2015). ISSN: 15731405. DOI: `10.1007/s11263-015-0816-y`. arXiv: `1409.0575`.

[67] SciPy Developers. *Scientific Computing Tools for Python — SciPy.org.* 2019.

[68] E. Dendy Sloan and Carolyn Ann Koh. *Clathrate hydrates of natural gases, thrid edition.* 2007. ISBN: 9781420008494.

[69] Matthew Spellings and Sharon C. Glotzer. 'Machine learning for crystal identification and discovery'. In: *AIChE Journal* (2018). ISSN: 15475905. DOI: `10.1002/aic.16157`. arXiv: `1710.09861`.

[70] Nitish Srivastava et al. 'Dropout: A simple way to prevent neural networks from overfitting'. In: *Journal of Machine Learning Research* (2014). ISSN: 15337928.

[71] Paul Steinhardt, David Nelson and Marco Ronchetti. 'Phys. Rev. B 28, 784 (1983): Bond-orientational order in liquids and glasses'. In: *Physical Review B* (1983). ISSN: 0163-1829. DOI: 10.1103/PhysRevB.28.784.

[72] G. W. Stewart. 'On the early history of the singular value decomposition'. In: *SIAM Review* (1993). ISSN: 00361445. DOI: 10.1137/1035134.

[73] Stephen M. Stigler. 'Gauss and the Invention of Least Squares'. In: *The Annals of Statistics* (1981). ISSN: 0090-5364. DOI: 10.1214/aos/1176345451.

[74] Frank H. Stillinger and Thomas A. Weber. 'Computer simulation of local order in condensed phases of silicon'. In: *Physical Review B* (1985). ISSN: 01631829. DOI: 10.1103/PhysRevB.31.5262.

[75] Alexander Stukowski. 'Visualization and analysis of atomistic simulation data with OVITO-the Open Visualization Tool'. In: *Modelling and Simulation in Materials Science and Engineering* (2010). ISSN: 09650393. DOI: 10.1088/0965-0393/18/1/015012.

[76] Henrik Andersen Sveinsson. 'Molecular dynamics modeling of mechanical failure processes in methane hydrates.' PhD thesis. University of Oslo, 2019. URL: http://urn.nb.no/URN:NBN:no-73734.

[77] Henrik Andersen Sveinsson. *Molecular dynamics simulations of polycrystalline methane hydrates under shear loading: Positions at maximum stress.* 2019. DOI: 10.5281/zenodo.3228754. URL: https://doi.org/10.5281/zenodo.3228754.

[78] Pieter Rein Ten Wolde, Maria J. Ruiz-Montero and Daan Frenkel. 'Numerical calculation of the rate of homogeneous gas-liquid nucleation in a Lennard-Jones system'. In: *Journal of Chemical Physics* (1999). ISSN: 00219606. DOI: 10.1063/1.477799.

[79] Yongjun Tian et al. 'Ultrahard nanotwinned cubic boron nitride'. In: *Nature* (2013). ISSN: 00280836. DOI: 10.1038/nature11728.

[80] Loup Verlet. 'Computer "experiments" on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules'. In: *Physical Review* (1967). ISSN: 0031899X. DOI: 10.1103/PhysRev.159.98.

[81] Sandra Vieira, Walter H.L. Pinaya and Andrea Mechelli. *Using deep learning to investigate the neuroimaging correlates of psychiatric and neurological disorders: Methods and applications.* 2017. DOI: 10.1016/j.neubiorev.2017.01.002.

[82] E. M. Wright and Richard Bellman. 'Adaptive Control Processes: A Guided Tour'. In: *The Mathematical Gazette* (1962). ISSN: 00255572. DOI: 10.2307/3611672.

Appendix A

Network Architectures

Table A.1: Convolutional architectures

	Input Dim $< 30 \times 30$	Input Dim $\geq 30 \times 30$
Architecture	Conv[16] Pool[2] Conv[32] Conv[64] Conv[128]	Conv[16] Pool[2] Conv[32] Conv[64] Pool[2] Conv[128]
Dropout Rate	None 0.3 0.5	
Activations	[Relu, Relu, Relu, Relu] [Relu, Relu, Relu, Tanh] [Relu, Relu, Tanh, Relu] ⋮ [Tanh, Tanh, Tanh, Tanh]	
Kernel Sizes	3×3 5×5	
Padding	Keep input dimensions	
Batch Sizes	512	
Max Epochs	150	
Optimizer	Adam	

Note: Table over convolutional neural networks tested. Separate architectures are implemented depending on the size of the input. The larger inputs utilize an extra pooling layer to reduce the feature map size. Numbers in brackets show the number of filters used per layer. Each network architecture was run for all combinations of parameters below. The activations functions tested was all combinations of length 4 of relu and the hyperbolic tangent function.

