Identifying Local Structures in Dipolar Colloid-Polymer Mixtures using Machine Learning

by

© **Vahid Sheigani**

A thesis submitted in partial fulfillment of the requirements for the degree of Bachelor of Science.

Department of Physics and Physical Oceanography

Memorial University

October 2023

St. John's, Newfoundland and Labrador, Canada

Abstract

A colloid-polymer mixture with an applied electric field is subject to two categories of forces, the induced dipole-dipole interactions and depletion forces due to the polymer. The combination of these forces with different polymer concentrations and external electric fields results in the formation of various structures with different local orders. To study these structures in more detail, the colloid-polymer system can be replicated computationally using the molecular dynamics method which enables us to calculate particle features such as bond order parameters computationally and further investigate these features using advanced methods such as machine learning. In this thesis, we apply a multi-step machine learning algorithm to dipolar-depletion systems and identify the local structure of atoms in the simulation box using the algorithm. The machine learning algorithm is a combination of multiple cutting-edge machine learning techniques including autoencoders, Gaussian mixture models, and a cluster merging technique. These algorithms are combined to create a multi-step process that can identify different structures of matter in any molecular dynamics simulation output. This algorithm utilizes unsupervised machine learning which does not require labeled data and is applicable to known and unknown local structures. The machine learning model can identify different local structures in colloid-polymer systems and the identified clusters of atoms in the systems is in complete agreement with our understanding of the systems.

Acknowledgements

I would like to thank my supervisor Dr. Ivan Saika-Voivod for introducing me to the field of condensed matter physics and for his invaluable insights into molecular dynamics simulations and for his guidance through conducting this research. I would also like to thank Cassandra Clowe-Coish for providing the LAMMPS configurations that became the basis for my MD simulations. I would also like to thank Drs. Anand Yethiraj and Shivani Semwal for providing insight into the colloid-polymer mixture experiment.

Table of contents

List of figures

Chapter 1

Introduction

1.1 Overview

The purpose of this study is to identify local structures present in colloid-polymer mixtures that include dipolar interactions. To achieve this goal, we first introduce dipolar-depletion systems and describe different forces in these systems. We then introduce molecular dynamics simulations and the software, LAMMPS [1], which we use to carry out the simulations, together with some of the details of simulating the dipolar-depletion systems. Then we describe averaged bond order parameters that we use as the input for the machine learning algorithm and the cutoff distance measure that we use to identify the particle neighbours. In chapter 2, we introduce step-by-step the unsupervised machine learning algorithm that we use to identify local structures. The first step of the machine learning algorithm is an autoencoder, which is a form of feed-forward neural network. The second step is a Gaussian mixture model (GMM) and the final step is a merging algorithm that we use to merge the GMM clusters and obtain the final results. The description of the machine learning algorithm is

followed by an illustrative case of a simple system containing one hexagonal close packed (hcp) and one face-centred cubic (fcc) crystal to validate the algorithm and compare the obtained results with our knowledge of the system. fcc and hcp crystal differences are as follows. The most direct difference between hcp and fcc crystals is in the atomic arrangements. Both hcp and fcc are close-packed, which means they have the maximum theoretical packing density of about 74%, but the fcc structure is a cube with an atom at all 8 corner positions, and at the center of all 6 faces and the hcp structure is a hexagonal prism with an atom at all 12 corner positions, an atom in the middle of both the top and bottom faces, and 3 atoms in the central layer. In both crystals, the structure consist of repeated layers of 2D triangular lattices. If the layers repeat with an abc-abc-abc... arrangement then the crystal is fcc. If the layers repeat with an ab-ab-ab... arrangement then the crystal is hcp. In chapter 3, we apply the machine learning algorithm on the dipolar-depletion systems simulated using molecular dynamics and report the results. Finally, in chapter 4, we discuss the results and provide a path for further research and continuation of this thesis topic.

1.2 Dipolar Depletion Systems

Lekkerkerker et al. [2] describe the colloid-polymer system as analogous to a restaurant room on two different occasions. In a regular time of the day, the tables are arranged in a typical restaurant fashion. But when the restaurant is booked for a busy party, the tables are pushed to a certain area of the restaurant (usually near a wall) to clear more area for the people present in the party. The new configuration allows more space for the movements of the guests. This transition of the tables and the apparent attraction between the tables is due to repulsive interaction between the guests. In other words, people do not like to be too close to each other, and also like

to have maximum freedom in the space while they still have access to the tables.

In other words, in a colloid-polymer mixture, the formation of a depletion layer – a shell where the centre of mass of a polymer does not enter – around the colloidal hard spheres creates an effective attractive force between the hard spheres of the colloid. In figure 1.1 the depletion layer is shown by the dashed circles around the hard spheres. When the depletion layers of the hard spheres overlap, the volume available for polymer chains increases. This means that the entropy of the mixture is maximized if the colloidal hard spheres are close together, and so the colloids act as if there were an attractive force between them. The picture sketched above first became clear in the 1950s through the work of Asakura and Oosawa [3].

The system that is used for the purpose of this thesis is the computational replica of the system used by Semwal et al. [4], one comprised of fluorescently labeled polymethyl methacrylate (PMMA) colloidal microspheres dispersed in a density-matched solvent mixture of cyclohexyl bromide (CHB) and cis-trans decalin. As a depletant, non-adsorbing polymer, polystyrene (PS) was used. To suppress Coulomb interactions and obtain a hard-sphere-like system, they added a salt, tetrabutylammonium bromide (TBAB). For this system, the colloidal particles behaved approximately as hard spheres. The colloid-polymer suspension was then transferred to an electric field cell. In the Semwal et al. experiments, the field strength E applied to the sample ranged between 0 and 0.53 V/μm, and the polymer concentration c_p ranged from 0 to 10.3 mg/ml. Semwal et al. characterized the system's structure through both the two-dimension radial distribution function $g(r)$, and the fraction of particles with high average bond order parameter $\overline{\psi}_s$, denoted as f_s, where s is an index quantifying the local symmetry. We only analyze the difference between systems based on the difference in their $\overline{\psi}_s$, as bond order parameters are the only features of the systems

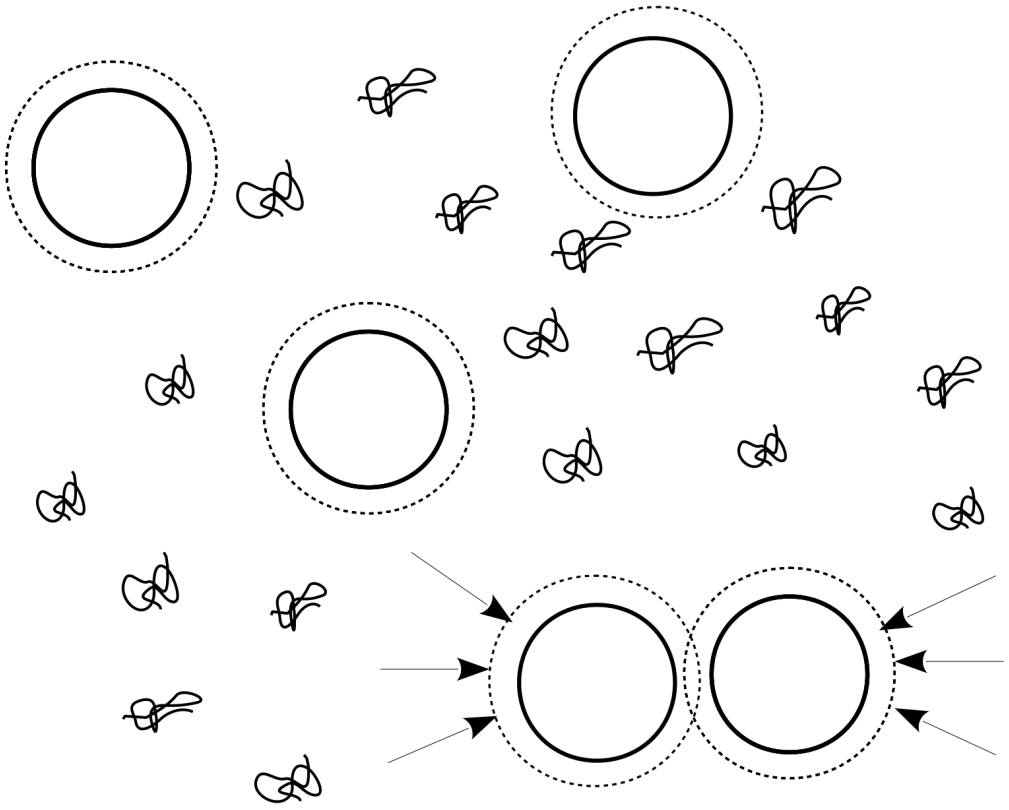


Figure 1.1: The colloidal spheres in a solution with non-adsorbing polymers. The depletion layers are indicated by short dash lines.

that are relevant to our research. The significance of these features and their usage in our machine learning algorithm is later described in Chapter 3.

When an electric field is applied to the suspension of colloidal particles with a dielectric constant mismatch between the particles and solvent, it creates a dipole moment parallel to the field in the particles. The stable structure at high electric fields (i.e., high dipolar strength) is known to be the body-centered tetragonal (bct) crystal [4]. A field-induced change in crystal structure will result in a change in the bond order parameters $\overline{\psi}_s$ and fractions f_s.

1.3 Molecular Dynamics Simulation

The molecular dynamics (MD) simulation method applies classical Newtonian mechanics to describe the movements of atoms. Since the modeling of the systems is based on a solid physical foundation, the MD method can be used to obtain the properties of the system with high accuracy. The MD simulation method is simple and computationally efficient compared to quantum mechanical methods and the results are in good agreement with the quantum mechanical models at sufficiently high temperatures [5]. Different types of force fields for various molecular systems, such as biomolecules, polymers, metals, and semiconductors have been developed for use in MD simulations. Large systems and sophisticated models can be simulated using MD as a result of the versatility of the tools and features developed for MD simulations.

There are several popular MD simulation software packages that also allow parallel computing including NAMD, AMBER, CHARMM, LAMMPS, and GROMACS. Among them, LAMMPS, and GROMACS are the most widely used due to their parallel computing features, free accessibility, and frequent updates. GROMACS is mainly

used in modeling biochemical molecules, such as proteins and lipids due to being capable of calculating complicated interactions. GROMACS also has been widely used in modeling non-biological systems including polymers [5].

1.4 LAMMPS

LAMMPS (large-scale atomic/molecular massively parallel simulator) is one of the most popular MD simulation software for the modeling of condensed matter [1]. LAMMPS is highly efficient and allows the usage of a variety of potential energies for a wide range of materials including solid-state materials and soft matter. LAMMPS is an open-source software with the flexibility that allows users to develop code based on the problem at hand [5].

1.4.1 LAMMPS Configuration

In this section, I explain the configuration that I use for modeling the experimental system created by Semwal et al. [4]. The modeling methods of this research are based on the work done by Semwal et al. [4] and Clowe-Coish [6].

To model the depletion interaction with molecular dynamics (MD) simulations, we use the general short-range pair potential presented by Wang et al. [7]. This potential is defined as:

$$\frac{U_c(r)}{k_B T} = \alpha\epsilon \left(\left[\frac{\sigma}{r}\right]^2 - 1 \right) \left(\left[\frac{r_c}{r}\right]^2 - 1 \right)^2, \tag{1.1}$$

where r is the distance between two particles, r_c is the cutoff distance for the potential, σ is the colloid diameter, and α is a dimensionless normalization parameter that ensures that the well depth is ϵ. We assume that this ϵ is the parameter that is

obtained experimentally and increases with polymer concentration c_p. Wang et al. [7] suggest a value of $r_c = 1.2\sigma$ to produce a narrow potential well. In this case, the location of the minimum of the potential is $r_{min} \approx 1.055\sigma$, a value that is consistent with the scale of the polymer-colloid size ratio $\xi = 0.066$ in Ref. [4]. In all our simulations, k_BT is set to 1.

To model the effect of the electric field, we add point dipolar interactions between the colloidal particles, where all dipole moment vectors are aligned along the z direction. The dipole-dipole interaction energy between two dipoles $\vec{p}_1$ at the origin and $\vec{p}_2$ at position $\vec{r}$ is:

$$U_d(\vec{r}) = \Lambda \left(\frac{\sigma}{r}\right)^3 \left[\vec{p}_1 \cdot \vec{p}_2 - 3\left(\vec{p}_1 \cdot \hat{r}\right)\left(\vec{p}_2 \cdot \hat{r}\right)\right] \tag{1.2}$$

In our system, $\vec{p}_i = p\hat{z}$. Λ is the strength of the dipolar interaction,

$$\Lambda p^2 = \frac{1}{16}\pi\epsilon_0\epsilon_s\sigma^3\beta^2E_0^2 \tag{1.3}$$

where ϵ_0 is the permittivity of free space, $\epsilon_s = 6.1$ is the dielectric constant of the solvent, $\epsilon_c = 2.6$ is that of the colloid, and β is the dielectric mismatch matching the experimental parameters in Ref. [4],

$$\beta = \frac{-1 + \epsilon_c/\epsilon_s}{2 + \epsilon_c/\epsilon_s} \approx -0.24. \tag{1.4}$$

MD simulations are performed for multiple values of dimensionless depletion strength ϵ, and for multiple values of dimensionless dipole moment $\mu = \sqrt{p^2\Lambda/k_BT}$. While the range of μ values that we use in the simulations is different from the values that were used in the experiment by Semwal et al., we believe that the state points simulated in this research yield dipolar structures representative of those observed in the

experiments. We use the values of $\mu = 0,\ 2,\ 4$ and $\epsilon = 0,\ 1,\ 3.25$ in this research to generate nine state points, reflecting all nine ϵ-μ pairs.

1.5 Bond Order Parameters

The complex vector q_{lm} is defined as,

$$q_{lm}(i) = \frac{1}{N_b(i)} \sum_{j=1}^{N_b(i)} Y_{lm}(\mathbf{r}_{ij}) \tag{1.5}$$

where $N_b(i)$ is the number of the nearest neighbors of particle i, l is a free integer parameter, and m is an integer that runs from $m = -l$ to $m = +l$. The $Y_{lm}(\mathbf{r}_{ij})$ functions are the spherical harmonics and $\mathbf{r}_{ij}$ is the vector pointing from particle i to particle j. A set of parameters that hold the information of the local structure are the local bond order parameters, or the Steinhardt order parameters, defined as,

$$q_l(i) = \sqrt{\frac{4\pi}{2l+1} \sum_{m=-l}^{l} |q_{lm}(i)|^2}. \tag{1.6}$$

Depending on the choice of l, these parameters are sensitive to different crystal symmetries. Each of them depends on the angles between the vectors to the neighbouring particles only and therefore these parameters are independent of the reference frame [8].

