Master's thesis

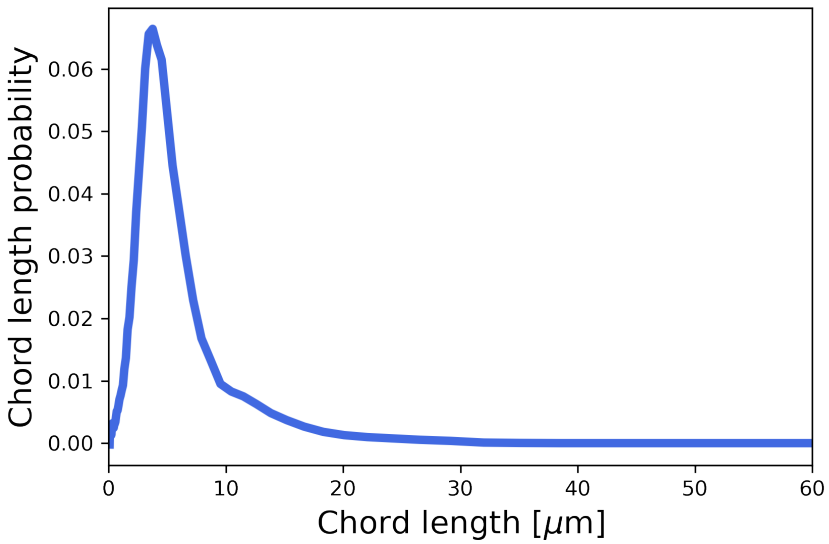
NTNU
Norwegian University of Science and Technology
Faculty of Natural Sciences
Department of Chemical Engineering

Solveig Sannes

A machine learning framework for predicting chord length distributions

An adaption to experimental data

Master's thesis in Chemical engineering
Supervisor: Ali Mesbah
Co-supervisor: Johannes Jäschke
June 2022



NTNU
Norwegian University of
Science and Technology

Solveig Sannes

A machine learning framework for predicting chord length distributions

An adaption to experimental data

Master's thesis in Chemical engineering
Supervisor: Ali Mesbah
Co-supervisor: Johannes Jäschke
June 2022

Norwegian University of Science and Technology
Faculty of Natural Sciences
Department of Chemical Engineering

NTNU
Norwegian University of
Science and Technology

Abstract

Real time monitoring and control of crystallization processes in process industry producing food, fine chemicals and pharmaceuticals is of major importance to assess quality and purity of the product. The lack of available online and in-situ measurement options have been identified as a bottleneck in this regard. The Focused Beam Reflectance Measurement (FBRM) probe can access online and in-situ measurements of the chord length distribution (CLD). However, this measurement only provides information related to the particle size, and thus a conversion to the particle size distribution (PSD) is needed. This problem has been subject to research in literature. There are several ways to address this problem, either by mapping PSDs to CLDs called the forward problem or the mapping of CLDs to PSD called the inverse problem. In this research, a framework to solve the forward problem is presented. The suggested framework consists of two main structures. The first part is a convolutional neural network (CNN) which maps PSDs to CLDs. The CNN is trained with in-silico generated data. This CNN model has already been developed from previous research efforts by members of the Mesbah research group at University of California, Berkeley. In this research, a second structure has been developed. It consists of a dimensionality reduction and correction layer to adapt the framework to an experimental dataset in the low data regime. It accounts for possible discrepancies between the experimental and simulated dataset and considers experimental conditions. The dimensionality reduction was performed with an autoencoder by reducing the dimensionality of the CLDs. The correction layer was modeled with a multi-output Gaussian process regression model.

The autoencoder was implemented and displayed great ability to compress and reconstruct CLDs from both the simulated and experimental dataset. With this tool, the CNN model was adapted and trained with the reduced dimensionality data. The new CNN model exceeded the previous model with regards to accurately predicting CLDs with the simulated dataset, however it failed to predict CLDs from the experimental data set. With the implementation of the correction layer, the final framework was successfully able to map the experimental PSDs to CLDs. This developed framework demonstrates how both simulated and experimental data can be leveraged to develop an accurate data-driven model for prediction without requiring substantial amount of experimental data. This framework also provides a robust model to generate enough data to address the inverse problem (CLD to PSD) which is of greater interest.

Preface

Declaration of Compliance

I, Solveig Sannes, hereby declare that this is an independent work according to the exam regulations of the Norwegian University of Science and Technology (NTNU).

Signature:

Solveig Sannes

Place and date: Telemark, Norway, June 2022

Acknowledgements

I have been lucky to be a part of the Mesbah research group at the Department of Chemical and Biomolecular Engineering at Univeristy of California, Berkeley the fall and spring semester of 2021/2022. This research has been a part of a project which is a collaboration between the Mesbah research group and Bayer AG. A special thanks to George Makrygiorgos, John Maggioni, and Ali Mesbah for help and guidance during this rewarding experience.

Contents

List of Figures

List of Tables

Terms and acronyms

AE Autoencoder

AI Active ingredient

API Active pharmaceutical ingredients

ANN Artificial neural network

CLD Chord length distribution

CNN Convolutional neural network

EI Expected improvment

FBRM Focused beam reflectance measurement

GCLD CLD predicted from the CNN model

GP Gaussian process

GPR Gaussian process regression

LCB Lower confidence bound

MAE Mean absoulte error

MSE Mean squared error

NaN Not a number

PCA Principal component analysis

PI Probability of improvement

PSD Particle size distribution

NN Neural network

RBF Radial basis function

ReLu Rectified linear unit

RPM Revolutions per minute

Chapter 1

Introduction

Crystallization is a widely used separation technique in many process industries producing food, fine chemicals and pharmaceuticals. The crystallization process has a major impact on important crystalline properties like purity, polymorphic form, particle size distributions (PSD). Not only do these properties have substantial effect on the efficiency of downstream processes such as filtration, drying, granulation, milling, storage, but also on product qualities[51]. Purity, shape and size of the active pharmaceutical ingredients (API) are especially crucial to in the pharmaceutical industry. These properties influence the development and design of the dosage forms due to its effect on formulation, manufacturability, dissolution and bio-performance of the dosage form[62]. Therefore, it is of major importance to pharmaceutical manufacturers to have the capability to regulate these physical properties to deliver a high quality product and to ensure consistency[62]. Consequently, this need has inspired different methods to control the particle shape and size during the crystallization process. For example, milling and micronization of particles after crystallization are frequently used by pharmaceutical manufacturers to achieve small particles that exhibit the necessary particles size characteristics [24]. Furthermore, the manipulation of operation parameters during crystallization is another way to manipulate crystal shape. This is can for example be achieved by supersaturation. Another approach is induced crystal breakage which manipulates the crystal shape by reducing the aspect ratio. This approach is often used for needle like crystals. Crystal growth rate modifiers have also been shown to significantly impact the shape of crystals [37].

The lack of sensors providing online and in-situ measurements of crystal shape have been identified as a restriction to improving design, monitoring and developing control methods in crystallization processes [42]. However, recent progress in image processing techniques have been able to provide new possible approaches for monitoring and providing in-situ and real time PSD measurements [5]. The PSD is a measurement that provides important information about crystal size. An illustration of a PSD is given in Figure 1.1. For 2-D crystals, the PSD is typically expressed as a two dimensional distribution with the crystal length and width on the x- and y-axis and the probability on the z-axis.

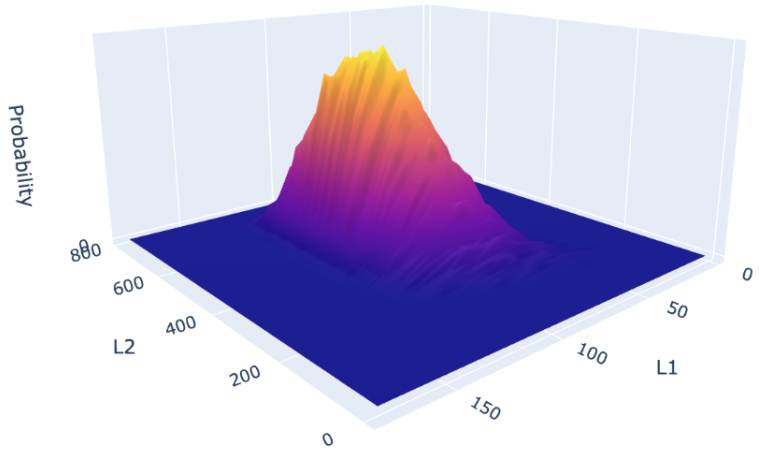


Figure 1.1: Illustration of a PSD.

The focused beam reflectance measurement (FBRM) is a tool frequently used to collect information related to particle shape and size. The main advantage of the FBRM is that it can produce online and in-situ measurements. The FBRM uses a high focused laser directed at the particles in a suspension and obtain information related to the PSD by measuring backscattering [8]. It has been broadly utilized because it is simple to implement and is able to take measurements at relatively high crystal concentrations [27]. A rotating focused laser beam scans through a suspension or slurry through a sapphire window mounted on a cylindrical probe. When the laser beam intersects a crystal, some is backscattered and recorded by a detector in the probe. The sensor in the probe records how long the backscattered signal lasted [63]. The measurement is produced by multiplying the signal length with the tangential rotating speed. This calculated quantity is called a chord length. The recorded chord lengths can then be presented in a histogram referred to as a chord length distribution (CLD). An example of a CLD is given in Figure 1.2. The x-axis typically represents the chord lengths or the logarithm of the chord lengths and y-axis represents the chord length probability.

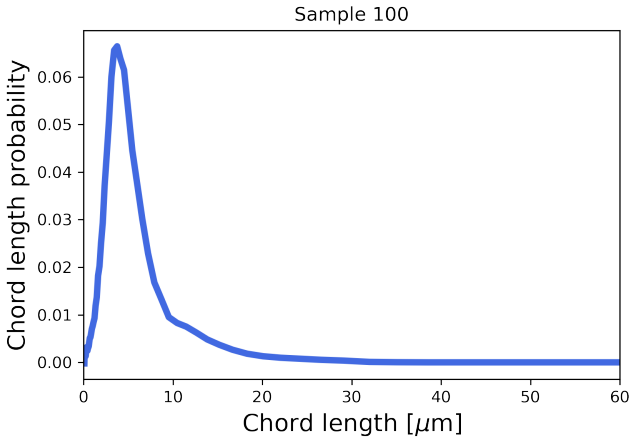


Figure 1.2: Example of a chord length distribution.

The drawback of the FBRM is that it generates CLDs. The chord length distribution contain information *related* to the particle size distribution, but does not record the actual sizes of the crystals. This is illustrated in Figure 1.3. In fact, the chord length recorded by the FBRM does not only depend on the actual size of the crystal but also on the orientation, suspension density, optical properties and solvent medium. The illustration displays some possible chord lengths recorded by the FBRM. The thick black line represents the chord length. The two first examples clearly illustrates how the chord length recorded is affected by the orientation of the crystal. In the bottom left corner, two crystals are overlapping such that the joined width of the crystals are recorded as the chord lengths. In the bottom right corner, the chord length coincides with the actual width of the crystal which would be an ideal case for recording particle size.

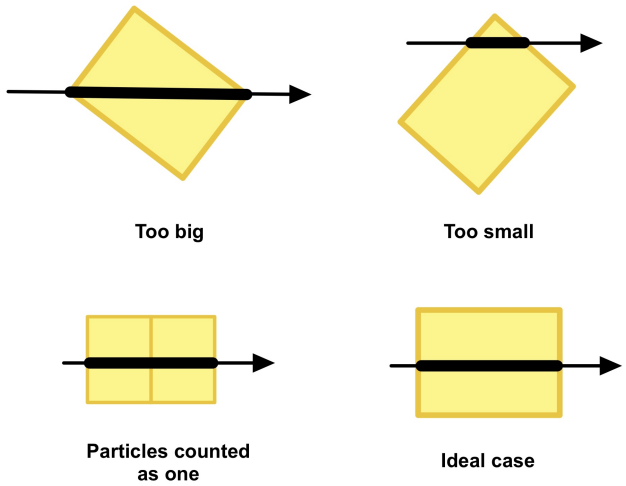


Figure 1.3: An illustration of possible chord lengths recorded by a FBRM probe.

Since the FBRM probe does not record the actual size of the crystal, the CLDs need to be related to the PSDs in order to extract useful information. Transforming a PSD to a CLD is referred to as the forward problem and transforming a CLD to a PSD is referred to as the inverse problem. Several attempts have been made to address both these mappings. Some earlier techniques are based on least square methods [33], [9]. In [33] an analytical solution is proposed to calculate PSDs from CLDs from a non-negative least squares method. This model was first validated by using simulated data. The second part of this paper [34], validated this model with experimental data. One drawback of this method is that the least square optimization can possibly lead to ill conditioned optimization problems. In addition, the accuracy of this method heavily depended on the particle shape since the aspect ratio was an important factor in the analytical model. Thus this method is not readily applicable to systems where the particle shapes are deviating from the ones defined in analytical model. Another analytical method was proposed in [59]. This approach attempts to map PSDs from CLDs by directly inverting the relationship. One major assumption in this approach is that the CLDs had similar shapes and could be represented by two moments. The method was able to invert the relationship, but cannot be universally applied as it was shown to be unstable in some cases. In [41] an iterative process with log-normal distributions was presented. For this method to be successful, the assumption that the particles are uniform within the system must hold. Consequently, this method is not applicable to systems with irregular particle shapes.

In more recent years, more attempts on have been made to use data-driven frameworks. [27] utilizes a data-driven approach to predict PSDs from CLDs. The general framework is separated into three steps: i. Compressing the CLD to a small set of parameters (average, standard deviation and slurry concentration), ii. a regression function to correlate this set of parameters with low order moments or a small number of percentiles of the PSD and iii. a two-layer network consisting of N node functions to predict the PSD histogram. For this approach to work, it is essential that the information of the CLD can be reduced to a small set of parameters called CLD descriptors. An expansion of this framework was presented in [26]. This extension enabled the model to predict CLDs with bimodal distributions by inclusion of higher order moments and a combination of two chord-selection modes. The main limitation of these models is that the moments of the CLDs must be known to perform the compression step. Realistically, there is no guarantee that real CLDs can be describe with a set of moments. An alternative to using the moments in the compression step was presented in [45]. In stead of compressing the CLD to a set of moments, the compression step was performed with a principal component analysis (PCA). In PCA a vector is compressed into a smaller set of uncorrelated numbers i.e the principal components. This approach did not result in better predictions. Artificial neural network(ANN) models have also been implemented to convert CLDs to PSDs for different types of particles [19],[22], [51]. In [51] an ANN model was developed to solve the forward problem, i.e transforming a 2D PSD to a CLD from 2D needle-shaped crystals. The data used to train the network was generated by using first principle, geometric-model based simulations. In [19] an ANN was developed to predict PSDs for sugar crystals from real-time FBRM data. This ANN model was trained with experimental data, which requires that sufficient amounts of experimental data is available. This framework successfully proved that an ANN model can be used as a soft sensor, providing online information about sugar crystallization processes. In the reviewed literature, both the forward and the inverse problem have been addressed. In this research, a data-driven framework has been developed to address the forward problem. Typically, the forward problem is mathematically well-posed and less computationally expensive compared to the inverse problem. As already encountered in the literature, the inverse problem can lead to ill-posed optimization problems, but is typically of greater interest, since it

will automatically produce a PSD from a CLD measurement. However, there are still significant advantages of solving the inverse problem first. The forward problem can easily be decomposed into smaller sub-problems which requires less data overall. Moreover, an effective model for solving the forward problem can be used to solve parameter optimization and/or control problems. In this type of problem, the CLD represents the objective function. Lastly, a model solving the forward problem can be used to simulate data to create a data-driven approach for the inverse problem, reducing the amounts of experimental data required. The framework developed in this research, offers a data-driven approach to map experimentally generated PSDs to CLDs without requiring a substantial amount of experimental data. In the reviewed literature, a few approaches are described offering data-driven frameworks that are developed either on simulated data or experimental data. Data-driven approaches typically requires a larger data-set, but in many cases it may not be possible or too expensive to generate enough experimental data. In this research, the developed framework demonstrates how both simulated and experimental data can be leveraged to develop an accurate data-driven model for prediction. This is the main advantage of this research, and together with the structure of the framework itself- made up of three important machine learning approaches, a convolutional neural network (CNN), and autoencoder (AE) and a Gaussian process regression model- it encompasses the novelty of the research. The CNN model in this framework has already been developed from previous research efforts of members of the Mesbah research group at Universe of California, Berkeley. This CNN model was trained with a simulated dataset and the aim of the research presented in thesis, has been to expand the overall framework to be able to make accurate predictions for experimental data. The expansion of the framework consists of a dimensionality reduction performed by an autoencoder and the implementation of a correction layer modeled with a Gaussian process regression model. The remaining part of this thesis is divided into three main chapters:

Chapter 2: Neural networks and the existing CNN model.
This chapter presents the fundamental theory and building blocks of the CNN model. This includes a broad description of neural networks in general, convolutional neural networks, Bayesian optimization and Gaussian process regression, the latter two being integral for the tuning of the model. This is followed by a description of the simulated dataset and the experimental dataset. Lastly, the performance of CNN model is presented and evaluated.

Chapter 3: Adaption of the convolutional neural network to experimental data. Chapter 3 presents the two main steps implemented to adapt the original framework to be able to make accurate predictions with experimental data. The theory underpinning the concepts is provided along with a thorough evaluation and discussion. With the building blocks of this framework put together, the final part of the chapter assesses the final framework and its performance.

Chapter 4: Final conclusion. The final conclusion briefly sums up the important findings of this research and provides suggestions to further work.

Chapter 2

Machine learning and neural networks

The use of neural networks is rapidly increasing. Neural networks are powerful tools because of their flexibility, strong performance and the ease at which they can be applied to a myriad of possible applications. There are several different types and structures of neural networks that can handle a variety of tasks like reinforcement learning, regression, image recognition, hyperparameter tuning and the list goes on. These types of frameworks have a lot of potential, almost only limited by imagination. The fundamental idea builds on arranging a large number of nodes in a hierarchy structure. Each node receives some input data and performs a simple mathematical operation and pass the output of forward to another node. In this way, the nodes in the structure are connected. The nodes are typically organized in layers stacked on top of each other. After the process of training, the network learns to recognize the important information in a data set. Once the network is sufficiently trained, it is able to extract the important information from an *unseen* dataset which is similar to the training dataset. In this chapter, a brief introduction to neural networks and the most important elements of typical architectures are presented. In this research a convolutional neural network (CNN) was used to map particle size distributions(PSDs) to chord length distribtuions(CLDs). Therefore, a section of this chapter is dedicated to a more detailed description of this specific type of neural network. Since there are many choices that can be made with regards to the design of the neural network which will have major impact on performance, a systemic approach of deciding these parameters should be considered. A popular method for tuning the hyperparemeters in a neural network is Bayesian optimization. An introduction to Bayesian optimization is thus explained in this chapter. The Bayesian optimization algorithms heavily relies on the use of Gaussian processes. Therefore, the theoretical background of this powerful non-parametric regression method is provided. With all the important concepts explained, the architecture of the CNN model is presented. Subsequently, the performance with both the simualted dataset and the dataset generated from experiments is presented and evaluated. If the datasets are similar enough, the CNN model should be able to make predictions for the experimental dataset too. The final evaluation includes some thoughts and limitations of the modeling choices and performance before moving on to the next chapter.

2.1 Neural networks

The structure of neural networks are inspired by how the biological neural networks are structured. Biological neural networks are complex systems whose full extent and capacity still eludes scientists today. The general workings of a single neuron can be explain as follows: A neuron receives a number of inputs (from the dendrites), which are processed and result in an output (potential in the axon). This process in is illustrated in Figure 2.1. Next to the biological neuron is an illustration of a single neuron in an artificial neural network, clearly inspired by the nature of a biological neuron. The neuron is one of the fundamental building blocks of an artificial neural network.

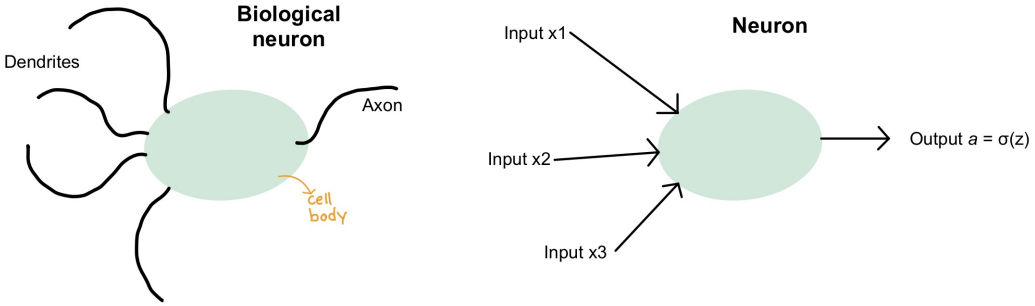


Figure 2.1: A visual comparison of a biological neuron and a neuron in neural networks.

The illustration of a neuron in a neural network in Figure 2.1 depicts the basic mathematical operation performed by the neuron. Each input of the neuron is assigned a weight, w_i, and added together along with the bias term b which sums up to z. This summation is given in Equation 2.1.

$$z = \sum_{i=1}^{N} w_i x_i + b \tag{2.1}$$

To produce the final output of the neuron, an activation function is used to transform z to the output value,

$$a(z) = \sigma(z) \tag{2.2}$$

here $a(z)$ is the output of the neuron and σ is the activation function. An activation function is necessary because it introduces non-linearity to the network model. Consequently, the network model is able to approximate non-linear functions. More neurons can be connected in a layer, and several layers can be stacked on top of each other and seen in Figure 2.2. This stacked structure introduces a complexity to the model which makes it able to recognize intricate and non-linear features of the input data.

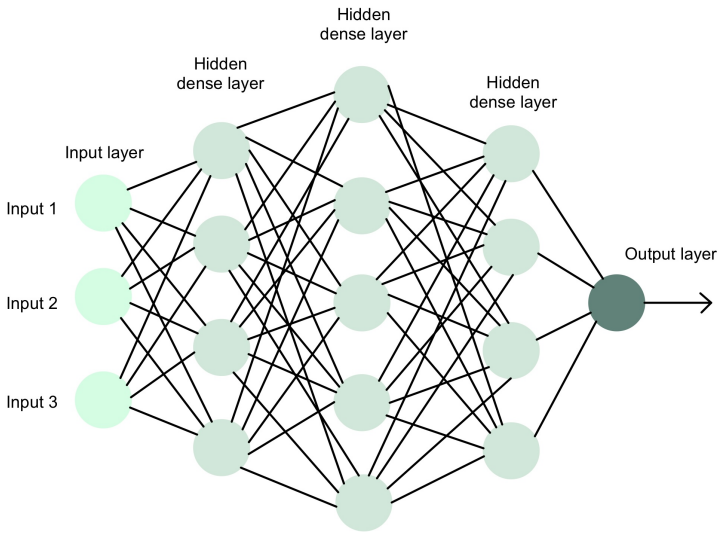


Figure 2.2: Illustration of neurons connected in layers.

If all the neurons in the previous layer are connected to the next layer, it is called a *dense layer*. Theoretically, a deep enough network with non-linear activation functions could approximate any non-linear function. Without an activation function between the layers of neurons, a network with many layers would still be equivalent to one linear layer [16]. Examples of commonly used activation functions are ReLu, tanh and Sigmoid. The choice of activation function depends on the problem at hand and the structure of the neural network. Some comments about the choice of activation function will be mentioned at the end of this sections.

The basic idea of neural network training is to systematically adjust the weights in the network until the prediction error is minimized. An illustration is provided in Figure 2.3. The network predicts an output consisting of two blue and three purple cells whereas the true output consists of three blue and two purple cells. The algorithm will thus adjust the weights and biases in the network to make a better prediction of the output.

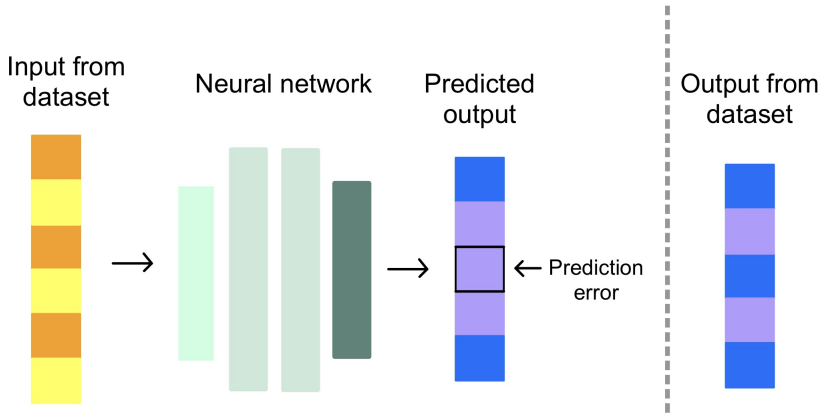


Figure 2.3: Illustration of prediction error in a neural network.

First, a data set with input data and corresponding output data is prepared. The data set is the split into a selected number of batches. The batches are then passed to the neural network consecutively. The input data is passed to the input layer and then to the first hidden layer and so on until it reaches the output layer. The result of this forward pass is stored and used in the next step which is a backwards pass. The algorithm will compare the predicted output with the output in the data and calculate the error between the two. Commonly used error functions are mean absolute error (MAE) or mean squared error (MSE), but it could also be a custom error function. The backwards pass calculates the error gradients of the weights and biases in the network by using the chain rule[16]. The algorithm then analyses how much each output of each node contributed to the total error. This reverse approach is called backpropagation. The weights and biases will be updated according to a gradient descent optimization algorithm based on the error gradients obtained from the backpropagation [52]. These updating algorithms are called optimizers and have different updating schemes. Some common optimizers are SGD, RMSPROP and ADAM[18]. The choice of optimizer can affect how fast the training progresses, how computationally expensive the training is and how well the weights and biases are updated to minimize the training error. A more in depth description of the individual optimizers is out of the scope of this thesis. The fitting algorithm will repeat this process for the remaining batches. When all the batches have passed through the network, the algorithm has completed one epoch. The algorithm will then complete the same process for the remaining number of epochs. Upon completion, the network will store the weights and biases of the best epoch and the model can be used to make predictions. One possible issue that can arise with the backpropagation algorithm is the vanishing gradient problem. This problem is encountered when the weights and biases are being updated. The chance of this occurring is affected by the architecture of the neural network for example by the depth of the network and/or the choice of activation function. Sometimes if the calculated gradient is small, it gradually vanishes during the backpropagation to the input layer [6]. Consequently, the network may not be able to update its weights appropriately and in the worst case, it could bring the training to a complete halt. Therefore, the choice of activation function can significantly influence the quality of the training. The activation functions Sigmoid, tanh and ReLu are illustrated in Figure 2.4.

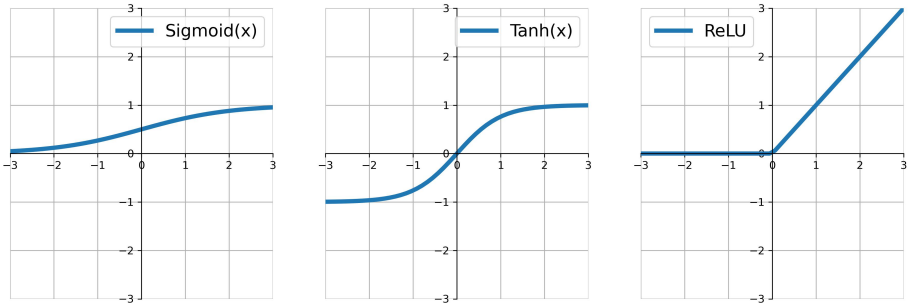


Figure 2.4: The activation functions Sigmoid, tanh and ReLU.

The Sigmoid function is a common activation function. It is a non-linear function with a smooth S-shape and will produce an output between 0 and 1. Another function is tanh. Compared to Sigmoid, tanh is symmetric around zero which means that it produces negative and positive outputs [48]. Another activation function is ReLu which stands for rectified liner unit. It is widely used and generally less prone to issues regarding vanishing gradients compared to Sigmoid and tanh.

The typical flow of neural network training is visualized in the flow chart in Figure 2.5.

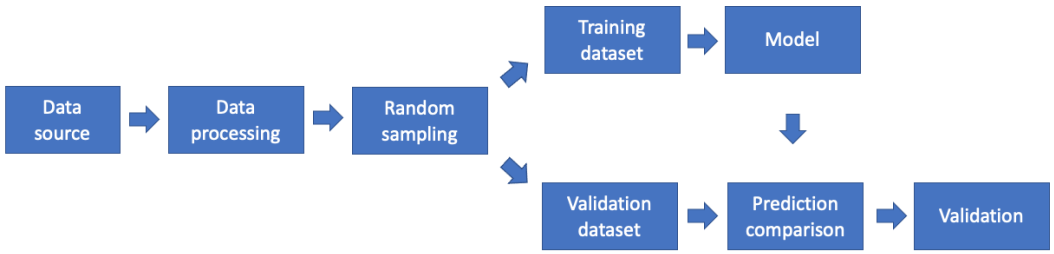


Figure 2.5: The flow chart illustrates the typical evolution of training a neural network.