Table A.2: Fully-connected architectures

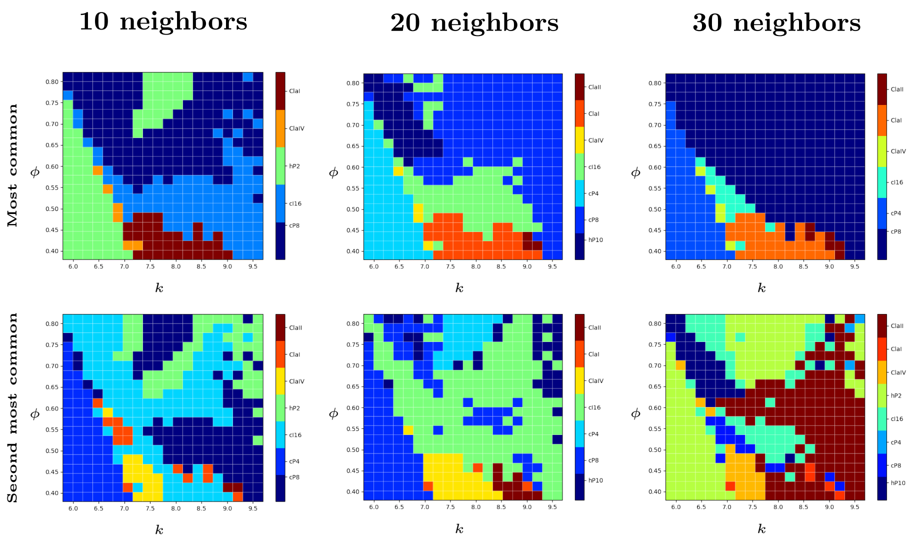
	Neighbors	Layer Sizes
Architecture	10	[66, 40, 28, 22]
	20	[266, 160, 114, 88]
	30	[600, 360, 257, 200]
	40	[1066, 640, 457, 355]
	50	[1666, 1000, 714, 555]
	60	[2400, 1440, 1028, 800]
	70	[3266, 1960, 1400, 1088]
	80	[4266, 2560, 1828, 1422]
Dropout Rate	None	
	0.3	
	0.5	
Activations	[Relu, Relu, Relu, Relu]	
	[Relu, Relu, Relu, Tanh]	
	[Relu, Relu, Relu, Sigmoid]	
	[Relu, Relu, Tanh, Sigmoid]	
	⋮	
	[Tanh, Tanh, Tanh, Tanh]	
	[Sigmoid, Sigmoid, Sigmoid, Sigmoid]	
Batch Sizes	512	
Max Epochs	150	
Optimizer	Adam	

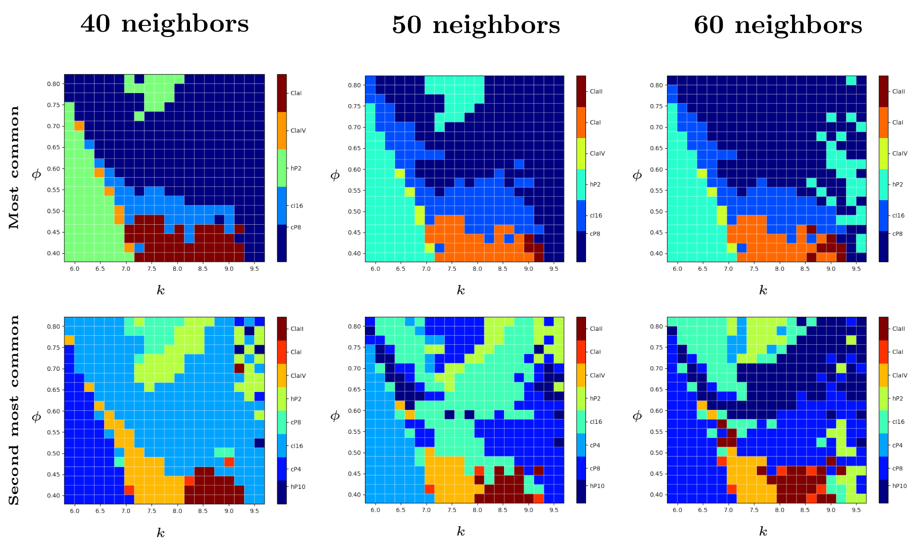
Note: Table over fully-connected neural networks tested. We used four hidden layers in all architectures but varied the layer sized depending on the input size. Each network architecture was run for all combinations of hyperparameters. The activations functions tested was all combinations of length 4 of relu, sigmoid and the hyperbolic tangent function.