1.5.1 Averaged Bond Order Parameters

The crystal structure determination described above can be improved by using the following averaged form of the local bond order parameters,

$$\bar{q}_l(i) = \sqrt{\frac{4\pi}{2l+1} \sum_{m=-l}^{l} |\bar{q}_{lm}(i)|^2} \tag{1.7}$$

where

$$\bar{q}_{lm}(i) = \frac{1}{\tilde{N}_b(i)} \sum_{k=0}^{\tilde{N}_b(i)} q_{lm}(k) \tag{1.8}$$

Here, the sum from $k = 0$ to $\tilde{N}_b(i)$ runs over all neighbors of particle i plus the particle i itself. Thus, to calculate $\bar{q}_l(i)$, one uses the local orientational order vectors q_{lm} averaged over particle i and its surroundings. While $q_l(i)$ holds the information of the structure of the first shell around particle i, its averaged version $\tilde{q}_l(i)$ also takes into account the second shell. [8]

1.5.2 Determining the cutoff distance

We use solid-angle based, nearest-neighbor algorithm (SANN) to determine the cut-off distance and find the neighbors of the particles [9]. SANN is a parameter-free algorithm which makes it a better choice for creation of a parameter-free machine learning algorithm. In addition, the cutoff distance is a local property, which makes the SANN algorithm suitable for systems with inhomogeneous densities. The SANN algorithm is preferred over the usual fixed cutoff algorithm because the fixed cutoff algorithm results are very sensitive to the value of the cutoff and determining the value of the cutoff for our system is difficult due to the complexity of the system's

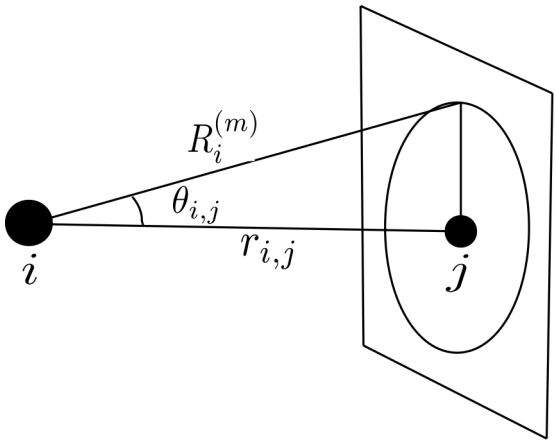


Figure 1.2: Definition of the solid angle $\theta_{i,j}$ for neighbour j of particle i.

structure. The cutoff required for the fixed cutoff method should be defined manually or using other algorithms which adds another step and more complexity to our algorithm while SANN, by itself, determines the cutoff locally and does not require a cutoff determination step.

Consider particle i located at position r_i surrounded by particles labeled by j. For each particle i, SANN determines an individual cutoff distance $R_i^{(m)}$, which we call the shell radius. First, we sort the particles j surrounding i such that $r_{i,j} \leq r_{i,j+1}$ for all j which also means:

$$r_{i,m} < R_i^{(m)} < r_{i,m+1} \tag{1.9}$$

Then starting with the particle closest to i, we associate an angle $\theta_{i,j}$ with each particle j, based on the distance between the particles $r_{i,j} = |r_j - r_i|$. Figure 1.2 depicts the definition of solid angle ($\theta_{i,j}$) associated with neighbour j of particle i. We define the neighbourhood of a particle i to consist of the nearest m particles such that the sum

of their solid angles ($\theta_{i,j}$) equals 4π.

$$4\pi = \sum_{j=1}^{m} 2\pi[1 - \cos(\theta_{i,j})] = \sum_{j=1}^{m} 2\pi[1 - r_{i,j}/R_i^{(m)}] \tag{1.10}$$

In other words, combining Eqs. 1.9 and 1.10 gives:

$$R_i^{(m)} = \frac{\sum_{j=1}^{m} r_{i,j}}{m-2} < r_{i,m+1} \tag{1.11}$$

To solve this inequality, we start with $m = 3$ and increase m iteratively. During each iteration, we evaluate Eq. 1.11 and the smallest m that satisfies this equation yields the number of neighbors $N_b(i)$ and the shell radius $R_i^{(m)}$. The m particles inside the shell radius are the particle i's nearest neighbors [9].

The process of finding neighbors and calculating bond order parameters of the systems simulated by LAMMPS is implemented in python's `pyscal` library which uses C++ as the backend for these calculations. In my research, I used this library for these calculations in all the systems.

Chapter 2

Machine Learning

2.1 Overview

In this section, I describe the machine learning algorithm that is used to classify different structures of atoms in the systems simulated by LAMMPS. The overall process consists of five steps. Step one is generating a sample system of particles using the LAMMPS software, which implements the molecular dynamics simulation method. Step two is importing the results of the LAMMPS simulation into the Python environment and calculating the values of the bond orientational order parameters of the system. Step three is implementing the autoencoder based on the article by Boattini et al. [10]. The output of the autoencoder is a set of points in a plane, where points that are close together represent particles with similar structural environments. Clusters of points represent particles that belong to the same structure. Step four is implementing the Gaussian mixture model (GMM), which identifies clusters, and using the Bayesian information criterion (BIC) values to find the optimal number of clusters for the GMM based on the article by Boattini et al. [10]. Step five is using

the cluster merging algorithm by Baudry et al. to merge the particularly similar clusters [11]. The optimal number of clusters of particles and the ability to label particles as belonging to a particular cluster/structure in the simulated box is the result of the final step. We first apply the algorithm to the test case of a system containing only face-centered cubic (fcc) and hexagonally close-packed (hcp) structures. We then apply the algorithm to the configurations drawn from the simulation of dipolar-depletion systems in Chapter 3.

The remainder of this chapter is organized as follows: First, I give a brief introduction to perceptrons, neural networks, and backpropagation, and then I describe a specific type of neural network called autoencoder, that we use in this research. Then, I describe the details of the autoencoder that we use such as regularized autoencoders and the weight-decay regularization followed by the Gaussian mixture model and the merging algorithm. Finally, I explain how all these machine learning methods come together and create a single machine learning algorithm by applying the machine learning algorithm on an illustrative case and providing the results.

2.2 Neural Networks

To understand Neural networks we should understand their most important components which are perceptrons and the backpropagation scheme that is used to train the network. This section summarizes the discussion of the neural networks in Refs. [12] and [13].

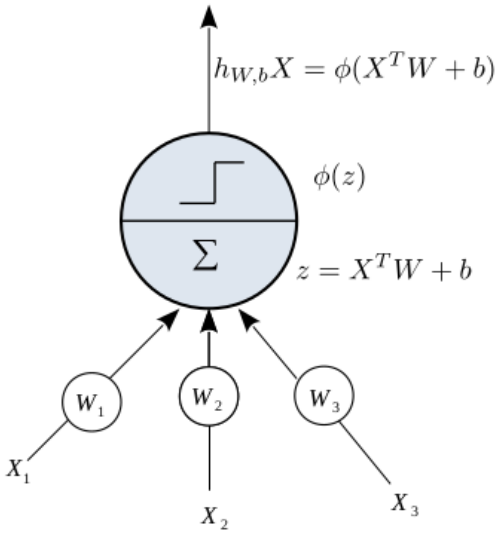


Figure 2.1: An artificial neuron that computes a weighted sum of its inputs and then applies a step function.

2.2.1 The Perceptron

Aurelien Geron in [12] describes perceptrons as follows. The perceptron (also called an artificial neuron) refers to a single unit of an artificial neural network (ANN). The inputs and output are real numbers, and each input connection has a weight associated with it. Figure 2.1 depicts a perceptron and its internal process. In this figure, X is the vector of inputs to the perceptron, W is the weights vector, W_1, W_2, and W_3 are the individual weights for the individual inputs X_1, X_2 and X_3, and b is the individual neuron's bias. The perceptron computes a weighted sum of its inputs and adds the biases ($X^T W + b$), then applies an activation function ϕ, which for example could be a step function, to that sum and outputs the result: $h_{W,b}(X) = \phi(z)$, where $z = X^T W + b$. A neural network is simply composed of multiple layers of perceptrons. The first layer of perceptrons (also called passthrough neurons) is fed the input dataset and outputs whatever input it is fed. The perceptrons in subsequent so-called hidden layers are connected to the previous layer of perceptrons, receiving the output of the

previous layer as their input. When all the neurons in a layer are connected to every neuron in the previous layer, the layer is called a *fully connected layer*, or a *dense layer*. To summarize, the output of a single neuron can be calculated using,

$$h_{W,b}(\mathbf{X}) = \phi(\mathbf{X}^T\mathbf{W} + \mathbf{b}) \tag{2.1}$$

In this equation:

- As always, $\mathbf{X}$ represents the matrix of inputs. It has one row per observation and one column per feature.
- The weight matrix $\mathbf{W}$ contains all the connection weights except for the ones from the bias neuron. It has one row per input neuron and one column per artificial neuron in the layer.
- The bias vector b contains all the connection weights between the bias neuron and the artificial neurons. It has one bias term per artificial neuron.
- The function ϕ is called the activation function: when the artificial neurons are TLUs, it is a step function (but we will discuss other activation functions shortly).

2.2.2 Gradient Descent Backpropagation

Sergios Theodoridis in Ref. [13] explains the backpropagation training scheme as follows: Let $(y_n, x_n),\ n = 1,\ 2, ...,\ N,$ be the set of training samples. Note that we have assumed multiple output variables assembled as a vector. We assume that the network comprises L layers; $L - 1$ hidden layers and one output layer. Each layer

consists of $k_r, r = 1, 2, ..., L$, neurons. Thus, the (target/desired) output vectors are

$$y_n = [y_{n1}, y_{n2}, ..., y_{nk_L}]^T \in R^{k_L},\ n = 1,\ 2,\ ,\ \ldots,\ N. \tag{2.2}$$

For the sake of the mathematical derivations, we also denote the number of input nodes as k_0; that is, $k_0 = l$, where l is the dimensionality of the input feature space. Let θ_j^r denote the vector of the synaptic weights associated with the jth neuron in the rth layer, with $j = 1,\ 2,\ \ldots,\ k_r$ and $r = 1,\ 2,\ \ldots, L$, where the bias term is included in θ_j^r , that is,

$$\theta_j^r := [\theta_{j0}^r, \theta_{j1}^r, ..., \theta_{jk_{r-1}}^r]^T. \tag{2.3}$$

The weights link the respective neuron to all neurons in layer k_{r-1}. The basic iterative step for the gradient descent scheme is written as

$$\theta_j^r(new) = \theta_j^r(old) + \Delta\theta_j^r \tag{2.4}$$

$$\Delta\theta_j^r \equiv \mu \frac{\partial J}{\partial \theta_j^r}\bigg|_{\theta_j^r(old)} \tag{2.5}$$

where the parameter μ is the learning-rate, an empirical parameter which we set to $\mu = 10^{-5}$ following [10], and J is the cost function defined as:

$$J = \sum_{n=1}^{N} \mathcal{L}(y_n,\ \hat{y}_n) \tag{2.6}$$

where $\hat{y}_n = f_\theta(x_n)$ is the target predicted by the neural network, and $\mathcal{L}$ is a loss function that quantifies the difference between predicted outputs and true values. A difficulty arises from calculating the gradients of a multi-layer network. To calculate the gradients in Eq. 2.5, for all neurons in all layers, the following steps should be

implemented in the neural network algorithms:

- Forward computations: For a given input vector x_n ,$n = 1, 2, \ldots, N$, use the current estimates of the parameters (weights) $[\theta_j^r(old)]$ and compute all the outputs of all the neurons in all layers.

- Backward computations: Using the above computed neuronal outputs together with the known target values, y_{nk_L} , of the output layer, compute the gradients of the cost function. This involves L steps, that is, as many as the number of layers. The sequence of the algorithmic steps is given below:

  - Compute the gradient of the cost function with respect to the parameters of the neurons of the last layer, i.e., $\frac{\partial J}{\partial \theta_j^L},\ j = \ 1,\ 2, \ldots .\ k_L$.

  - **For** $r = L - 1$ to 1, **Do**

    Compute the gradients with respect to the parameters associated with the neurons of the rth layer with respect to the parameters of the layer $r + 1$ that have been computed in the previous step

  - **End For**

The backward computations scheme is a direct application of the chain rule for derivatives, and it starts with the initial step of computing the derivatives associated with the last (output) layer, which turns out to be straightforward. Then the algorithm "flows" backward in the hierarchy of layers. This is because of the nature of the multi-layer network, where the outputs, layer after layer, are formed as functions of functions [13].

2.3 Autoencoders

An autoencoder is a neural network that is trained to attempt to reproduce its input as its output. Internally, it has a hidden layer h that describes the code or the process used to recreate the input. The network consists of two parts: an encoder function $h = f(x)$ and a decoder that produces a reconstruction of the input $r = g(h)$. If an autoencoder succeeds in simply learning to exactly set $g(f(x)) = x$ everywhere, then it is not generally useful. Instead, autoencoders are designed in a way to be unable to learn to copy the input blindly and perfectly. To make autoencoders more useful, they are usually restricted in ways that allow them to reproduce input values only approximately. The autoencoder learns important properties of the input because it is restricted to copying only the important features of the data.

Feedforward neural networks are a form of fully-connected neural network. The inputs are fed to the first layer, and subsequently the outputs of a layer are fed forward as inputs to the next layer. Weights are adjusted backward using back-propagation. Autoencoders are really a special case of feedforward networks, so they are trained with the same techniques, typically gradient descent following gradients computed by back-propagation. Autoencoders are mostly used for dimensionality reduction or feature learning. In our algorithm, they are used for both purposes. We train the autoencoder to reduce the dimensionality of the input vectors and learn which features are the most important in minimizing the network loss function.

2.3.1 Undercomplete Autoencoders

While copying the input to the output is not useful, in the autoencoder's case we are typically not interested in the output of the decoder. Instead, we hope that

training the autoencoder will result in h taking on useful properties. One way for the autoencoder to figure out the useful features of the training dataset is to constrain h to have a smaller dimension than x. An autoencoder whose bottleneck dimension is less than the input dimension is called undercomplete. The smaller bottleneck dimension forces the autoencoder to capture the most important features of the training data. The learning process is described simply as minimizing a loss function.

$$\mathcal{L}(x, g(f(x))) \tag{2.7}$$

where $\mathcal{L}$ is a loss function penalizing $g(f(x))$ for being dissimilar from x, such as the mean squared error (MSE).

2.3.2 Regularized Autoencoders

Undercomplete autoencoders, with bottleneck layer dimension less than the input dimension, can learn the most salient features of the data distribution. The bottleneck later is the layer with smallest number of neurons. We have seen that these autoencoders fail to learn anything useful if the encoder and decoder are given too much capacity. A similar problem occurs if the bottleneck layer is allowed to have a dimension equal to the input, and in the overcomplete case in which the bottleneck layer has a dimension greater than the input. In these cases, the autoencoder can learn to copy the input to the output without learning anything useful about the data distribution.

Regularized autoencoders provide the ability to train any architecture of autoencoder successfully, determining the bottleneck dimension, the number of hidden layers, and the number of neurons in each layer (including both the encoder and decoder)

based on the complexity of the distribution of the input data. Regularized autoencoders use a loss function that encourages the model to have other properties such as smallness of the derivative of the representation function (i.e., avoid very large weights), and robustness to noise besides the ability to copy its input to its output. A regularized autoencoder can be overcomplete but still learn something useful about the data distribution. The strategy for regularizing an autoencoder is to use a penalty Ω such that:

$$Loss = \mathcal{L}(x, g(f(x))) + \Omega(h, x), \tag{2.8}$$

where $\mathcal{L}(x, g(f(x)))$ can be any loss function such as mean squared error (MSE), and $\Omega(h, x)$ is the penalty term. In this research, we use weight-decay autoencoders to avoid overfitting the autoencoder on the training data set.

2.3.3 Weight Decay Regularization

One of the simplest regularizers is the weight decay regularizer. This regularizer simply uses some of the squares of the weights of the layers as the penalty term. The weight decay regularizer is defined as:

$$Loss = \mathcal{L}(x, g(f(x))) + \lambda \sum_i {w_i}^2 \tag{2.9}$$

where the sum runs over all the weights of the neural network and thus depends on the shape of the network, and λ is the coefficient of the penalty term. The justification for this penalty term is that producing an overfitted result requires large values of the weights. This regularizer encourages the network weights to be small and thus prevents overfitting [14].

2.4 Gaussian Mixture Models

A GMM is a probabilistic model that assumes that the data points are generated from a mixture of multiple Gaussian distributions with unknown parameters. All the data points that are generated from a single Gaussian distribution form a cluster. The task of clustering is to assign a number of points, $x_1, ..., x_N$, into K groups or clusters. Points that are assigned to the same cluster must be more "similar" than points that are assigned to different clusters. Some clustering algorithms need the number of clusters, K, to be provided by the user as an input variable. Other schemes treat it as a free parameter to be recovered from the data by the algorithm. There are also several GMM variants. The simplest variant is implemented in the `GaussianMixture` class of the `scikit-learn` library in Python. In this simple variant, you need to know the number of Gaussian distributions that you want to use for the clustering algorithm in advance.