As seen in this flow chart, the data is often processed before training to ensure that the quality of the data is appropriate for the training. Data-processing is often integral in order to obtain a model that can successfully make predictions as the quality of a neural network model heavily relies on the dataset. There are a few steps that can be easily implemented to modify the data set in order to ensure that it does not make the fitting of the weights harder than it needs to be.

Cleaning up the data

A given data set may not be complete. In a real data set, there will most likely be a few NULL or NaN values[32]. These are values that the system cannot handle which must be taken care of

before the training can be initiated. One way to remedy this, is to leave out the rows or columns of the data that contain the NaN values. This is a very simple approach, however the amount of data dropped relative to the size of the entire data set must be taken into account, as is could lead to too much information loss. Another alternative approach is mean imputation. In this approach, the NaN values are substituted with the mean of the observed data [25].

Scaling the data

A neural network attempts to map input variables to output variables by learning. The different variables in the datset may be from very different domains and recorded with different units and thus the data may be of different orders of magnitude. If the data is not scaled, it can prove difficult to find a model that handles both data of small orders of magnitude and large orders of magnitude well. For example, very large inputs can lead to very large weights in the system which is an indicator of an unstable network[13]. Two very common ways of scaling a dataset are min-max normalization and standardization. In min-max normalization, each value in the dataset is normalized by fist subtracting the minimum value followed by a division of difference of the maximum and minimum value of the respective variable. As a result, the datset will only consist of values ranging from 0 to 1.

$$x_{i,normal} = \frac{x_i - X_{\min}}{X_{\max} - X_{\min}} \tag{2.3}$$

In data standardization, the mean of the respective variable is subtracted from the data followed by a division of the standard deviation,

$$x_{i,standard} = \frac{x_i - \mu}{\sigma} \tag{2.4}$$

This will transform the data set into having a zero mean and a standard deviation of 1 for each of the respective variables.

Model development and evaluation

Before the training can start the dataset needs to be split into a training set and a validation set. This splitting of the dataset is done randomly and a 70-80/30-20 split is common. The appropriate splitting ratio depends on the dataset, the neural network architecture and how much data is available. Once the training and validation data is prepared, the training set can be used to train the neural network and produce a model. When the network is trained sufficiently, that is, the model error is sufficiently small, the model will be tested on the validation set. The neural network model will make predictions based on the validation set input. If the prediction error is satisfactory low, the model is validated.

2.1.1 Finding a good fit

The aim of neural network models is to find a function f which maps inputs x to the output y,

$$y = f(x) \tag{2.5}$$

How effective f is depends on how good f is to predict new output values of a dataset *unseen* by the model. This leads to the concept of generalization which explains how well a machine learning

model responds to data which is not already seen by the model during training. If the model cannot make predictions from another set of data from the same domain, it is not very useful. Overfitting and underfitting are terms used to describe the generalization ability of a machine learning model [15]. Neither overfitting and underfitting are desirable as both leads to poor performance. In Figure 2.6, three models with different levels of fitting are illustrated.

Figure 2.6: Illustration of an overfitted model, a model with a good fit and and underfitted model.

Overfitting occurs when the model is able to learn the statistical noise in the training data set such that it negatively influences performance. Noise in the data will be interpreted as actual concepts and will make generalization hard. If a model is overfitted, it will struggle to make good predictions with a different data set from the same domain. Underfitting occurs when the model has not been able to learn the important concepts in the data. When this happens the model will not be able to generalize to a new data set either. Underfitting is easier to detect as the training error will be large: The model is not able to adjust the weights to fit the training data. Typically, this can be solved by providing a larger dataset. There are several techniques developed to prevent overfitting in machine learning algorithms as it is a relatively common problem. This is due to the a large number of weights and biases in neural networks that can be adjusted to exactly match the training data. Dropout is a technique that randomly drops a fraction of nodes during the network training[40]. This is an attempt to reduce co-adaption between nodes. Co-adaption can lead to units experiencing highly correlated behavior. As a consequence, the nodes work less independently and may rely too much on other nodes. This phenomenon is illustrated in Figure 2.7.

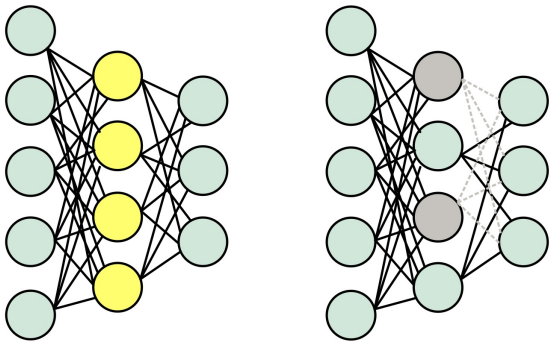


Figure 2.7: The network on the left displays nodes exhibiting co-adaption. (yellow circles). The network on the right tries to counteract this effect by dropping nodes (grey circles).

If dropout is added to a neural network layer, it forces nodes to probabilistically take on more or less responsibility in the layer [11]. If successful, the correlated behavior will be disbanded and the model will be more robust and better at making predictions with new data. Another technique to prevent overfitting is early stopping. This is a simple technique where the network will stop the training before it learns the noise in the data[64].

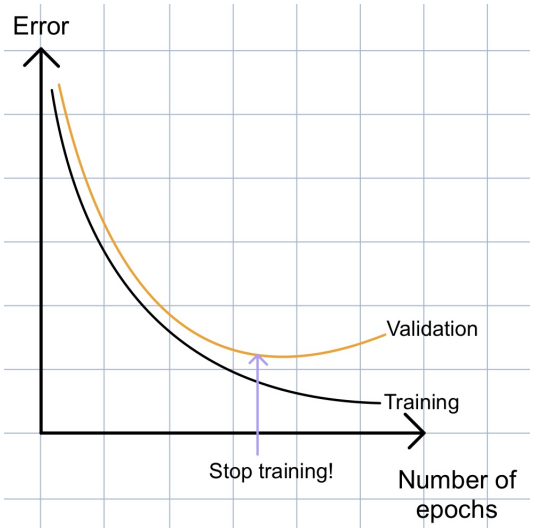


Figure 2.8: Illustration of early stopping. The training is stopped as the validation error starts to increase even if the training error was still decreasing.

The training will stop if the training error does not improve sufficiently or the validation error starts to increase. There is no point to continue the training after this point as the model will not be able to make better predictions. In addition, early stopping can make the training more effective as the training might stop before all the epochs are completed. Another technique is adding $\ell 1$ and $\ell 2$ regularization. This technique aims to reduce the size of weights and biases which can be a

source of instability and consequently cause overfitting[15]. If there are large weights in the system, the model may have been able to learn the statistical noise in the data. By adding regularization, the loss function will also take into account the size of the weights, thus preventing large weights from accumulating in the system. The most common regularizers are $\ell 1$ and $\ell 2$ regularization given in Equation 2.6 and Equation 2.7, respectively[56].

$$\ell 1 = Error(Y, \hat{Y}) + p \sum_{j=0}^{M} (\mid w_i \mid) \tag{2.6}$$

$$\ell 2 = Error(Y, \hat{Y}) + p \sum_{j=0}^{M} (w_i^2) \tag{2.7}$$

Here Error(Y,$\hat{\text{Y}}$) is the error between the predicted $\hat{\text{Y}}$ and the Y in the data set, w_i is the ith weight and p decides how much the large weights should be penalized. $\ell 1$ regularization effectively shrinks smaller weights to zero, but compared to $\ell 2$, the larger weights in $\ell 1$ are not penalized as much. Therefore, $\ell 1$ will have more weights that are zero, but also some weights that will possibly have larger values than $\ell 2$. Since $\ell 1$ exhibit these characteristics, $\ell 1$ often used in feature extraction, because important features are enhanced and insignificant ones are ignored [29]. The $\ell 2$ regularizer attempts to make all weights smaller, but does not force the weights to zero. Therefore, $\ell 2$ is better suited for more complex data patterns such as regression problems.

2.2 Convolutional neural networks

CNNs are inspired by how the animal vision works and are frequently used in photo recognition and classification tasks. It is a deep learning model that can process data with a grid patterns [52]. CNNs consists of a series of several elements and a typical structure is given Figure 2.9.

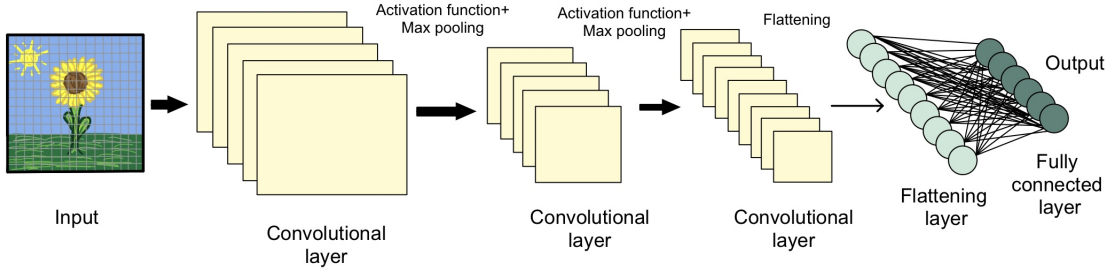


Figure 2.9: Example CNN

The structure is generally made up of a few important building blocks: convolutional layers, pooling layers, a flattening layer and fully connected layers. In Figure 2.9 an image is the input to the stacked architecture. The architecture is made up of a series of convolutional layers and max pooling layers. The output of the last convolutional layer is flattened before it is passed to a fully connected layer which produces the output.

Convolutional layers are constructed to extract features from an input image and consequently learn to recognize features. The neurons in a convolutional layer are arranged as feature

maps [44]. Each neuron in the feature map has a receptive field which is connected to a neighborhood of neurons in a previous convolutional layer via a kernel[55]. This is illustrated in Figure 2.10. The kernel is an array that moves across every pixel of the input image and performs a mathematical operation. Typical sizes of the kernel are 3x3, 5x5 and 7x7. By going over the pixels in the image, the important features can be extracted. The inputs are thus convolved with the learned weights to compute the convolved feature i.e the new feature map. The convolved feature is then transformed by a non-linear activation function like Sigmoid, tanh and ReLU.

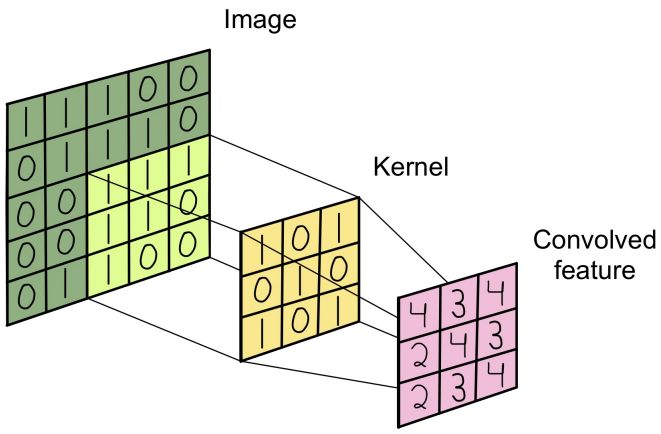


Figure 2.10: Illustration of a kernel moving across an image performing convolution.

In Figure 2.10, the result of the convolution operation is illustrated: A kernel matrix $\left(\begin{smallmatrix}1&0&1\\0&1&0\\1&0&1\end{smallmatrix}\right)$ has moved across the all the pixels of the image. The difference between two consecutive kernel positions is defined as the stride length. In the figure, the kernel has a stride length of 1. The convolved feature is obtained by the sum of the element-wise product between input tensor and the kernel. For example, the convolution calculation of the bottom right corner (light green) will be 1x1+1x0+1x1+1x0+1x1+0x0+1x1+0x0+0x0=4. The bottom right cell of the convolved feature (pink matrix) is thus 4. In this way, the kernel moves across all pixels and is able extract the important features of the input image.Convolutional layers can be stacked, such that the output of one, is the input to the next one. As the feature map shrinks after the convolution operation, padding is often added to counteract this. Zero-padding is typically implemented which adds rows and columns of zeros to the edges of the input image before the convolution operation occurs. Thus the input dimensions will be retained after successive convolution operations [52]. **Pooling layers** are used to reduce the size of the convolved feature. This helps to reduce the computational power required, to extract the dominant features and it also works as a noise suppressant[44]. The most common types of pooling are max and average pooling. The max poling operation works such that the 2x2 kernel matrix moves across the feature map and determines the larges value within the kernel and records it. On the left, the kernel has moved across feature map with a stride length of 2. On the right, the average pooling operation calculates the average of all the values in the kernel and records it. **Flattening layers** converts the data from the previous convolutional layer into a 1-D vector which will be the input to the fully connected layer. In other words, the output of the last convolution layer is transformed into one long feature vector [16]. **Fully connected layers** are dense layers and the last layers of the network. For categorization problems, the last dense layer

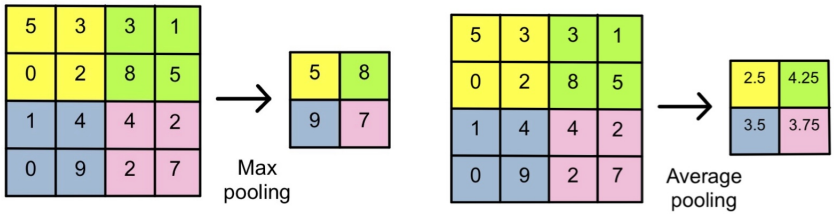


Figure 2.11: Illustration of max and average pooling.

utilizes a Soft Max activation function. A Soft Max function generates the probability that a given input ends up in a particular class and has the form [3],

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{2.8}$$

where K is the number of classes and z_i is the i-th element of the input vector. The CNN is trained the same way as described in section 2.1: The weights and biases in the network are adjusted to reduce the prediction error. The weights and biases will be updated according to the updating scheme of the optimizer based on the error gradients obtained from the backpropagation algorithm. When designing a CNN or any other type neural network there are a lot of possibilities and many choices with regards to the structure that need to be made. How many layers are appropriate? Which activation function should be used? These are parameters that defines the network architecture and need to be decided before the training takes place. These parameters are referred to as hyperparameters and are the variables that describes the architecture of the deep learning model itself. The hyperparameters will have major impact on the learning process and the weights and biases that the algorithm eventually ends up learning [39]. Tuning the hyperparameters is not an easy task: The hyperparameters that are chosen will heavily influence performance and without a systemic approach, finding the right set would be like finding a needle in a haystack. Some important hyperparameters are learning rate, activation functions, number of layer, dropout rate, kernel size and padding. The learning rate is a hyperparameter that affects the optimizer. The learning rate affects the size of the step length of the gradient descent optimization algorithm. There are several approaches to tune the hyperparameters. The most common methods are grid search methods, random search, Bayesian optimization and genetic optimization. In grid search and random search a search space is defined as a grid of hyperparameter values or as a bounded domain of hyperparameters values. In grid search, all the points will be tested whereas in random search, random points will be tested [14]. In genetic algorithms, each hyperparameter is defined as a gene and each solution is a combination of the different genes. Then the solutions create offspring and only the best ones survives. Consequently, the final solution will be the solution with the best set of genes, that is, the model with the most optimal hyperparamters [35]. In this research, Bayesian optimization was used to tune the parameters in the CNN model. Bayesian optimization is a black-box estimator which is a popular method for tuning hyperparameters in neural networks. The next section will give a detailed description of the principles of Bayesian optimization.

2.3 Bayesian optimization and Gaussian process regression

In this section Bayesian optimization and Gaussian process regression are explained. Bayesian optimization is a popular approach to hyperparameter tuning. It has gained recognition due to its flexibility, efficiency and the relative ease at which it can be implemented. Bayesian optimization often utilizes Gaussian processes regression as a part of its framework. Gaussian process regression is a non-parametric regression approach and can be used when the nature of the underlying function is unknown or hard to analytically evaluate [46].

2.3.1 Bayesian optimization

Bayesian optimization has gained popularity over the past years. It is used in many big data applications where there are a lot of possible design choices that leads to parameters needing tuning. From a mathematical perspective, Bayesian optimization is concerned about either maximising the accuracy or minimizing the error an unknown objective function[47],

$$\mathbf{x}^* = \arg\max_{\mathbf{x}\in\mathcal{X}} f(\mathbf{x}) \tag{2.9}$$

where f is a black-box function with no simple closed form. $\mathbf{x}$ are arbitrary points in the domain at which f is evaluated. However, evaluating f can be difficult as f might be non-convex, multinodal and the derivatives with respect to $\mathbf{x}$ might not be accessible. For these reasons, the evaluation of f could prove to be costly. The problem at hand decides the nature of the function f. In this research, f is a deep neural convolutional network with tunable parameters. Bayesian optimization is very effective in problems involving neural networks: It takes into consideration the previous iterations of the optimization when predicting new guesses for the hyperparameters, which makes the search more efficient. Fundamentally speaking, Bayesian optimization is a method that uses Bayes theorem to search for a maximum or minimum. Bayes' theorem is given by

$$P(A \mid B) = \frac{P(B \mid P(A)}{P(B)} \tag{2.10}$$

and is a way to calculate conditional probability. By using the total probability theorem, P(B) can be written as,

$$P(B) = \Sigma_{i=1}^{n} P(B \mid A_i)P(A_i) \tag{2.11}$$

and P(B) can be interpreted as a normalizing constant. Therefore, the expression can be simplified to describe a proportional quantity[12],

$$P(A \mid B) \propto P(B \mid A)P(A) \tag{2.12}$$

The expression above can be intuitively interpreted as,

$$\text{posterior} = \text{likelihood} \cdot \text{prior} \tag{2.13}$$

The posterior can be thought of as the current theory or hypothesis that about which inference is desired. In other words, what is the probability to observe A given the knowledge available about B. The prior, the set of data or observations about A and the likelihood describes the probability of B given A[10]. This gives a basis to make predictions about the unknown objective function. In Bayesian optimization, the data and samples observed will be collected in D=D(x_1, x_2, ..., f(x_1),

f(x_2), ...) as the optimization progresses. Thus, D will collect the result of the evaluations and the points at which the function f was evaluated. If P(f) is set to the prior, then the likelihood function is then defined as P(D | f). P(D | f) can be interpreted as: How likely is the data D that has been observed, given what is known about the prior, P(f) [10]? The expression for posterior distribution P(f | D) is then,

$$P(f \mid D) \propto P(D \mid f)P(f) \tag{2.14}$$

The posterior includes all the previous history of the evaluated points and is updated at every step. Since the nature of the objective function is often unknown, hard or computationally expensive to analytically evaluate, a surrogate function is often used to approximate the posterior function. It is used to calculate an output score given a set of input hyperparameters. In other words it calculates,

$$P(\text{accuracy} \mid \text{hyperparameters}) \tag{2.15}$$

Probabilistically, the surrogate function represents the conditional probability of the objective function f given the available information of the data D. Some common surrogate functions serving this purpose are random forest regression, tree-structured Parzen Estimator and Gaussian processes[31]. In this research, the surrogate function was modeled by a Gaussian process(GP). The theoretical basis of Gaussian processes will be described in the next section.

An indispensable element of the Bayesian optimization algorithm is the ability to effectively search for new potential points to be sampled i.e new sets of hyperparameters. In order to search the parameter space smartly, a utility function, often referred to as an acquisition function, is maximized. It is a function that is relatively cheap to evaluate and will generate samples that will be used in the next evaluation. The acquisition function will have higher values were the GP predicts the objective function to be high (i.e high accuracy) and in areas of low exploration (i.e high uncertainty) [10]. Some typical acquisition functions are LCB, PI and EI and will be described more in detail below.

The Bayesian optimization algorithm comprises of a sequential solving of Equation 2.9 for a given set of values, $\mathbf{x}$. The general algorithm can be described in the following way [47],

for *n=1,2,3...,* **do**
  find $\mathbf{x}_{n+1}$ by optimizing acquisition α function over the GP
  $\mathbf{x}_{n+1}$=arg max $\alpha(\mathbf{x};D_n)$
  evaluate the objective function at $\mathbf{x}_{n+1}$ to obtain y_{n+1}
  augment data for $D_{n+1}=D_n,(\mathbf{x}_{n+1}, y_{n+1})$
  update statistical model (GP model)
end

Algorithm 1: Pseudo code for Bayesian optimization

Here, α is the acquisition function, y is the sampled value of the objective function(y=f($\mathbf{x}$)), and D is the collection of previous points and evaluations. The algorithm runs for a predetermined number of iterations and return a set of hyperparamters that provided the highest accuracy. In this research the package skopt.gp_minimize from scikit-optimize was used to perform the Bayesian optimization. This package provides library functions that performs Bayesian optimization by using a Gaussian process for approximating the surrogate function. The default acquisition function is set to gp_hedge which for every iteration, chooses between PI, EI and LCB in a probabilistic manner. Probability of improvement (PI) aims to maximize the probability of improving the current optimal solution[60]. It searches for points near the current optimal values to find a better optimizer. Therefore, it is likely that PI converges to a local optimal solution. The expected improvement (EI)

function, aims to maximize the expected value in the vicinity of the current optimal point. If the function value is less than the expected value, the algorithm will explore other parts of the domain instead [58]. Therefore, unlike PI, EI is not likely to converge to the local solution optimum. LCB (lower confidence bound) is an acquisition function that decides whether the next sampling point should be close to the current optimum (exploitation) or rather explore another zone of the sample space with lower confidence (exploration). This trade-off between exploitation and exploration is determined by a parameter κ[58].

Bayesian optimization is an algorithm that can be used to effectively search through a hyper-parameter space in order to find an optimal set of hyperparameters for a neural network. The next section will explain Gaussian processes and Gaussian process regression which are used to model the surrogate function in the Bayesian optimization algorithm.

2.3.2 Gaussian process regression

The Gaussian process regression model can be interpreted as a distribution of functions given a set of points, and inference is made in this function space [43]. It has gained popularity due to its flexibility, but also because it provides uncertainty measurements to its predictions[57]. In addition, this non-parametric regression method works well in the low data regime [1]. A Gaussian process is defined as a collection of random variables, which have a joint multivariate Gaussian distribution that is entirely specified by its mean function and covariance function [46]. The mean function and covariance function can be thought of as analogous to the mean and covariance matrix in parametric Gaussian models[21]. A Gaussian process is typically expressed as,

$$f(\mathbf{x}) = \mathcal{GP}(m(\mathbf{x}), k(\mathbf{x}_i, \mathbf{x}_j) \tag{2.16}$$

m is a function which returns the mean of its input, and k($\mathbf{x}_i$,$\mathbf{x}_j$) represents the covariance or kernel function.

$$m(\mathbf{x}) = \mathbf{E}[f(\mathbf{x})] \tag{2.17}$$

$$k(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{E}[(f(\mathbf{x}_i) - m(\mathbf{x}_i))(f(\mathbf{x}_j) - m(\mathbf{x}_j))] \tag{2.18}$$

The Gaussian process model is a distribution over functions and determined by its mean and covariance. The smoothness of the function is defined by the covariance [57]. If $\mathbf{x}_i$ and $\mathbf{x}_j$ are regarded as close in value by the covariance function in Equation 2.18, then the outputs f($\mathbf{x}_i$) and f($\mathbf{x}_j$) will be similar. There are different forms of covariance functions k($\mathbf{x}_i$,$\mathbf{x}_j$) that can be utilized in the Gaussian process. The radial basis function (RBF) is often used to model the covariance function and will be described more in detail at the end of this section. The prior m($\mathbf{x}$) is often set to zero to make computation cheaper and to only do interference based on the covariance function. This assumes that the data is normalized to a zero mean. To achieve this empirically, the mean of the observations is subtracted from the observations[46]. An illustration of a Gaussian process regression is given in Figure 2.12. The green points represent the observed data $\mathbf{X} = [x_1,..., x_n]$ and $\mathbf{X} = [x_1,..., x_n]$. The blue line represents the mean function estimated by the Gaussian process given the observed points. New predictions can be made at new points $\mathbf{X}_*$ and will lie on the the blue line f($\mathbf{X}_*$) =$\mathbf{f}_*$.

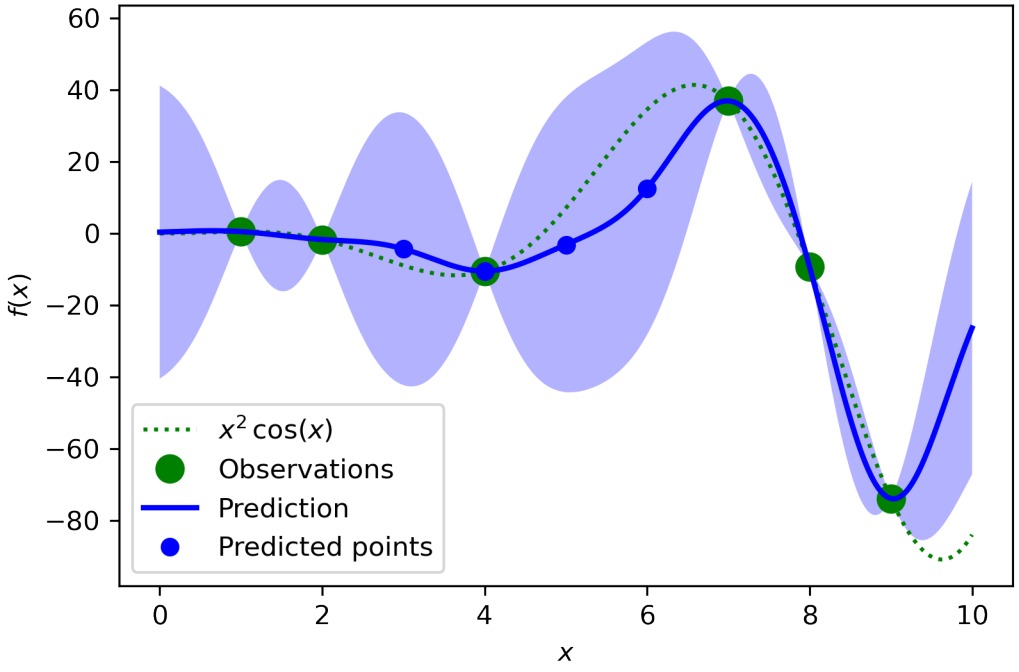


Figure 2.12: The figure illustrates a Gaussian process. The dashed, green line represents the true function. The green points represents the observed points and the blue line represent ts the predicted function. The blue shaded area around the blue line represents the uncertainty of the blue line.