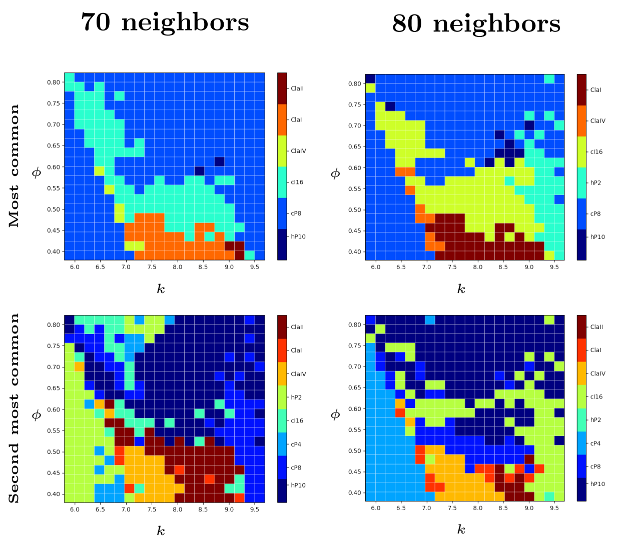
Appendix B

Oscillating Pair Potential Results

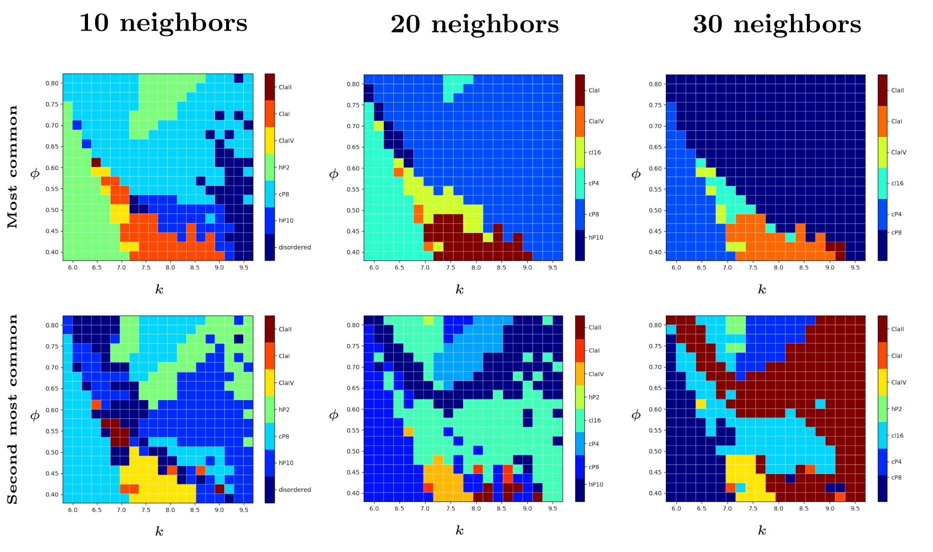
B.1 Convolutional Network

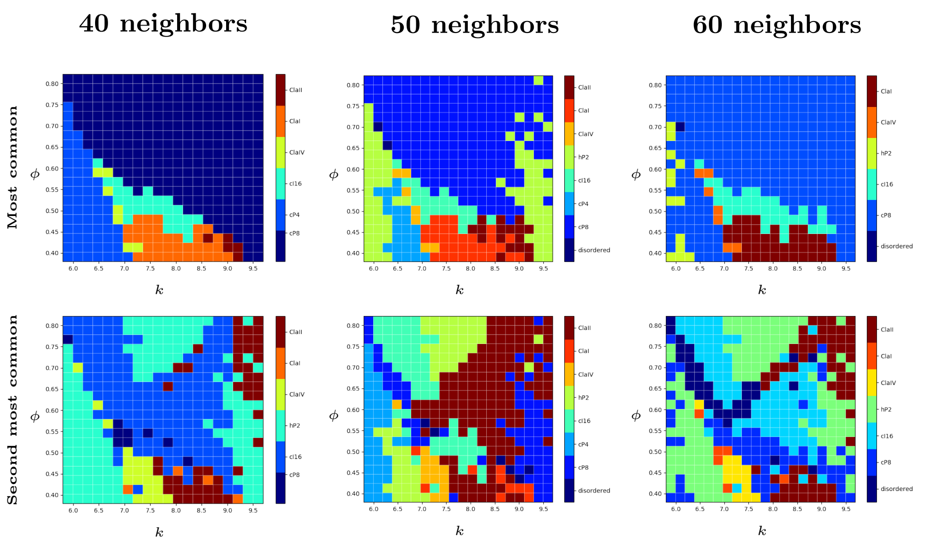


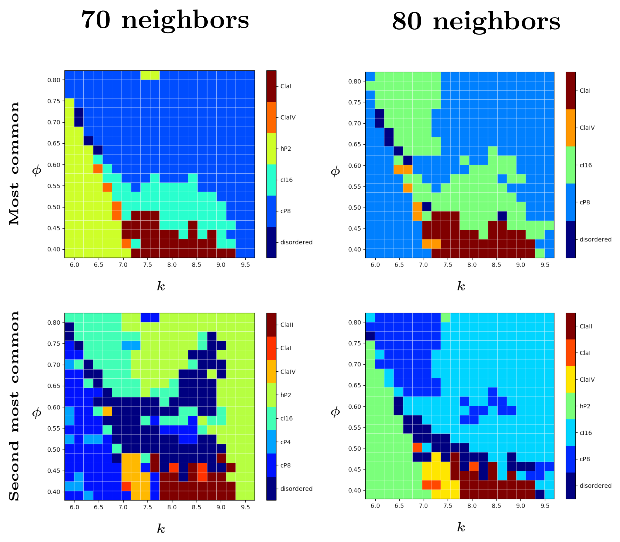

40 neighbors
50 neighbors
60 neighbors
Most common
Second most common
ϕ
k
ClaII
ClaI
ClaIV
hP2
cI16
cP8
cP4
hP10


70 neighbors
80 neighbors
Most common
Second most common
ϕ
k
ClaII
ClaI
ClaIV
hP2
cI16
cP4
cP8
hP10