The other major issue in clustering is quantifying "similarity". Different definitions end up with different clustering results. A clustering is a specific allocation of the points to clusters. Thus, in general, any clustering algorithm provides a suboptimal solution. Gaussian mixture modeling is among the popular clustering algorithms. The main assumption is that the points which belong to the same cluster are distributed according to the same Gaussian distribution (this is how similarity is defined in this case), of unknown mean and covariance matrix. This is can be written mathematically as:

$$p(x) = \sum_{k=1}^{K} P_k p(k|x_n) \tag{2.10}$$

where P_k is the parameter weighting the specific contributing probability distribution function (PDF). To guarantee that $p(x)$ is a PDF, the weighting parameters must be nonnegative and add up to one. Each mixture component defines a different

cluster. Thus, the goal is to run the *Expectation Maximization* (EM) algorithm over the available data points to provide, after convergence, the probabilities $p(k|x_n)$, $k = 1, 2, \ldots, K$, $n = 1, 2, \ldots, N$, where each k corresponds to a cluster. Then, each point is assigned to cluster k according to the following rule: Assign x_n to cluster k that maximizes $p(i|x_n)$, $i = 1, 2, \ldots, K$ [13]. The number of clusters K is a parameter of the model that is provided by the user. To calculate K we use a common technique that chooses the value of k by minimizing the Bayesian information criterion (BIC).

2.5 Bayesian Information Criterion (BIC)

The BIC balances the number of model parameters d and the number of data points n against the maximum likelihood function $\hat{L}$ defined as,

$$BIC = -2\log(\hat{L}) + \log(n)d \tag{2.11}$$

where n is the number of samples, and d is the number of model parameters. For a Gaussian model, the maximum log-likelihood is defined as,

$$\log(\hat{L}) = -\frac{n}{2}\log(2\pi) - \frac{n}{2}\ln(\sigma^2) - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{2\sigma^2} \tag{2.12}$$

where σ^2 is an estimate of the noise variance, and y_i and $\hat{y}_i$ are respectively the true and predicted targets. The noise variance is defined as,

$$\sigma^2 = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i')^2}{n - p} \tag{2.13}$$

where p is the number of data features and $\hat{y}_i'$ is the predicted target using an ordinary least squares regression. Note that Eq. 2.13 is only valid when $n > p$ [13].

While this procedure works perfectly for clusters that are actually generated from a mixture of separate multivariate normal distributions, the clusters that underlie our data are very often far from being Gaussian distributed in space. As a consequence, a single cluster in the data may be detected as two or more mixture components (if its distribution is indeed better approximated by a mixture of Gaussians than by a single Gaussian function), meaning that the number of clusters in the data may, in general, be different from the number of components found by minimizing the BIC [10].

2.6 Merging Algorithm

To overcome this problem, we use the method proposed by Baudry et al. [11]. The idea is to first use the BIC in order to find a GMM with N_G components that fits the data well. Then, a sequence of candidate clusterings with $K = N_G, N_G - 1, \ldots, 1$ clusters is formed by successively merging a pair of components. At each step, the two mixture components to be merged are chosen to minimize the entropy of the resulting clustering, defined as

$$S_K = -\sum_{i=1}^{n}\sum_{j=1}^{K} p_{ij} \ln(p_{ij}) \tag{2.14}$$

where n is the number of observations and K is the number of clusters. The entropy of the classification decreases with increasing probability that a point belongs to a specific cluster. As a result, higher probabilities result in a smaller entropy. Finally, the optimal number of clusters is found by looking for the existence of an elbow in the entropy S_K as a function of K [10].

2.7 Illustrative Case

In order to validate and optimize the machine learning algorithm for our purpose, the machine learning algorithm is applied to a specific case that only includes fcc and hcp crystals. We then compared the results of the algorithm with our knowledge of the system to validate the results of the algorithm.

The first step is of the algorithm is to read the results of the LAMMPS simulation in Python and calculate the bond orientational order parameters(BOPs) associated with each crystal using the `pyscal` Python library. [15] Using C++ as the backend for the calculations of BOPs, `pyscal` is one of the fastest libraries for these calculations compared to other libraries that are also tested for this task including the `partycls` Python library. For the purpose of this research, a total of eight BOP values (l ranging from 2 to 9 in Eq. 1.8) is calculated and used as the input vector for the training of the autoencoder.

In our research, we employ feedforward and fully connected regularized autoencoder with weight decay. The number of input and output layer neurons is determined by the dimensionality of the input vector (eight in our case). The autoencoder consists of two hidden layers each consisting of eighty neurons (10 times the number of input neurons). The bottleneck layer of the autoencoder consists of two neurons which is the low-dimensional projection of the input vector and thus is used to reduce the dimensionality of the input vectors from an N by eight matrix to an N by two matrix. The weights and biases of the autoencoder are initialized with the normalization initialization proposed by Glorot and Bengio [16].

Figure 2.2 shows the architecture of the autoencoder used for the purpose of this research. This structure follows the structure of the autoencoder that is designed

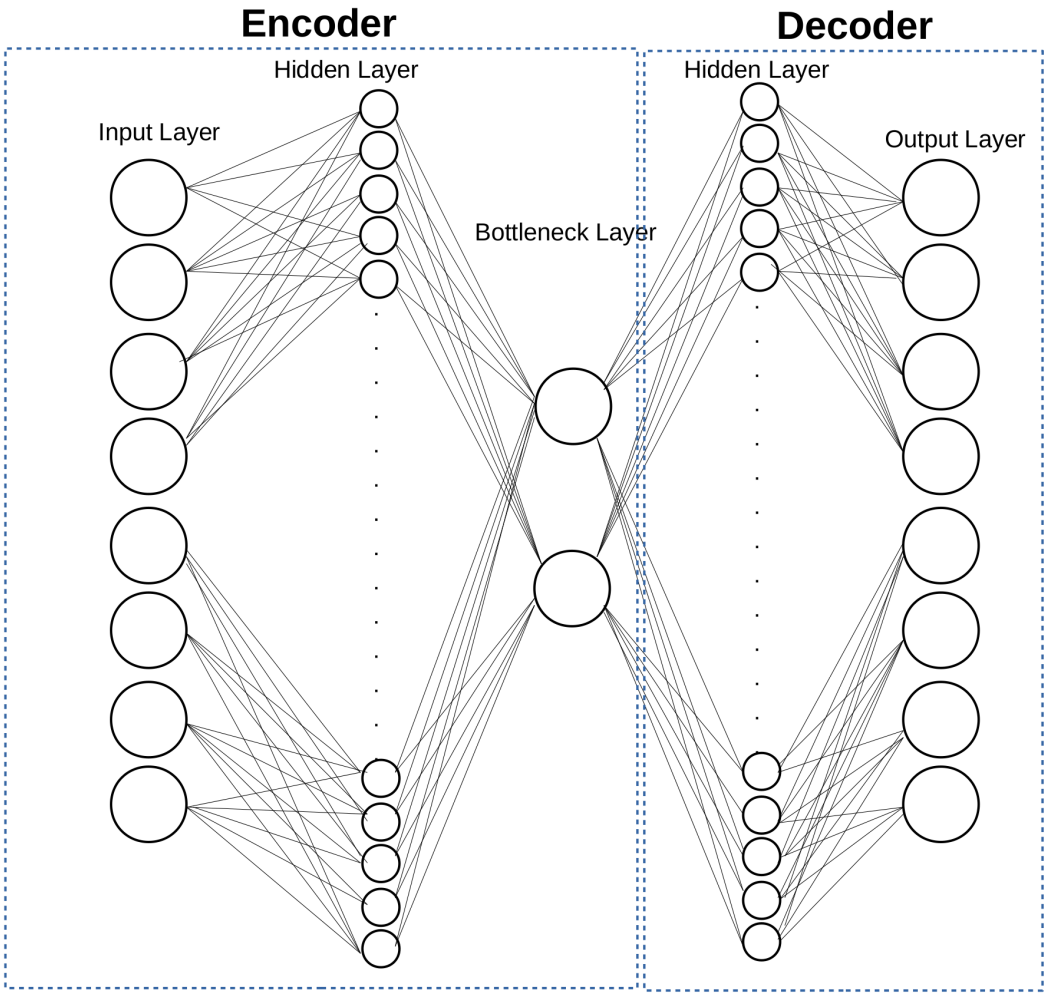


Figure 2.2: Architecture of a neural-network based autoencoder. The encoder network finds a low-dimensional representation of the input, from which the decoder reconstructs an approximation of the input as output.

by Boattini et al.[10] for the purpose of classification of crystal and liquid systems. Note that the number of hidden layers, the number of neurons in the bottleneck layer, and the hyperparameters of the encoder (including the batch size and the number of epochs) were determined empirically by monitoring the result for the case of classification of crystal systems. The number of epochs is the number of times that the training procedure of the neural network is applied to the entire input dataset.

Among the hyperparameters, the number of epochs is of high importance because changing the number of epochs could actually change the distribution of points in lower-dimensionality space. To choose the optimum number of epochs for our autoencoder, we trained the autoencoder for a large number of epochs, a thousand in our case, and found the optimal number of epochs based on the graph of the validation loss versus the number of epochs that we used to train the autoencoder. We divide our dataset into training and validation datasets which we then feed to the neural network. The training dataset is used for training and the validation dataset is used at the end of training to validate the results on a dataset that has not been seen by the neural network. Validation loss is the value of the loss of function calculated using the validation dataset at the end of each epoch. The graph is shown in figure 2.3. As it can be understood from the graph, the validation loss does not improve substantially after two hundred epochs. Figure 2.4 shows a close-up of the validation loss graph, where the number of epochs is two hundred. It can be understood from the graphs that the validation loss curve decreases much more slowly beyond two hundred epochs so we chose to train our autoencoders for two hundred epochs.

To uncover the inner workings of the autoencoder, we need to implement a method to figure out which features of the input vector are more important for the autoencoder and by how much. Machine learning and especially neural networks are often

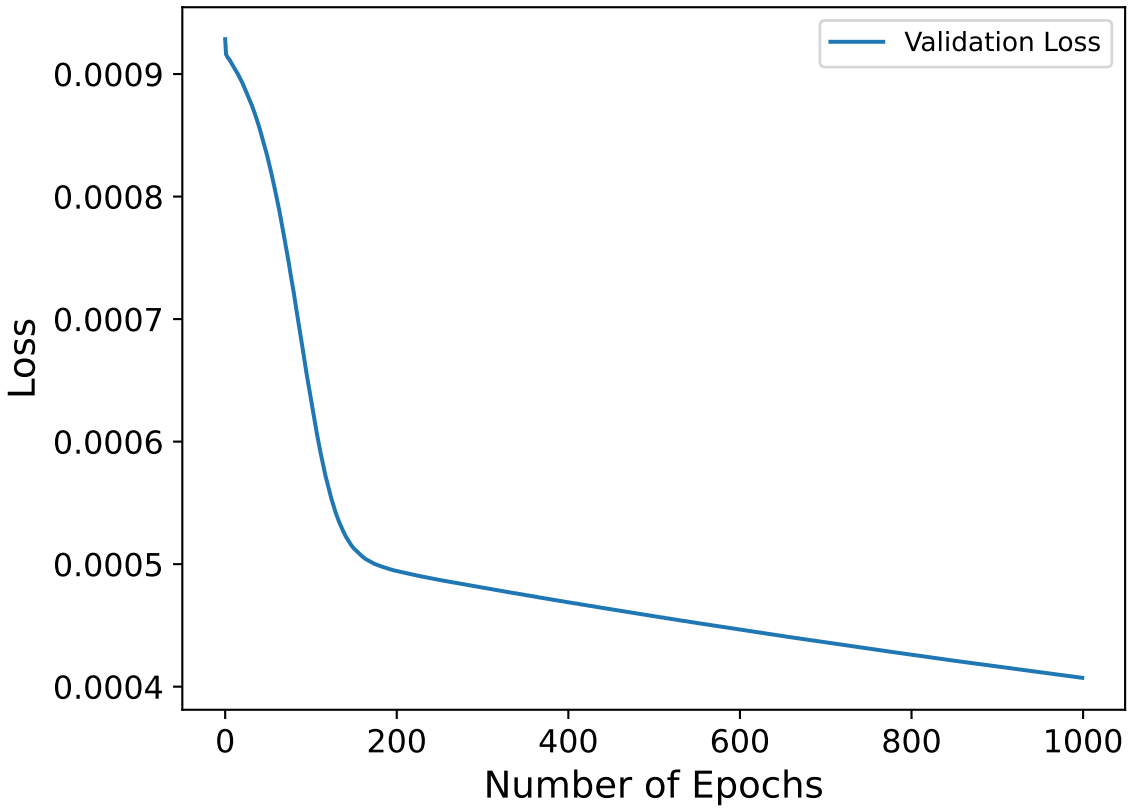


Figure 2.3: Validation loss vs number of epochs for 1000 epochs of training

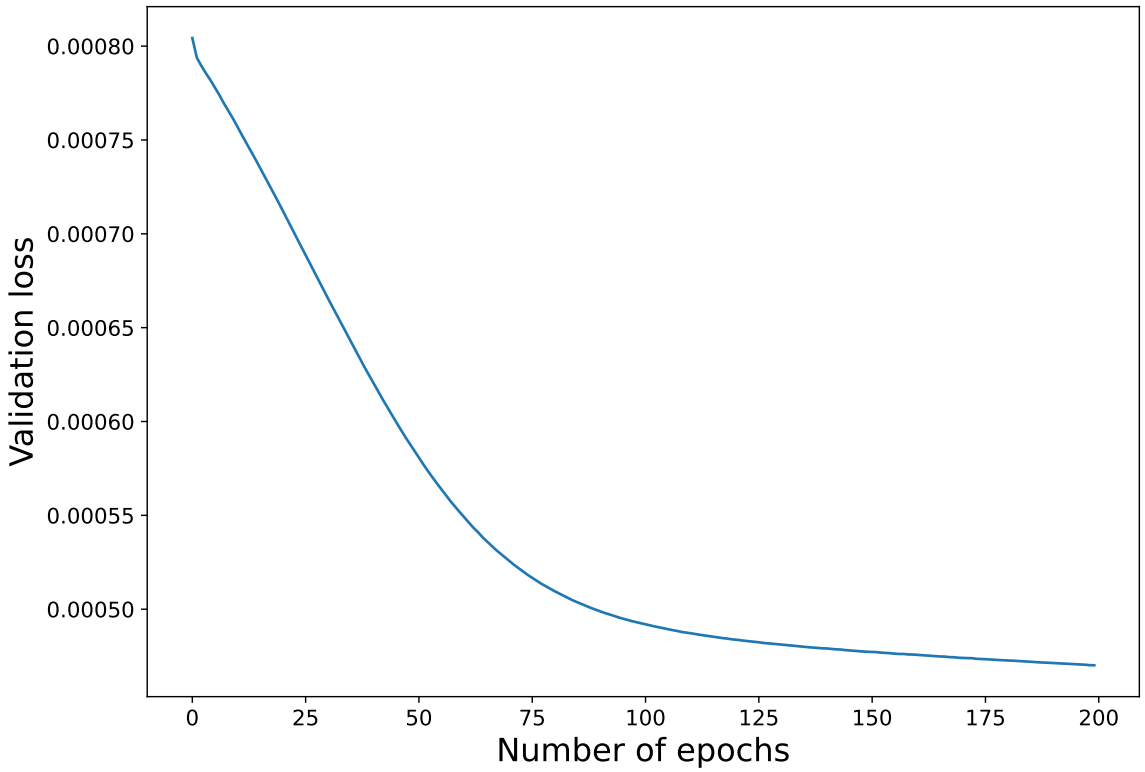


Figure 2.4: Validation loss versus the number of epochs for 200 epochs of training

regarded as blackboxes. This phrase means that they do something but it is difficult or sometimes impossible to understand how. In our case, the autoencoder reduces the dimensionality of the input vector from eight features to two important features which are the non-linear combination of input vector features with some weights and biases, and recreates the input vector with a very small error. But how? To find the answer to this question there is no straightforward method. Drawing the graph of the autoencoder or exporting the weights and biases would not give us any useful information. Instead, we use the input perturbation method introduced by Scardi et al. [17] to create the feature importance graph shown in Figure 2.5. The input perturbation method assesses the variation in the MSE of the autoencoder by adding, in turn, a small amount of white noise to the kth input while holding all the other inputs at their observed values. Here, we set the white noise to 10% and 50% of each input. The input variables that lead to the largest increase in the mean squared error are the ones that have the most relative influence. In both cases, a quantitative measure of the relative importance, RI_k, of the kth input can be obtained as

$$RI_k = \frac{\Delta E_k}{\sum_{j=1}^{d} \Delta E_j} \tag{2.15}$$

where ΔE_k is the variation in the MSE caused by the change applied to the kth input and the sum in the denominator runs over all the input variables. Figure 2.5 shows that $q6$ and $q8$ are the most important features of the input vector and other features of very low importance for this system of fcc and hcp crystals.