The joint distribution of $\mathbf{f}$ and $\mathbf{f}_*$ can be written as,

$$\begin{bmatrix}\mathbf{f}\\ \mathbf{f}_*\end{bmatrix} \sim \mathcal{N}(\begin{bmatrix}m(\mathbf{X})\\ m(\mathbf{X}_*)\end{bmatrix}, \begin{bmatrix}\mathbf{K} & \mathbf{K}_*\\ \mathbf{K}_*{}^T & \mathbf{K}_{**}\end{bmatrix}) \tag{2.19}$$

Here, $\mathbf{K} = \mathrm{k}(\mathbf{X},\mathbf{X})$, $\mathbf{K} = \mathrm{k}(\mathbf{X},\mathbf{X}_*)$ and $\mathbf{K}_{**}= \mathrm{k}(\mathbf{X}_*,\mathbf{X}_*)$ and $(\mathrm{m}(\mathbf{X}), \mathrm{m}(\mathbf{X}_*)) =(0,0)$. The probability density distribution can can be written in a more compressed form,

$$P(\mathbf{f}, \mathbf{f}_* \mid \mathbf{X}, \mathbf{X}_*) \tag{2.20}$$

The next step is to express the conditional distribution. In other words, what is the probability that $\mathbf{f}_*$ is observed given $\mathbf{f}$?

$$P(\mathbf{f}_* \mid \mathbf{f}, \mathbf{X}, \mathbf{X}_*) \tag{2.21}$$

The derivation of the conditional distribution of two jointly Gaussian random variables is given in section A.1 in Appendix A. The conditional distribution can be expressed as follows,

$$\mathbf{f}_* \mid \mathbf{f}, \mathbf{X}, \mathbf{X}_* \sim \mathcal{N}(\bar{\mathbf{f}}_*, \bar{\sigma}^2_{\mathbf{f}_*}) \tag{2.22}$$

where $\bar{\mathbf{f}}_*$ and $\bar{\sigma}^2_{\mathbf{f}_*}$ are given as,

$$\begin{aligned}\bar{\mathbf{f}}_* &= m(\mathbf{X}_*) + \mathbf{K}_*\mathbf{K}^{-1}(\mathbf{f} - m(\mathbf{X})) \\ &= \mathbf{K}_*\mathbf{K}^{-1}\mathbf{f}\end{aligned} \tag{2.23}$$

$$\bar{\sigma}^2_{\mathbf{f}_*} = \mathbf{K}_{**} - {\mathbf{K}_*}^T\mathbf{K}^{-1}\mathbf{K}_* \tag{2.24}$$

Note (m($\mathbf{X}$), m($\mathbf{X}_*$)) =(0,0). This is assuming there is no noise present in observed data. However, more realistically, the observed data will most likely be somewhat noisy. Therefore, the observed value will be $\mathbf{y} = \mathbf{f}_i + \epsilon_i$ where $\epsilon_i \sim \mathcal{N}(0,\sigma_n^2)$. Assuming that ϵ_i are independent, identically distributed (i.i.d) Gaussian distributions, the prior of the noisy distributions will be,

$$\mathbf{y} \sim \mathcal{N}(0, \mathbf{K} + I\sigma_n^2) \tag{2.25}$$

The joint distribution then becomes,

$$\begin{bmatrix}\mathbf{y} \\ \mathbf{f}_*\end{bmatrix} \sim \mathcal{N}(\mathbf{0}, \begin{bmatrix}\mathbf{K} + I\sigma_n^2 & \mathbf{K}_* \\ {\mathbf{K}_*}^T & \mathbf{K}_{**}\end{bmatrix}) \tag{2.26}$$

Thus, the conditional distribution can be expressed as,

$$\mathbf{f}_* \mid \mathbf{y}, \mathbf{X}, \mathbf{X}_* \sim \mathcal{N}(\bar{\mathbf{f}}_*, \bar{\sigma}^2_{\mathbf{f}_*}) \tag{2.27}$$

where $\bar{\mathbf{f}}_*$ and $\bar{\sigma}^2_{\mathbf{f}_*}$ are given as,

$$\begin{aligned}\bar{\mathbf{f}}_* &\triangleq \mathbb{E}[\bar{\mathbf{f}}_* \mid \mathbf{y}, \mathbf{X}, \mathbf{X}_*] \\ &= \mathbf{K}_*(\mathbf{K} - I\sigma_n^2)\mathbf{y}\end{aligned} \tag{2.28}$$

$$\bar{\sigma}^2_{\mathbf{f}_*} = \mathbf{K}_{**} - {\mathbf{K}_*}^T(\mathbf{K} - I\sigma_n^2)^{-1}\mathbf{K}_* \tag{2.29}$$

There are different functions that can be used to express k($\mathbf{x}_i$,$\mathbf{x}_j$). The most common one and often used in hyper parameter tuning is the radial basis function (RBF) given as,

$$k(\mathbf{x_i}, \mathbf{x_j}) = \sigma_f^2 \exp(-\frac{1}{2}(\mathbf{x_i} - \mathbf{x_j})^T M^{-1}(\mathbf{x_i} - \mathbf{x_j})) \tag{2.30}$$

where M = $\begin{bmatrix}l^2 & 0 \\ 0 & l^2\end{bmatrix}$. In the RBF function there are two parameters that can take on different values, σ_f^2 and l. σ_f^2 is signal noise and it can be thought of as how much the predicted function can vary vertically. l is the length scale and it can informally be thought of as quickly the function can change significantly between two points [43]. This effect can be seen visually: If l is high, then it will give a smoother function. If l is small, it will give a wigglier function.

2.4 Existing CNN model and performance

2.4.1 Datasets used in the model development

There are two main dataset used in developing the final framework in this research. The data set used to create the CNN model is obtained through simulations and is referred to as ether the simulated dataset. The second data set is obtained from a series of experiments produced in the laboratory of Bayer AG in Leverkusen. The generation of these datasets is out of the scope of this research, but a description of both datasets will be provided below.

Simulated dataset

The simulated dataset is based on simulations of needle-like particles that can be approximated as square-based prisms. An illustration is provided in Figure 2.13.

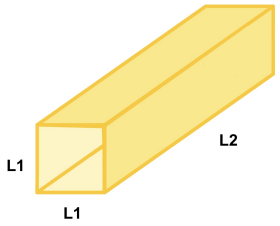
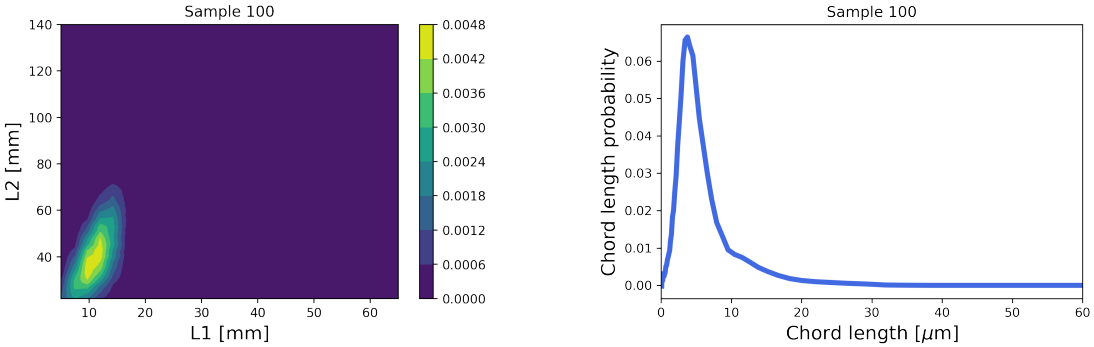


Figure 2.13: Approximation of a needle-like crystal as a square-based prism.

Since it is assumed that both sides of the base of the crystals are of the same length, the simulated PSDs are 2D distributions made by generating a large number of 2-D crystals with width L_1 and length L_2. The width L_1 is drawn from a normal distribution and L_2 is obtained by multiplying L_1 by the aspect ratio (AR). The AR is set to greater than 1 to obtain needle shaped crystal representations. The 2D PSD is then obtained by arranging the crystal representations in a histogram. The bins on the x-axis represent L_1 and the bind on the y-axis represents L_2. The probability density is plotted on the z-axis. The discretization grid is set to 30x300 i.e L_1 will be sorted into 30 bins and L_2 will be sorted into 300 bins. L_1, L_2 and the probability density will be saved in a tensor in addition to Δ L_1, Δ L_2 which are the difference between the midpoint of two consecutive bins. The resulting tensor describing the PSD will have a dimension of 30x300x5 for each sample, where 5 represents the five channels L_1, L_2, the probability density, Δ L_1 and Δ L_2. The corresponding CLDs are approximated with a geometric model. An algorithm creates a polytope based on the two lengths, L_1 and L_2 for each crystal representation in the PSD tensor. Each polytope is subjected to a series of rotations and then the projection of the polytope is calculated. The projection represents the laser intersecting a crystal to imitate the measurement of the FBRM probe. The length of a projection will be represent the chord length. When the algorithm has successfully iterated through all possible combinations of L_1 and L_2 in the PSD, the measured chord lengths are combined in a chord length distribution by sorting the chord lengths in a histogram with 100 bins ranging from 0.1 μm to 1000 μm. A visualisation of a simulated PSD and the corresponding simulated CLD is presented Figure 2.14.



(a) PSD (b) CLD

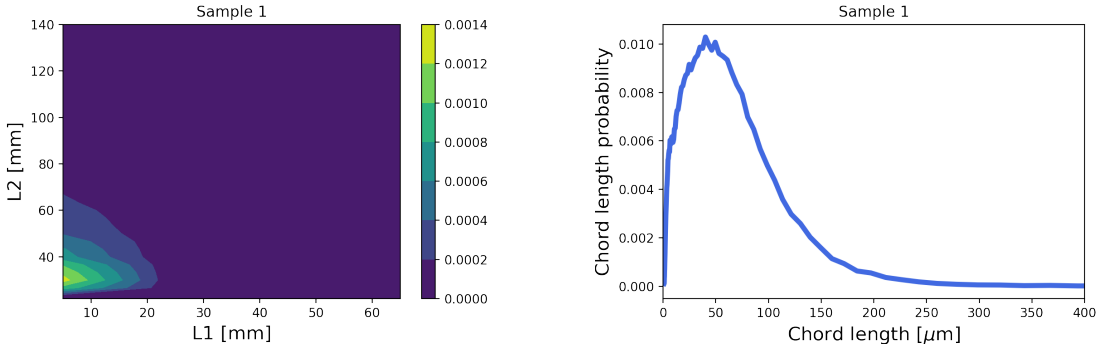
Figure 2.14: The PSD and the corresponding CLD of sample 100.

Experimental dataset

Three different active ingredients were used to generate the dataset of 26 experiments in total. The active ingredients will be referred to as AI1, AI2 and AI3. AI1 is the most stable form of D-mannitol produced by Sigma Aldrich. AI2 and AI3 are two active ingredients chosen from the Bayer AG portfolio and are of interests for the agrochemical and pharmaceutical divisions. AI2 and AI3 were produced in Leverkusen according to the standard recipes of the patented Bayer processes. All the active ingredients have needle-like and rod-like shapes. A good approximation to describe these shapes, is as squared-based shapes. The aspect ratio (AR) is defined as the ratio between the smallest side, L_1 to the longest side, L_2, of the prism. For needle-like shapes, the aspect ratio is greater than 1. In these experiments, the aspect ratio was ranging from 1.1 to 18. Two parameters were varied during each experiment: the stirring rate in revolutions per minute (RPM) and the mass suspension density. The stirring rate varied from 200 to 600 RPM and the suspension density varied from 1%wt to 9 %wt. The list of experiments carried out for each active ingredient and the corresponding experimental conditions is summarized in Table A.1. Two different solvents were used for the active ingredients, thus the systems had slightly different diffraction indices. AI1 was suspended in acetone, whereas AI2 and AI3 were suspended in water. The experiments were carried out in constant temperarture (300K) in a 500 mL unbaffeled glass reactor with a 4 blade impeller. The suspension volume for each experiment was 450 mL. The chord length distribution associated with each system were measured with a Focused Beam Reflectance Measurement device. The device was a ParticleTrack G400 from Mettler Toledo[54]. The FBRM probe is inserted directly into the suspension at an angle which allows the particles to flow past the probe window with ease. A focused laser beam passes through this sapphire window and the detector in the probe detects the backscattered light produced by the individual particles sizes and structures[53]. The duration of each signal detected is multiplied by the scan speed and the resulting measurement is defined as the chord length. The measured chord lengths can be presented in a histogram ie. a chord length distribution (CLD). Each CLD measurement of each set of experimental conditions was measured for 30 minutes with a sampling rate of 10 seconds.

In order to measure the particle size distribution (PSD) of the active ingredient in each system, a QicPic device was used. The device was QicPic L02 from Sympatec [49]. The device uses an image analysis technique and operates in a similar way to a modern microscope. It is able to

capture physical properties of each single particle by capturing with special optics the particles in the frame. The information about the particles physical properties is interpreted by an evaluation software which then presents the information in a PSD[50]. Around 500, 000 individual particles were used to determine each PSD. The minimum resolution of the QicPic device was set to 0.5 μm and the minimum particle size was set to 1.0 μm. The PSDs were measured after each FBRM measurement to eliminate the possibility of breakage during the measurement. The PSD information for the 26 experiments were saved with the dimensions 30x300 with five channels i.e in a tensor of shape 26x30x300x5. The five channels are L_1, L, PSD value, Δ L_1, Δ L_2. A visualisation of the measured PSD of sample 1 and the corresponding measured CLD is presented in Figure 2.15.



(a) PSD (b) CLD

Figure 2.15: The PSD and the corresponding CLD of sample 1.

2.4.2 Network architecture

The CNN model and subsequent models in this research were developed in Python 3.7. The neural network models were largely constructed with the TensorFlow library, version 2.9.1. The remaining packages and libraries used in this research are listed Table A.2 in Appendix A. In Table 2.1 the main components of the CNN architecture are presented. Some hyperparameters are marked as 'Tunable' which means that the optimal value will be decided from the Bayesian optimization. An illustration of the structure is given in Figure 2.16.

The input to the CNN model is a PSD tensor with dimensions 30x300x5. As illustrated in Figure 2.16, the CNN model architecture consist three main blocks consisting of a stack of convolutional layers, a max pooling layer and a dropout layer. The number of convolutional layers stacked in each block and the dropout rate in the dropout layers are tunable hyperparameters. The dimensions of the kernel performing the convolutional operation and the kernel performing the max pooling operation are given in Table 2.1. The padding in the convolutional operation and in the max pooling operation are both set to zero-padding to avoid shrinkage of the feature map. Between the convolutional layers and the max pooling layer there is an activation function which is also a tunable hyperparameter. After passing through these three main blocks, the feature map tensor is flattened to a 1-D vector and passed to a dense layer consisting of 100 nodes. There is a linear activation function after the dense layer which produces the output of 100 bins representing the CLD. The loss function used in the training is the mean squared error and the optimizer is set to Adam. The learning rate for this optimizer is a tunable hyperparameter. The next step is to tune

Table 2.1: The table list the main components of the CNN model.

Element in network	Number	Kernel dimension/rate
Input image	1	-
Convolutional layers	Tunable	5x5
Max pooling layer	1	2x2
Dropout layer	1	Tunable
Convolutional layers	Tunable	3x3
Max pooling layer	1	2x2
Dropout layer	1	Tunable
Convolutional layers	Tunable	3x3
Max pooling layer	1	2x2
Dropout layer	1	Tunable
Flattening layer	1	-
Dense layer	1	100

the unknown hyperparamters to obtain the final architecture of the CNN model. The Bayesian optimization was run with 25 iterations and the number of epochs in the CNN model was set to 200. Each hyperparameter to be tuned in the optimization was assigned a given range of possible values or belonging to a specific category. The parameters tuned in the optimization and their individual bounds or categories are given in Table 2.2,

Table 2.2: The table shows the possible ranges or categories for the different hyperparameters tuned in the Bayesian optimization.

Hyperparameter	Possible values/category	Initial values/category
Learning rate	0.0005-0.01	0.001
Drop out rate	0.0-0.3	0.0
Activation function	ReLu, tanh	tanh
Number of layers block 1	10-18	16
Number of layers block 2	24-40	32
Number of layers block 3	48-68	64

The progress of the Baysian optimization throughout the iterations is given in Figure A.1 in Appendix A. The hyperparamters with the best score in the Bayesian optimization after 25 iterations are given in Table 2.3.

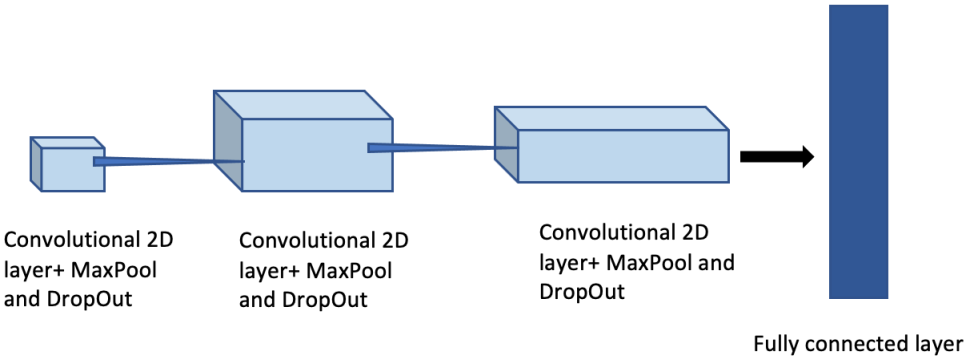


Figure 2.16: Simple schematic illustration of the existing CNN architecture.

Table 2.3: The table shows the optimal hyperparamters after the optimization.

Hyperparameter	Value/category
Learning rate	0.001
Drop out rate	0.0
Activation function	tanh
Number of layers block 1	16
Number of layers block 2	32
Number of layers block 3	64

With the optimal hyperparameters found, the next step is to train the CNN model with this final architecture. The network was trained and the number of epochs was set to 400.

2.4.3 Performance of network

The training and the validation loss of the best epoch is given in Table 2.4,

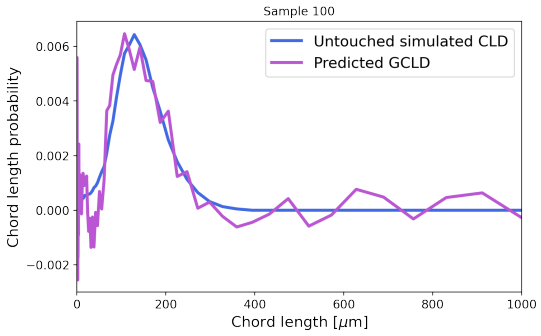
Table 2.4: The training and validation mean absolute error of the best epoch.

Error	Value
Training	4.8859e-05
Validation	4.6222e-05

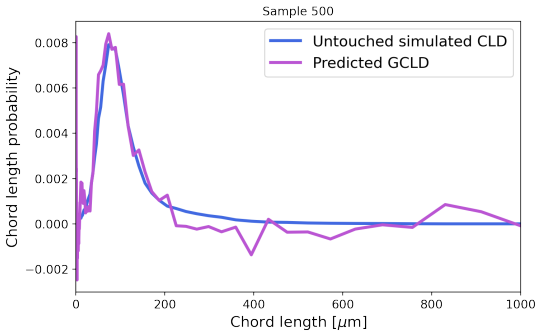
The training loss and the validation loss across all the epochs are given in Appendix A in Figure A.2. To assess the performance of the CNN model, some samples from the simulated and experimental dataset are plotted next.

Performance with simulated data

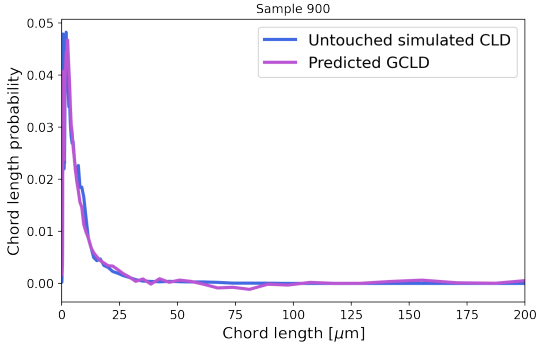
In Figure 2.17 some samples of the validation dataset are plotted. Two lines are plotted: The blue line shows the untouched CLD and the purple line shows the prediction of the CLD by the CNN model. It is labeled GCLD because the CLDs in the simulated dataset used to train the CNN model are modeled with a geometric model. For the remaining part of this thesis, the prediction of the CNN will be referred to as a GCLD.



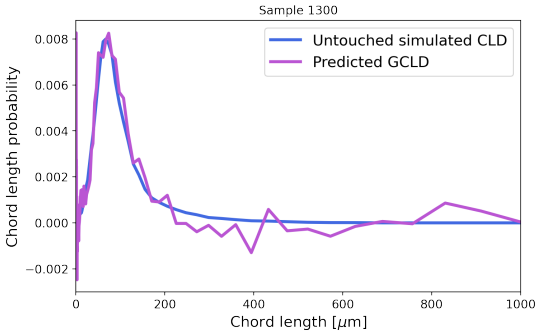
(a) Sample 100



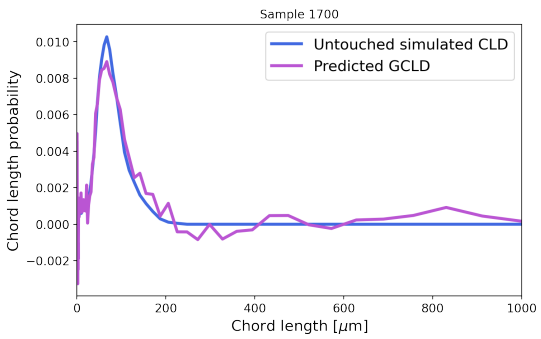
(b) Sample 500



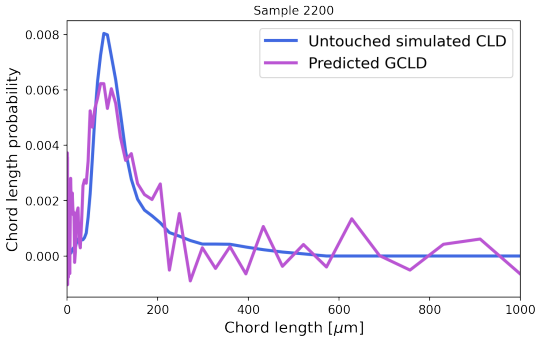
(c) Sample 900



(d) Sample 1300



(e) Sample 1700

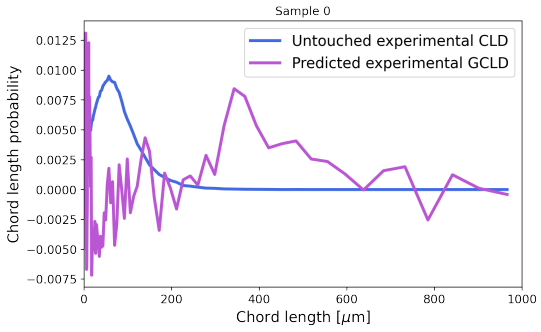


(f) Sample 2200

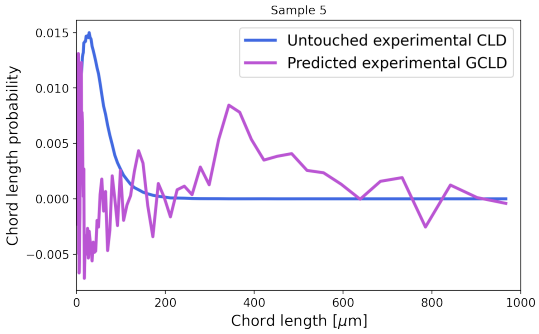
Figure 2.17: The subfigures display the untouched simulated CLD and the predicted GCLD for a selection of samples.

Performance with experimental data

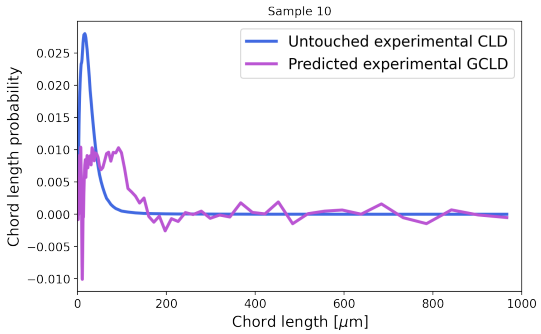
In Figure 2.18 the some samples of the experimental dataset are plotted. Two lines are plotted: The blue line shows the untouched experimental CLD and the purple line shows the predicted GCLD.



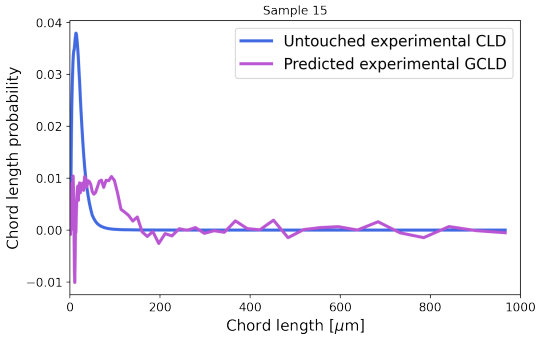
(a) Sample 0: RPM is 200 and suspension density is 1%wt.



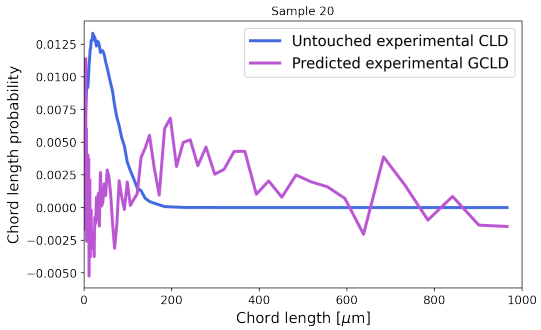
(b) Sample 5: RPM is 600 and suspension density is 3%wt.



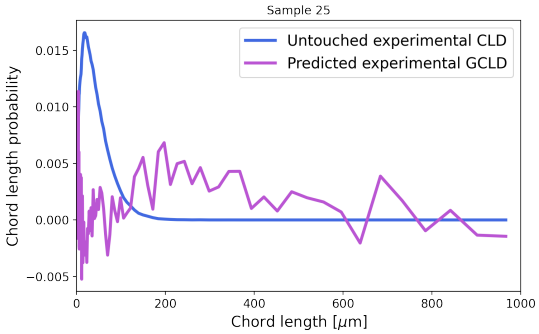
(c) Sample 10: RPM is 400 and suspension density is 1%wt.



(d) Sample 15: RPM is 200 and suspension density is 9%wt.



(e) Sample 20: RPM is 200 and suspension density is 3%wt.



(f) Sample 25: RPM is 600 and suspension density is 9%wt.

Figure 2.18: The subfigures display the untouched experimental CLD and the predicted GCLD for a selection of samples.

2.4.4 Model evaluation and discussion

In Figure 2.17 some samples of the validation set of the simulated data are plotted. The CNN model is largely able to predict the general shape of the CLDs, however all the predicted GCLDs display some non-smooth behavior: The GCLDs seem to exhibit spurious tails where the distributions should be close to zero. This indicates that the convolutional operations through the layers ultimately activates some of the nodes in the dense layer due to prediction error, thus producing somewhat noisy predictions. In addition, all the GCLDs except for sample 900 show negative predicted values. The y-axis represents the chord length probability and thus negative values are theoretically not feasible. The Bayesian optimization found that a drop out rate of 0.0 was optimal, which is an indication that the model is not overfitted. If it was, the drop out rate would probably be higher as it is a tool implemented to combat overfitting. Furthermore, in table Table 2.4 the training and validation error are given. The table shows that the validation error is lower than the training error. This difference in error is true for all epochs after around 50 epochs and can

be seen in Figure A.2 in Appendix A. This is an indication that the network may be underfitted as the validation set performs better than the training set. One way to remedy this could be to expand the dataset or to change the network architecture to a deeper structure. Another option is to perform a dimensionality reduction such that the mapping of the input data to the output data is simpler.

In Figure 2.18 some samples of the experimental data set are plotted. By inspection of these samples, it is clear that the CNN model fails to predict the GCLDs and thus fails to provide any useful information from the experimental data. This indicates that the assumption about the datasets being relatively similar may not hold. Consequently, the CNN model is not able to extract the important features of the input data and map them to the right output. Furthermore,there seem be no correlation between the prediction and the respective experimental conditions for each sample. Higher or lower values of the stirring rate or mass suspension data yield neither better nor worse predictions. Although, this may change if the CNN mapping is improved, as not much can be interpreted from these samples. However, there seem to be a correlation between the active ingredient of each sample and the CNN prediction. There are three similar patterns of the predicted GCLDs. The CNN model produce similar predictions for the pairs made up of Sample 0 and 5, 10 and 15, and 20 and 25. The samples in these three pairs have the same active ingredients AI1, AI2 and AI3 respectively. A summary of the experiments is given Table A.1 in Appendix A. This indicates the samples with the same active ingredients have similar PSDs and are mapped to the same shape by the CNN model.

Moreover, the final structure of the CNN model is defined by its hyperparameters. The hyperparameters are tuned with the Bayesian optimization. The Bayesian optimization is a efficient approach to search through a myriad of possible combinations of hyperparameters. Although the tuning will return an optimal set of hyperparameters, it cannot not guarantee that this is the universally best set possible. This is one limitation of the Baysian optimization, because the final result will invariably be a product of the available dataset, the possible ranges set for each hyperparamters and the number of iterations of the tuning.

2.4.5 Conclusion and the next steps

The CNN model was developed to map PSDs to CLDs and was trained with a simulated dataset. The model was able to predict CLDs reasonably well with the simulated dataset, that is, it was able to predict the general shape of the CLDs. However, there is still much room for improvement of the model: The predicted CLDs were not smooth and had negative values which are theoretically impossible. Overall, the model is somewhat successful in the overarching objective, however it needs some refinement to accurately do this mapping with simulated data. When the model was tested with experimental data, it failed to predict the CLDs. The CNN model is also not able to distinguish between samples of different experimental conditions and maps the ones with the same active ingredient to the same shape.

The next step in the development of this model is to be able to adapt it to make good predictions with experimental data. One approach would be to generate more experimental or simulated datasets with which to train the model. Generating more experimental and simulated data can be time consuming and expensive, in addition to possibly being difficult to generate simulated data that imitates experimental data accurately enough. The next chapter will present an alternative approach to create a model framework that can make good predictions even if the experimental dataset is sparse.

Chapter 3

Adapting to experimental data

In the previous chapter, it was seen that the CNN model performs satisfactory with simulated data, but fails to make predictions for the experimental dataset. This implies that the datset may be too different for the CNN model to be able to accurately interpret the important features in the experimental dataset. A simple solution would be to train the CNN model with a larger experimental dataset or attempt generate more simulated data which is more similar to the experimental data. However, this may be time consuming or financially or computationally expensive. Also, there is no guarantee that the experimental data can be imitated by simulations accurately enough. The following approach attempts to adapt the developed framework to make good predictions with experimental data. In order to do this, a correction layer is implemented which also takes in experimental conditions which may be affecting the measured distributions. Since the experimental data is in the low data regime, a dimensional reduction of the CLDs is performed. This provides a simpler mapping for the correction layer but also a simpler mapping for the CNN model and a faster training. A Gaussian process regression model is implemented to serve as the correction layer and an autoencoder is implemented to perform the dimensionality reduction. In this chapter, these concepts are be explained, however most of the essential element are based on theory introduced in chapter 2. The implementation and the performance of the dimensionality reduction are assessed first, then the results and evaluation of the correction layer and the resulting framework are assessed.