B.2 Dense Network



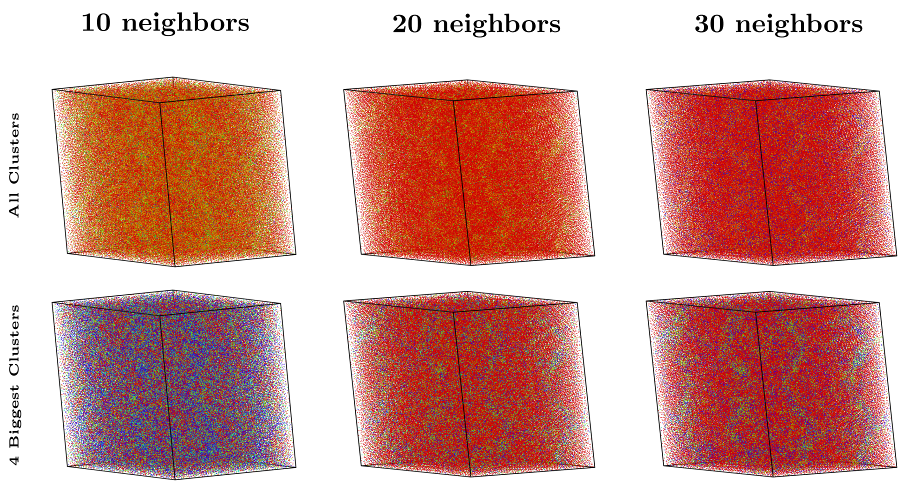


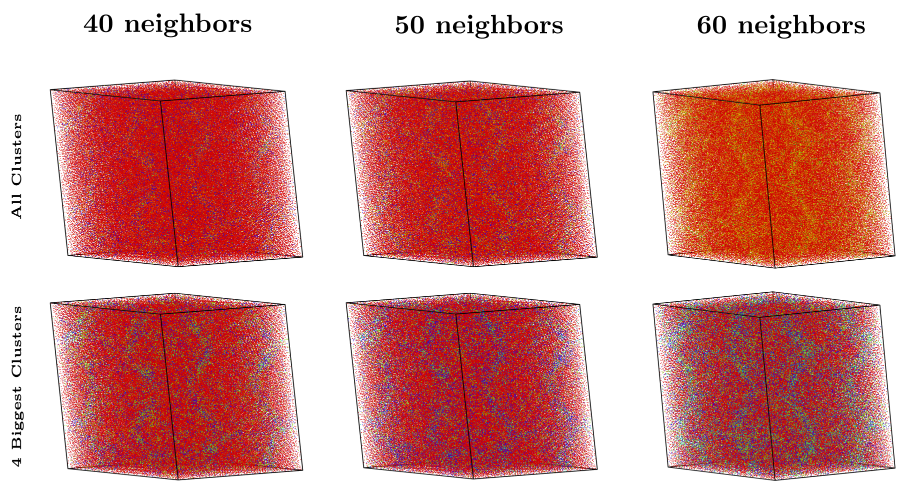

70 neighbors
80 neighbors
Most common
Second most common
ϕ
k
0.80
0.75
0.70
0.65
0.60
0.55
0.50
0.45
0.40
6.0
6.5
7.0
7.5
8.0
8.5
9.0
9.5
ClaII
ClaI
ClaIV
hP2
cI16
cP4
cP8
disordered