The number of Gaussian components in the mixture, N_G, is usually found by minimizing the Bayesian information criterion (BIC), which measures how well a GMM fits the observed data while penalizing models with many parameters to prevent overfitting. Figure 2.6 shows the value of BIC for the different number of clusters used

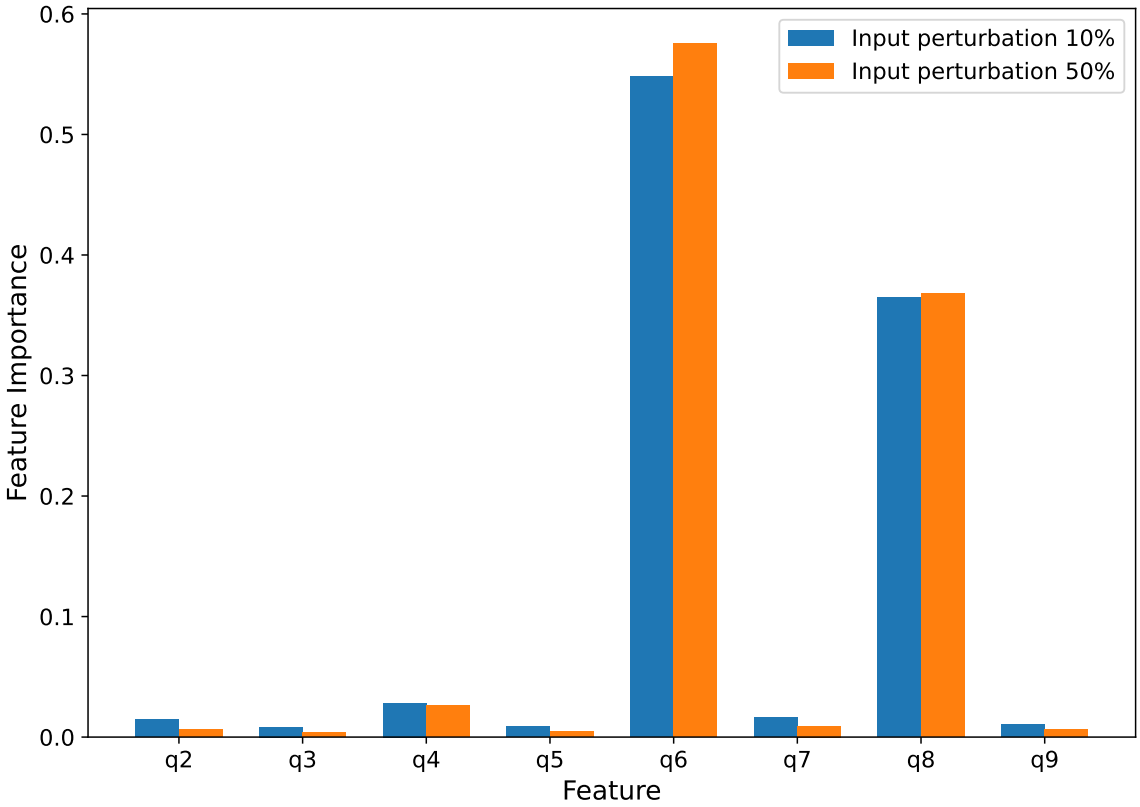


Figure 2.5: The graph of autoencoder feature importance calculated using the input perturbation method for 10% and 50% white noise

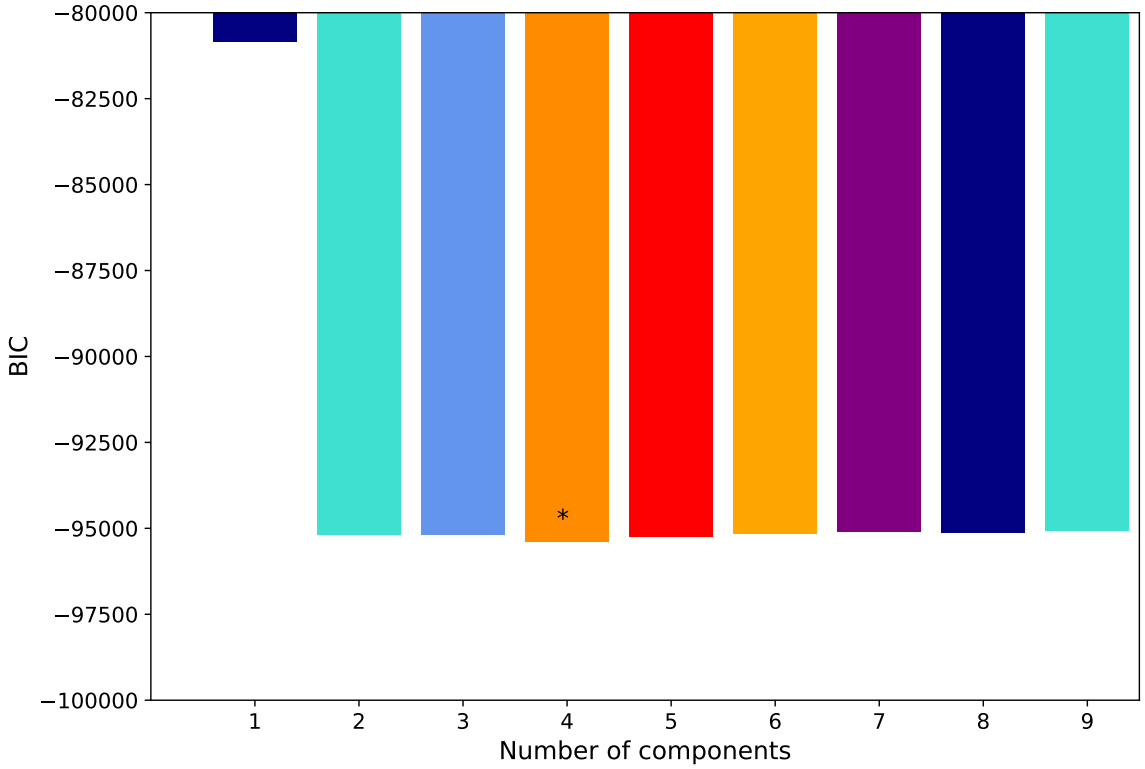


Figure 2.6: The graph of BIC values for the number of clusters from 1 to 9. The star denotes the number of components with the lowest BIC.

in the GMM. It is understood from the graph that the minimum BIC can be achieved by using four clusters for the number of components of the Gaussian mixture model which makes it the best number of clusters for our illustrative case.

Figure 2.7 shows two outputs of the encoder part of the autoencoder $y1$ and $y2$ plotted in the same graph. The colours represent the clusters distinguished by the GMM. The number of components for this GMM is set to four based on the results of the BIC minimization graph. While minimizing the BIC gives an estimate of the Gaussian distributions that reproduce our data points, it usually gives a number that is greater than an actual number of clusters present in our data points, as we clearly see from Fig 2.7. As pointed out in Ref. [10], this fact justifies the need for adding another piece to our algorithm that merges the redundant clusters.

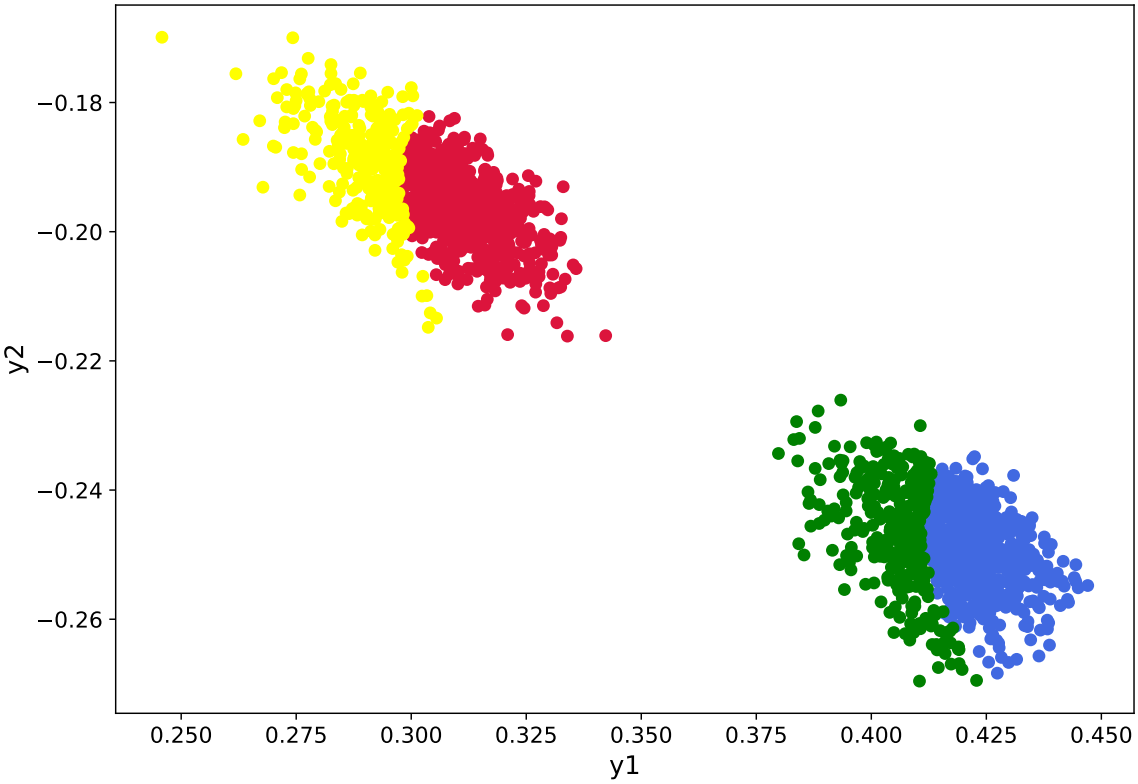


Figure 2.7: The graph of $y1$ versus $y2$ and the clusters. The number of components of the GMM is set to 4.

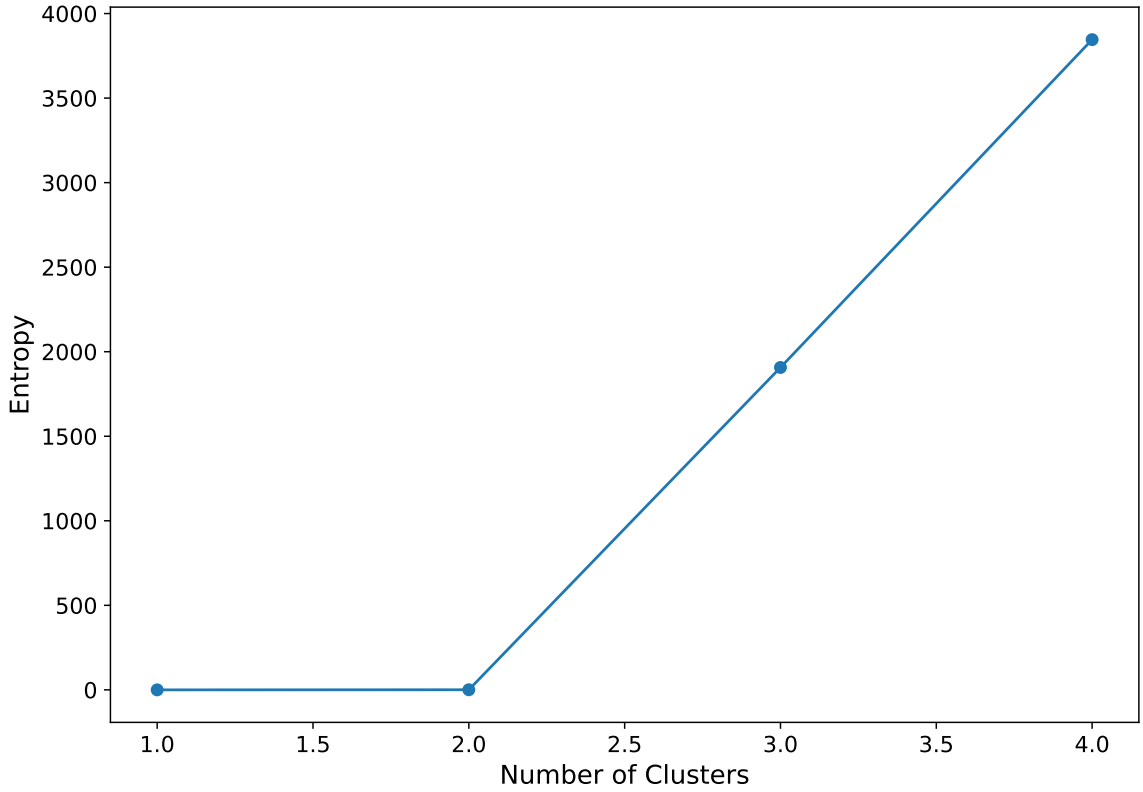


Figure 2.8: The graph of entropy of the GMM clusters after merging from 4 to 1 clusters. Two, the location of the elbow of the graph, is the best number of clusters.

The output of a trained GMM, p_{ij}, is the probability that the ith data point belongs to the jth component of the GMM. Using these probabilities, the value of entropy defined in equation 2.14 is calculated for each pair of clusters, and the pair of clusters that reduces the entropy by the greatest amount is merged. The graph of entropy versus the number of clusters is generated by repeating the merging procedure and the best number of clusters is found using the elbow method. Figure 2.8 shows that after merging four clusters to three and then to two clusters, the entropy is reduced to a value close to zero, and so two clusters is the best number of components for the GMM clustering algorithm.