3.1 Dimensionality reduction

A dimensionality reduction of the CLDs was implemented for two main reasons. The experimental data available is the low data regime, therefore if the regression is performed in a space with lower dimensionality it will provide a simpler mapping: Trying to fit a well performing regression model with 100 parameters is much harder (if even possible with 26 datasets) than fitting a model with 10 or 5 parameters. It also requires less computational power. Secondly, reducing the dimensionality of the problem also allows for a faster training and simpler mapping for the convolutional neural network. The ideal dimensionality reduction is able to compress and reconstruct the original input with minimal loss of information. There are several approaches typically found in literature. For example, in the reviewed literature in the introduction, one approach to dimensionality reduction was to reduce the CLD to its moments. However, the ability to do this heavily depends on the nature of the distribution, and the is no guarantee, in fact, highly unlikely that all CLDs can be reduced to their moments. Another approach is principal component analysis (PCA), which reduces a high dimensional input space into a space where the maximal variance is displayed[4]. PCA has been frequently used in literature for dimensionality reduction tasks, but since it is essentially a linear transformation it will most likely struggle to reconstruct data with more complex features. In order to ensure that most CLDs regardless of its shape and possible complex features could be compressed and reconstructed, an autoencoder(AE) was implemented to perform the dimensionlity reduction in this research. An AE is a type of neural network with several possible applications. AE are typically used reduce dimentionality, noise reduction, feature extraction and image compression[7]. Since most of the terminology regarding neural networks has already been defined in section 2.1 and it will not be repeated here. In the next section, the general idea behind the autoencoder is be explained. The autoencoder is then added to the existing framework. With this new addition, the performance of the convolutional neural network is be evaluated before moving on to the implementation of the correction layer.

3.1.1 Autoencoder

An autoencoder(AE) is a type of neural network that consist of two main parts: an encoder and a decoder. The encoder takes in an input, reduces it to what is called a latent space (latent means hidden). The decoder takes in the output of the encoder, ie. the input reduced to the latent space, and attempts to reconstruct the original input. Structurally, an AE is a feed forward neural network that is made up of an input layer, hidden layers and an output layer. If an AE has more hidden layers, it is called a deep AE. With this stacked structure, the AE is able to capture more complex and non-linearity of features in the input data[30].

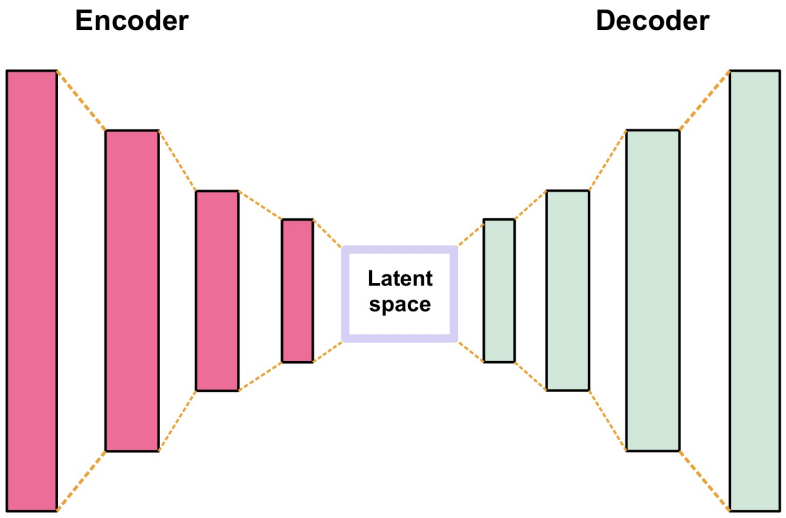


Figure 3.1: Illustration of a deep autoencoder showing the encoder, latent space and the decoder.

In Figure 3.1 an illustration of a deep AE is given. The pink part of the structure makes up the encoder and the green part makes up the decoder. The structure consists of one input layer, three hidden layers on each side of the latent space, and one output layer. An AE can also be build using convolutional layers as hidden layers. In this case, a flattening layer is needed after the last convolutional layer. The latent space in the middle of the encoder and decoder, holds the input data encoded in the reduced space. Mathematically, the output of the autoencoder can be expressed as

$$\hat{\mathbf{x}} = g(f(\mathbf{x})) \tag{3.1}$$

where f($\mathbf{x}$) represents the encoder function and g(f($\mathbf{x}$)) represents the decoder function [2]. Between the layers, there is an activation function to introduce non-linearity to the structure. When the autoencoder is trained, the input data and the output data are the same dataset since the AE is trying to reconstruct the original input. The objective of the training of the AE is to reduce the reconstruction error. The ideal AE exhibits two ideal characteristics: the training error is low such that it is able to reconstruct the input data accurately and it is not be too overfitted such that it simply memorizes the data [28]. In order to avoid overfitting, regularization or drop out layers can be added to introduce some noise in the data.

3.1.2 Structure

The autoencoder was build with the structure given in Table 3.1 and illustration is given in Figure 3.2. The dimension of the latent space must be chosen such that it is a trade-off between low dimensionlity such that it is an effective dimensionality reduction tool, but not too low such that too much information is lost to be able to accurately reconstruct the original input. The chosen dimension of the latent space was 5. The structure of the encoder is made up of an input layer with 100 nodes and then two dense layers, which first reduces the dimensionality to 25 nodes, then

to 5. The decoder consist of a dense layer of 5 nodes, then another dense layer of 25 and lastly an output layer of 100 nodes. Between each dense layer a tanh activation function was used except for the last layer. The last layer has a sigmoid activation function which maps the input to a real number between 0 to 1. This is done such that the output of the AE can be interpreted as pixels.

Table 3.1: The table shows the network architecture of the autoencoder.

Typer of layer	Nodes	Activation function
Input layer	100	-
Dense layer	25	tanh
Dense layer	5	tanh
Latent space		
Dense layer	5	tanh
Dense layer	25	sigmoid
Output layer	100	-

In addition, $\ell 2$ regularization layer was added after the first and the second dense layer. $\ell 2$ was chosen over $\ell 1$ since $\ell 2$ is more suited for complex data patters as found in distributions. ADAM was chosen as the optimizer and the loss function was set to mean squared error.

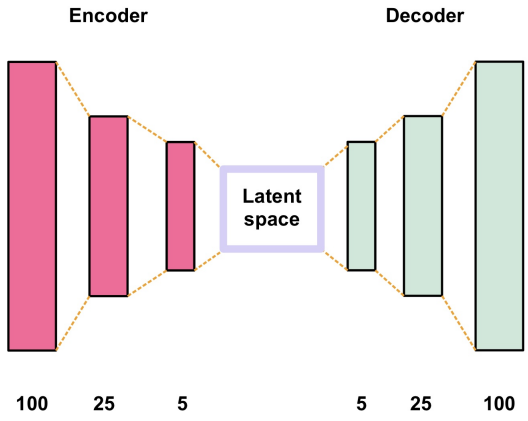
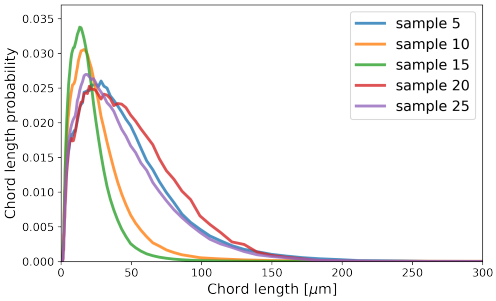
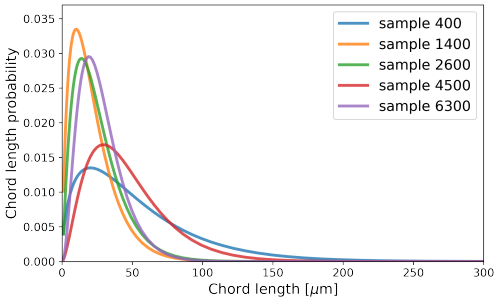


Figure 3.2: Illustration of the final structure of the autoencoder.

In addition, more simulated CLDs were generated to train the autoencoder. The simulated CLDs were generated from gamma distributions resembling the experimental CLDs. This was accomplished by generating 6400 different gamma distributions. Some samples of the gamma distributions and the experimental CLDs are given in Figure 3.3.



(a) Some samples of the experimental CLDs



(b) Some samples of the gamma distributions imitating the experimental CLDs.

Figure 3.3: The decoded predictions, the decoded original data and the original data plotted

An augmented data set was constructed from the gamma distributions and CLDs from the simulated dataset. The autoencoder was trained with the augmented dataset with a 80-20 data split. The number of epochs were set to 1500.

3.1.3 Autoencoder performance

The final training and validation loss in training of the autoencoder are given in Table 3.2.

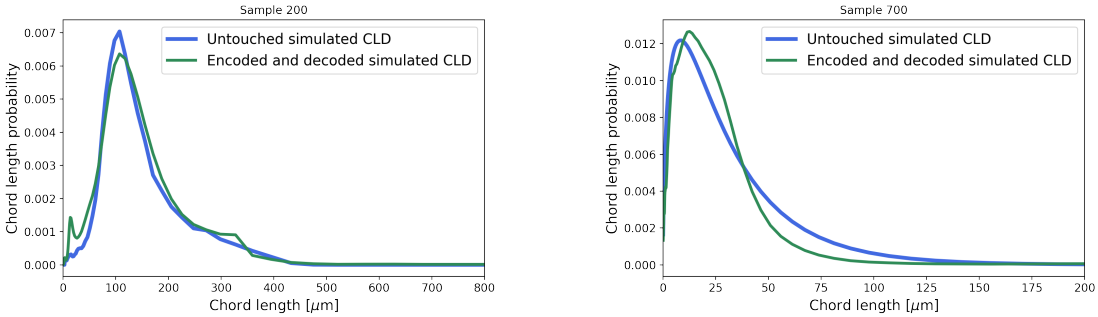
Table 3.2: The final training and validation loss

Training loss	Validation loss
1.4966e-6	1.5723e-6

The training loss and the validation loss across all the epochs are given in Appendix B in Figure B.1. To assess the performance of the autoencoder with both simulated data and experimental data, some samples from each dataset are plotted below.

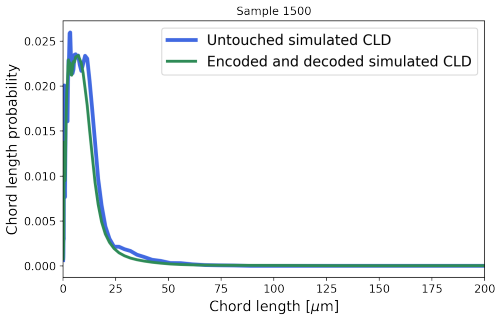
Simulated data

In Figure 3.5 the reconstruction ability of the autoencoder with simulated data is displayed. The simulated CLD and its encoded and decoded version is presented in each subfigure.

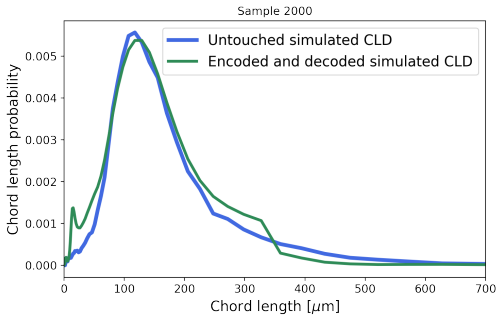


(a) Sample 200

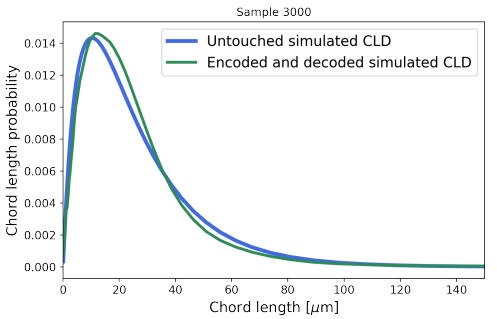
(b) Sample 700



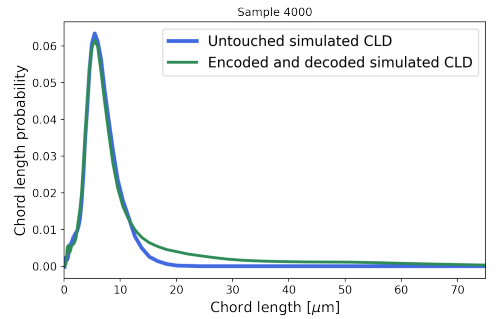
(c) Sample 1500



(d) Sample 2000



(e) Sample 3000

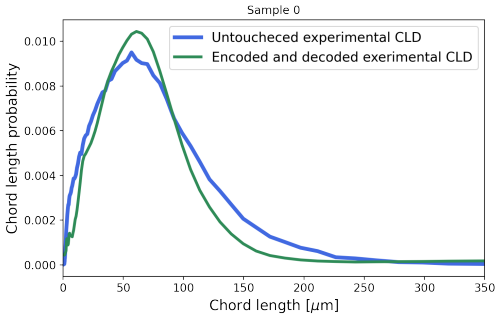


(f) Sample 4000

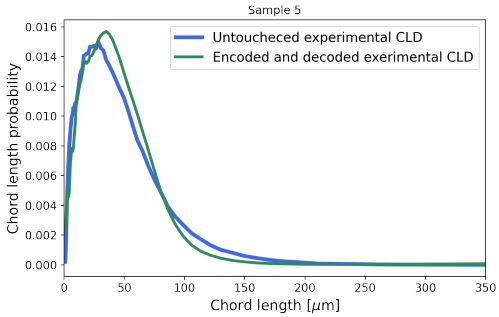
Figure 3.4: The figure shows samples of the simulated CLDs and simulated CLDs that have been encoded and decoded.

Experimental data

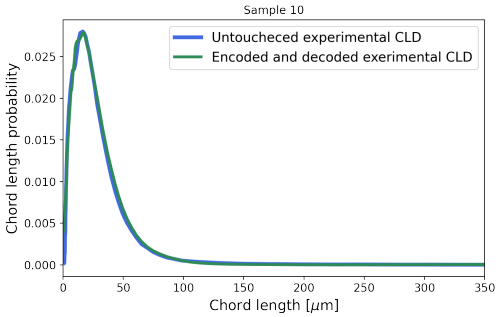
In Figure 3.4 the reconstruction ability of the autoencoder with experimental data is displayed. The experimental CLD and its encoded and decoded version is presented in each subfigure.



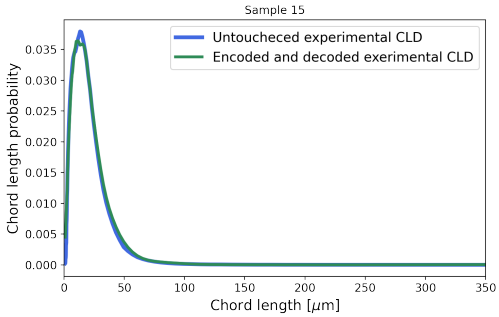
(a) Sample 0: RPM is 200 and suspension density is 1%wt.



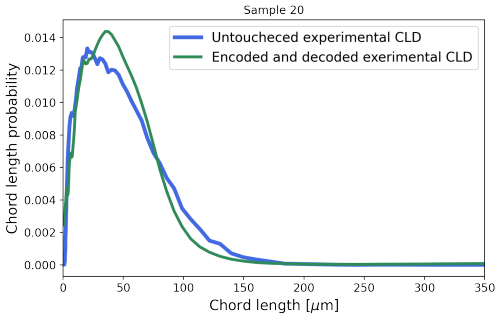
(b) Sample 10: RPM is 600 and suspension density is 3%wt.



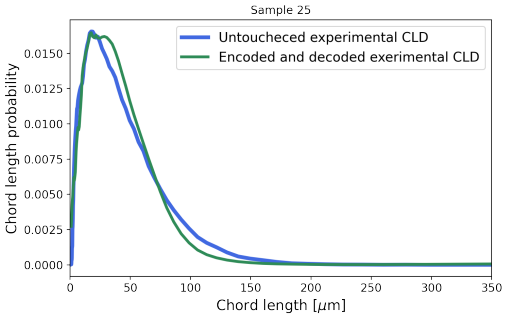
(c) Sample 10: RPM is 400 and suspension density is 1%wt.



(d) Sample 15: RPM is 200 and suspension density is 9%wt.



(e) Sample 20: RPM is 200 and suspension density is 3%wt.



(f) Sample 25: RPM is 600 and suspension density is 9%wt.

Figure 3.5: The figure shows samples of the experimental CLDs and experimental CLDs that have been encoded and decoded.

3.1.4 CNN model with CLDs in the latent space

After the training, the autoencoder was obtained consisting of an encoder and decoder part. Subsequently, the CLDs from the simulated dataset were encoded by passing the data to the encoder. The simulated dataset now consist of the PSD as input and the latent space CLDs as output. Consequently, the last dense layer in the CNN model, needs to be changed from 100 nodes to 5. Since the dataset and the CNN structure are now different, the hyperparameters of CNN needs to be re-tuned. The Bayesian optimization was therefore performed to obtain the new set of hyperparameters of the CNN model. The tuning was done with a 80-20 training split. The Bayesian optimization was run with 25 iterations and the number of epochs in the CNN model was set to 200. As before, each hyperparameter to be tuned in the optimization, was assigned a given range of possible values or belonging to a specific category. The bounds on the individual hyperparameter are given in Table 3.3

Table 3.3: The table shows the possible ranges or categories for the different hyperparameters tuned in the Bayesian optimization.

Hyperparameter	Possible values/category	Initial values/category
Learning rate	0.0005-0.01	0.001
Drop out rate	0.0-0.3	0.0
Activation function	ReLu, tanh	tanh
Number of layers block 1	10-18	16
Number of layers block 2	24-40	32
Number of layers block 3	48-68	64

The set of hyperparameters yielding the best score in the Bayesian optimization is given in Table 3.4. The scores of all the iterations of the tuning are given in Appendix B in Figure B.2.

Table 3.4: The table shows the optimal hyperparamters after the optimization.

Hyperparameter	Value/category
Learning rate	0.001
Drop out rate	0.0
Activation function	tanh
Number of layers block 1	16
Number of layers block 2	32
Number of layers block 3	64

When the optimal hyperparametrs were found, the CNN was trained with these hyperparameters. The simulated dataset was randomly split into a training set and a validation set with a 80/20 split. The model was trained with 200 epochs.

3.1.5 Performance of final CNN model

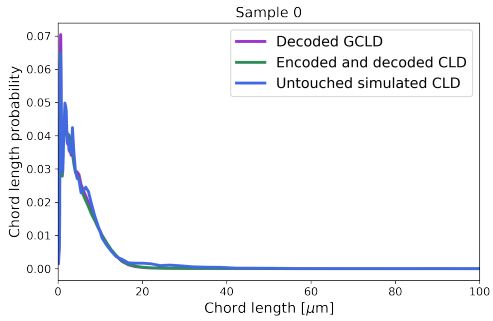
Table 3.5: CNN training loss

Training loss	Validation loss
3.5018e-5	3.5492e-5

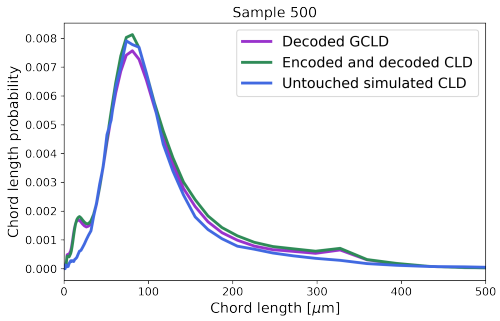
The training loss and the validation loss across all the epochs are given in Appendix B in Figure B.3. The performance of the CNN model will be presented in the figures below, both with the simulated data and then with the experimental data.

Simulated data

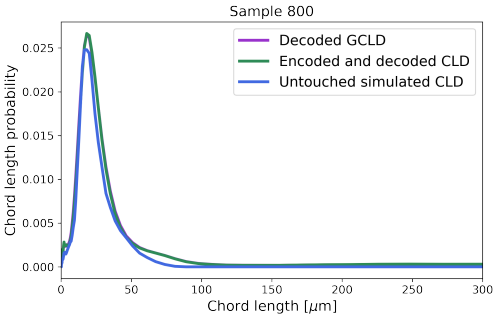
In Figure 3.6 the some samples of the validation dataset are plotted. Three lines are plotted: The blue line shows the untouched CLD, the green line shows the encoded and decoded CLD and the purple line shows the decoded GCLD. As before, output of the CNN prediction is denoted GCLD as the CLDs in the dataset used to train the CNN model are generated from a geometric model. The experimental CLD and its encoded and decoded version is presented in each subfigure.



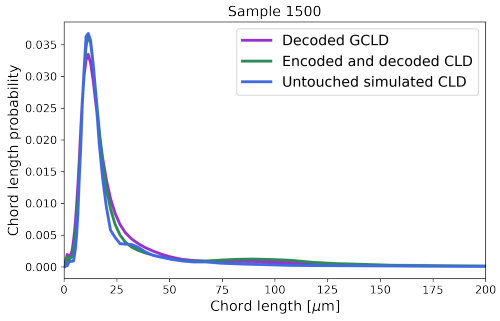
(a) Sample 0



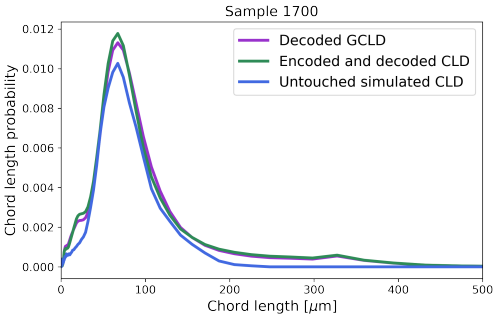
(b) Sample 500



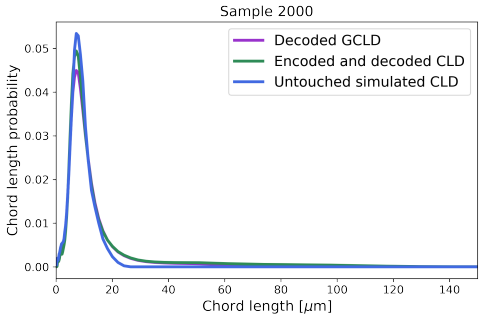
(c) Sample 800



(d) Sample 1500



(e) Sample 1700

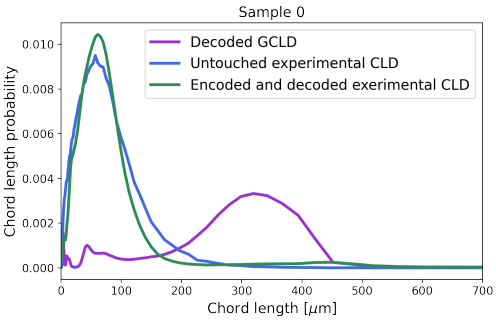


(f) Sample 2000

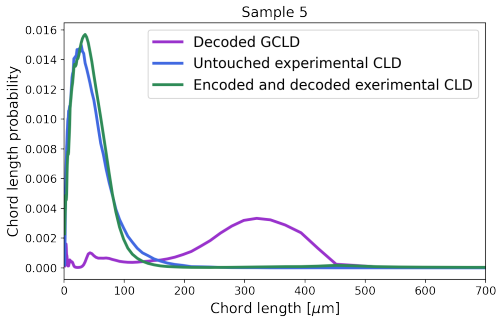
Figure 3.6: The decoded predictions, the decoded original data and the original data plotted

Experimental data

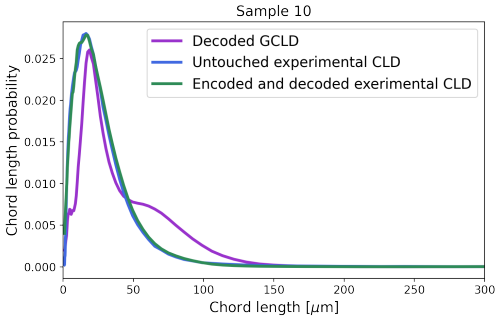
In Figure 3.7 the some samples of the experimental dataset are plotted. Three lines are plotted: The blue line shows the untouched experimental CLD, the green line represents the encoded and decoded experimental CLD and the purple line shows the decoded GCLD.



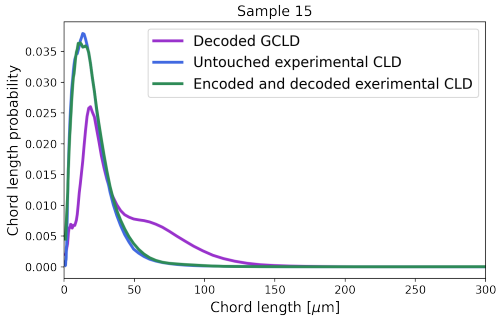
(a) Sample 0: RPM is 200
and suspension density is 1 % wt



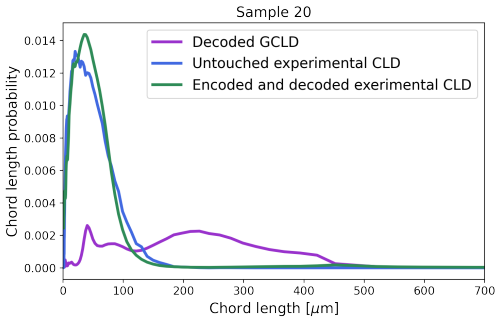
(b) Sample 5: RPM is 600
and suspension density is 3% wt



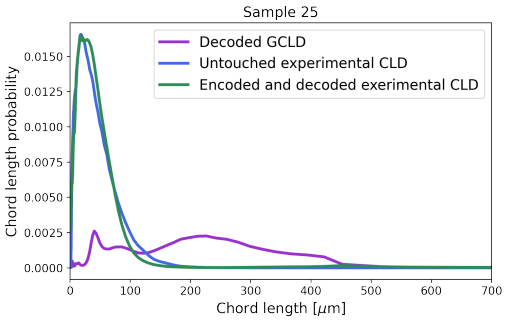
(c) Sample 10: RPM is 400 and suspension density is 1% wt



(d) Sample 15: RPM is 200 and suspension density is 9% wt



(e) Sample 20: RPM is 200 and suspension density is 3% wt



(f) Sample 25: RPM is 600 and suspension density is 9% wt

Figure 3.7: The untouched experimental CLD and the decoded GCLD.

3.1.6 Discussion

The performance of the autoencoder was presented in subsection 3.1.3 and the performance of the subsequent reduced space CNN model was presented in subsection 3.1.4. The first part of this discussion will evaluate the performance of the autoencoder and the second part will evaluate the impact of the dimensionality reduction on the performance of the CNN model.

The performance of the autoencoder is displayed by plotting a selection of samples in Figure 3.4 and Figure 3.5. In Figure 3.4 the reconstruction ability of the AE with a selection of validation samples from the simulated data is shown. The blue line is the untouched CLD and the green line is the result of encoding the CLD and subsequently decoding the output of the encoder. The autoencoder is able to perform the reconstruction well without significant loss of information. Moving on to Figure 3.5, in which the performance with of experimental data is presented. The autoencoder is able to reconstruct the experimental CLDs without much information loss. This result demonstrates the robustness of the autoencoder: It is able to produce strong results even if it has never seen CLDs from the experimental dataset. Also, this illustrates that enriching the dataset with

simulated data similar to the experimental dataset can be an effective way to boost performance if the data can be readily generated.