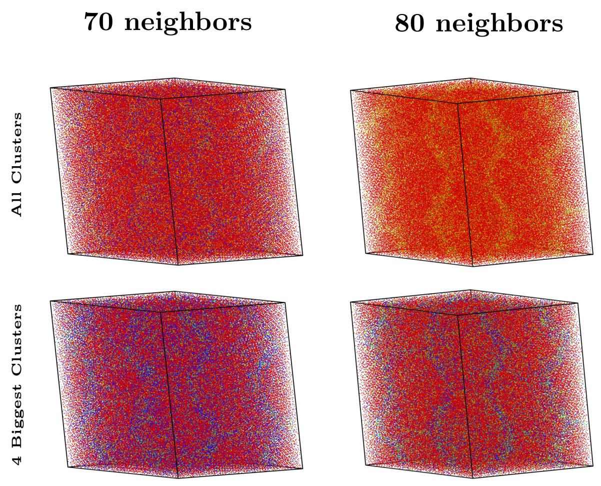
Appendix C

Methane Hydrate Results

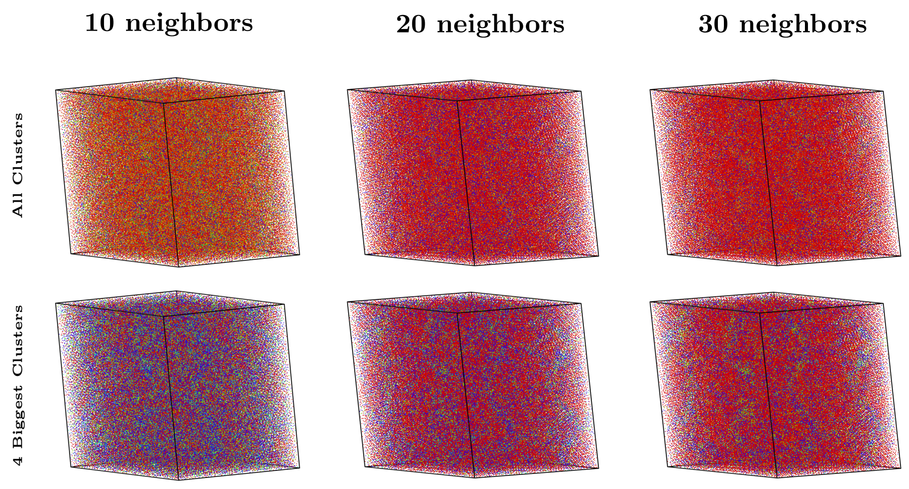
C.1 Convolutional Network

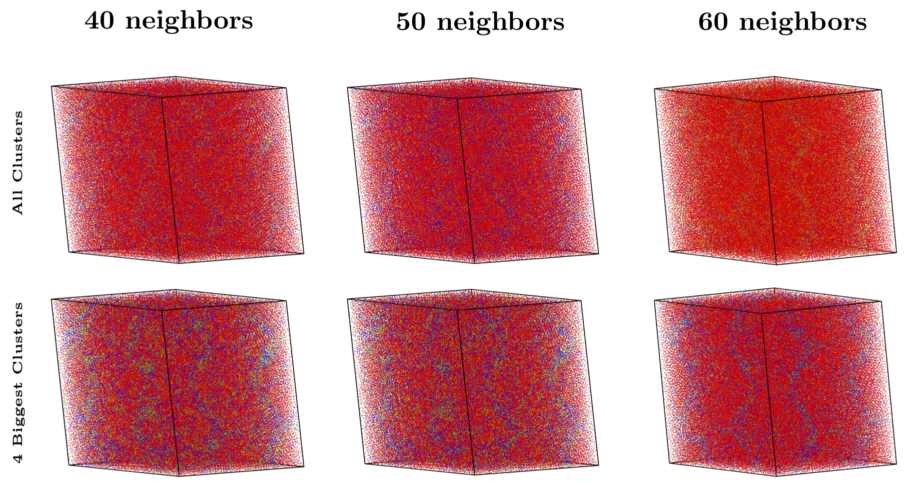


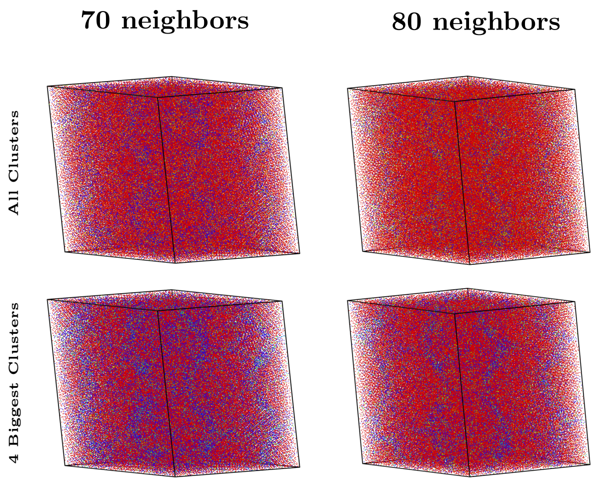

40 neighbors
50 neighbors
60 neighbors
All Clusters
4 Biggest Clusters


70 neighbors
80 neighbors
All Clusters
4 Biggest Clusters

C.2 Dense Network






70 neighbors
80 neighbors
All Clusters
4 Biggest Clusters

Appendix D

Clustering Results

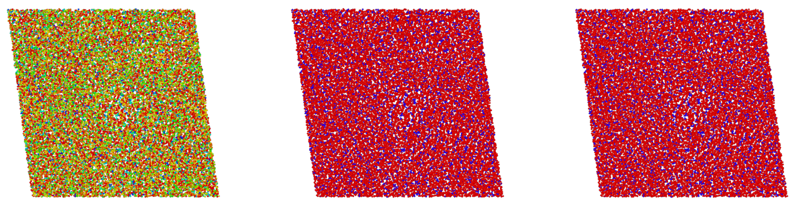
D.1 Gaussian Mixtures

Table D.1: Autoencoder

Sorted by best Calinski-Harbasz			
Score		Parameters	
Calinski	417665.15	Neighbors	60
Silhouette	0.52	$MinPts$	-
Davies	0.53	ε	-
Sorted by best Davies-Boulding			
Score		Parameters	
Calinski	238287.03	Neighbors	60
Silhouette	0.86	$MinPts$	-
Davies	0.26	ε	-
Sorted by best Silhouette			
Score		Parameters	
Calinski	238287.03	Neighbors	60
Silhouette	0.86	$MinPts$	-
Davies	0.26	ε	-