Figure 2.9 shows the graph of $y1$ versus $y2$ with colours representing the clusters distinguished using a merged GMM. The number of components of the GMM is set

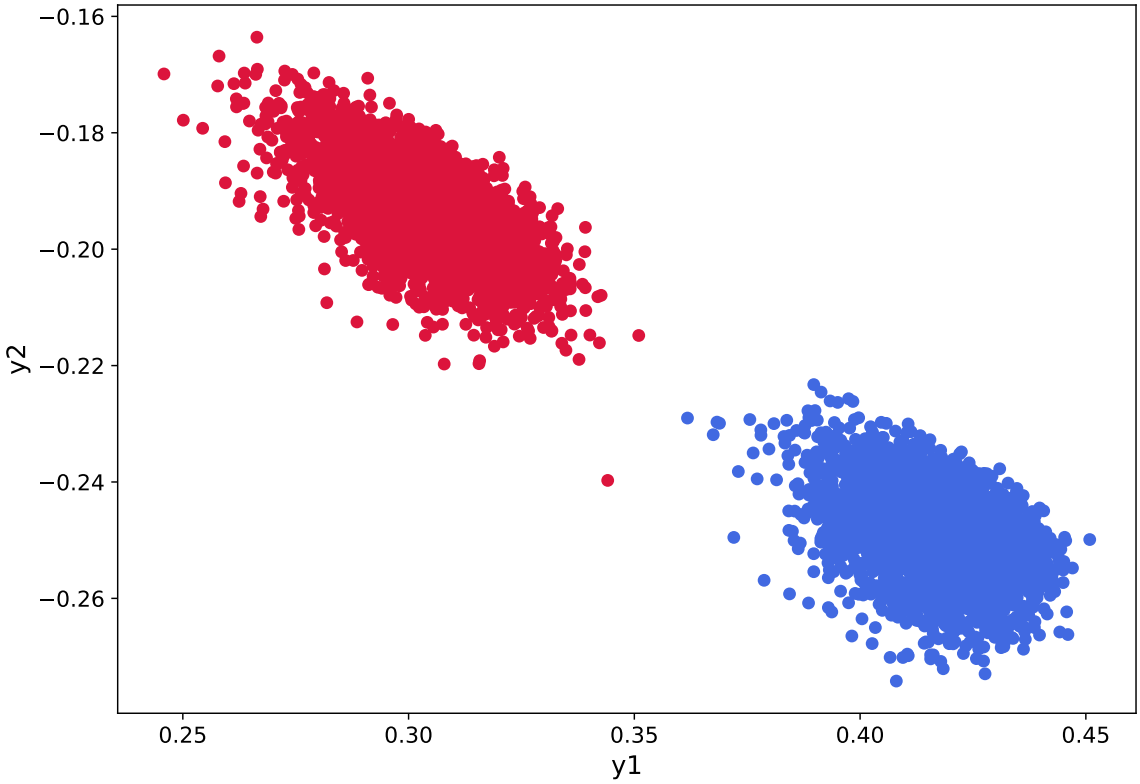


Figure 2.9: The graph of $y1$ versus $y2$ and the clusters. The number of components of the GMM is set to 2.

to two based on the output of the merging algorithm. The graph clearly shows that in the lower-dimensionality space generated by the autoencoder, the clusters that are identified by the GMM using the number of clusters that the merging algorithm finds, are correct.

Figure 2.10 shows a 3D representation of the two crystals. The colouring of the atoms is based on the labels that are the output of the Gaussian mixture model. Blue atoms are the atoms that belong to the first cluster and red are the atoms that belong to the second cluster. Figure 2.11 shows the cross-section of the crystals with colours describing the clusters of the atoms.

The graph of the probability distributions of the two most important BOPs ($q6$ and $q8$) of the two crystals is shown in Figure 2.12. The graph shows that the $q6$

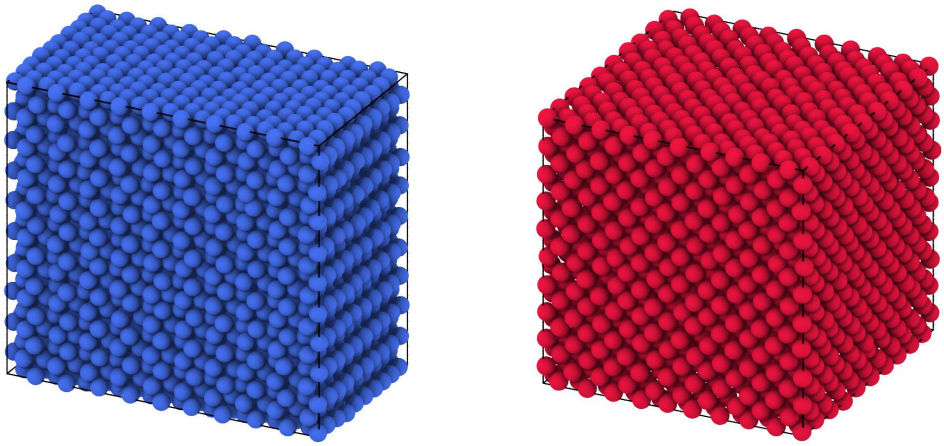


Figure 2.10: 3D representation of thermalized hcp (left) and thermalized fcc (right) crystals side by side. Colours identify different clusters found by the machine learning algorithm: blue for the first cluster and red for the second cluster.

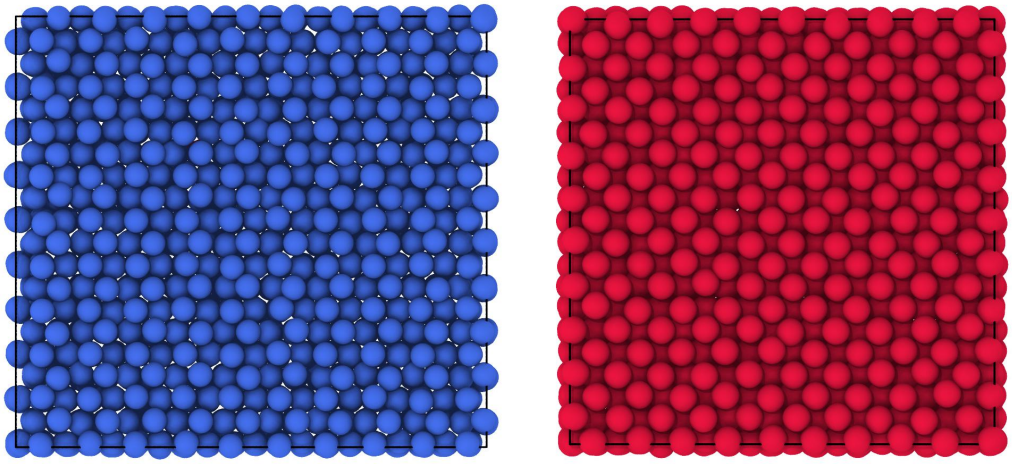


Figure 2.11: 2D representation of thermalized hcp (left) and thermalized fcc (right) crystals side by side. colours identify different clusters found by the machine learning algorithm: blue for the first cluster and red for the second cluster.

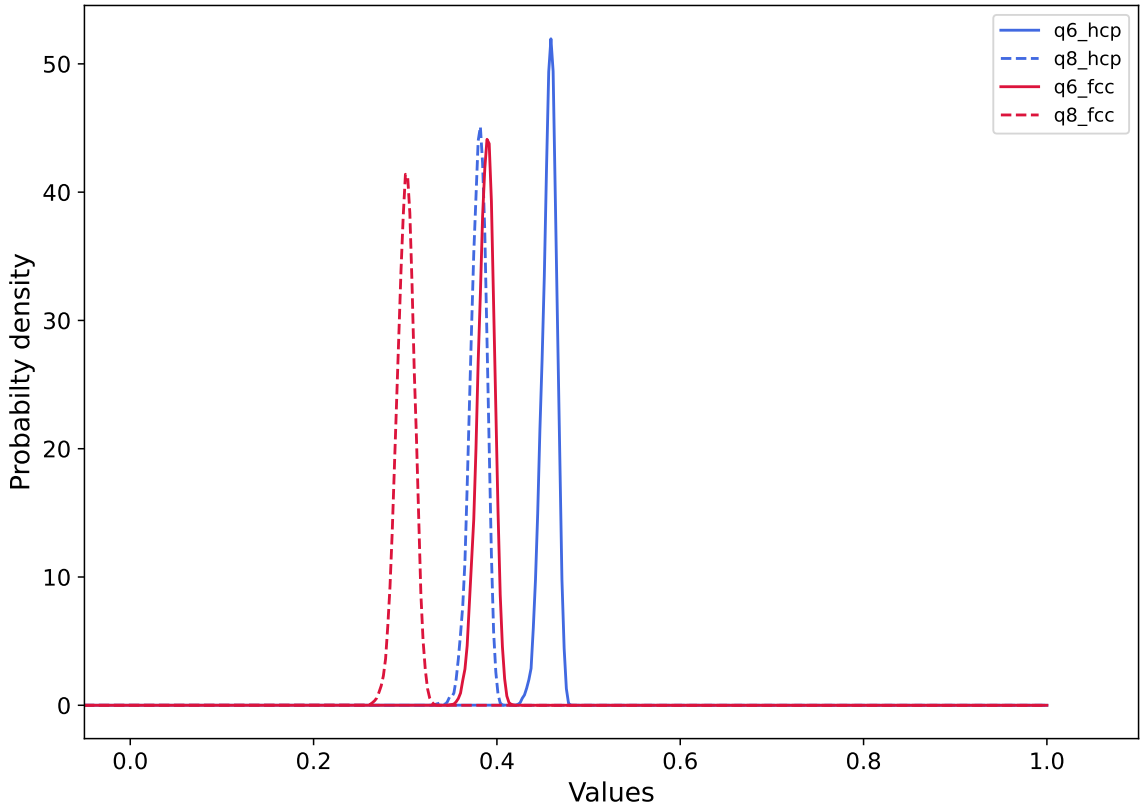


Figure 2.12: The graph of probability distribution of the $q6$ and $q8$ values of the hcp and fcc crystals

values of the hcp crystal (blue curves) peak at a greater value than the $q6$ values of the fcc crystal (red curves). The same is true for $q8$ values; the orange curve peaks at a greater value than the red curve.

Figure 2.13 shows the graph of $q6$ versus $q8$ for the simple illustrative case which only includes hcp and fcc clusters. As can be seen from the graph, the two crystal structures are clearly distinguished in terms of their q6 and q8 values. We shall see that such clean separation is not always achieved.

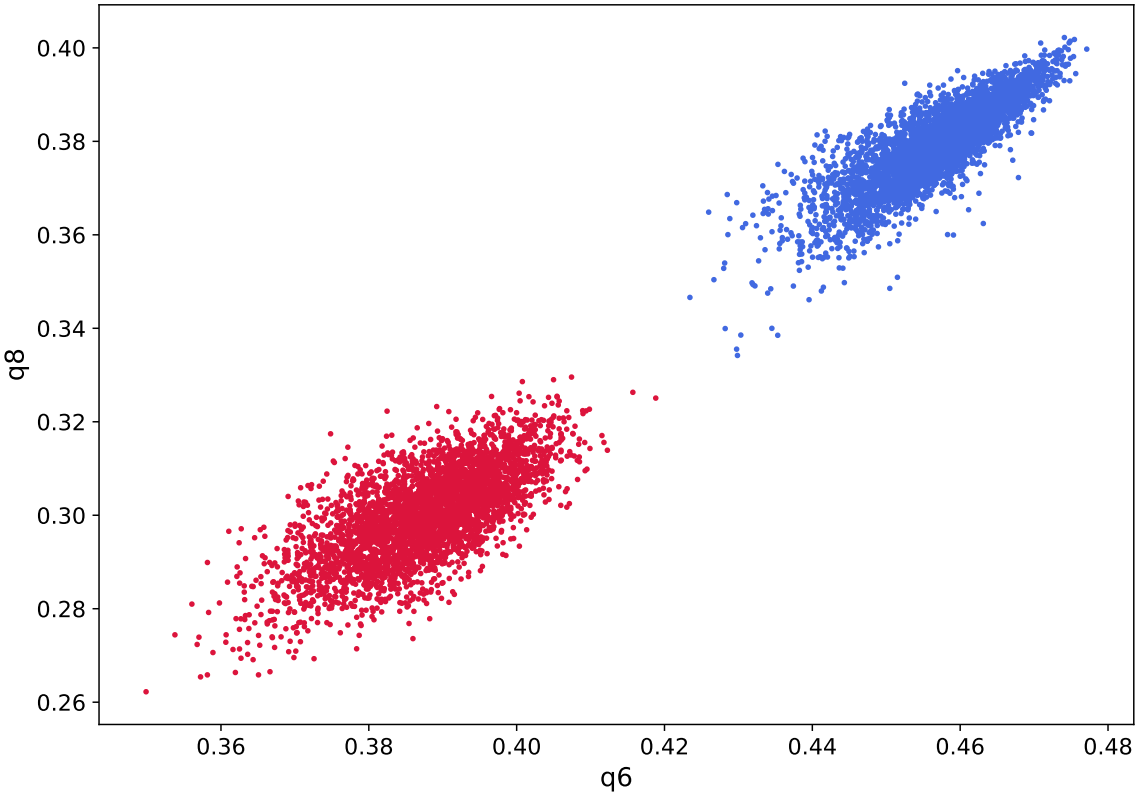


Figure 2.13: The graph of $q6$ versus $q8$. The different colours represent the clusters in the space of $y1$ and $y2$.

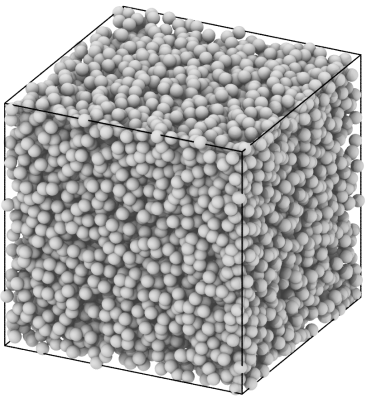
Chapter 3

Results

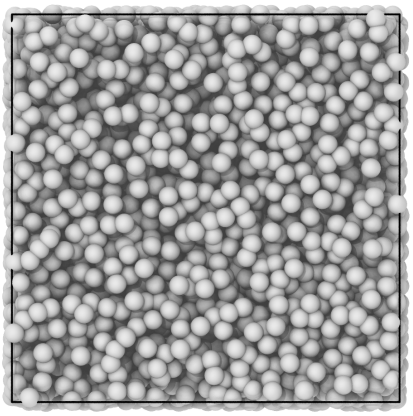
The results of MD simulation for the case of $\mu = 0$ are shown in Figure 3.1. As can be seen in the results for $\mu = 0$, $\epsilon = 0$, and $\epsilon = 1$, we see liquid-like structure. For the case of $\mu = 0$ and $\epsilon = 3.25$, we see that a denser structure forms as the system begins to phase separate. The simulation results for $\mu = 2$, which are shown in Figure 3.2, depict the formation of more ordered structures, especially for higher values of ϵ. Figure 3.3 which is for $\mu = 4$ and different values of ϵ shows the formation of dipolar chains and also the formation of body-centered tetragonal (bct) crystallites, which appear to become disrupted at $\epsilon = 3.25$.

The following results are obtained after applying the machine learning algorithm to the LAMMPS output. In particular, the machine learning algorithm is trained on the BOP values of each system separately and then the atoms in each system are used for the clustering process. In other words, the algorithm is applied to each state point separately; e.g. when analyzing the data from $\mu = 0$ and $\epsilon = 0$, no information from other values of μ and ϵ is used.

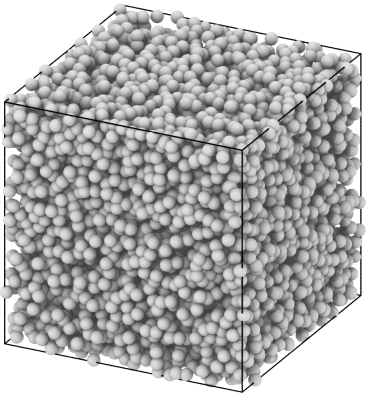
For the case of $\mu = 0$ and all values of the ϵ and for the case of $\mu = 2$ and all values



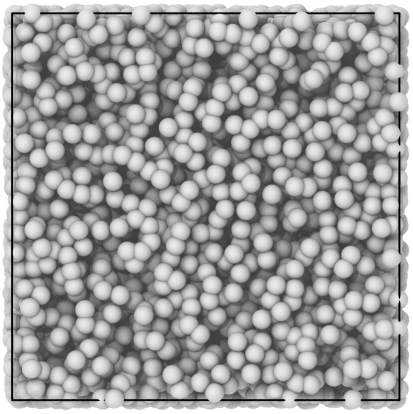
(a) $\mu = 0$, $\epsilon = 0$. View off axis.



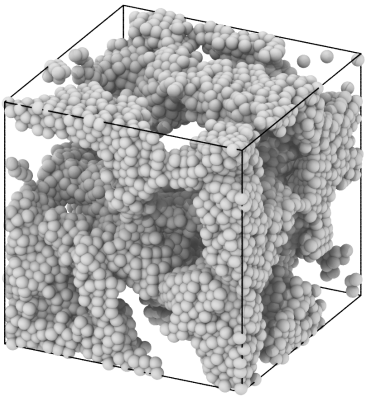
(b) $\mu = 0$, $\epsilon = 0$. View down the Z axis.