Since the AE proves to perform satisfactory as a dimensionality reduction tool, it could be applied to reduce the dimensionality of the problem. The CLDs in the simulated dataset were reduced to the latent space and the model was trained again with the hyperparameters obtained from the Bayesian optimization. The performance of the CNN model with some validation samples from the simulated dataset is given in Figure 3.6. There are three lines are plotted in each subplot: The blue line is the untouched CLD, the green line is the encoded and decoded CLD and the red line is the decoded GCLD i.e the decoded prediction of the CNN model. These plots show a major improvement in performance compared to the previous model without the dimensional reduction shown in Figure 2.17. Strikingly, the decoded GCLDs are able to almost perfectly capture the shape of the untouched simulated CLD. Furthermore, the turbulent nature of the previous GCLDs is gone and the model does not predict negative values. This suggests that implementing the autoencoder aided in two aspects: Model accuracy and noise reduction. Evidently, a simpler mapping of a PSD to a latent space CLD results in a better fitted model. Furthermore, the optimal drop out rate from the Bayesian optimization was 0.0. Again, this is an indication that the model was not overfitted. The dropout rate would probably be higher if the CNN was prone to overfitting. In addition, the training in Table 3.5 is slightly lower than the validation error and is persistent across epochs as shown in Figure B.3 in Appendix B. Thus there is no indication that the model was underfitted, in fact, the CNN model seems to be very well fitted to the data. Furthermore, a common application of autoencoders is as noise reduction tools. After the implementation of the autoencoder, the turbulent nature present in the previous model seems to have disappeared. Since the predicted GCLD is in the latent space, possible noise in the prediction is more likely to disappear as the prediction is forced into five possible outputs instead of a 100. Therefore, only the most important features will be extracted and smaller signals will be neglected. If there was no error present in Figure 3.6, all the plotted lines would be the same. The discrepancy between the lines stems from two possible sources: Error due to information loss from the dimensionality reduction and prediction error from the CNN model. If the decoded GCLD (purple line) and the encoded and decoded CLD (green line) are similar, but differ from the untouched CLD (blue line), the error is from the encoding/decoding operation. In other words, the CNN model is able to make a good prediction, but the decoder is not able to perfectly reconstruct the CLD. This can be seen in sample 1700 in Figure 3.6. If the discrepancy is caused by prediction error however, then the green and the blue line will coincide and the purple line will differ from the two. In this situation, the autoencoder is able to perfectly reconstruct the CLD: the encoded and decoded CLD (green line) is the same as the untouched CLD (blue line). The error of the decoded GCLD (purple line) must thus be introduced from the prediction. Error due to prediction is present in sample 1500 in Figure 3.6. Lastly, the discrepancy between the untouched CLD (blue line) and decoded GCLD (purple line) can be due to error introduced from both sources. In this case, the decoded GCLD (purple line) and the encoded and decoded CLD (green line) will be different from each other, in addition to both being different from the untouched CLD (blue line). Error from both sources is seen in sample 500 in Figure 3.6.

The performance of the CNN model with a selection of experimental data is given in Figure 3.7. Three lines are plotted: The untouched experimental CLD (blue line), the encoded and decoded experimental CLD (green line) and the decoded GCLD (purple line). These plots indicate that the model largely fails to predict the CLDs from the experimental data. The same pattern as found in the performance of the previous model is observed here too: There are three pairs, sample 0 and 5,

10 and 15, and 20 and 25 that have almost indistinguishable distribution shapes. The samples in these pairs are made with the same active ingredients. This is an indication that the experimental data differs too much from the simulated data used to train the model. Thus, the samples with the same active ingredient and similar PSD, are mapped to the same, wrong shape. The discrepancy between the untouched CLD (blue line) and the decoded GCLD (purple line) is almost solely due to prediction error of the CNN. This can be seen from the figure as the encoded and decoded CLD (green line) is almost identical to the untouched experimental CLD (blue line). Even though the CNN model largely fails to predict the CLD from the experimental data, it still performs better than the previous model. The previous model was not able to predict any coherent patters as seen in Figure 2.18. This model however, is somewhat able to capture the general shape of the samples with the active ingredient AI2 that, is sample 10 and 15. This may indicate that the data with AI2 samples is more similar to the stimulated data used to train the CNN model. To investigate this, the performance of the network with the samples with AI2 are plotted in Figure B.4, Figure B.5 and Figure B.6 in Appendix B. As seen in these figures, the CNN model is able to produce relatively good predictions for the GCLDs in Figure B.4 for the samples with 1 wt% suspension density. A possible explanation for this behavior may be assigned to the observation that high suspension densities have been identified as a possible limiting factor of the FBRM probe[23]. However, given the data available there is not enough evidence to establish this connection. This can only be interpreted as an indication for possible instrument limitation and is something that can be investigated further.

3.1.7 Conclusion

An autoencoder was implemented as a dimensionality reduction tool. The AE was successful in serving this purpose and was able to reconstruct CLDs from the both the simulated and experimental dataset with relative ease. As a result, this allowed the dimensionality of the problem to be reduced. Consequently, this lead a simpler mapping for the CNN model and it was able to make better predictions for both the simulated and experimental data set. For the experimental data set however, the predictions are still far from satisfactory and the overall framework needs to be altered to support the experimental data. The CNN model performance can most likely be improved with an augmented dataset, simulated or experimental, to make better predictions. However, as generating more data, stimulated or experimental, may be time consuming or financially/computationally expensive, another approach is discussed in the next section for the implementation of the correction layer.

3.2 Correction layer

The CNN model was trained with the simulated data. The aim of adding a correction layer was to achieve a framework that is able to predict CLDs from experimental data. The underlying hypothesis was that the simulated datset and the experimental dataset may not be not similar enough or influenced by experimental conditions such that the CNN would not be able to make accurate predictions on it own. As seen in the previous section, this hypothesis proved to be true. The CNN was not able to readily predict the CLD from the experimental data. Thus in order to correct for this for this, a correction layer was added. The correction layer also takes in two experimental parameters, the stirring rate and the mass suspension density, to account for different experimental conditions that may influence the final measurements. High suspension density is known to affect the readings of the FBRM probe [23]. At higher suspension densities, there will be many particles in the laser path. This can affect the readings as it creates multiple deviations to the light beam before it reaches the detector in the FBRM probe[22]. The effect of stirring rate have shown to have effect on particle count by the FBRM probe. Low stir speed increases the probability of settling of crystals in the system, thus reducing the particle count[61]. An illustration of the desired mapping by the correction layer is given in Figure 3.8. The correction layer model will be calibrated with a selection of samples of the experimental dataset. First, the PSDs will be the input to the CNN model to obtain a predicted CLD representation. This prediction, {X1, X2, X3, X4, X5}, will serve as the input to the correction later in addition to the stirring rate (RPM) and suspension density in the given experiment. The output {Y1, Y2, Y3, Y4, Y5} represents the encoded experimental CLD of the respective experiment. By fitting this correction layer with a selection of the experimental samples, the ideal model is able to make accurate predictions with the validation set. Subsequently, the output can be decoded to the full space CLD predictions.

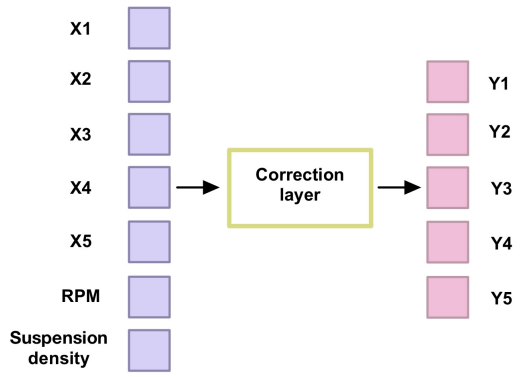


Figure 3.8: An illustration of the correction layer mapping.

A correction layer can be build with a regression model. There is a main divide between parametric and non-parametric regression models. Parametric regression models have an assumed form. In other words, the model has a set of fixed parameters that need to be fitted to the data [36]. Therefore, performance of such models heavily depends on the chosen function form and if the data exhibit this assumed relationship. Typical parametric regression models are linear, non-linear and logarithmic regression models. However, since the underlying relationship of the data is unknown,

an efficient parametric regression models may be hard to find. Moreover, the experimental data is in the low data regime, which can prove the fitting challenging (if even possible) as a larger data set is usually required. Therefore, a non-parametric regression model might be the better option. Non-parametric models do not require a fixed form to be specified a priori, but is determined by the dataset [36]. Neural networks and Gaussian processes fall into this category. As it is hard to determine the relationship of the mapping of the correction layer, a non-parametric regression model was chosen for this research. One possible option would involve building a second neural network to do the mapping. However, due to the lack of data available this option will probably lead to poor performance. Thus, to serve as the correction layer, a Gaussian process regression(GPR) model was selected. GPR does not require knowledge about the underlying relationship and works well in the low data regime. The fundamental idea behind a Gaussian process regression was explained in subsection 2.3.2. The next section will therefore not repeat the theory presented in the previous section, but briefly explain how the GPR model described in subsection 2.3.2 can also be extended to multi-output models.

3.2.1 Multi-output Gaussian process regression

In subsection 2.3.2, the fundamental principles of a Gaussian process regression for a single output vector was presented. The simplest way to use a GPR for multi-output is to construct a GPR model for each output. This approach assumes that the outputs are independent. If the outputs are correlated the Gaussian process regression model can be expanded by an additional covariance matrix to describe this dependency. In this research, the outputs were assumed to be independent due to the random sampling of chord lengths by the FBRM probe. In other words, it is assumed that crystals in the suspension are randomly oriented such that the chord lengths recorded by the FBRM probe are independent. Furthermore, it is also assumed that the independency is maintained through the transformation to the latent space due to the independent and identically distributed (iid) assumption central in machine learning algorithms[20]. This assumption states that the samples in the datasets are assumed to be iid. Effectively, it simplifies machine learning algorithms by assuming that the data distributions do not change over time or space and that each sample is independent of one another [17]. If these assumptions holds, then the resulting model will be simpler, easier to implement and less computational expensive.

The lantent space dimension was set to five points. Therefore, the same number of GPR models is needed such that there will be one model for each output. Each of the GPR models will performing the mapping,

$$f : x \mapsto y, x \in \mathbb{R}^7 \text{ and } y \in \mathbb{R}^1 \tag{3.2}$$

one for each output. Each x vector has a dimension of 7 made up of 5 numbers representing the latent space prediction from the CNN model and 2 numbers describing experimental conditions (stirring rate and suspension density). This vector, will be mapped to each of the five outputs with five different GPR models.

3.2.2 Final model

The Gaussian regression process was implemented in python with the scikit-learn library. The covariance matrix function was set to the RBF function described in subsection 2.3.2. Before the fitting of Gaussian process models, the data was standardized such that each input and output had

a mean of 0 and a standard deviation of 1. This prepossessing of the data was implemented to achieve a better fit to the data as all the datapoints will be of the same order of magnitude. If the datapoints are of different orders of magnitude, it can be hard to find a regression model that is able to accommodate larger and smaller input values at the same time. The noise was set to 0.1 to take into account that there may be some noise in the observed data. Since Gaussian process regression is a flexible regression model, it can be prone to overfitting. To decide how many samples should be kept in the validation set, the fitting was run several times with different number of validation points. The result is given in Figure 3.9 The figure was generated by running the fitting process 100 times for each number of validation samples. For each fitting, the relative error between the encoded experimental CLD and the prediction of the correction layer was calculated. The relative error was calculated using equation Equation C.2 in Appendix C using the vector 2-norm given in Equation C.1. The average of the relative errors of the validation samples was calculated. Each point in the figure represents the average relative error for one fitting. Note that this average relative error is calculated before decoding and represents the error in the latent space.

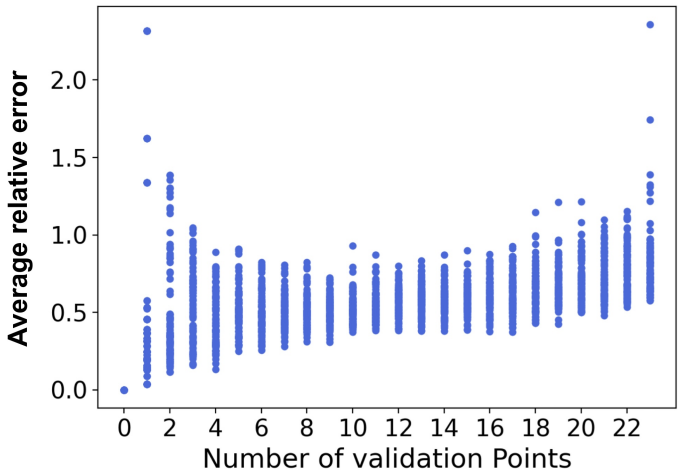


Figure 3.9: Each point represent the relative error between the encoded experimental CLD and prediction of the correction layer in the latent space for each fitting.

By inspecting the figure, the number of validation points in the range 5-17 proved to produce good fittings with regards to low relative error and low variance. In Figure 3.10, the average relative error of the 100 fittings between the decoded prediction of correction layer and the full space experimental CLD is shown. This relative average error will also take into account the error contribution from the autoencoder.

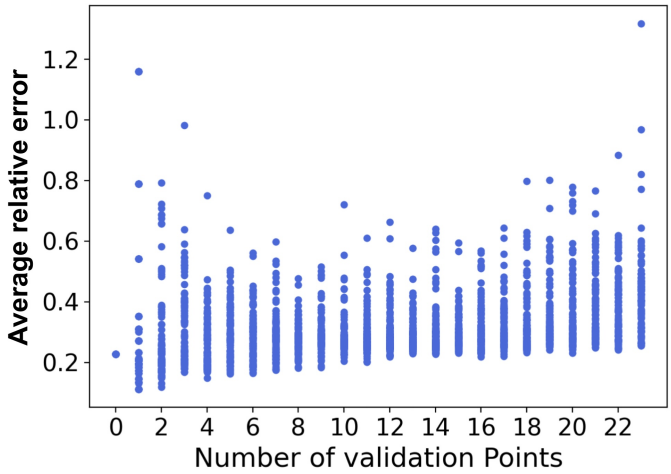


Figure 3.10: Each point represent the average relative error between the experimental CLD and the decoded prediction of the correction layer for each fitting.

The mean and the standard deviation of the average relative error of the 100 fittings for each of the different number of validation points are plotted in Figure 3.11. By inspection, 8 validation points yield a low mean error and low standard deviation. Thus, 8 experiments were left out during the fitting of model and used for validation.

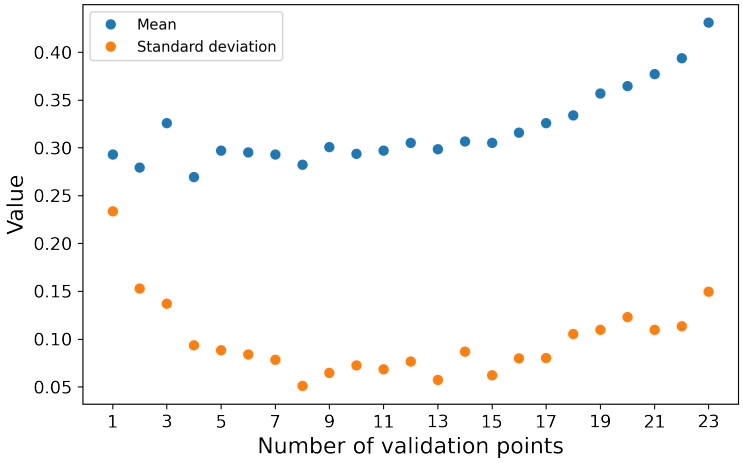


Figure 3.11: The figure shows the mean and the standard deviation of the error of the 100 fittings for the given number of validation points.

To get a final overview of the framework, a simple illustration is presented in Figure 3.12. It displays all the steps of the mapping of an experimentally generated PSD to a CLD. A PSD is the

input to the convolutional neural network. The CNN model makes a prediction of the CLD in the latent space. This prediciton along with two experimental conditions, is the input to the correction layer i.e the Gaussian process regression model. The prediction made by the correction layer is then decoded to a full space CLD by the decoder. The next section will present the performance of this final framework.

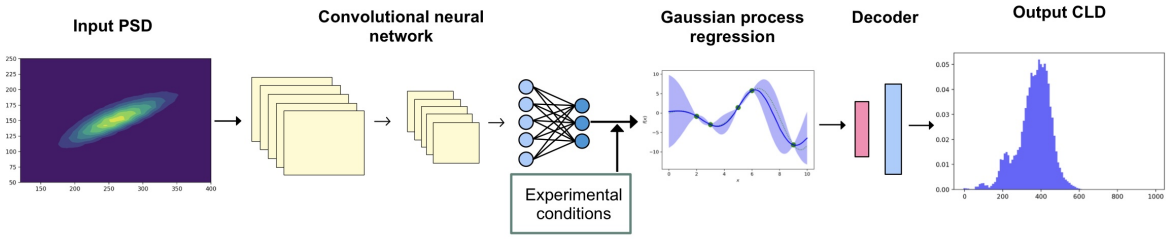


Figure 3.12: Simple illustration of final framework

3.2.3 Performance of the final framework

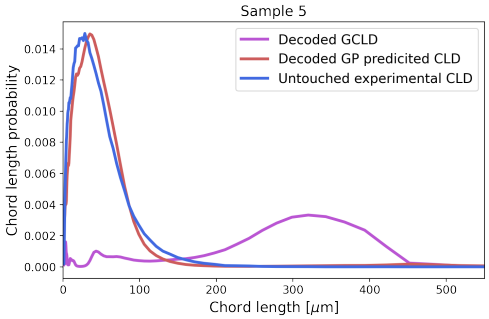
In this section, the performance of the final framework is presented. The fist section illustrates the performance of the validation samples with and without the correction layer. The second section presents the performance of the correction later and the autoencoder with the validation samples.

Performance of the correction layer

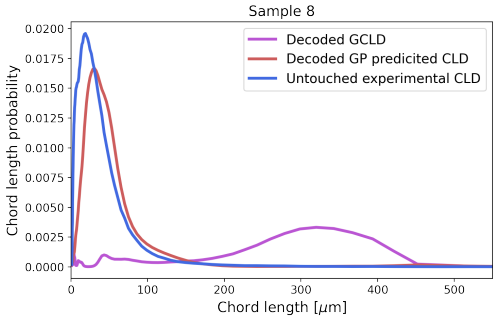
In Table 3.6 the relative error of the CLD prediction with and without the correction layer of each validation sample is presented. In Figure 3.13 the validation samples are plotted. Three lines are plotted. The untouched experimental CLD (blue line), the decoded prediction by the correction layer (red line) and the decoded prediction without the correction layer(purple line).

Table 3.6: Relative errors between the decoded prediction of the correction layer and the untouched experimental CLD for each validation sample. The active ingredient(AI) for each sample is also provided.

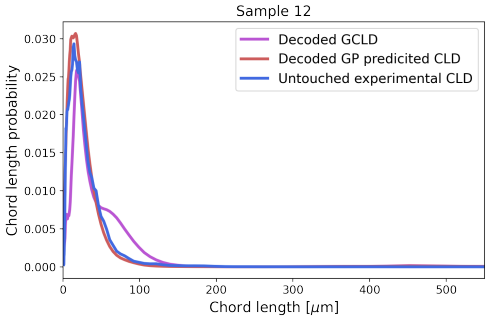
Sample	Without correction layer [%]	With correction layer [%]	AI
5	96.7	20.7	1
8	97.5	58.4	1
12	47.3	16.6	2
16	62.5	11.2	2
17	63.1	15.2	2
18	93.5	21.6	3
23	93.4	37.0	3
25	95.3	16.7	3



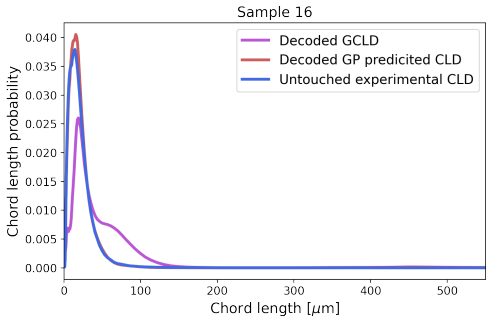
(a) Sample 5: RPM is 400 and suspension density is 3 % wt



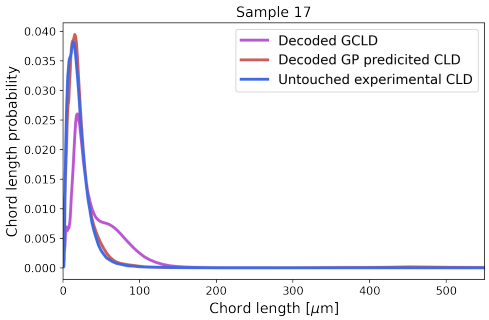
(b) Sample 8: RPM is 600 and suspension density is 9 % wt



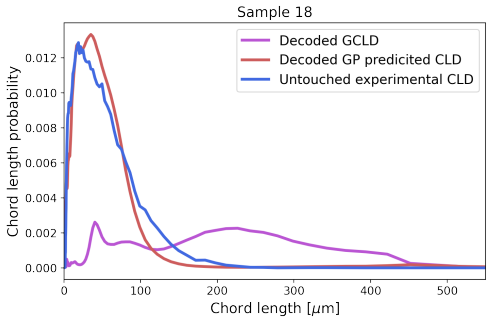
(c) Sample 12: RPM is 200 and suspension density is 3 % wt



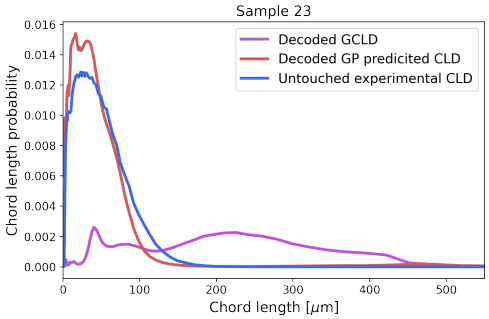
(d)
Sample 16: RPM is 400 and suspension density is 9 % wt



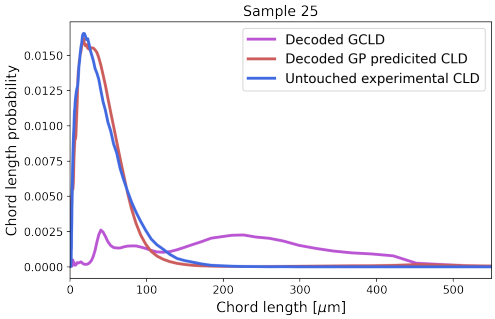
(e) Sample 17: RPM is 600 and suspension density is 9 % wt



(f) Sample 18: RPM is 400 and suspension density is 1 % wt



(g) Sample 23: RPM is 200 and suspension density is 9 % wt

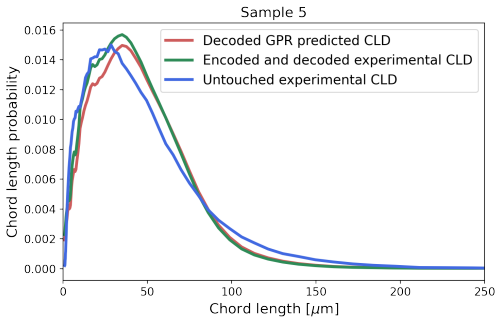


(h) Sample 25: RPM is 200 and suspension density is 9 % wt

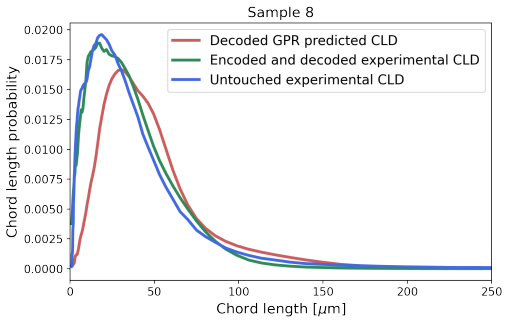
Figure 3.13: The untouched experimental CLD, the decoded GPR prediction and the decoded GCLD for each of the validation samples are plotted.

The correction layer performance and the autoencoder

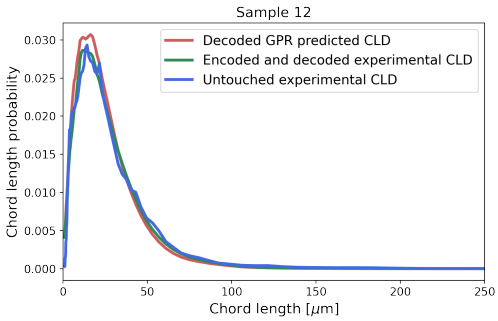
In Figure 3.14 the validation samples are plotted. Three lines are plotted. The untouched experimental CLD (blue line), the decoded prediction by the correction layer (red line) and the encoded and decoded experimental CLD(green line).



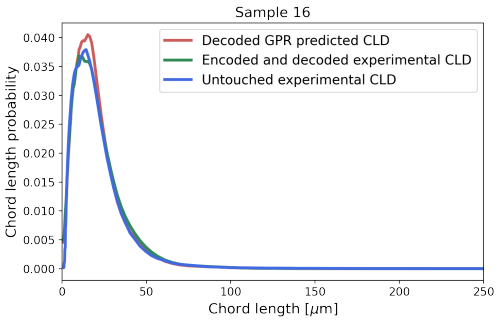
(a) Sample 5: RPM is 600 and suspension density is 3 % wt



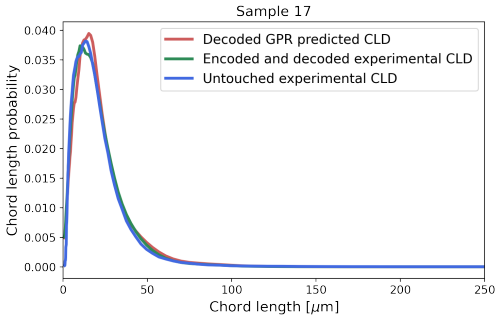
(b) Sample 8: RPM is 600 and suspension density is 9 % wt



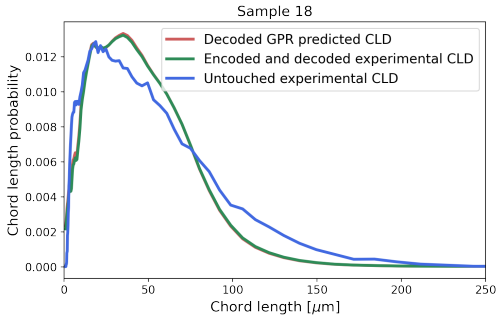
(c) Sample 12: RPM is 200 and suspension density is 3 % wt



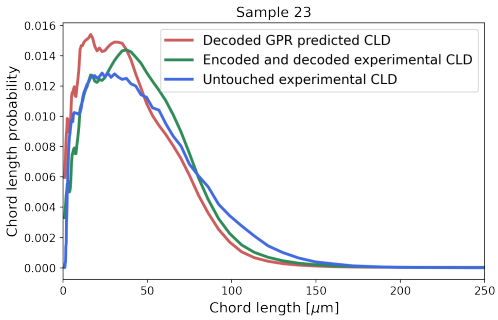
(d) Sample 16: RPM is 400 and suspension density is 9 % wt



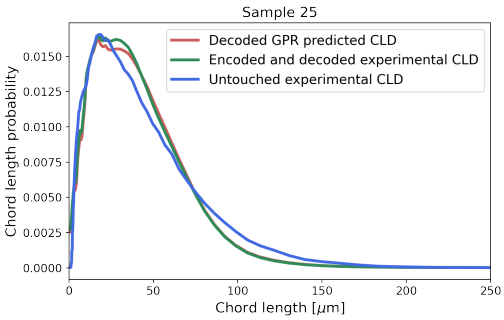
(e) Sample 17: RPM is 600 and suspension density is 9 % wt



(f) Sample 18: RPM is 400 and suspension density is 1 % wt



(g) Sample 23: RPM is 200 and suspension density is 9 % wt



(h) Sample 25: RPM is 600 and suspension density is 9 % wt

Figure 3.14: The untouched experimental CLD, the decoded GPR prediction and the encoded and decoded CLD for each of the validation samples are plotted.

3.2.4 Discussion

The performance of the final model with the eight randomly selected validation samples are presented in Figure 3.13. There are three lines plotted in each subfigure: The blue line represents the untouched experimental CLD, the red line represents decoded prediction by correction layer and the purple line represents the decoded GCLD ie. the prediction without the correction layer. By inspecting the subfigures, it is clear that the correction layer does a remarkable job correcting the predictions made by the convolutional neural network. The CNN model fails to predict the CLD for all samples except for sample 12, 16 and 17 which all are made with the active ingredient AI2. Though the performance of the CNN model is far from satisfactory for the samples with AI1 and AI3, the erroneous prediction is corrected by the Gaussian process regression model. The relative errors between the untouched experimental CLD (blue line) and the predictions with and without the correction layer are given in Table 3.6. The relative error with the correction layer for samples with AI2 are the lowest. Naturally, this is in line with what is expected as these samples already

had the smallest relative error without the correction layer as listed in the table. Despite the high relative error for the samples with AI1 and AI3 before the correction layer as seen in Figure 3.13 and Table 3.6, the correction layer is able to successfully correct the GCLDs to a satisfactory result. With the efficiency and ease of the performance of the correction layer, it seems that the independence assumption made in the modelling step, holds. Therefore, building a GPR model with the assumption that the there is no correlation between the output is sufficient in this case.