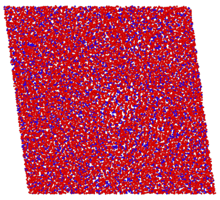
Table D.2: PCA

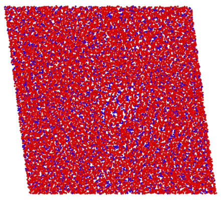
Sorted by best Calinski-Harbasz			
Score		Parameters	
Calinski	920.64	Neighbors	20
Silhouette	0.07	$MinPts$	-
Davies	4.72	ε	-
Sorted by best Davies-Boulding			
Score		Parameters	
Calinski	732.77	Neighbors	30
Silhouette	0.06	$MinPts$	-
Davies	4.7	ε	-
Sorted by best Silhouette			
Score		Parameters	
Calinski	920.64	Neighbors	20
Silhouette	0.07	$MinPts$	-
Davies	4.72	ε	-



(a) Calinski (b) Davies (c) Silhouette

Figure D.1: Best clustering results for gaussian mixture models using autoencoder reduction.


(a) Calinski


(b) Davies

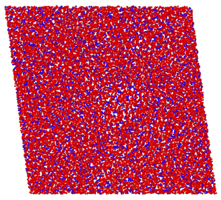

(c) Silhouette

Figure D.2: Best clustering results for gaussian mixture models using PCA reduction.

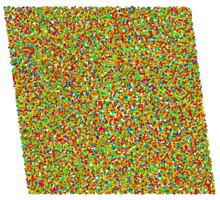
D.2 Agglomerative Clustering

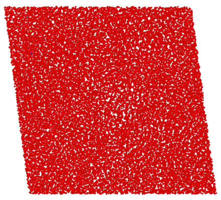
Table D.3: Autoencoder

Score		Parameters	
Sorted by best Calinski-Harbasz			
Calinski	368796.71	Neighbors	60
Silhouette	0.49	$MinPts$	-
Davies	0.55	ε	-
Sorted by best Davies-Boulding			
Calinski	1.0	Neighbors	60
Silhouette	0.0	$MinPts$	-
Davies	0.0	ε	-
Sorted by best Silhouette			
Calinski	249274.13	Neighbors	60
Silhouette	0.86	$MinPts$	-
Davies	0.24	ε	-

Table D.4: PCA

Score		Parameters	
Sorted by best Calinski-Harbasz			
Calinski	1104.08	Neighbors	20
Silhouette	0.09	$MinPts$	-
Davies	3.86	ε	-
Sorted by best Davies-Boulding			
Calinski	1104.08	Neighbors	20
Silhouette	0.09	$MinPts$	-
Davies	3.86	ε	-
Sorted by best Silhouette			
Calinski	1104.08	Neighbors	20
Silhouette	0.09	$MinPts$	-
Davies	3.86	ε	-


(a) Calinski


(b) Davies

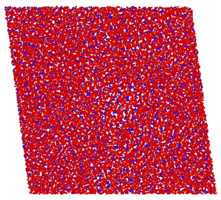

(c) Silhouette

Figure D.3: Best clustering results for agglomerative clustering using autoencoder reduction.

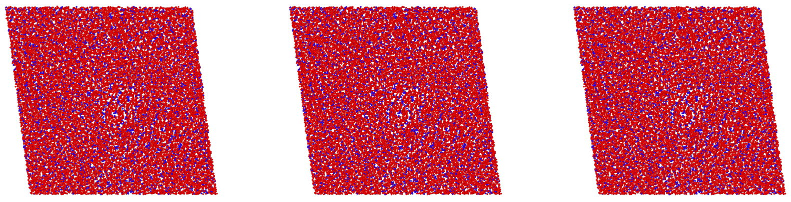

(a) Calinski (b) Davies (c) Silhouette

Figure D.4: Best clustering results for agglomerative clustering using PCA reduction.

D.3 OPTICS

Table D.5: Autoencoder

Sorted by best Calinski-Harbasz			
Score		Parameters	
Calinski	741.62	Neighbors	40
Silhouette	-0.25	$MinPts$	50
Davies	4.83	ε	0.01
Sorted by best Davies-Boulding			
Score		Parameters	
Calinski	71.21	Neighbors	20
Silhouette	0.34	$MinPts$	20
Davies	0.57	ε	20
Sorted by best Silhouette			
Score		Parameters	
Calinski	21.28	Neighbors	60
Silhouette	0.54	$MinPts$	2
Davies	8.31	ε	0.0001