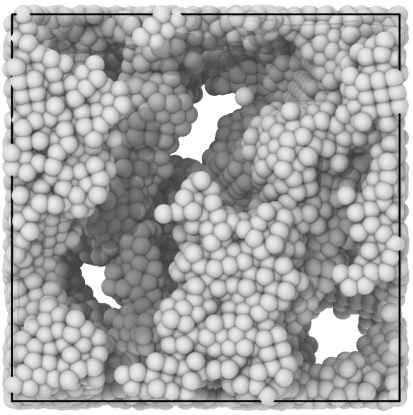
(c) $\mu = 0$, $\epsilon = 1$. View off axis.



(d) $\mu = 0$, $\epsilon = 1$. View down the Z axis.

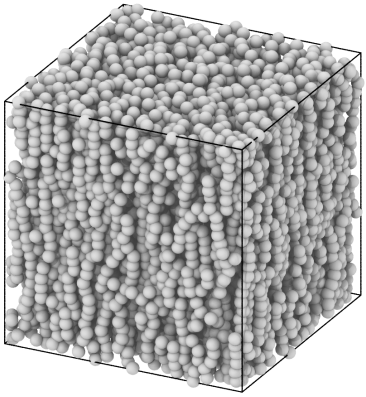


(e) $\mu = 0$, $\epsilon = 3.25$. View off axis.

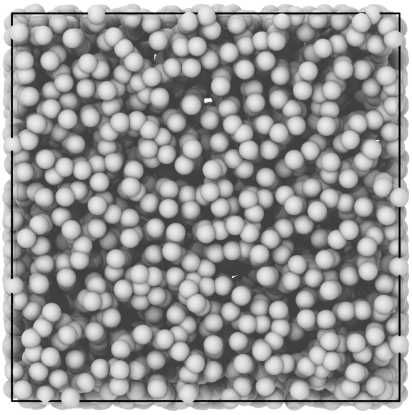


(f) $\mu = 0$, $\epsilon = 3.25$. View down the Z axis.

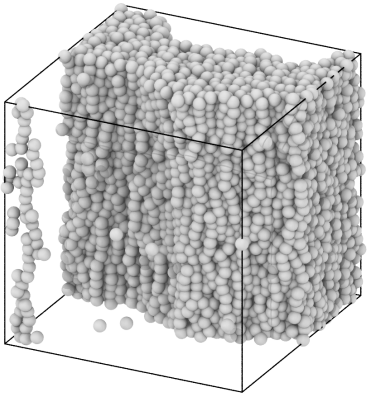
Figure 3.1: Simulation results for $\mu = 0$ and different values of ϵ.



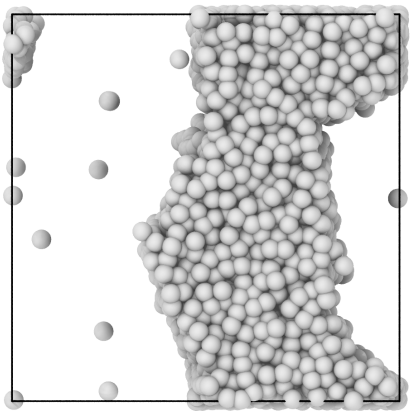
(a) $\mu = 2$, $\epsilon = 0$. View off axis.



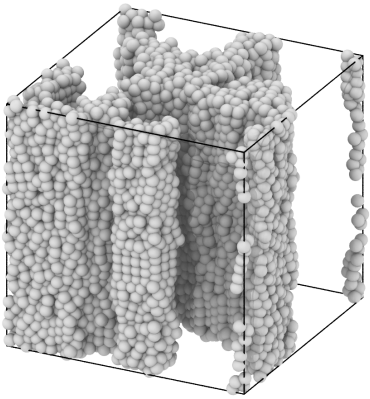
(b) $\mu = 2$, $\epsilon = 0$. View down the Z axis.



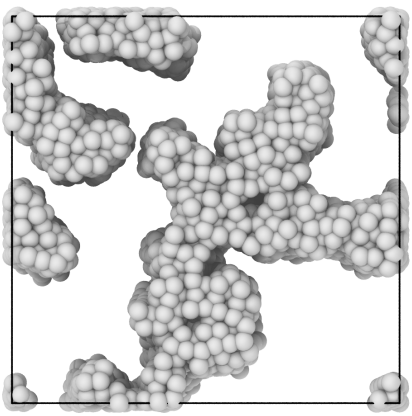
(c) $\mu = 2$, $\epsilon = 1$. View off axis.



(d) $\mu = 2$, $\epsilon = 1$. View down the Z axis.

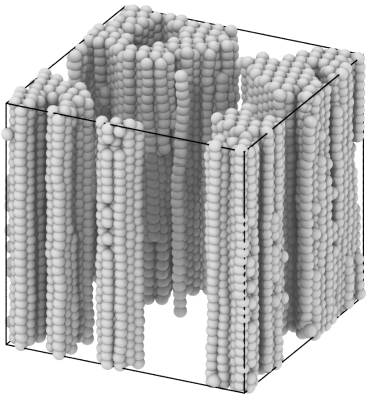


(e) $\mu = 2$, $\epsilon = 1$. View off axis.

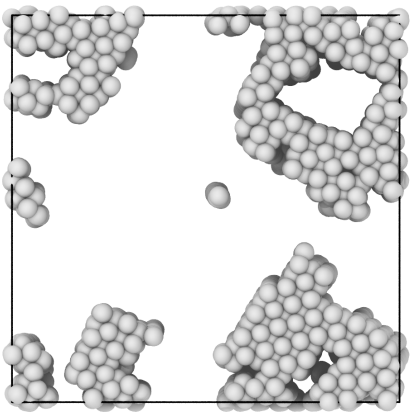


(f) $\mu = 2$, $\epsilon = 1$. View down the Z axis.

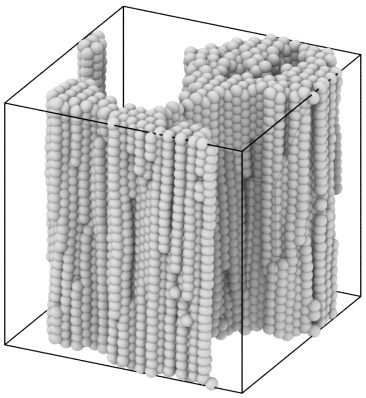
Figure 3.2: Simulation results for $\mu = 2$ and different values of ϵ.



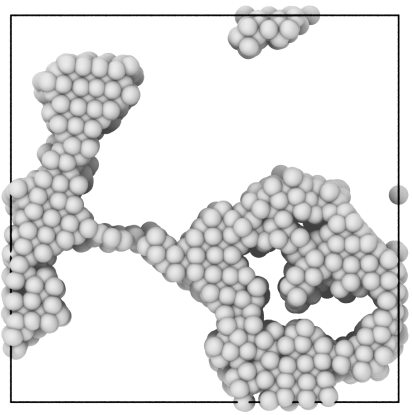
(a) $\mu = 4$, $\epsilon = 0$. View off axis.



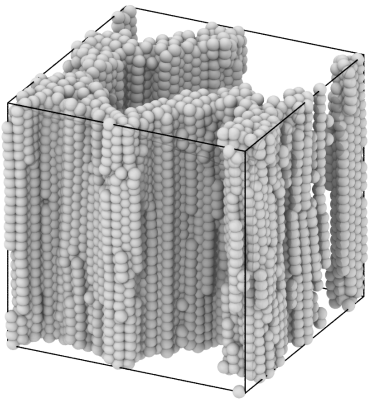
(b) $\mu = 4$, $\epsilon = 0$. View down the Z axis.



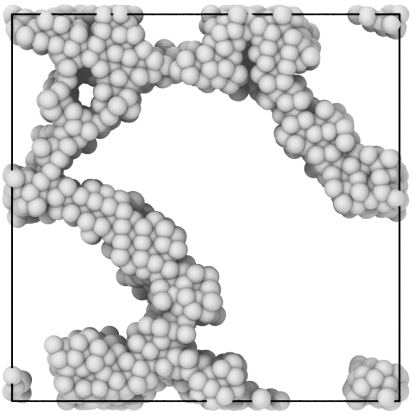
(c) $\mu = 4$, $\epsilon = 1$. View off axis.



(d) $\mu = 4$, $\epsilon = 1$. View down the Z axis.



(e) $\mu = 4$, $\epsilon = 3.25$. View off axis.



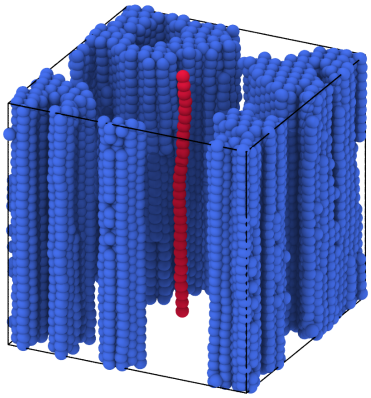
(f) $\mu = 4$, $\epsilon = 3.25$. View down the Z axis.

Figure 3.3: Simulation results for $\mu = 4$ and different values of ϵ.

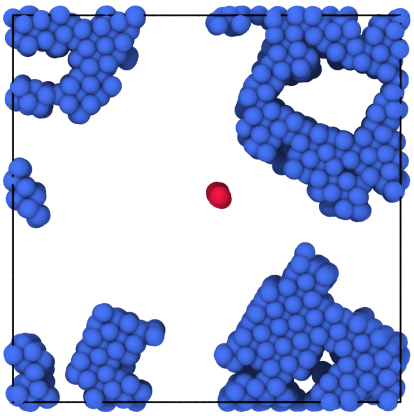
of the ϵ, only one single cluster type is identified in the systems meaning that only one type of structure is found. While the results might be reasonable for $\mu = 0$ and low values of the ϵ, for high values of ϵ and for the case of $\mu = 2$, one would expect the algorithm to detect new local structures. Figure 3.4 shows the results of applying the clustering algorithm on the systems with $\mu = 4$. In this figure, for the system with $\mu = 4$ and $\epsilon = 0$ and also $\epsilon = 1$ the clustering algorithm detects two types of atoms. The red cluster in figure 3.4a and 3.4b, clearly identifies a single separated chain. On the other hand, for the case of $\mu = 4$ and $\epsilon = 1$, the red cluster in 3.4c and 3.4d corresponds to a local structure that is not bct. While the result of the applying clustering algorithm on the mentioned cases looks reasonable, the clustering algorithm results for the case $\mu = 4$ and $\epsilon = 3.25$ does not agree with our expectations. Figure 3.4e and 3.4f show that the clustering algorithm detects only one type of structure for the $\mu = 4$ and $\epsilon = 3.25$ system, while we expect at least the two types of structures. Unfortunately, this system is only identified as having one type of structure, which is an unreasonable result.

To summarize, applying the machine learning algorithm on one state point at a time yields results that do not agree with our understanding of the system. We suspect that this might be due to the fact that the set of samples that we use to train our machine learning model is too small or too sparse, meaning that the training set is not a good representative of all the values that we want to identify using the clustering algorithm.

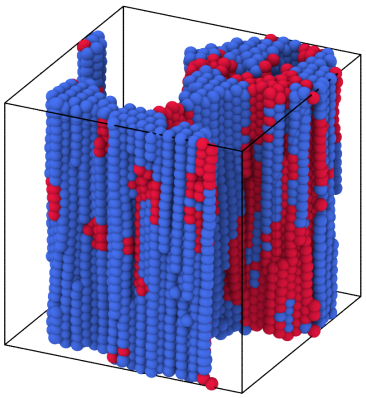
To solve this problem, we feed the algorithm data from all nine state points that we have created. This approach includes a sample from all the BOPs from all the local structures that we want to identify and so it is a good representation of all the state points. To create this dataset, we take the first octant of all the state points



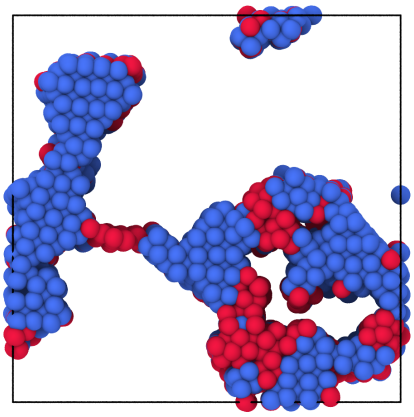
(a) $\mu = 4$, $\epsilon = 0$. View off axis



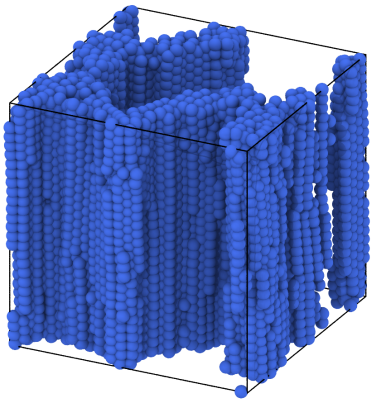
(b) $\mu = 4$, $\epsilon = 0$. View down the Z axis



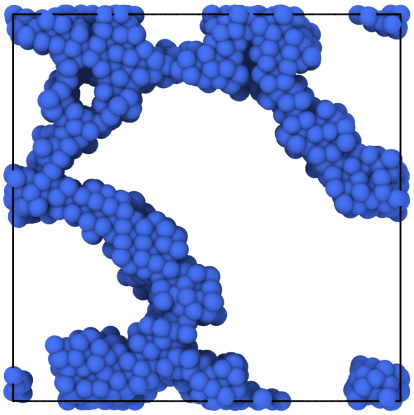
(c) $\mu = 4$, $\epsilon = 1$. View off axis



(d) $\mu = 4$, $\epsilon = 1$. View down the Z axis



(e) $\mu = 4$, $\epsilon = 3.25$. View off axis



(f) $\mu = 4$, $\epsilon = 3.25$. View down the Z axis

Figure 3.4: The results of the clustering algorithm for $\mu = 4$ and different values of ϵ

and use it as input to the machine learning algorithm.

Figure 3.5 shows the result of applying the Gaussian mixture model on the output of the autoencoder for this combined data set. The number of clusters of the Gaussian mixture model is set to seven, which is the number of clusters that minimizes the BIC. Applying the next step of the algorithm (cluster merging process) on the combined system gives five clusters as the optimal number of clusters for this system. Figure 3.6 shows the result of the cluster merging process.

Using the labels obtained from the merging process, we visualize the combined system (all the octants). Figures 3.7 and 3.8 shows the system from different angles. As can be seen in the figures, each cluster of atoms clearly identifies a different local structure. The majority of the ($\mu = 0$, $\epsilon = 0$), ($\mu = 0$, $\epsilon = 1$), ($\mu = 2$, $\epsilon = 0$) and ($\mu = 2$, $\epsilon = 1$) state points consist of the blue cluster, which represents fluid-like structures at low values of μ and ϵ. The gray cluster shows up in the ($\mu = 2$, $\epsilon = 1$) state point for the first time and consists of particles with relatively high local density and some sort of local non-crystalline order like a dense liquid. The red cluster dominates the systems with high values of μ and represents the bct structures that are formed at high μ. The green structure is also present in state points with a high value of μ and especially a high value of ϵ, and is representative of the crystalline structures other than bct, such as fcc and hcp. Finally, the purple structure is formed from sheet-like structures with local orders similar to bct but in sheet form.