For each validation sample in Figure 3.14, the encoded and decoded experimental CLD is plotted (green line) along with the untouched experimental CLD (blue line) and the predicted CLD with the correction layer (red line). The predicted CLD with the correction layer will have three main sources of error: the CNN model prediction, the Gaussian process regression and from the dimensional reduction. In Figure 3.13 the error from the CNN model is clearly visible from the discrepancy between the untouched experimental CLD (blue line) and the decoded GCDL (purple line). Almost all the samples plotted in this figure display major deviation from the untouched experimental CLD. In Figure 3.14, the error due to the autoencoder and the grouped error of the CNN model and the correction layer are visualized. For example, in sample 8, the blue and the green line coincides relatively well and the red line deviates. This indicates that the error present is introduced from the CNN model/and or the correction layer. In sample 18 however, the green line and the red line coincides but differ from the blue line, which signifies that the error present stems the compression/reconstruction operation by the autoencoder. In sample 23, all the plotted curves differ slightly from each other which means that all three sources of error contributed to the total prediction error.

As seen in Figure 3.13 the CNN largely fails to make accurate predictions with the experimental data. To improve the CNN, one can attempt to generate more simulated data more similar to the experimental data. Moreover, the performance of CNN, GPR model and AE is directly related to the chosen structure of these machine learning models. Therefore, improved performance could possibly be attainable by changing these structures. Due to this fact, Bayesian optimization was introduced in order to find the optimal hyperparameters to define the optimal structure of the CNN. Although the Bayesian optimization is an effective way to tune the hyperparameters, it does not guarantee that the optimal hyperparameters are the universally the best hyperparameters. The obtained hyperparameters are ultimately still dependent on the available data, bounds on the range of the hyperparameters, the number of epochs and iterations. If the generation of more simulated data or structural changes could lead to more accurate predictions by the CNN, the framework can be used as a tool to asses the effect of experimental conditions on the measured experimental data. With the results presented, there is no way establish any solid connections between measurement and experimental conditions. Despite this limitation, the framework as whole is able to predict accurate CLDs from experimental PSD data, and can thus serve as a robust model to generate more data to develop a data-driven model tackling the inverse problem.

Chapter 4

Final evaluation and future work

While the monitoring and control of crystallization processes are important for the production of many products in the process industry, the lack of good in-situ and online measurements has proven to be a bottleneck in the advancement of such methods. Therefore, several attempts have been made to resolve this issue as product purity and quality are important for the performance and characteristics of the final crystalline product. The use of the FBRM probe is widely applied as it provides online and in-situ measurements. However, as the FBRM probe measures chord lengths instead of the actual particle size, various attempts have been made to relate the PSD to the CLD or vice versa. In this research, a CNN-based framework was developed to perform the mapping of the forward problem. The main objective of the research presented in this thesis was to adapt an existing CNN model trained with simulated data to be able to make predictions with experimental data. The existing CNN model was shown to be able to make satisfactory predictions with the simulated data, but failed to make predictions for the experimental data. This is not too surprising as the CNN model was only trained with simulated data. This indicates that the experimental dataset was too different for the model to be able to recognize the important features in the data. One possible solution to this issue is to enrich the dataset with either experimental data or more similar simulated data if possible. However, as the generation of new data may be time consuming and financially and/or computationally expensive another approach was suggested that does not require more data generation. Instead, a correction layer was implemented after the CNN prediction along with a dimensonality reduction of the problem. An autoencoder was implemented to perform the dimensionality reduction. The autoencoder was able to compress and reconstruct the CLDs from the simualted dataset and the experiemntal data set with relative ease. With this effective tool for dimensionality reduction, the CNN model structure was altered to accommodate the dimensionality reduction. The dimensionality reduction proved to enhance the performance of the adjusted CNN due to a simpler mapping. A multi-output Gaussian process regression was implemented to serve as the correction layer. The GPR model also used two experimental conditions as additional input data. The GPR model proved to be a powerful model to serve as a correction layer. Thus with the implementation of the correction layer and the dimensionality reduction, the final framework was able to accurately predict experimental PSDs to CLDs with as little as 26 datasets. This developed framework demonstrates how both simulated and experimental data can be leveraged to develop an accurate data-driven model for prediction without requiring substantial amount of experimental data. In addition, it also provides a robust model to generate enough data

to address the inverse problem (CLD to PSD) which is of greater interest.

Several possible improvements and directions for further research can be undertaken given these findings. With regards to improving the current framework there are several things that can be changed like enriching the simulated dataset such that the CNN is able to make better predictions. This can be interesting with regards to investigating which experimental conditions may or may not have greater influence on the measurements. Moreover, it would also be interesting to see if the framework could be adjusted to interpret crystal populations consisting of different shapes other than needle-like crystals like platelets or mixed shapes. In this research, the forward problem was addressed i.e mapping a PSD to a CLD. With this tool for generating PSD and CLD data, the next step further could be to develop a data-driven model for the inverse problem in a similar manner.

Bibliography

[1] Idan Achituve et al. "Personalized Federated Learning with Gaussian Processes". In: *arXiv* (2021). URL: https://doi.org/10.48550/arXiv.2106.15482.

[2] Abdulaziz Almalaq and George Edwards. "A Review of Deep Learning Methods Applied on Load Forecasting". In: *IEEE* (2017). URL: 10.1109/ICMLA.2017.0-110.

[3] Arc. *Convolutional Neural Network, An Introduction to Convolutional Neural Networks*. 2018. URL: https://towardsdatascience.com/convolutional-neural-network-17fb77e76c05 (visited on 10/25/2021).

[4] Shereena V. B. and Julie M. David. "SIGNIFICANCE OF DIMENSIONALITY REDUCTION IN IMAGE PROCESSING". In: *Signal Image Processing* 6 (2015). URL: 10.5121/sipij.2015.6303.

[5] Naim Bajcinca et al. "Optimal control solutions for crystal shape manipulation". In: *Computer Aided Chemical Engineering* (2010). URL: https://doi.org/10.1016/S1570-7946(10)28126-9.

[6] Sunitha Basodi et al. "Gradient Amplification: An Efficient Way to Train Deep Neural Networks". In: *BIG DATA MINING AND ANALYTICS* (2020). URL: https://doi.org/10.48550/arXiv.2006.10560.

[7] D. Binu and B.R. Rajakumar. *Artificial Intelligence in Data Mining. Deep learning methods for data classification*. Academic Press, 2021. URL: https://doi.org/10.1016/C2019-0-01255-1.

[8] A. Blanco et al. "Flocculation Monitoring: Focused Beam Reflectance Measurement as a Measurement Tool". In: *The Canadian Journal of Chemical Engineering* 80 (2002). URL: https://doi.org/10.1002/cjce.5450800403.

[9] H. H. J. Bloemen and M. G. M. De Kroon. "Transformation of Chord Length Distributions into Particle Size Distributions Using Least Squares Techniques". In: *Particulate Science and Technology* (2007). URL: https://doi.org/10.1080/02726350500212996.

[10] Eric Brochu, Vlad M. Cora, and Nando de Freitas. "A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning". In: *ArXiv* (2010). URL: https://doi.org/10.48550/arXiv.1012.2599.

[11] Jason Brownlee. *A Gentle Introduction to Dropout for Regularizing Deep Neural Networks*. 2018. URL: https://machinelearningmastery.com/dropout-for-regularizing-deep-neural-networks/.

[12] Jason Brownlee. *How to Implement Bayesian Optimization from Scratch in Python*. 2019. URL: https://machinelearningmastery.com/what-is-bayesian-optimization/ (visited on 11/29/2021).

[13] Jason Brownlee. *How to use Data Scaling Improve Deep Learning Model Stability and Performance*. 2020. URL: https://machinelearningmastery.com/how-to-improve-neural-network-stability-and-modeling-performance-with-data-scaling/ (visited on 11/14/2021).

[14] Jason Brownlee. *Hyperparameter Optimization With Random Search and Grid Search*. 2020. URL: https://machinelearningmastery.com/hyperparameter-optimization-with-random-search-and-grid-search/.

[15] Jason Brownlee. *Overfitting and Underfitting With Machine Learning Algorithms*. 2016. URL: https://machinelearningmastery.com/overfitting-and-underfitting-with-machine-learning-algorithms/.

[16] Van P. CAREY. "Lecture notes week 3, ME249-4F21". In: (2021).

[17] Sundaresh Chandran. *Significance of I.I.D in Machine Learning*. URL: https://medium.datadriveninvestor.com/significance-of-i-i-d-in-machine-learning-281da0d0cbef. (accessed: 16.06.2022).

[18] Dami Choi et al. "On Empirical Comparisons of Optimizers for Deep Learning". In: *arXiv* (2020).

[19] C.E. Crestani et al. "An artificial neural network model applied to convert sucrose chord length distributions into particle size distributions". In: *Powder Technology* (2021). URL: https://doi.org/10.1016/j.powtec.2021.01.075.

[20] Trevor Darrell et al. "Machine Learning with Interdependent and Non-identically Distributed Data". In: *Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik* (2015). URL: 10.4230/DagRep.5.4.18.

[21] Samuel J. Gershman and David M. Blei. "A tutorial on Bayesian nonparametric models". In: *Journal of Mathematical Psychology* (2012). URL: https://doi.org/10.1016/j.jmp.2011.08.004.

[22] R. Guardani, R.S. Onimaru, and F.C.A. Crespo. "Neural network model for the on-line monitoring of a crystallization process". In: *Brazilian Journal of Chemical Engineering* (2001). URL: https://doi.org/10.1590/S0104-66322001000300006.

[23] Jörg Heinrich and Joachim Ulrich. "Application of Laser-BackscatteringInstruments for In Situ Monitoring ofCrystallization Processes – A Review". In: *Chemical Engineering Technology* (2012). URL: 10.1002/ceat.20110034.

[24] Raimundo Ho et al. "Effect of Milling on Particle Shape and Surface Energy Heterogeneity of Needle-Shaped Crystals". In: *Pharmaceutical research* (2012). URL: https://doi.org/10.1007/s11095-012-0842-.

[25] Jianglin Huang, Yan-Fu Li, and Min Xie. "An empirical analysis of data preprocessing for machine learning-basedsoftware cost estimation". In: *Information and Software Technology* (2015). URL: 10.1016/j.infsof.2015.07.004.

[26] Roberto Irizarry, Akshaya Nataraj, and Jochen Schoell. "CLD-to-PSD model to predict bimodal distributions and changes in modality and particle morphology". In: *Elsevier* (2020).

[27] Roberto Irizarry et al. "Data-driven model and model paradigm to predict 1D and 2D particle size distribution from measured chord-length distribution". In: *Chemical Engineering Science* (2017). URL: https://doi.org/10.1016/j.ces.2017.01.042.

[28] JEREMY JORDAN. *Introduction to autoencoders*. 2018. URL: https://www.jeremyjordan.me/autoencoders/ (visited on 05/11/2022).

[29] Ozgur Demir Kavuk et al. "Prediction using step-wise L1, L2 regularization and feature selection for small data sets with large number of features". In: *BMC Bioinformatics* (2011).

[30] Minjeong Kim et al. *Biomedical Information Technology, second edition. Deep learning in biomedical image analysis*. Academic Press, 2020. URL: https://doi.org/10.1016/B978-0-12-816034-3.00008-0.

[31] Will Koehrsen. *A Conceptual Explanation of Bayesian Hyperparameter Optimization for Machine Learning*. 2018. URL: https://towardsdatascience.com/a-conceptual-explanation-of-bayesian-model-based-hyperparameter-optimization-for-machine-learning-b8172278050f.

[32] Dhairya Kumar. *Introduction to Data Preprocessing in Machine Learning*. 2018. URL: https://towardsdatascience.com/introduction-to-data-preprocessing-in-machine-learning-a9fa83a5dc9d (visited on 11/14/2021).

[33] Mingzhong Li and Derek Wilkinson. "Determination of non-spherical particle size distribution from chord length measurements. Part 1: Theoretical analysis". In: *Chemical Engineering Science* (2005). URL: https://doi.org/10.1016/j.ces.2005.01.008.

[34] Mingzhong Li, Derek Wilkinson, and Kumar Patchigolla. "Determination of non-spherical particle size distribution from chord length measurements. Part 2: Experimental validation". In: *Chemical Engineering Science* (2005). URL: https://doi.org/10.1016/j.ces.2005.04.019.

[35] Sehla Loussaief and Afef Abdelkrim. "Convolutional Neural Network Hyper-Parameters Optimization based on Genetic Algorithms". In: *International Journal of Advanced Computer Science and Applications* (2018).

[36] Hamdy F. F. Mahmoud. "Parametric versus Semi and Nonparametric Regression Models". In: *arXiv* (2019). URL: https://doi.org/10.48550/arXiv.1906.10221.

[37] Aniruddha Majumder and Zoltan K. Nagy. "Prediction and control of crystal shape distribution in the presence of crystal growth modifiers". In: *Computer Aided Chemical Engineering* (2013).

[38] Mark Meuller. *Lecture notes in MECENG 231B: Chapter 7, the optimal state estimator*. (accessed: 15.06.2022).

[39] Kizito Nyuytiymbiy. *Parameters and Hyperparameters in Machine Learning and Deep Learning*. 2020. URL: https://towardsdatascience.com/parameters-and-hyperparameters-aa609601a9ac (visited on 11/20/2021).

[40] Adam P.Piotrowski, Jaroslaw J. Napiorkowski, and Agnieszka E.Piotrowskab. "Impact of deep learning-based dropout on shallow neural networks applied to stream temperature modelling". In: *Earth-Science Reviews* (2020). URL: https://doi.org/10.1016/j.earscirev.2019.103076.

[41] Ajinkya V. Pandit and Vivek V. Ranade. "Chord Length Distribution to Particle Size Distribution". In: *AIChE Journal* (2016). URL: https://doi.org/10.1002/aic.15338.

[42] Daniel B. Patience and James B. Rawlings. "Particle-Shape Monitoring and Control in Crystallization Processes". In: *AIChE Journal* (2001). URL: h10.1002/aic.690470922.

[43] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning. Chapter 2: Regression*. The MIT Press, 2005. URL: https://doi.org/10.7551/mitpress/3206.001.0001.

[44] Sumit Saha. *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way*. 2018. URL: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53 (visited on 10/24/2021).

[45] Jochen Schoell et al. "Determining particle-size distributions from chord length measurements for different particle morphologies". In: *AIChE Journal* (2019). URL: https://doi.org/10.1002/aic.16560.

[46] Eric Schulz, Maarten Speekenbrink, and Andreas Krause. "A tutorial on Gaussian process regression: Modelling, exploring, and exploiting functions". In: *Journal of Mathematical Psychology* (2018). URL: https://doi.org/10.1016/j.jmp.2018.03.001.

[47] Bobak Shahriar et al. "Taking the Human Out of the Loop: A Review of Bayesian Optimization". In: *IEEE* 104 (2016). URL: 10.1109/JPROC.2015.2494218.

[48] Sagar Sharma, Simone Sharma, and Anidhya Athaiya. "Activation functions in neural networks". In: *towards data science* 6.12 (2017), pp. 310–316.

[49] SympaTec. *The universal shapefinder for particle characterisation ranging from below 1 µm to 34,000 µm*. URL: https://www.sympatec.com/en/particle-measurement/sensors/dynamic-image-analysis/qicpic/.

[50] Sympatec. *Dynamic Image Analysis*. URL: https://www.sympatec.com/en/particle-measurement/sensors/dynamic-image-analysis/.

[51] Botond Szilágyi and Zoltán K. Nagy. "Aspect Ratio Distribution and Chord Length Distribution Driven Modeling of Crystallization of Two-Dimensional Crystals for Real-Time Model-Based Applications". In: *Crystal Growth Design 18 (9)* (2018). URL: 10.1021/acs.cgd.8b00758.

[52] Rikiya Yamashita Mizuho Nishio Richard Kinh Gian Do Kaori Togashi. "Convolutional neural networks: an overview and application in radiology". In: *Springer* (2018).

[53] Mettler Toledo. *FBRM Method of Measurement*. URL: https://www.mt.com/us/en/home/library/videos/automated-reactors/Lasentec-FBRM-Method-of-Measurement.html.

[54] Mettler Toledo. *ParticleTrack G400*. URL: https://www.mt.com/us/en/home/products/L1_AutochemProducts/FBRM-PVM-Particle-System-Characterization/FBRM/fbrm-g400.html.

[55] Jonathan Tompson et al. "Efficient Object Localization Using Convolutional Networks". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2015.

[56] Neelam Tyagi. *L2 and L1 Regularization in Machine Learning*. 2021. URL: https://www.analyticssteps.com/blogs/l2-and-l1-regularization-machine-learning.

[57] Jie Wang. "An Intuitive Tutorial to Gaussian Processes Regression". In: *arXiv* (2022). URL: https://doi.org/10.48550/arXiv.2009.10862.

[58] Jia Wu et al. "Hyperparameter Optimization for Machine Learning Models Based on Bayesian Optimization". In: *JOURNAL OF ELECTRONIC SCIENCE AND TECHNOLOGY* 17 (2019). URL: https://doi.org/10.11989/JEST.1674-862X.80904120.

[59] E.J.W Wynn. "Relationship between particle-size and chord-length distributions in focused beam reflectance measurement: Stability of direct inversion and weighting". In: *Powder Technology* 133 (July 2003), pp. 125–133. URL: 10.1016/S0032-5910(03)00084-6.

[60] Liang Yan et al. "Bayesian Optimization Based on K-Optimality". In: *Entropy* (2018). URL: 10.3390/e20080594.

[61] Xiaochuan Yang et al. "Risk Considerations on Developing a Continuous Crystallization System for Carbamazepine". In: *Organic Process Research Development* (2017). URL: https://doi.org/10.1021/acs.oprd.7b00130.

[62] Weili Yu et al. "What is the "typical" particle shape of active pharmaceutical ingredients?" In: *Powder Technology* (2017). URL: https://doi.org/10.1016/j.powtec.2017.02.043.

[63] Zai Qun Yu, Pui Shan Chow, and Reginald B. H. Tan. "Interpretation of Focused Beam Reflectance Measurement (FBRM) Data via Simulated Crystallization". In: *Organic Process Research Development* 12 (2008). URL: https://doi.org/10.1002/cjce.5450800403.

[64] Andrea Caponnetto Yuan Yao Lorenzo Rosasco. "On Early Stopping in Gradient Descent Learning". In: *Constructive Approximation 26* (2007).

Appendix A

Existing model

A.1 Derivation of the conditional distribution of multivariate normal distributions

The derivation of conditional distributions of multivariate normal distributions is provided according to the derivation in [38]. Let $x \in \mathbb{R}^{N_x}$ and $y \in \mathbb{R}^{N_y}$ be two jointly Gaussian random variable such that ζ =(x,y) $\in \mathbb{R}^{N_x+N_y}$ is a Gaussian random variable with $\zeta \sim \mathcal{N}(\mu_\zeta, \Sigma_{\zeta\zeta})$ and

$$\mu_\zeta = \begin{bmatrix} \mu_x \\ \mu_y \end{bmatrix} \text{ and } \Sigma_{\zeta\zeta} = \begin{bmatrix} \Sigma_{xx} & \Sigma_{xy} \\ \Sigma_{xy}^T & \Sigma_{yy} \end{bmatrix} \tag{A.1}$$

The next step is to show that x is conditionally Gaussian, that is f(x|y) is a Gaussian distribution

$$\begin{aligned} f_{x|y}(x \mid \bar{y}) =& \frac{f_{x,y}(x,\bar{y})}{f_y(\bar{y})} = \frac{f_\zeta((x,\bar{y}))}{f_y(\bar{y})} \\ =& \frac{1}{(2\pi)^{(N_x+N_y)/2} \det\left(\Sigma_{\zeta\zeta}\right)^{1/2}} \exp\left(-\frac{1}{2}\left(\begin{bmatrix} x \\ \bar{y} \end{bmatrix} - \mu_\zeta\right)^T \Sigma_{\zeta\zeta}^{-1}\left(\begin{bmatrix} x \\ \bar{y} \end{bmatrix} - \mu_\zeta\right)\right) \\ & \left(\frac{1}{(2\pi)^{N_y/2} \det\left(\Sigma_{yy}\right)^{1/2}} \exp\left(-\frac{1}{2}\left(\bar{y}-\mu_y\right)^T \Sigma_{yy}^{-1}\left(\bar{y}-\mu_y\right)\right)\right)^{-1} \\ & \propto \exp\left(-\frac{1}{2}\left(\begin{bmatrix} x \\ \bar{y} \end{bmatrix} - \mu_\zeta\right)^T \Sigma_{\zeta\zeta}^{-1}\left(\begin{bmatrix} x \\ \bar{y} \end{bmatrix} - \mu_\zeta\right)\right) \\ =& \exp\left(-\frac{1}{2}\begin{bmatrix} x-\mu_x \\ \bar{y}-\mu_y \end{bmatrix}^T \begin{bmatrix} \Sigma_{xx} & \Sigma_{xy} \\ \Sigma_{xy}^T & \Sigma_{yy} \end{bmatrix}^{-1} \begin{bmatrix} x-\mu_x \\ \bar{y}-\mu_y \end{bmatrix}\right) \end{aligned} \tag{A.2}$$

here $\bar{y}$ is treated as a known constant. The result is quadratic in x and has a functional form of the Gaussian. x conditioned on y is thus a Gaussian random variable. In this expression, the

proportionality constant is ignored, but it will be whatever value needed for the integrated PDF to sum to 1. Now the mean and the variance of f(x|y) can be found such that,

$$f_{x|y}(x \mid \bar{y}) \propto \exp\left(-\frac{1}{2}(x-\bar{\mu})^T \bar{\Sigma}^{-1}(x-\bar{\mu})\right) \tag{A.3}$$

The inverse of the matrix $\Sigma_{\zeta\zeta}$ is given below,

$$\begin{bmatrix} \Sigma_{xx} & \Sigma_{xy} \\ \Sigma_{xy}^T & \Sigma_{yy} \end{bmatrix}^{-1} = \begin{bmatrix} \left(\Sigma_{xx} - \Sigma_{xy}\Sigma_{yy}^{-1}\Sigma_{xy}^T\right)^{-1} & -\Sigma_{xx}^{-1}\Sigma_{xy}\left(\Sigma_{yy} - \Sigma_{xy}^T\Sigma_{xx}^{-1}\Sigma_{xy}\right)^{-1} \\ -\left(\Sigma_{yy} - \Sigma_{xy}^T\Sigma_{xx}^{-1}\Sigma_{xy}\right)^{-1}\Sigma_{xy}^T\Sigma_{xx}^{-1} & \left(\Sigma_{yy} - \Sigma_{xy}^T\Sigma_{xx}^{-1}\Sigma_{xy}\right)^{-1} \end{bmatrix}$$

The variance $\bar{\Sigma}$ can be determined from inspection by comparing Equation A.2 and Equation A.3. $\bar{\mu}$ and $\bar{\Sigma}$ are computed such that the first and second order terms in x are equal. Note that the terms that do not contain x are ignored as these are simply constants. This yields,

$$\bar{\Sigma} = \Sigma_{xx} - \Sigma_{xy}\Sigma_{yy}^{-1}\Sigma_{xx}^T \tag{A.4}$$

Comparing first-order terms in x gives,

$$x^T\bar{\Sigma}^{-1}\bar{\mu} = x^T\bar{\Sigma}^{-1}\mu_x + x^T\Sigma_{xx}^{-1}\Sigma_{xy}\left(\Sigma_{yy} - \Sigma_{xy}^T\Sigma_{xx}^{-1}\Sigma_{xy}\right)^{-1}\left(\bar{y} - \mu_y\right) \tag{A.5}$$

which must hold for all x. With some manipulation,

$$\bar{\mu} = \mu_x + \Sigma_{xy}\Sigma_{yy}^{-1}\left(\bar{y} - \mu_y\right) \tag{A.6}$$

A.2 Experimental data summary

Table A.1: Summary of experiments

Sample	Active ingredient	RPM	Mass suspension density [% wt]
0	AI1	200	1
1	AI1	400	1
2	AI1	600	1
3	AI1	200	3
4	AI1	400	3
5	AI1	600	3
6	AI1	200	9
7	AI1	400	9
8	AI1	600	9
9	AI2	200	1
10	AI2	400	1
11	AI2	600	1
12	AI2	200	3
13	AI2	400	3
14	AI2	600	3
15	AI2	200	9
16	AI2	400	9
17	AI2	600	9
18	AI3	400	1
19	AI3	600	1
20	AI3	200	3
21	AI3	400	3
22	AI3	600	3
23	AI3	200	9
24	AI3	400	9
25	AI3	600	9

A.3 List of software

Table A.2: The table displays the software and python libraries used in this research along with the respective version.

Software	Version
Python	3.7.11
Scipy	1.7.3
Tensorflow	2.9.1
Keras	2.9.0
Pandas	1.3.4
Matplotlib	3.5.1
Scipy	1.7.3
Seaborn	0.11.2
Plotly	5.8.0
Pyny3d	0.1.1
Scikit-learn	1.0.2
Scikit-optimize	0.9.0

A.4 Plots from the Bayesian optimization

Figure A.1 shows the score of each iteration in the Bayesian optimization.

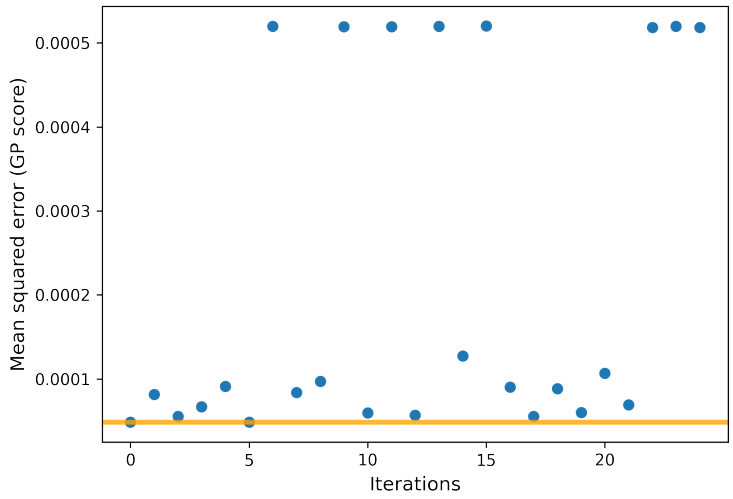


Figure A.1: Displays the mean absolute error for each iteration in the Bayesian optimization for the model without regularization.

The Figure A.2 displays the training and validation loss for each epoch of the training of the CNN model.

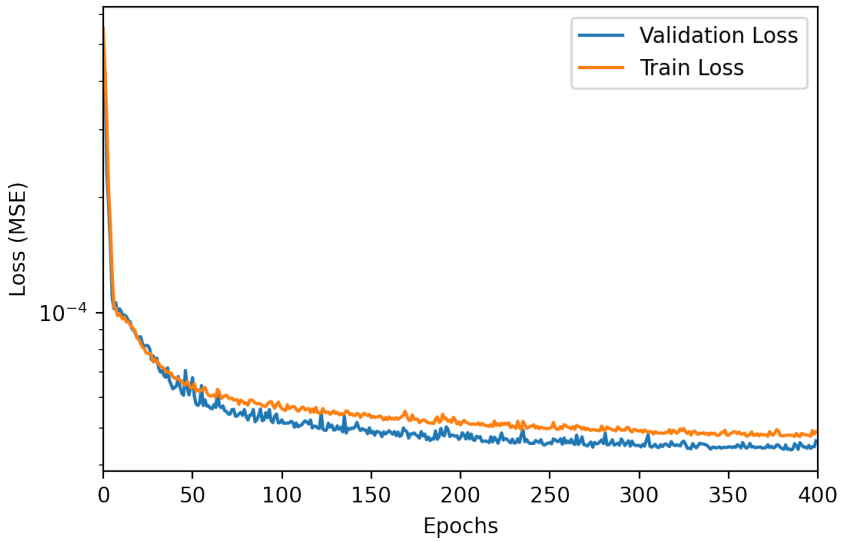


Figure A.2: The figure illustrates the training and validation loss of the CNN across epochs.

Appendix B

Dimensionality reduction

B.1 Training of the autoencoder

The Figure B.1 displays the training and validation loss in each epoch of the training of the autoencoder.

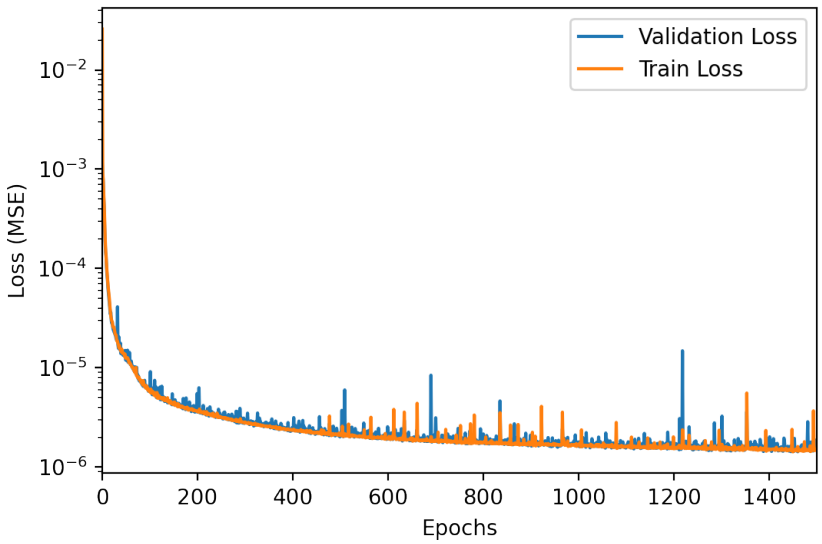


Figure B.1: The figure illustrates the training and validation loss of the autoencoder across epochs.