Table D.6: PCA

Sorted by best Calinski-Harbasz			
Score		Parameters	
Calinski	2.25	Neighbors	20
Silhouette	-0.05	$MinPts$	5
Davies	2.36	ε	0.001
Sorted by best Davies-Boulding			
Score		Parameters	
Calinski	1.52	Neighbors	20
Silhouette	-0.19	$MinPts$	2
Davies	1.72	ε	0.0001
Sorted by best Silhouette			
Score		Parameters	
Calinski	1.12	Neighbors	70
Silhouette	-0.04	$MinPts$	2
Davies	2.06	ε	0.01

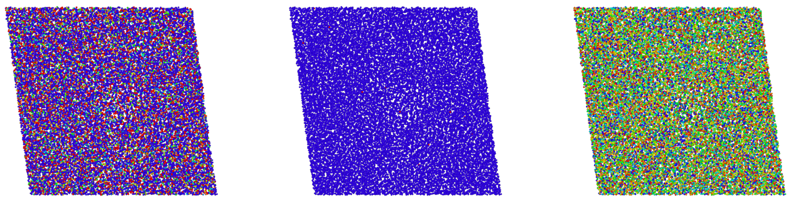
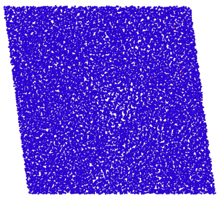
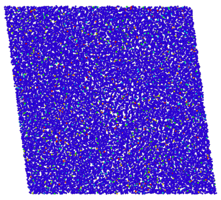

(a) Calinski (b) Davies (c) Silhouette

Figure D.5: Best clustering results for OPTICS using autoencoder reduction.


(a) Calinski


(b) Davies

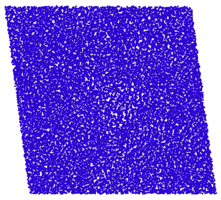

(c) Silhouette

Figure D.6: Best clustering results for OPTICS using PCA reduction.

D.4 DBSCAN

Table D.7: Autoencoder

Sorted by best Calinski-Harbasz			
Score		Parameters	
Calinski	130624.75	Neighbors	60
Silhouette	0.77	$MinPts$	40
Davies	0.96	ε	0.1

Sorted by best Davies-Boulding			
Score		Parameters	
Calinski	21.01	Neighbors	70
Silhouette	0.72	$MinPts$	5
Davies	0.17	ε	1

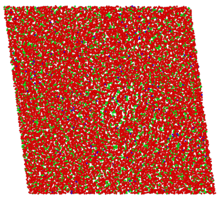
Sorted by best Silhouette			
Score		Parameters	
Calinski	123495.48	Neighbors	60
Silhouette	0.81	$MinPts$	20
Davies	0.5	ε	0.1

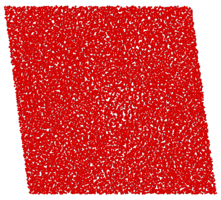
Table D.8: PCA

Sorted by best Calinski-Harbasz			
Score		Parameters	
Calinski	4.86	Neighbors	40
Silhouette	-0.02	$MinPts$	10
Davies	4.13	ε	10

Sorted by best Davies-Boulding			
Score		Parameters	
Calinski	1.46	Neighbors	30
Silhouette	0.1	$MinPts$	10
Davies	0.83	ε	10

Sorted by best Silhouette			
Score		Parameters	
Calinski	1.46	Neighbors	30
Silhouette	0.1	$MinPts$	2
Davies	0.83	ε	10


(a) Calinski


(b) Davies

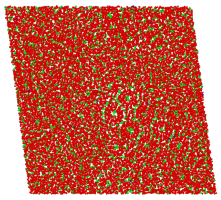

(c) Silhouette

Figure D.7: Best clustering results for DBSCAN using autoencoder reduction.

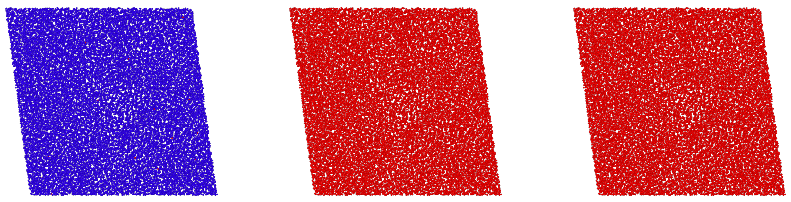

(a) Calinski (b) Davies (c) Silhouette

Figure D.8: Best clustering results for DBSCAN using PCA reduction.