Figure 3.9 shows the graph of $q6$ versus $q8$ for the combined system. The comparison of figure 3.9 and figure 3.6 leads to an interesting conclusion. While one might assume that using $q6$ and $q8$ as the input for clustering and then the merging algorithm might yield the same results, the graph of $q6$ versus $q8$ shows that the outputs of the autoencoder, $y1$, and $y2$ do not only represent $q6$ and $q8$ because one of the

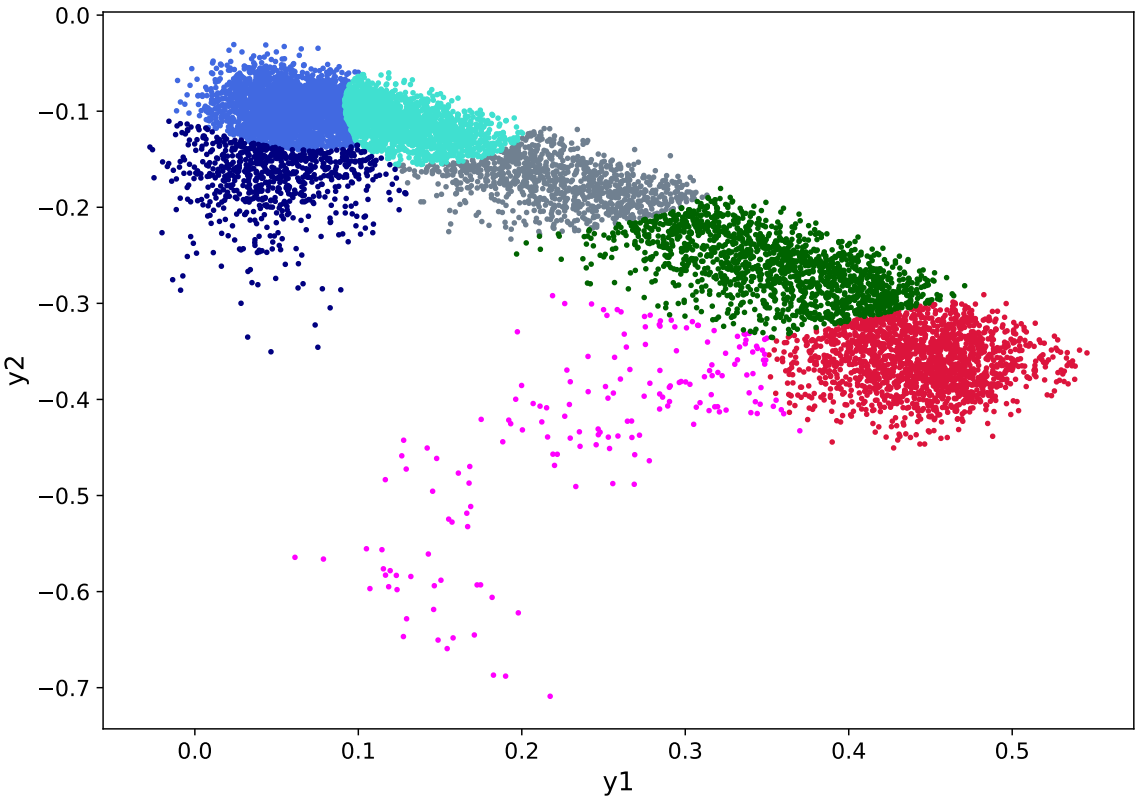


Figure 3.5: The scatter plot of $y1$ versus $y2$. colours identify the clusters detected by the Gaussian mixture model

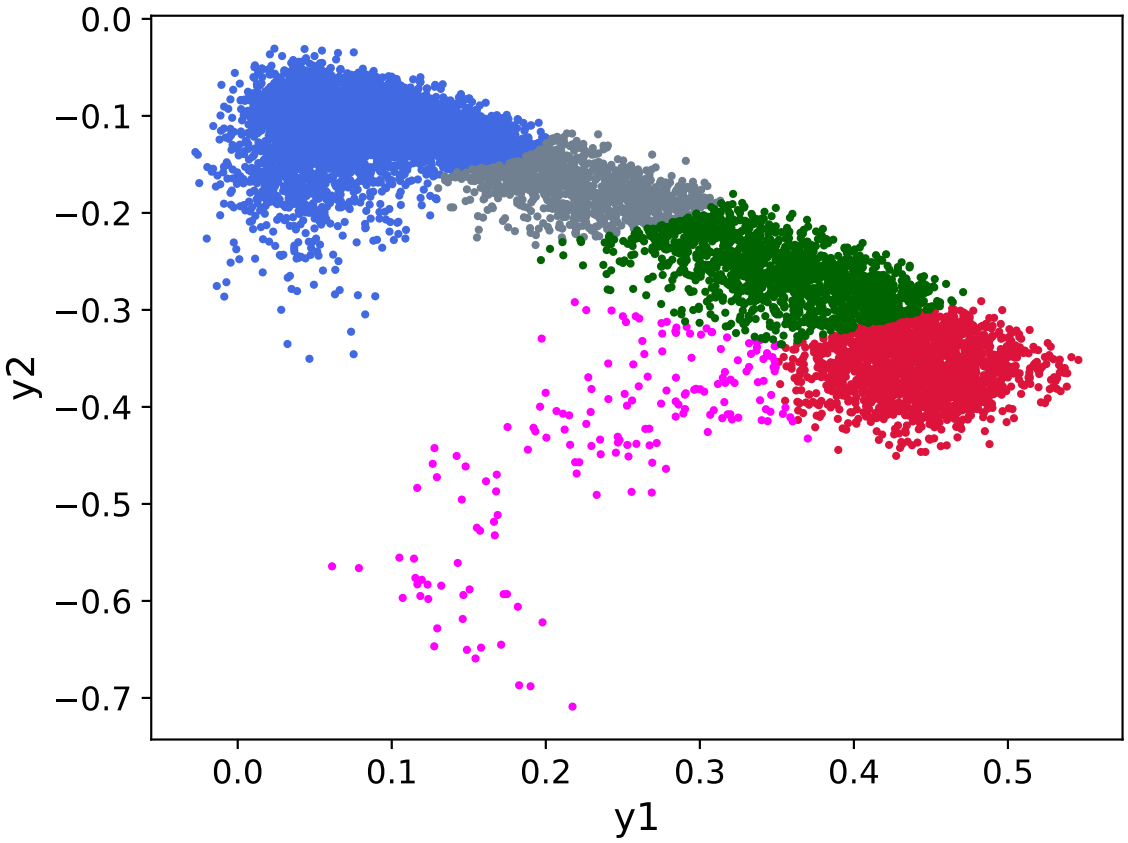


Figure 3.6: The scatter plot of $y1$ versus $y2$. colours identify the clusters after the cluster merging process

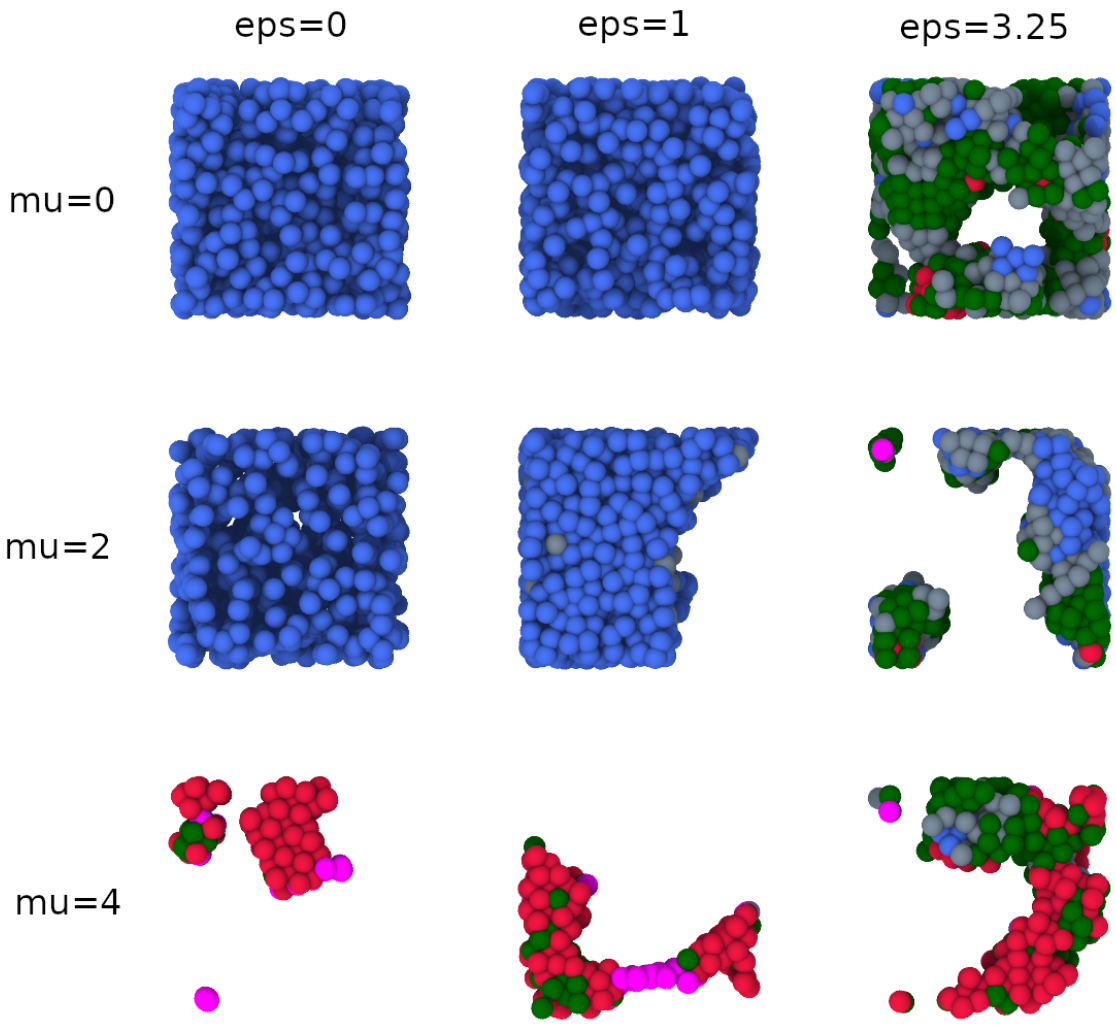


Figure 3.7: The combined system. colours represent clusters. View down the Z axis

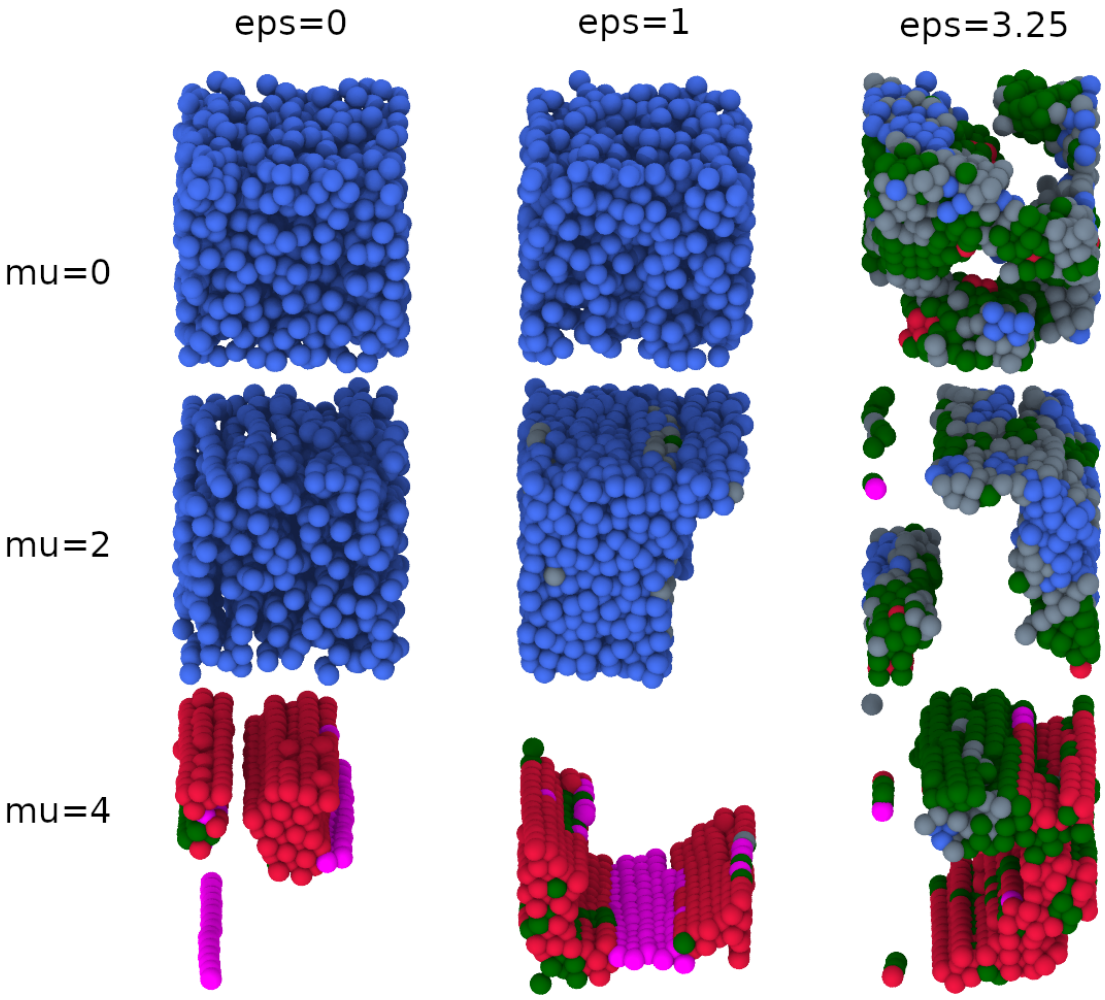


Figure 3.8: The combined system. colours represent clusters. View off axis

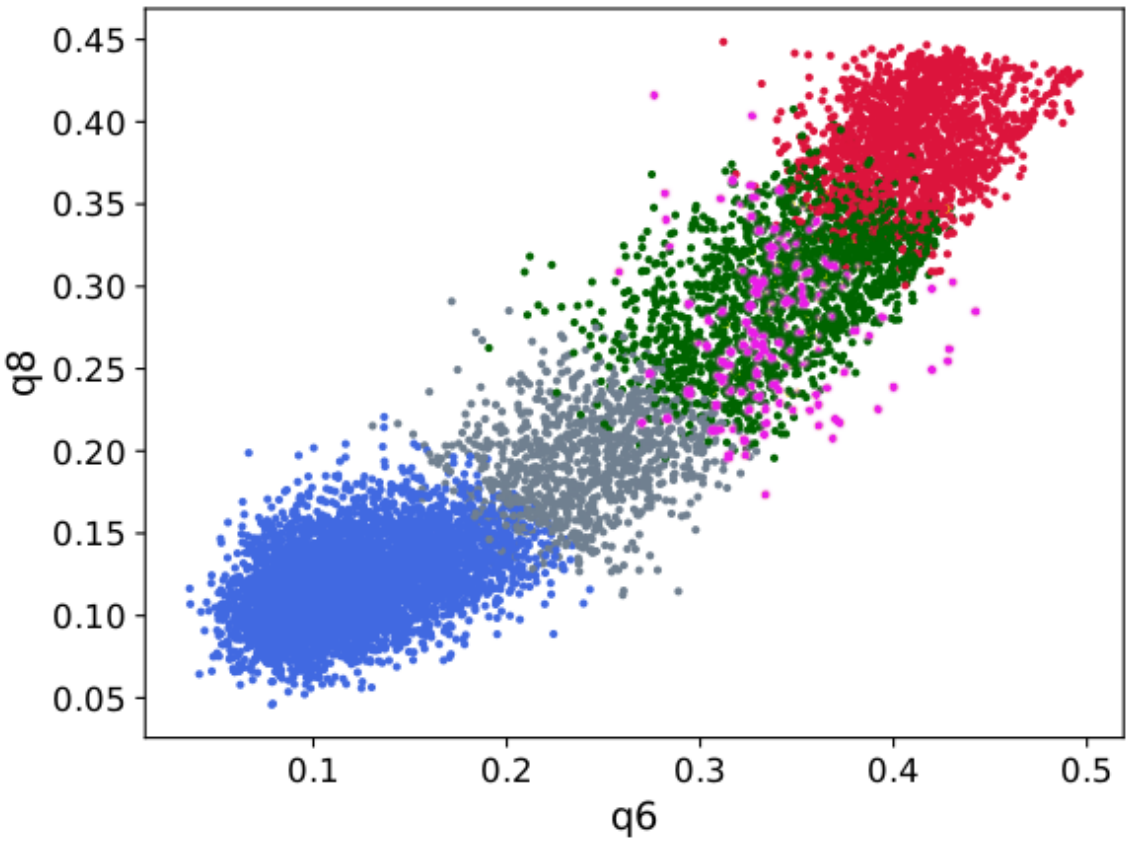


Figure 3.9: The graph of $q6$ versus $q8$ for the combined system. The colours represent different clusters

clusters that are identified from the graph $y1$ and $y2$ (the pink cluster) is completely unidentifiable in the graph of $q6$ and $q8$. This means that even though $y1$ and $y2$ depend on $q6$ and $q8$ as their most important components, they are a non-linear combination of all eight features.