B.2 Training of the CNN with the CLDs in the dataset in the reduced space

The Figure B.2 shows the score of each iteration in the Bayesian optimization.

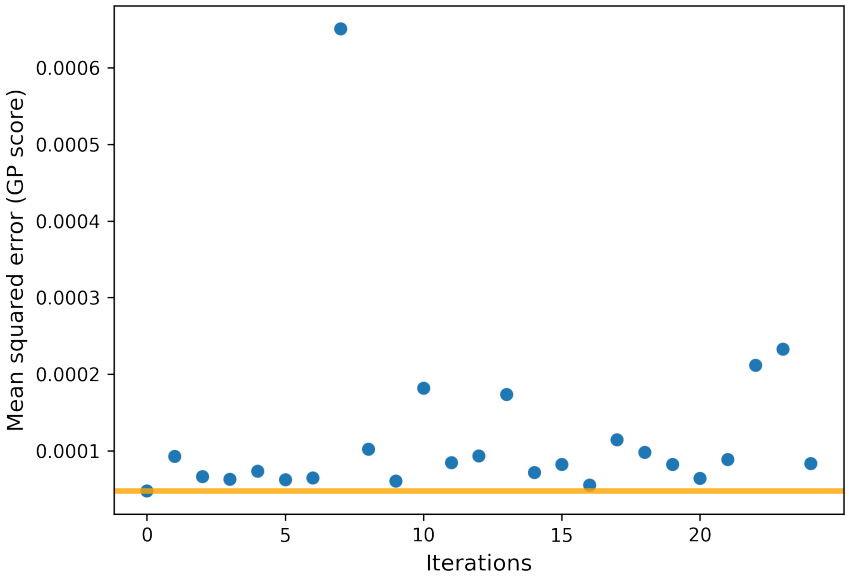


Figure B.2: The score of each iteration of the Gaussian process regression.

The Figure B.3 shows the training error and validation error in each epoch of the training of the CNN model with the CLDs in dataset in the reduced space.

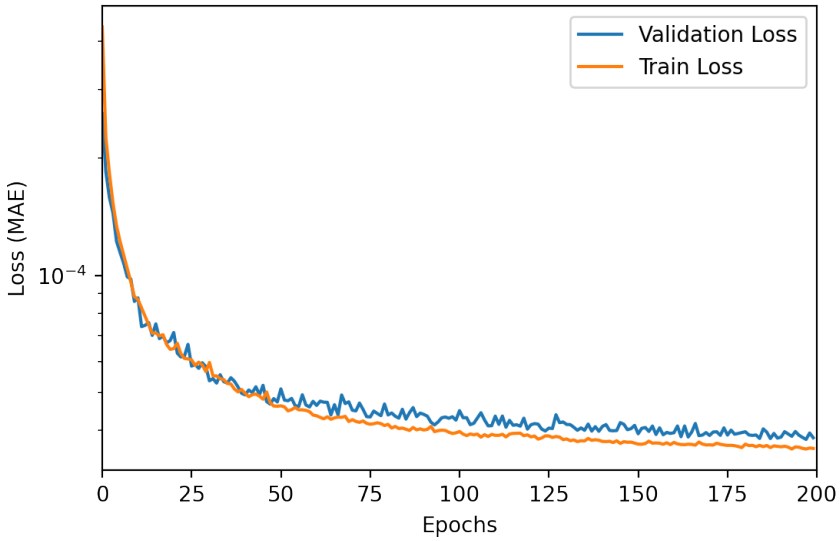
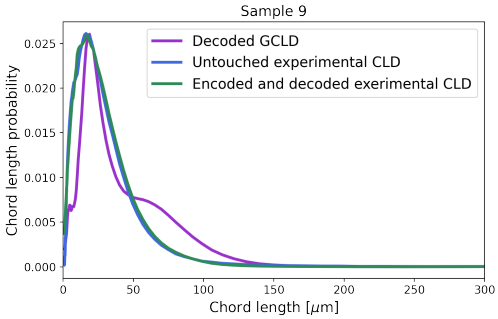


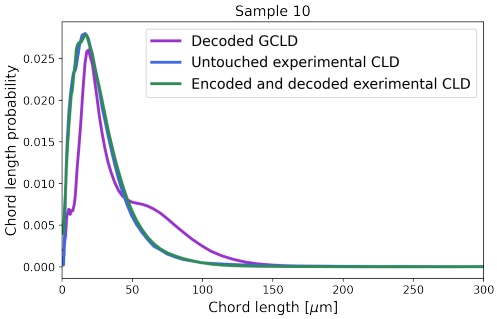
Figure B.3: The figure illustrates the training and validation loss of the CNN across epochs.

B.3 Samples with the active ingredient AI2

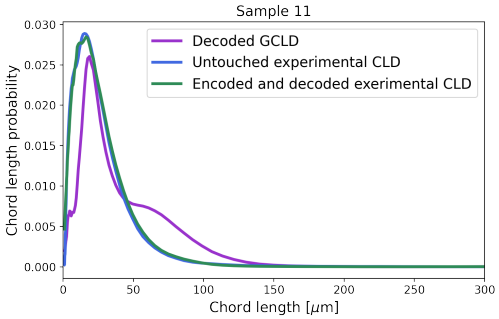
Figure B.4, Figure B.5 and Figure B.6 the untouched experimental CLD, the encoded and decoded CLD and the decoded GCLD of all the samples with active ingredient AI2.



(a) Sample 9: RPM is 200 and suspension density is 1

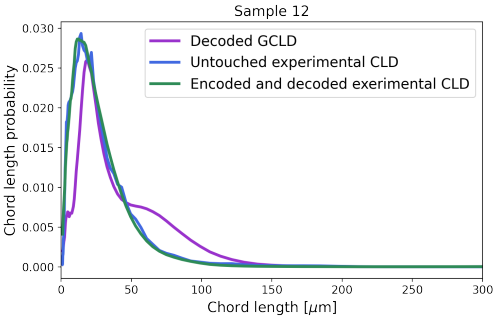


(b) Sample 10: RPM is 400 and suspension density is 1

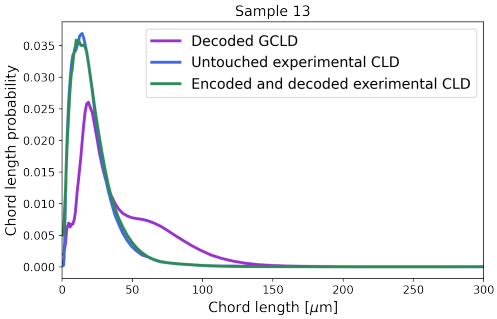


(c) Sample 11: RPM is 600 and suspension density is 1

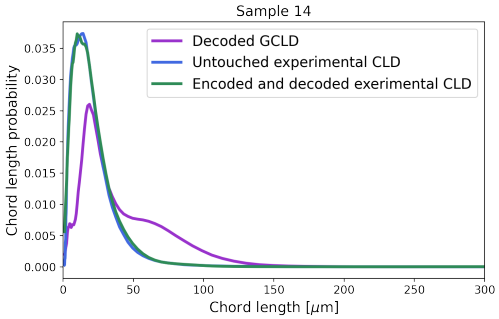
Figure B.4: The untouched experimental CLD, the encoded and decoded CLD and the decoded GCLD of all the sample 9, 10 and 11.



(a) Sample 12: RPM is 200 and suspension density is 3

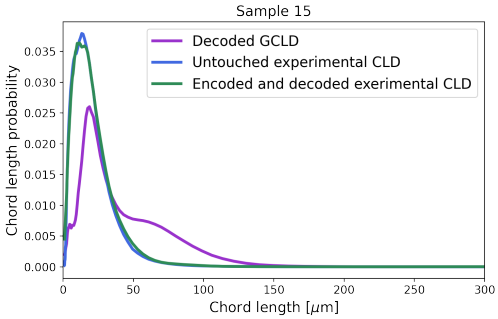


(b) Sample 13: RPM is 400 and suspension density is 3

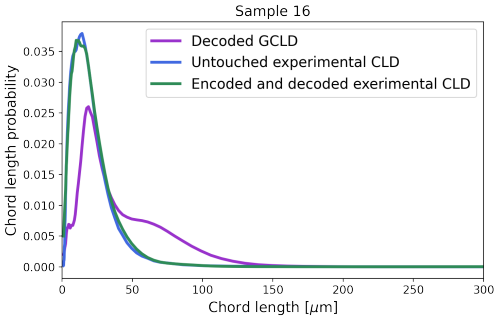


(c) Sample 14: RPM is 600 and suspension density is 3

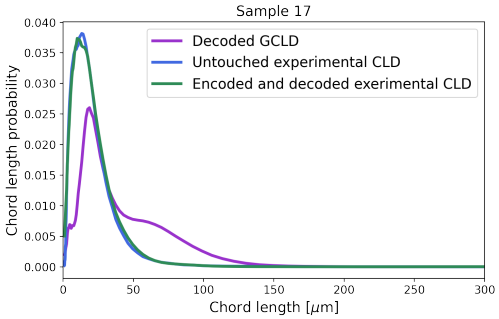
Figure B.5: The untouched experimental CLD, the encoded and decoded CLD and the decoded GCLD of all the sample 12, 13 and 14.



(a) Sample 15: RPM is 200 and suspension density is 9



(b) Sample 16: RPM is 400 and suspension density is 9



(c) Sample 17: RPM is 600 and suspension density is 9

Figure B.6: The untouched experimental CLD, the encoded and decoded CLD and the decoded GCLD of all the sample 15, 16 and 17.

Appendix C

Correction layer

C.1 Error calculations

The equation for the 2-norm is given in Equation C.1.

$$\|\mathbf{x}\|_2 = (\Sigma_{i=1}^{n} |x_i|^2)^{\frac{1}{2}} \tag{C.1}$$

The expression for the relative error calcualtion is given in Equation C.2

$$\text{relative error} = \frac{\|\alpha - \hat{\alpha}\|_2}{\|\alpha\|_2} \tag{C.2}$$

Here, α is the true value and $\hat{\alpha}$ is the predicted value.

Appendix D

Python code

D.1 Code for the implementation of the autoencoder, CNN model and tuning of hyperparameters with Bayesian optimization

[ ]:
```
"""
This code implements the autoencoder, CNN model and the Bayesian tuning of␣
↪hyperparameters. The structure of\
the code is as follows:
0. Import the simulated data and generate gamma distributions to augment the␣
↪data set used in the training\
of the autoencoder
1. Define and train Autoencoder + plots illustration performance
2. Define CNN model
3. Define Bayesian optimization model and perform the optimization to find the␣
↪optimal hyperparameters
4. Train CNN
5. Make predictions and plot performance
6. Save CNN model and autoencoder
Contributors: Solveig Sannes and George Makrygiorgos. Spring 2022
"""
```

[ ]:
```
#import packages
from __future__ import print_function
import warnings
warnings.filterwarnings('ignore')

import numpy as np
import scipy
import sys
import copy
from scipy.integrate import quad
from sklearn import preprocessing
from sklearn.utils import shuffle
from sklearn.model_selection import train_test_split

#For plotting
import matplotlib.pyplot as plt
%matplotlib inline

import sklearn
# suppress tensorflow compilation warnings
import os
import tensorflow as tf
from tensorflow.keras import backend as K
```

```
from tensorflow.python.framework import ops
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
seed=0
np.random.seed(seed) # fix random seed

import skopt
import pandas as pd
from skopt import gbrt_minimize, gp_minimize
from skopt.utils import use_named_args
from skopt.space import Real, Categorical, Integer
from tensorflow.keras.utils import plot_model


from  tensorflow.keras.models import Sequential
from  tensorflow.keras import optimizers
from  tensorflow.keras import losses
from  tensorflow.keras.layers import Dense, Dropout, Flatten, Conv2D,␣
↪MaxPool2D, Activation, LeakyReLU
from tensorflow.keras import utils
from tensorflow.python.keras.models import load_model

from tensorflow.keras.wrappers.scikit_learn import KerasRegressor
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
```

Data Preprocess

Load the initial data from simulated dataset experiments

[ ]:
```
Needle_Data = np.load('GM_combined_needle_train.npz')
```

[ ]:
```
Xraw= Needle_Data['Xtrain']
Yraw = Needle_Data['Ytrain'] #Yraw is before the dimensionality reduction
```

[ ]:
```
X = Xraw.copy()
```

[ ]:
```
#plotting simulated data to check if the data scaled correctly
sample = 110

l1 =X[sample,:,0,0]#yaxis
l2 = X[sample,0,:,1] #xaxis
Z = X[sample,:,:,2].T #frequency
x, y =np.meshgrid(l1, l2)
```

```
plt.figure(dpi=200)
plt.contourf(x, y, Z)
plt.colorbar()
plt.xlabel('L1', fontsize=15)
plt.ylabel('L2', fontsize=15)
plt.show()
```

Create *Gamma* Distributions similar to the experimental ones

[ ]:
```
#import chord lenthgs
ch_lengths = np.loadtxt('ch_len.csv')
```

[ ]:
```
from scipy import stats
def gamma_sample(x,a,b):
    y = stats.gamma.pdf(x, a, scale = 1/b)
    return y
```

[ ]:
```
gvals = gamma_sample(ch_lengths,3, 0.03)
plt.plot(ch_lengths, gvals)
plt.xlim([0,150])
```

[ ]:
```
#create and arry with N*N number of gamma distributions
N = 100
#creating values for the parameters a and beta
a_val = np.linspace(1.5,3.2, N)
scale_val = np.linspace(0.03,0.14,N)

sample_dist = np.zeros((N,N,len(ch_lengths))) #creating vector to store gamma␣
↪distributions

#generating gamma distribtuions with different values for a and beta within the␣
↪set ranges
for i in range(N):
    a = a_val[i]
    for j in range(N):
        scale = scale_val[j]
        x = ch_lengths
        sample_dist[i][j] = gamma_sample(x,a,scale)
#reshaping
simulated_dist = sample_dist.reshape(N*N,100)
```

[ ]:
```
#plot some samples
plt.figure(dpi=300)
indices = [0,100,1000,2000,3000]
for j in range(5):
```

```
    plt.plot(ch_lengths, simulated_dist[indices[j]])

plt.xlabel('x', fontsize=15)
plt.ylabel('f(x)',fontsize=15)
plt.savefig('gammadist')
```

[ ]:
```
#add simualted gamma distributions to simualted CLDs
YrawAugmented = np.concatenate((Yraw,simulated_dist), axis=0)
```

Implementing the autoencoder

[ ]:
```
#defining the autoencoder model
def autoencoder_model(Yraw,ldim):
    """
    The NN-based autoencoder of this
    section* is hardocoded in the function
    Input: Data (Nsamples*Features)
    Output: Encoder, Decoder, Autoencoder
    """
    #defining the training and validation set
    X_train,X_test,Y_train,Y_test =␣
 ↪train_test_split(Yraw,Yraw,shuffle=True,train_size=0.8,random_state=42)

    odim = Yraw.shape[1]
    nbins = odim
    encoding_dim = ldim

    #defining the structure

    actf = 'tanh'
    input_cld = tf.keras.layers.Input(shape=(nbins,))

    encoded0 = tf.keras.layers.Dense(25, activation=actf,
                                     activity_regularizer=tf.keras.regularizers.
 ↪l2(10e-5))(input_cld)
    encoded = tf.keras.layers.Dense(encoding_dim, activation=actf,
                                     activity_regularizer=tf.keras.regularizers.
 ↪l2(10e-5))(encoded0)

    encoder = tf.keras.models.Model(input_cld, encoded, name="encoder_model")

    decoder_input = tf.keras.layers.Input(shape=(ldim), name="decoder_input")
    decoded = tf.keras.layers.Dense(25, activation=actf)(decoder_input)
    decoded = tf.keras.layers.Dense(nbins, activation='sigmoid')(decoded)
```

```
    decoder = tf.keras.models.Model(decoder_input, decoded,␣
↪name="decoder_model")

    ae_input = tf.keras.layers.Input(shape=(nbins,), name="AE_input")
    ae_encoder_output = encoder(ae_input)
    ae_decoder_output = decoder(ae_encoder_output)
    autoencoder = tf.keras.models.Model(ae_input, ae_decoder_output, name="AE")

    autoencoder.compile(optimizer='adam', loss='mean_squared_error')
    autoencoder.summary()
    autoencoder.fit(X_train, X_train,
                    epochs=1500,
                    batch_size=30,
                    shuffle=True,
                    validation_data=(X_test, X_test))


    return autoencoder,encoder,decoder,X_test
```

Define Encoding Dimension and Train AE

[ ]:
```
#defining encoding dim
encoding_dimension = 5;

#obtain autoencoder
autoencoder, encoder, decoder, testSet=␣
↪autoencoder_model(YrawAugmented,encoding_dimension)
#save autoencoder
autoencoder.save('autoencoder.h5')
```

[ ]:
```
#plotting training and validation loss across epochs
plt.figure(3,dpi=200)
plt.semilogy(autoencoder.history.history['val_loss'])
plt.semilogy(autoencoder.history.history['loss'])
plt.xlabel('Epochs')
plt.ylabel('Loss (MSE)')
plt.legend(['Validation Loss','Train Loss'])
plt.xlim([0,1500])
plt.savefig('training_loss_AE')
```

Encode simulated CLDs

[ ]:
```
#load model and encode CLDs
autoencoder = load_model('autoencoder.h5')
encoder = autoencoder.layers[1]
decoder = autoencoder.layers[2]
encoder.summary()
Y=encoder(Yraw).numpy()
```

Plot some samples to show performance of autoencoder

[ ]:
```
#samples of simualted data
x=np.linspace(-1,3,100)
x=10**x
sample = [200,700,1500,2000,3000,4000]
enDeSim = np.zeros((6,100))
for i in range(len(sample)):
    encoded= encoder(testSet[sample[i]].reshape(1,100)).numpy()
    enDeSim[i] = decoder(encoded).numpy()
```

[ ]:
```
for i in range(len(sample)):
    plt.figure(figsize=[8,5],dpi=200)
    l= 'Sample '+ str(sample[i])
    xrange=[800,200,200,700,150,75]
    plt.title(l)
    plt.plot(x,testSet[sample[i]], color='royalblue', label ='Untouched␣
 ↪simulated CLD', linewidth=4)
    plt.plot(x,enDeSim[i], color='seagreen',label='Encoded and decoded␣
 ↪simulated CLD',linewidth=3)
    plt.xlim([0,xrange[i]])
    plt.xlabel('Chord length [$\mu$m]',fontsize=15)
    plt.ylabel('Chord length probability', fontsize=15)
    plt.xticks(fontsize=12)
    plt.yticks(fontsize=12)
    plt.yticks(fontsize=12)
    plt.legend(fontsize=15)
    plt.savefig('resultsAE/Sample'+ str(sample[i]))
```

[ ]:
```
#samples of experimental data
expCLD = np.load('experimental_CLD.npy')
enDe = np.zeros((26,100))
for i in range(26):
    encoded= encoder(expCLD[i].reshape(1,100)).numpy()
    enDe[i] = decoder(encoded).numpy()
```

[ ]:
```
for i in range(0,26,5):
    plt.figure(figsize=[8,5],dpi=200)
    l= 'Sample '+ str(i)
    plt.title(l)
    plt.plot(ch_lengths,expCLD[i], color='royalblue', label ='Untoucheced␣
↪experimental CLD',linewidth=4)
    plt.plot(ch_lengths,enDe[i], color='seagreen',label='Encoded and decoded␣
↪exerimental CLD', linewidth=3)
    plt.xlim([0,350])
    plt.xlabel('Chord length [$\mu$m]',fontsize=15)
    plt.ylabel('Chord length probability', fontsize=15)
    plt.xticks(fontsize=12)
    plt.yticks(fontsize=12)
    plt.legend(fontsize=14)
    plt.savefig('resultsAE/ExpSample'+ str(i))
```

Define CNN Model

[ ]:
```
#Define DNN model structure
from tensorflow.python.keras.layers import advanced_activations #Needed to add␣
↪our custom activation
def get_model(latent_dim, qval,activation,learning_rate,
              dropout_rate,filters1, filters2, filters3,trainable,training):
    """
    Inputs: latent_dim (latent dimensionality of CLD)
            activation (activation function)
            learning_rate
            dropout rate
            filters(1-3)
            trainable (True: the layers are trained - has to do with Transfer␣
↪Learning)
            training (Has to do with whether I am passing a previous model's␣
↪parameters)

    Output: CNN Model
    """
    trainable=True

    model = Sequential()

    model.add(Conv2D(filters1, (5, 5), strides=( 2, 2), padding='same',␣
↪input_shape=(30, 300, 5) ,trainable=trainable))
    model.add(Activation(activation))
    model.add(MaxPool2D((2, 2), padding='same'))
    model.add(Dropout(dropout_rate))
```

```
    model.add(Conv2D(filters2, (3, 3), strides=( 2, 2 ),␣
↪padding='same',trainable=trainable))
    model.add(Activation(activation))
    model.add(MaxPool2D((2, 2), padding='same'))
    model.add(Dropout(dropout_rate))

    model.add(Conv2D(filters3, (3, 3), strides=( 2, 2 ),␣
↪padding='same',trainable=trainable))
    model.add(Activation(activation))
    model.add(MaxPool2D((2, 2), padding='same'))
    model.add(Dropout(dropout_rate))
    model.add(Flatten())
    model.add(Dense(latent_dim, activation= 'linear' ,trainable=True))

    optimizer = optimizers.Adam(lr=learning_rate,amsgrad=True)
    model.compile(optimizer=optimizer,
              loss = 'mean_squared_error')

    return model
```

[ ]:
```
def train_model(tuning,x0,latent_dim,batch_size,epochs,X,Y):
    indices = X.shape[0] #Number of total samples
    indices=np.arange(indices)

    #defines validation and training samples and save the indices
    X_train,X_test,Y_train,Y_test,indices_train,indices_test  = \
    train_test_split(X,Y,indices,shuffle=True,train_size=0.8,random_state=42)
    data_splits = [X_train,X_test,Y_train,Y_test,indices_train,indices_test]

    #Retrive encoding dim
    ldim = Y.shape[1]

    # Assign values to hyperparemters
    if tuning =="False":
        qval = x0[0]
        activation = x0[1]
        learning_rate = x0[2]
        dropout_rate = x0[3]
        filters1 = x0[4]
        filters2 = x0[5]
        filters3 = x0[6]

        #Compile Model
        CNNModel = get_model(ldim,qval,activation,learning_rate,dropout_rate,
```

```
                            filters1, filters2, filters3, True,None)

        CNNModel.summary()
        CNNModel.fit(X_train,Y_train,
                batch_size = batch_size,
                epochs= epochs,
                validation_data = (X_test, Y_test),
                verbose = 1)
    else:
        print("No Training...")
    return CNNModel, data_splits
```

Bayesian optimization

[ ]:
```
#Defining bounds on hyperparameters to be tuned
dim_penalty_value = Real(low=1e-3, high=1e-2, name='penalty_value')
dim_learning_rate = Real(low=5e-4, high=1e-2, name='learning_rate')
dim_dropout_rate = Real(low=0.0, high=0.3, name='dropout_rate')
dim_activation = Categorical(categories=['relu','tanh'],
                             name='activation')
dim_filters1 = Integer(low = 10, high = 18, name = 'filters1')
dim_filters2 = Integer(low = 24, high = 40, name = 'filters2')
dim_filters3 = Integer(low = 48, high = 68, name = 'filters3')

dimensions = [dim_penalty_value,
              dim_learning_rate,
              dim_dropout_rate,
              dim_activation,
              dim_filters1,
              dim_filters2,
              dim_filters3
             ]

latent_dim =5
#defining intial condition
default_parameters = [0.01, 1e-3, 0,'tanh', 16, 32, 64]
```

[ ]:
```
#defining the loss function in the Byaesian optimization
@use_named_args(dimensions=dimensions) #Important for the hyperopt to work
def loss_obj(penalty_value,learning_rate,dropout_rate,activation,filters1,
    filters2,filters3):

    #size of latent space
    latent_dim=5
```

```
    CNNModel =␣
↪get_model(latent_dim,penalty_value,activation,learning_rate,dropout_rate,

    filters1,filters2, filters3,trainable=True,training=False) #True-activates␣
↪dropout during prediction
    CNNModel.summary()
    epochs = 200
    batch_size =128


    CNNModel.fit(X_trainBO,Y_trainBO,
            batch_size = batch_size,
            epochs = epochs,
            validation_data = (X_testBO, Y_testBO),
            verbose = 1)

    score = CNNModel.evaluate(X_testBO, Y_testBO, verbose=0)
    print('New GP iteration with score : ', score)

    # Delete currencyy CNN model with these hyper-parameters from memory.
    del CNNModel
    # Clear the Keras session, otherwise it will keep adding new
    # models to the same TensorFlow graph each time we create
    # a model with a different set of hyper-parameters.
    K.clear_session()
    ops.reset_default_graph()
    return score
```

[ ]:
```
#Defining training set for BO
X_trainBO,X_testBO,Y_trainBO,Y_testBO =␣
↪train_test_split(X,Y,shuffle=True,train_size=0.8,random_state=42)

#running the Bayesian optimization
gp_result = gp_minimize(func=loss_obj,
                            dimensions=dimensions,
                            n_calls=25,
                            noise= 1e-8,
                            n_jobs=1,
                            acq_func="EI",
                            x0=default_parameters)

#plot the score for each iteration
plt.figure(dpi=300)
plt.plot(np.arange(0,25),gp_result.func_vals,'o')
plt.xticks(range(0,25,5))
```

```
plt.axhline(y=np.min(gp_result.func_vals), color='orange', alpha=0.8,␣
 ↪linewidth=3)
plt.xlabel("Iterations")
plt.ylabel("Mean Absolute Error (Validation)")
plt.savefig('reg_l2')
```

[ ]:
```
#setting the hyperparameters to the optimal hyperparameters obtained from the␣
 ↪Bayesian optimization
opt_params= gp_result.x
latent_dim =5

qval = opt_params[0]
learning_rate = opt_params[1]
activation = opt_params[3]
dropout_rate = opt_params[2]
filters1 = opt_params[4]
filters2 = opt_params[5]
filters3 = opt_params[6]
```

Complining model with new hyperparameters

[ ]:
```
#Compile Model
cnn_model =␣
 ↪get_model(latent_dim,qval,activation,learning_rate,dropout_rate,filters1,␣
 ↪filters2, filters3,trainable=True,training=False) #True-activates dropout␣
 ↪during prediction
cnn_model.summary()
```

Training model with new hyperparametes

[ ]:
```
x0 = [qval,activation,learning_rate,dropout_rate,filters1,filters2,filters3]

cnn_model, data_splits = train_model('False',x0,Y.shape[1], 20, 200, X,Y)
```

Save CNN model and plot convergence

[ ]:
```
cnn_model.save('cnn_model.h5')

cnn_weights = load_model('cnn_model.h5')

plt.figure(2,dpi=200)
plt.semilogy(cnn_model.history.history['val_loss'])
plt.semilogy(cnn_model.history.history['loss'])
plt.xlabel('Epochs')
```

```
plt.ylabel('Loss (MSE)')
plt.legend(['Validation Loss','Train Loss'])
plt.xlim([0,200])
plt.savefig('training_loss_CNN')
```

Test Predictions

[ ]:
```
def geometric_cnn_predictions(cnn_model,decoder,dataset,Yraw):
    """
    The function uses the CNN model to predict CLDs which are subsequently␣
 ↪deocded
    """
    X_train,X_test,Y_train,Y_test,ind_train,ind_test = data_splits
    predicted_latent = cnn_model.predict(X_test)
    true_latent = Y_test

    #decode the predicted CLD and the encoded simualted CLD
    decoded_prediction =decoder(predicted_latent).numpy()
    decoded_true = decoder(true_latent).numpy()

    return decoded_prediction,decoded_true
```