Chapter 4

Discussion and Future Work

As discussed in the results section, the results of applying the machine learning algorithm on dipolar-depletion systems vary with changing the training set. While using a single state point BOP values as the training set, for the case of $\mu = 0$ and $\mu = 2$ leads to reasonable results, the results for the state points with $\mu = 4$ and different ϵ do not agree with our understanding of the physics of the systems. Checking multiple data sets for the training of the machine learning algorithm showed that the best results are achieved by using a combination of all state points for the training of the machine learning algorithm. We created this data set by attaching the first octants of all state points together. Using the combined system as the training set yields the most promising results in this research. The mentioned results are in good agreement with our understanding of the system.

This algorithm helps us answer important questions about the nature of the dipolar-depletion systems. For example, to simulate the dipolar depletion system with $\mu = 2$, and $\epsilon = 3.25$, two possible methods can be adopted. In the first method, we create a simple-cubic crystal and apply both dipole-dipole interactions ($\mu = 2$)

and short-range depletion interactions ($\epsilon = 3.25$) on the system simultaneously from the beginning of the simulation and let it run until the system reaches equilibrium. In the second method, we create the simple-cubic crystal, apply $\mu = 2$ and let it run until the system reaches equilibrium and then restart the simulation with the value of $\epsilon = 3.25$ applied until the system reaches the equilibrium again. The question is do we get different results as the result of these two methods? In other words, do the simulation results depend on the path or are they path-independent? The machine learning algorithm allows us to answer this question by identifying the different structures in the final states of the simulations. Preliminary analysis shows that the final configurations of both simulations have the same distribution of structures. While this suggests that the results are path independent, we leave a detailed study for future work.

Although the results of applying the machine learning algorithm on dipolar-depletion systems in this research are promising, there is always room for improvement. As a general rule, we know that in machine learning methods bigger training data sets lead to better results. In our research, since the results of training the model on only one state point were inconsistent and unacceptable we can reasonably conclude that we need bigger, more comprehensive data sets to improve the accuracy of our machine learning algorithm results. To achieve this goal, we can simulate multiple other state points and train the machine learning algorithm while including those state points to get better results.

While we only used an octant of each state-point and only eight bond order parameters to create our input vector, we realized that the amount of needed computational power needed was already too high for a laptop and training the machine learning

algorithm was taking too long. As a result, adding more features and using bigger systems will only be possible if we are using a system with high computational power. Additionally, there are various other features that can be extracted from the simulation results. While we only extract eight bond order parameters from the simulation results and feed them as the input of the machine learning algorithm, there are dozens of other features such as Voronoi volume, symmetry features, etc. that can be extracted in the feature extraction process to improve the clustering algorithm results.

In my view, the main ambiguity of this machine learning algorithm is the fact that even though for some state points we know what local structure each cluster of particles corresponds to, for other state points we can only guess. I believe the best solution for this problem is to use supervised learning in addition to unsupervised learning. While in the unsupervised machine learning methods the dataset does not have any labels, in supervised learning machine learning methods each observation has an identifying label that specifies the category that the observation belongs to (in our case different types of crystals). The machine learning model is trained on the labeled dataset. The trained model predicts the labels of unseen test datasets. Using supervised learning techniques will give the label for each cluster and the probability that each atom belongs to each cluster, which is what we eventually want as the result.

Appendix A

Source Codes

A.1 Bond Order Parameters

```
import pyscal.traj_process as ptp
import pyscal as pc

trajfile = "thermalized_hcp.xyz"
files = ptp.split_trajectory(trajfile)
part_sys1 = pc.System()
part_sys1.read_inputfile(files[1], format="lammps-dump")
part_sys1.find_neighbors(method='cutoff', cutoff=cutoff)
part_sys1.calculate_q(qs, averaged=True)
q_hcp = part_sys1.get_qvals(qs, averaged=True)

trajfile = "thermalized_fcc.xyz"
files = ptp.split_trajectory(trajfile)
```

```
part_sys2 = pc.System()
part_sys2.read_inputfile(files[1], format="lammps-dump")
part_sys2.find_neighbors(method='cutoff', cutoff=cutoff)
part_sys2.calculate_q(qs, averaged=True)
q_fcc = part_sys2.get_qvals(qs, averaged=True)
```

A.2 Autoencoder

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import backend as K

lam = 1e-5

def contractive_loss(y_pred, y_true):
    mse = K.mean(K.square(y_true - y_pred), axis=1)

    W1 = stacked_encoder.get_layer('encoder_l1').weights[0]
    W1 = K.transpose(W1)

    W2 = stacked_encoder.get_layer('encoder_l2').weights[0]
    W2 = K.transpose(W2)

    W3 = stacked_encoder.get_layer('encoder_l3').weights[0]
    W3 = K.transpose(W3)
```

```
    W4 = stacked_decoder.get_layer('decoder_l1').weights[0]
    W4 = K.transpose(W4)

    W5 = stacked_decoder.get_layer('decoder_l2').weights[0]
    W5 = K.transpose(W5)

    contractive = K.sum(lam * W1**2, axis=(0,1)) + \
                  K.sum(lam * W2**2, axis=(0,1)) + \
                  K.sum(lam * W3**2, axis=(0,1)) + \
                  K.sum(lam * W4**2, axis=(0,1)) + \
                  K.sum(lam * W5**2, axis=(0,1))
    return mse + contractive

stacked_encoder = keras.models.Sequential([
    keras.layers.Dense(8, name='encoder_l1', input_shape=[8]),
    keras.layers.Dense(80, activation="tanh",
                       name='encoder_l2',
                       kernel_initializer="glorot_normal"),
    keras.layers.Dense(2, activation='linear',
                       name='encoder_l3',
                       kernel_initializer="glorot_normal"),
])
stacked_decoder = keras.models.Sequential([
    keras.layers.Dense(80, activation="tanh",
                       name='decoder_l1',
                       input_shape=[2],
```

```
                        kernel_initializer="glorot_normal"),
    keras.layers.Dense(8, activation='linear',
                        name='decoder_l2',
                        kernel_initializer="glorot_normal"),
])
stacked_ae = keras.models.Sequential([stacked_encoder,
                                      stacked_decoder])
stacked_ae.compile(loss=contractive_loss,
                   optimizer=keras.optimizers.SGD(learning_rate=0.01,
                                                  momentum=0.9),
                   metrics=['mean_squared_error'])

history = stacked_ae.fit(X_train,
                         X_train,
                         epochs=200,
                         batch_size=128,
                         validation_data=(X_test, X_test))
```

A.3 Autoencoder Feature Importance

```
import matplotlib.pyplot as plt

params = [0.1, 0.5]
plt.figure(figsize=(10,7))
width= 0.35
ax = plt.gca()
```

```
origin_mse = np.mean((K.mean(K.square(X - stacked_ae(X)),
                     axis=1)).numpy())
for j ,param in enumerate(params):
    importances = []
    for i in range(X.shape[1]):
        my_rand = np.random.uniform(low=0,
                                    high=param,
                                    size=(X.shape[0], 1))
        new_col = np.multiply(my_rand, X[:, i].reshape(-1,1))
        purt_X = X.copy()
        purt_X[:, i] = purt_X[:,i] + new_col.flatten()
        mse = K.mean(K.square(purt_X - stacked_ae(X)), axis=1)
        importances.append(abs(np.mean(mse.numpy() - origin_mse)))
    RI = importances/np.sum(importances)
    plt.bar([j*width+x for x in range(2, len(RI)+2)], RI,
            width=width,
            label=f'Input purturbation {str(int(param*100))}%')

ax.set_xticks([x+width/2 for x in range(2,len(importances)+2)])
ax.set_xticklabels(['q2', 'q3', 'q4', 'q5', 'q6', 'q7', 'q8', 'q9'],
                   fontsize=12)

ax.tick_params(axis='both', which='major', labelsize=12)
ax.set_xlabel("Feature", fontsize=14)
ax.set_ylabel("Feature Importance",
```

```
                fontsize=14)
ax.legend(fontsize=12)
plt.show()
```

A.4 Gaussian Mixture Model

```
# training gaussian mixture model
import pandas as pd
from sklearn.mixture import GaussianMixture

plt.figure(figsize=(10,7))
gmm = GaussianMixture(n_components=n_clusters,
                      covariance_type='full',
                      random_state=42)
gmm.fit(X_hat)

#predictions from gmm
labels = gmm.predict(X_hat_test)
df = pd.DataFrame(X_hat_test)
df['cluster'] = labels
df.columns = ['y1', 'y2', 'cluster']

for k in range(0, n_clusters):
    data = df[df["cluster"] == k]
    plt.scatter(data["y1"], data["y2"], s=4, color=colors[k])
```

```
plt.xlabel("y1")
plt.ylabel("y2")
plt.show()
```

A.5 Merging Algorithm

```
from partycls.helpers import merge_clusters, _compute_ICL_ent
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 7))

# Use weights from GMM to merge the clusters into `n_cluster_min`
# this returns new weights and new labels
weights = gmm.predict_proba(X_hat_test)
weights[np.where(weights < 1e-13)] = 1e-10
s_ls = []
nrange = range(n_clusters, 0, -1)
for i in nrange:
    new_weights, new_labels = merge_clusters(weights,
                                             n_clusters_min=i)
    new_weights[np.where(new_weights < 1e-13)] = 1e-10
    ent = _compute_ICL_ent(new_weights, 1e-15)
    s_ls.append(ent)

s_ls = np.array(s_ls)
```

```
plt.scatter(nrange, s_ls)
plt.plot(nrange, s_ls)
plt.xlabel("Number of Clusters")
plt.ylabel("Entropy")
plt.show()
```

A.6 Particle System Visualization using Ovito

```
try:
    from ovito.io import import_file
except ImportError:
    print('install ovito to display the particles')

from ovito.vis import Viewport, TachyonRenderer
import os
import tempfile
from IPython.display import Image
from matplotlib import colors as cls

view='top'
size = (1024, 1024)
outfile = 'g7.png'

box_size = []
box_size.append(part_sys.box[0][0])
box_size.append(part_sys.box[1][1])
```

```
box_size.append(part_sys.box[2][2])

# Get a temporary file to write the sample
fh = tempfile.NamedTemporaryFile('w', suffix='.xyz', delete=False)
tmp_file = fh.name

fh.write('{}\n'.format(len(part_sys.atoms)))
fh.write('Properties=species:S:1:pos:R:3:radius:R:1:color:R:3\
          Lattice="{},0.,0.,0.,{},0.,0.,0.,{}"\n'.format(*box_size))
for i, p in enumerate(part_sys.atoms):
    fh.write('{} {} {} {} {} {} {} {} \
              \n'.format(1, *p.pos, 0.7,
              *cls.to_rgb(colors[labels[i]])))
fh.close()

# Ovito stuff. Can be customized by client code.
pipeline = import_file(tmp_file)
pipeline.add_to_scene()
cell_vis = pipeline.source.data.cell.vis
cell_vis.line_width = 0.

views = {   'top':    Viewport.Type.Top,
            'bottom': Viewport.Type.Bottom,
            'front':  Viewport.Type.Front,
            'back':   Viewport.Type.Back,
            'left':   Viewport.Type.Left,
```

```
            'right':  Viewport.Type.Right}
vp_type = views[view]
vp = Viewport(type=vp_type)
vp.zoom_all()

vp.render_image(filename=outfile,
                size=size,
                renderer=TachyonRenderer())

# Scene is a singleton, so we must clear it
pipeline.remove_from_scene()
del pipeline

# remove temporary exyz file
os.remove(tmp_file)
```

Bibliography

[1] Aidan P. Thompson, H. Metin Aktulga, Richard Berger, Dan S. Bolintineanu, W. Michael Brown, Paul S. Crozier, Pieter J. in 't Veld, Axel Kohlmeyer, Stan G. Moore, Trung Dac Nguyen, Ray Shan, Mark J. Stevens, Julien Tranchida, Christian Trott, and Steven J. Plimpton. LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales. *Computer Physics Communications*, 271:108171, 2022.

[2] Henk N.W. Lekkerkerker and Remco Tuinier. *Colloids and the Depletion Interaction*, volume 833 of *Lecture Notes in Physics*. Springer Netherlands, 2011.

[3] Sho Asakura and Fumio Oosawa. On interaction between two bodies immersed in a solution of macromolecules. *The Journal of Chemical Physics*, 22(7):1255–1256, 1954.

[4] Shivani Semwal, Cassandra Clowe-Coish, Ivan Saika-Voivod, and Anand Yethiraj. Tunable colloids with dipolar and depletion interactions: Toward field-switchable crystals and gels. *Phys. Rev. X*, 12:041021, Nov 2022.

[5] Kun Zhou and Bo Liu. *Molecular Dynamics Simulation*. Elsevier Inc., 2022.

[6] Cassandra Clowe-Coish. *Computer Simulation of Dipolar-Depletion Colloids*. Honour's thesis, Memorial University of Newfoundland, 2022.

[7] Xipeng Wang, Simón Ramírez-Hinestrosa, Jure Dobnikar, and Daan Frenkel. The lennard-jones potential: when (not) to use it. *Physical Chemistry Chemical Physics*, 22(19):10624–10633, 2020.

[8] Wolfgang Lechner and Christoph Dellago. Accurate determination of crystal structures based on averaged local bond order parameters. *The Journal of Chemical Physics*, 129(11):114707, 2008.

[9] Jacobus A. van Meel, Laura Filion, Chantal Valeriani, and Daan Frenkel. A parameter-free, solid-angle based, nearest-neighbor algorithm. *The Journal of Chemical Physics*, 136(23):234107, 2012.

[10] Emanuele Boattini, Marjolein Dijkstra, and Laura Filion. Unsupervised learning for local structure detection in colloidal systems. *The Journal of Chemical Physics*, 151(15):154901, 2019.

[11] Jean-Patrick Baudry, Adrian E. Raftery, Gilles Celeux, Kenneth Lo, and Raphaël Gottardo. Combining mixture components for clustering. *Journal of Computational and Graphical Statistics*, 19(2):332–353, 2010.

[12] Aurelien Geron. *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow*. O'reilly, 2nd edition, 2019.

[13] Sergios Theodoridis. *Machine Learning: A Bayesian and Optimization Perspective*. Academic Press, 2nd edition, 2020.

[14] Christopher M Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, Inc., New York, NY, USA, 1995.

[15] Sarath Menon, Grisell Díaz Leines, and Jutta Rogal. pyscal: A python module for structural analysis of atomic environments. *Journal of Open Source Software*, 4(43):1824, 2019.

[16] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010. ISSN: 1938-7228.

[17] Michele Scardi and Lawrence W Harding. Developing an empirical model of phytoplankton primary production: a neural network case study. *Ecological Modelling*, 120(2):213–223, 1999.