[ ]:
```
#load CNN model
cnn_model = load_model('cnn_model.h5')
#make predicitons
dec_prediction,dec_true=
    geometric_cnn_predictions(cnn_model,decoder,data_splits,Yraw)

ind_train = data_splits[4]
ind_test = data_splits[5]
#list to select samples
klist=[0,500,800,1500,1700,2000]
#define range of xaxis
xrange=[100,500,300,200,500,150]
#make x-axis for plot
x=np.linspace(-1,3,100)
x=10**x
#plot a selection of samples
for k in range(len(klist)):
   # original_index=ind_test[klist[k]]
    original_index = ind_test[klist[k]] #to find same CLD in Yraw

    plt.figure(figsize=[8,5],dpi=300)
    plt.title('Sample '+ str(klist[k]),fontsize=15)
```

```
    plt.plot(x,dec_prediction[klist[k]],color='darkorchid',label='Decoded␣
↪GCLD',linewidth=3)
    plt.plot(x,dec_true[klist[k]].reshape(100,),color='seagreen',label='Encoded␣
↪and decoded CLD',linewidth=3)
    plt.plot(x,Yraw[original_index],color='royalblue',label='Untouched␣
↪simulated CLD',linewidth=3)
    plt.ylabel('Chord length probability',fontsize=15)
    plt.xlabel('Chord length [$\mu$m]',fontsize=15)
    plt.xticks(fontsize=12)
    plt.yticks(fontsize=12)
    plt.xlim([0,xrange[k]])
    plt.legend(fontsize=15)
    plt.savefig('resultsCNN/predAE'+ str(k)
```

[ ]:
```
#load experimental data to make predicitons
expPSD = np.load('experimentalPSSD.npy')
expCLD=np.load('experimental_CLD.npy')
enDeGCLD = np.zeros((26,100))
#make predicitons for the experimental PSDs and decode the prediciton
for i in range(26):
    predict= cnn_model.predict(expPSD[i].reshape(1,30,300,5))
    enDeGCLD[i] = decoder(predict).numpy()
```

[ ]:
```
#select samples to be plotted
sample= np.arange(0,26,1)
for i in range(len(sample)):
    plt.figure(figsize=[8,5], dpi=200)
    plt.title('Sample '+str(sample[i]),fontsize=15)
    plt.plot(ch_lengths,enDeGCLD[sample[i]],color='darkorchid',␣
↪linewidth=3,label='Decoded GCLD')
    plt.plot(ch_lengths, expCLD[sample[i]], color='royalblue', linewidth=3,␣
↪label='Untouched experimental CLD')
    plt.plot(ch_lengths,enDe[sample[i]], color='seagreen',label='Encoded and␣
↪decoded exerimental CLD', linewidth=3)
    plt.ylabel('Chord length probability',fontsize=15)
    plt.xlabel('Chord length [$\mu$m]',fontsize=15)
    plt.xticks(fontsize=12)
    plt.yticks(fontsize=12)
    plt.xlim([0,300])
    plt.legend(fontsize=15)
    plt.savefig('population2/2PopPredEXP'+ str(sample[i]))
```

Save CNN, Encoder, Autoencoder

[ ]:
```
#Save CNN, Encoder, Autoencoder and decoder as .h5 files
cnn_model.save('cnn_model.h5')
autoencoder.save('autoencoder.h5')
```

D.2 Code for the implementation of the correction layer i.e the Gaussian process regression model

[ ]:
```
"""
This code implments the Gaussian prcoess regression model. First, the dataset␣
↪is imported with \
the experimental conditions. Note that the data is alread in the latent space.␣
↪The input vector is then\
augmented to include the expeiremntal conditions. The imported data is then␣
↪standardized. The AE and the \
experimental data is then imported. The GPR model is created. First some error␣
↪clulations are made.\
The GPR model is fitted 100 times for each number of validation points and the␣
↪averge error of \
the relative errors for each sample for each fitting is calculated and plotted.␣
↪This examines \
what number of vaidation points provides the best fitting. When the appropriate␣
↪number of vlaidation points\
is found, the final GPR model is implemented. After the prediciton, the␣
↪prediciton is reverted back from\
the stanardized space. The last part of the code plots the prdictions and␣
↪calcualtes the relative error.
Contributors: Solveig Sannes and George Makrygiorgos. Spring 2022

"""
```

[73]:
```
#import the necessary packages
import numpy as np
import pandas as pd
import os, pdb
from pylab import *

# Plotting tool
import matplotlib.pyplot as plt

#modules for GPR
from sklearn.gaussian_process import GaussianProcessRegressor
```

```
from sklearn.gaussian_process.kernels import ConstantKernel, RBF

# tensorflow and sklearn library
import os
import tensorflow as tf
from tensorflow.keras import backend as K
from tensorflow.python.framework import ops
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
from tensorflow.python.keras.models import load_model
from  tensorflow.keras.models import Sequential
from  tensorflow.keras import optimizers
from  tensorflow.keras import losses
from  tensorflow.keras.layers import Dense, Dropout, \
Flatten, Conv2D, MaxPool2D, Activation, LeakyReLU
from tensorflow.keras import utils
from tensorflow.python.keras.models import load_model
from sklearn.model_selection import train_test_split
```

[2]:
```
#import the summary of experiments with experimental conditions
data_summary_df = pd.read_excel (r'Data_Summary.xlsx')
print (data_summary_df)
exp_names = []
num_exp = len(data_summary_df["Experiment"])
```

Loading the GCLD, RCLD and experimental conditions

[3]:
```
#loading experimental parameters
rpms=[]
ww=[]
for k in range(num_exp):
    rpms.append(data_summary_df["RPM"][k])
    ww.append(data_summary_df["Susp % w/w"][k])
```

[4]:
```
#Defining dimensions
encoding_dim = 5
exp_params = 2
```

[5]:
```
#loading the predicted CLD and experimental CLDs in latent space
Xdata=np.load('latent_GCLD_pred.npy')
Ydata=np.load('latent_RCLD.npy')
```

[6]:
```
# Adding the experimental conditions to the GCLD vector
X_sample=np.zeros((Xdata.shape[0],encoding_dim+exp_params))
X_sample[:,0:encoding_dim]=Xdata
X_sample[:,encoding_dim]=rpms[:-1]
```

```
X_sample[:,encoding_dim+1]=ww[:-1]
```

[7]:
```
#Creating a vector to contain the predicitons
test_mean_cov=np.zeros((X_sample.shape[0],Ydata.shape[1],2))
```

Standardizing the data before Gaussian Process Regression

[8]:
```
#function which standardizes the input
def standardize(sample):
    mu = np.mean(sample,axis=0)
    sig = np.std(sample, axis =0)
    standard = sample
    for i in range(len(sample[0])):
        standard[:,i] = (sample[:,i]-mu[i])/sig[i]
    return standard, mu, sig
```

[9]:
```
#function which reverts the input back to the original space
def revertStandardize(sample, mu,sig):
    revStand= sample
    for i in range(5):
        revStand[i] = (sample[i])*sig[i] + mu[i]
    return revStand
```

[10]:
```
#Storing the origina vectors before stadardization
YdataOriginal = np.array(Ydata.shape)
X_sampleOriginal = np.array(X_sample.shape)

YdataOriginal = Ydata.copy()
X_sampleOriginal = X_sample.copy()
```

[12]:
```
#Standardizing the data and saving the means and the standard deviations
Ydata, muY, sigY = standardize(Ydata)
X_sample, muX, sigX = standardize(X_sample)
```

Loading the autoencoder and the untouched experimental CLDs

[14]:
```
#loading autoencoder
autoencoder = load_model('autoencoder.h5')
#extracting the encoder from the autoencoder
decoder = autoencoder.layers[2]
decoder.summary()
```

[15]:
```
#load experimental data in full space
experimental_CLD=np.load('experimental_CLD.npy')
```

Gaussian Process Regression

Error calculations given the number of validation samples

[16]:
```
def gp_train(X,Y,Nval,edim):
    #edim := encoding dimension
    exp_params = 2;

    #saving test indices in order to compare to the experimental data later
    indices = X.shape[0] #Number of total samples
    indices=np.arange(indices)

    #defining taining and validation set
    if Nval>0:
        X_train, X_val, Y_train, Y_val, indices_train,indices_test =␣
↪train_test_split(X, Y, indices ,test_size=Nval,random_state=None)

    else:
        X_train = X
        Y_train = Y
        X_val= X
        Y_val= Y
        indices_train= indices
        indices_test= indices

    #Define GP parameters
    noise = 0.0001
    rbf =  ConstantKernel(1.0) * RBF(1, (1e-6,1e4))
    gpr = GaussianProcessRegressor(kernel=rbf, alpha=noise**2)

    if Nval>0:
        test_mean = np.zeros((Nval,edim,1))
        error = np.zeros((Nval,edim,1))
        iter_range = Nval
    else:
        test_mean = np.zeros((1,edim,1))
        error = np.zeros((1,edim,1))
        iter_range = 1


    #iter_range is the same as number of validation points
    for i in range(iter_range):
        predicted=np.zeros((1,Ydata.shape[1]))
        truth=np.zeros((1,Ydata.shape[1]))

        #For given validation  point, find all labels and compare to true value
```

```
        for j in range(Ydata.shape[1]):
            current_gp = gpr.fit(X_train[:,:], Y_train[:,j])
            #make prediction with the current model
            mean,covar=current_gp.predict(X_val[i,:].
↪reshape(1,edim+exp_params), return_cov=True)
            predicted[0][j] = mean[0]
            truth[0][j]=Y_val[i][j]

        #Reverting the predictions back from the standardized space
        revertCLD = np.zeros((1,edim))
        decodedPredicted =np.zeros((1,edim))
        revertCLD = revertStandardize(predicted[0].copy(), muY, sigY)
        decodedPredicted =decoder(revertCLD.reshape(1,edim)).numpy()

        #Selecting the correct CLD from the experimental data
        UntoucedExperimental = experimental_CLD[indices_test[i]]

        #calcuating the relative error between the prediction and the␣
↪experimental CLD in the latent space
        samp_error = np.linalg.norm(truth - predicted)/np.linalg.norm(truth)
        #calcuating the relative error between the decoded prediction and the␣
↪experimental CLD
        sampleErrorDecoded = \
        np.linalg.norm(UntoucedExperimental - decodedPredicted)/np.linalg.
↪norm(UntoucedExperimental)

        #save it
        errorsDecoded.append(sampleErrorDecoded)
        errors.append(samp_error)

    #return the mean of the error based on validation points (scalar)
    return np.mean(errors), np.mean(errorsDecoded)
```

[17]:
```
#this function returns the average error of all the N fittings of the GPR
def error_converge(N, Nval, X, Y,edim):
    av_error = np.zeros(N)
    av_errorDecoded = np.zeros(N)
    for j in range(N):
        #returns aveargе mean and error given the data and number of validation␣
↪sets
        av_error[j], av_errorDecoded[j] = gp_train(X,Y, Nval,edim)
    return  av_error, av_errorDecoded #np.mean(av_error)
```

[35]:
```

```

```
#setting the number of validations samples and number of fittings of the GPR␣
 ↪model
MaxNval = 24
N = 100 #number of evaluations
edim = 5
error_nval = np.zeros((MaxNval,N))
error_nvalDecoded = np.zeros((MaxNval,N))


for i in range(0,MaxNval):
    error_nval[i,:], error_nvalDecoded[i,:]  = error_converge(N, i, X_sample,␣
 ↪Ydata,edim)
```

Plotting the average relative error in latent space

[38]:
```
colorlist='royalblue'

plt.figure(figsize=[7,5],dpi=200)
for k in range(error_nval.shape[1]):
    #x = np.linspace(0, MaxNval-1, MaxNval)
    plt.plot(range(0,MaxNval),error_nval[:,k],'o', markersize=4,color=colorlist)
    plt.xticks(range(0,MaxNval,2),fontsize=16)
    plt.yticks(fontsize=16)
    plt.xlabel('Number of validation Points',fontsize=18)
    plt.ylabel('Relative error',fontsize=18)
plt.savefig('Relative error')
```

Plotting the average relative error of the decoded CLDs

[39]:
```
#Decoded error plot
colorlist='royalblue'

plt.figure(figsize=[7,5],dpi=200)
for k in range(error_nval.shape[1]):
    x = np.linspace(0, MaxNval-1, MaxNval)
    plt.plot(range(0,MaxNval),error_nvalDecoded[:,k],'o',␣
 ↪markersize=4,color=colorlist)
    plt.xticks(range(0,MaxNval,2),fontsize=16)
    plt.yticks(fontsize=16)
    plt.xlabel('Number of validation Points',fontsize=18)
    plt.ylabel(' Decoded relative error',fontsize=18)
plt.savefig('Relative decoded error')
```

[45]:
```
#calculating the mean and the standard deviation of the average relative error
 ↪of the decoded CLDs

means=np.mean(error_nvalDecoded,axis=1)
var=np.std(error_nvalDecoded,axis=1)
plt.figure(figsize=[8,5], dpi=300)
plt.plot(np.arange(1,24,1),means[1:],'o',label='Mean')
plt.plot(np.arange(1,24,1),var[1:],'o',label='Standard deviation')
plt.xticks(np.arange(1,24,2), fontsize=12)
plt.yticks(fontsize=12)
plt.xlabel('Number of validation points',fontsize=15)
plt.ylabel('Value', fontsize=15)
plt.legend()
plt.savefig('SampleSelect')
```

Code to leave out a number of validation samples

[59]:
```
a=np.arange(1,26)
ValidationN = 8
sample=np.random.choice(a, ValidationN, replace=False)
```

[61]:
```
#Implementing the GPR model
noise = 0.0001

rbf = ConstantKernel(1) *RBF(1, (1e-3,1e3))
gpr = GaussianProcessRegressor(kernel=rbf, alpha=noise**2)

#vector for saving the calculated means and covariances
test_mean_cov=np.zeros((2,Ydata.shape[1]))

#creating the validation and test set
validation_sample =np.zeros((ValidationN,7))
for i in range(ValidationN):
    validation_sample[i] = X_sample[sample[i]]

Xtest=X_sample.copy()
Ytest=Ydata.copy()


Xtest=np.delete(Xtest,sample,0) #leaving out samples
Ytest=np.delete(Ytest,sample,0)
```

```
#creating vector to save GP predicitons
GP_result=np.zeros((ValidationN,Ydata.shape[1]))

#fitting the model to the data and make predicitons
for i in range(len(sample)):
    for j in range(Ydata.shape[1]):
        #print(Ydata[:,j])
        current_gp = []
        current_gp = gpr.fit(Xtest, Ytest[:,j].reshape(X_sample.
 ↪shape[0]-ValidationN,1))
        mean,covar=current_gp.predict(validation_sample[i].
 ↪reshape(1,encoding_dim+exp_params), \
                                      return_cov=True)

        print("======================")
        print('The mean is')
        print(mean[0][0])
        print('The covariance')
        print(covar[0][0])
        print('The true latent value is')
        print(Ydata[sample[i],j])
        print("======================")
        test_mean_cov[0,j]= mean[0][0]
        test_mean_cov[1,j]= covar[0][0]

    GP_result[i]=test_mean_cov[0,:]
```

Revert back samples after standardization

[62]:
```
#Reverting back to non-stndardized form
for i in range(len(GP_result)):
    GP_result[i] = revertStandardize(GP_result[i], muY, sigY)
```

Decode the predictions from the Gaussian Process Regression

[64]:
```
decoded_CLD=np.zeros((ValidationN,100)) #vector for storing decoded GP␣
 ↪predicitons
decoded_true_CLD =np.zeros((ValidationN,100))  #vector for storing decoded CLD␣
 ↪predicitons

for i in range(len(GP_result)):
    decoded_CLD[i] =decoder(GP_result[i].reshape(1,encoding_dim)).numpy()␣
 ↪#test_mean_cov grabs only 1 value
```

```
    decoded_true_CLD[i] =decoder(YdataOriginal[sample[i],:].
↪reshape(1,encoding_dim)).numpy()
```

Plot the results

[65]:
```
#Loading real chord length bins
ch_lenghts = np.loadtxt('ch_len.csv')
```

Plot the performance of the Gaussian Process Regression in the validation samples

[71]:
```
for i in range(ValidationN):
    plt.figure(figsize=[8,5],dpi=200)
    label='sample'+ str(sample[i])
    plt.title('Sample '+ str(sample[i]), fontsize=15)
    plt.plot(ch_lenghts.reshape(100,),decoded_CLD[i], color='indianred',\
             label='Decoded GPR predicted CLD',linewidth=3)
    plt.plot(ch_lenghts.reshape(100,),decoded_true_CLD[i],color='seagreen',\
             label='Encoded and decoded experimental CLD',linewidth=3)
    plt.plot(ch_lenghts,experimental_CLD[sample[i]],color='royalblue',\
             label='Untouched experimental CLD',linewidth=3)
    plt.xlabel('Chord length [$\mu$m]',fontsize=15)
    plt.ylabel('Chord length probability', fontsize=15)
    plt.xticks(fontsize=12)
    plt.yticks(fontsize=12)
    plt.xlim([0,250])
    plt.legend(fontsize=14)
    plt.savefig('results/'+label)
```

Prediction of GCLD from experimental PSSD without the Gaussian process regression

[67]:
```
#load experimental PSSD
experimentalPSSD = np.load('experimentalPSSD.npy')

#load CNN network
cnn_model=load_model('cnn_model.h5')

#predict latent GCLD
latent_GCLD_pred= np.zeros((ValidationN,5))
for i in range(ValidationN):
    latent_GCLD_pred[i]=cnn_model.predict(experimentalPSSD[sample[i]].
↪reshape(1,30,300,5))[0]

#decode from latent to full space
```

```
GCLD_pred =np.zeros((ValidationN,100))
for i in range(ValidationN):
    GCLD_pred[i]= decoder(latent_GCLD_pred[i].reshape(1,encoding_dim)).numpy()
```

[68]:
```
#Plot the results
for i in range(ValidationN):
    plt.figure(figsize=[8,5],dpi=200)
    label='sample'+ str(sample[i])
    plt.title('Sample '+ str(sample[i]), fontsize=15)
    plt.plot(ch_lenghts.reshape(100,),GCLD_pred[i].reshape(100,),\
             color='mediumorchid', label ='Decoded GCLD',linewidth=3)
    plt.plot(ch_lenghts.reshape(100,),decoded_CLD[i].reshape(100,), \
             color='indianred',label='Decoded GP predicited CLD',linewidth=3)
    plt.plot(ch_lenghts.reshape(100,),experimental_CLD[sample[i]].
 ↪reshape(100,), \
             color='royalblue', label='Untouched experimental CLD',linewidth=3)
    plt.xlabel('Chord length [$\mu$m]',fontsize=15)
    plt.ylabel('Chord length probability', fontsize=15)
    plt.xticks(fontsize=12)
    plt.yticks(fontsize=12)
    plt.xlim([0,550])
    plt.legend(fontsize=14)
    plt.savefig('results/GCLD'+label)
```

[69]:
```
#Calculate the error of the prdicitons
errorGP = np.zeros(ValidationN)
errorGCLD = np.zeros(ValidationN)
for i in range(ValidationN):
    errorGP[i] = \np.linalg.norm(experimental_CLD[sample[i]].reshape(100,) -␣
 ↪decoded_CLD[i].reshape(100,))\
    /np.linalg.norm(experimental_CLD[sample[i]].reshape(100,))
    errorGCLD[i] = np.linalg.norm(experimental_CLD[sample[i]].reshape(100,) -␣
 ↪GCLD_pred[i].reshape(100,))\
    /np.linalg.norm(experimental_CLD[sample[i]].reshape(100,))
```

[70]:
```
for i in range(ValidationN):
    print('Sample ', sample[i],': Relative error GPR:',errorGP[i]*100,'␣
 ↪Relative error GCLD: ',errorGCLD[i]*100)
```

D.3 Code for generating simulated PSDs

[ ]:
```
"""
This code generates simualted PSDs. A large number of crystals are simualted by␣
 ↪defining the \
```

```
width L1 and obtaining the length L2 by multiplying with the aspect ratio. All␣
 ↪the crystals are gathered \
in a 2D histogram which represent the PSD. The PSDs are saved in a tesor with␣
 ↪dimensions NoSamplesx30x300x5.
Contributors: John Tsortos, John Maggioni and Solveig Sannes. Spring␣
 ↪2020-Spring 2022.
"""
```

[2]:
```
# Import the required modules
import numpy as np
import pandas as pd
import pylab as pl
import scipy, glob, os, fnmatch, re, sklearn, time, sys
import matplotlib.pyplot as plt
# Parallel Computing
from joblib import Parallel, delayed
# Convex Hull
from scipy.spatial import ConvexHull, convex_hull_plot_2d , distance
from scipy.interpolate import interp1d, griddata
from scipy.signal import savgol_filter as svf
from scipy.integrate import simps
# Draw 3D and 2D polygons
import pyny3d.geoms as pyny
```

[81]:
```
# Number of Random Particles to approximate each Population
Np = int(5e4)
# Number of Random Populations
Npop = 10
# Type of Crystal
Type = 'Needles' # Alternative: Platelets



# Select a Distribution
# Set AR > 1 for needles and 0 < AR <1. for platelets

L1PBE = np.zeros((Npop,30))
L3PBE = np.zeros((Npop,300))
f = np.zeros((Npop,L1PBE.shape[1],L3PBE.shape[1]))
PSSD = np.zeros((Npop, L1PBE.shape[1],L3PBE.shape[1],5))


for i in range(0,Npop):

    Lm1 = np.random.uniform(10,50) #defining mean of L1
```

```
    cv1 = 0.15 + np.random.uniform(-0.01,0.01) #defining the variance

    L1 = np.random.normal(Lm1,cv1*Lm1, Np) #generate L1s
    #Generate an aspect ratio
    AR = np.random.uniform(1,1.5,Np)
    #Define L3
    L3 = L1*AR

    # Define the "discretisation grid"
    L1g = np.linspace(0., Lm1*(1+3*cv1), L1PBE.shape[1]+1)
    L3g = np.linspace(0.,(AR*Lm1*(1+3*cv1)).max(), L3PBE.shape[1]+1)
    # Actual Cells width
    dL1 = np.diff(L1g)
    dL3 = np.diff(L3g)

    # Representative points in each cell (mid-points)
    L1PBE[i,:] = L1g[:-1]+dL1/2
    L3PBE[i,:] = L3g[:-1]+dL3/2

    # Approximate with 2D Histogram and smooth the results
    A = svf(svf(np.
↪histogram2d(L1,L3,bins=[L1g,L3g],density=True)[0],3,1,axis=0),3,1)
    # REmove the unphysical zeros due to smoothing
    A[A<0.] = 0.
    # Normalise the Integral
    A /= simps(simps(A,L3PBE[i,:]),L1PBE[i,:])
    f[i,:,:] = A


    #Saving the PSSDs in a vector with shape (nSamples,30,300,5)
    for k in range(PSSD.shape[2]):
        PSSD[i,:,k,0]= L1PBE[i]
        PSSD[i,:,k,3]= dL1
    for j in range(PSSD.shape[1]):
        PSSD[i,j,:,1]=L3PBE[i]
        PSSD[i,j,:,4]=dL3
    PSSD[i,:,:,2]= A

    #%% Check for Consistency

if Type == 'Needles':

    if np.any(L1>L3):

        print("ERROR: Aspect Ratio not consistent")
```

```
        sys.exit()

if Type == 'Platelet':

    if np.any(L3>L1):

        print("ERROR: Aspect Ratio not consistent")
        sys.exit()
# Save the Simulated DAta
np.savez('PSSD_populations.npz', Xtrain=PSSD)
```

D.4 Code for generating the simulated CLDs

[ ]:
```
"""
This code generates the CLDs from the PSD data with a geometric model. Each␣
↪crystal in the PSD is\
subject to rotations and projections. The generated CLDs are then saved.

Contributors: John Tsortos, John Maggioni and Solveig Sannes. Spring␣
↪2020-Spring 2022.
"""
```

[ ]:
```
#Importing packages
import pdb
import math
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
import numpy as np
import pandas as pd
import scipy, glob, os, fnmatch, re, sklearn, time, sys
import matplotlib.pyplot as plt
import seaborn as sns
# Parallel Computing
from joblib import Parallel, delayed
# Convex Hull
from scipy.spatial import ConvexHull, convex_hull_plot_2d
# Draw 3D and 2D polygons
import pyny3d.geoms as pyny
from mpl_toolkits.mplot3d import Axes3D
```

[ ]:
```
def rotationnmatrix(phi,theta,psi):
    # This function yields a rotation matrix in the 3D-Euclidean Space
    # based on the Euler Angles, psi, theta, phi.
    # phi rotates around the original axis z
```

```
    # theta rotates around the axis x generated by the first rotation
    # psi rotates around the axis z generated after the first two rations
    # Each rotation is represented by a 3 dimensional matrix
    # The output of this function is a tuple "P" containing the aforementioned matrices

    # The three angles, scalars
    psi     = float(psi)
    theta   = float(theta)
    phi     = float(phi)

    # The three rotation matrices
    P3 = np.array([[np.cos(psi),-np.sin(psi),0.],
                   [np.sin(psi),np.cos(psi),0.],
                   [0.,0.,1.]])
    P2 = np.array([[1.,0.,0.],
                   [0.,np.cos(theta),-np.sin(theta)],
                   [0.,np.sin(theta),np.cos(theta)]])
    P1 = np.array([[np.cos(phi),-np.sin(phi),0.],
                   [np.sin(phi),np.cos(phi),0.],
                   [0.,0.,1.]])

    return (P1,P2,P3)
```

[ ]:
```
C = lambda L1,L2,L3: np.array([[0.,0.,0.],[L1,0.,0.],[L1,L2,0.],[0.,L2,0.],[0.,0.,L3],[L1,0.,L3],[L1,L2,L3],[0.,L2,L3]]).T
```

[ ]:
```
def projection2D(psi,theta,phi,L1,L2,L3):

    # Rotation Matrix
    P = rotationnmatrix(psi,theta,phi)
    # Projection Matrix onto X1-X2 Plane
    Pxy = np.array([[1.,0.,0.],[0.,1.,0.]])

    # Original Crystal
    C0 = C(L1,L2,L3)
    # Rotated Crystal
    Cr = P[0]@P[1]@P[2]@C0
    # Projected Crystal
    Cp = Pxy@Cr

    Feret =  abs(Cp.T.min(0)-Cp.T.max(0))
    # Feret = np.array([Feret.max(),Feret.min()])
    # Feret = np.array([F0.max(),F0.min()])
```

```
    return C0,Cr,Cp,Feret
```

[ ]:
```
def Ideal_CLD_generator(x):
    # This function generates the ideal CLD distribution
    # based on a purely geometric model.

    # Inputs are the Lengths of the particle and the angles of rotation
    # C is the function producing a Crystal modelled as a polytope.

    (L1,L2,L3),C,(phi,theta,psi) = x

    # Rotation Matrices
    # Rotation around the original z-axis by phi
    P1 = np.array([[np.cos(phi),-np.sin(phi),0.],
                   [np.sin(phi),np.cos(phi),0.],
                   [0.,0.,1.]])
    # Rotation by theta around the x-axis after the rotation by phi
    P2 = np.array([[1.,0.,0.],
                   [0.,np.cos(theta),-np.sin(theta)],
                   [0.,np.sin(theta),np.cos(theta)]])
    # Rotation by psi around the z-axis after the rotation by theta
    P3 = np.array([[np.cos(psi),-np.sin(psi),0.],
                   [np.sin(psi),np.cos(psi),0.],
                   [0.,0.,1.]])

    # Original Crystal
    C0 = C(L1,L2,L3)
    # Rotated Crystal
    Cr = P3@P2@P1@C0
    # Projected Crystal
    Cp = Pxy@Cr
    # Feret Diameters along X1 and X2
    Feret =  abs(Cp.T.min(0)-Cp.T.max(0))
    weight = Feret[1]

    ### Generate the chord length distribution associated with
    # the specific projections of the individual particle

    # Find the convex hull of the projection
    hull = ConvexHull(Cp.T)
    I = np.array([np.arange(0.,hull.vertices.size-1),np.arange(1.,hull.vertices.
↪size)])
    I = np.append(I.T,np.array([[0.,hull.vertices.size]]),axis=0)

    # Random side thorugh which the chords are drwan
```

```
    N = np.asarray([np.random.choice(hull.vertices.size-1, 2,replace=False) for
↪i in range(0,Nc)])
    # Random values for the parametrisation of the segments
    # beloging the perimeter of the projection and used to draw the chord
    T1 = np.random.uniform(0.,1.,(Nc))
    T1 = np.array([T1,1-T1])
    T2 = np.random.uniform(0.,1.,(Nc))
    T2 = np.array([T2,1-T2])

    # Find the coordinates of the random Chord
    X1 = np.asarray([Cp.T[I.astype(int)[N[i,0],:]].T@T1[:,[i]] for i in
↪range(0,Nc)]).reshape(Nc,2)
    X2 = np.asarray([Cp.T[I.astype(int)[N[i,1],:]].T@T2[:,[i]] for i in
↪range(0,Nc)]).reshape(Nc,2)
    # Compute the Chord length (Euclidean distance between the individual
↪extremities of each chord)
    Chord = np.sqrt(((X1-X2)**2).sum(1))#[:,None]

    return np.append(Chord,weight)
```

Here we load the pre-existing data to re-generate the labels

[ ]:
```
data = np.load('PSSD_populations.npz')
X = data['Xtrain']
PSSD = X
```

Here generate CLDs from the PSDs

[ ]:
```
# Matrix to store log chord histogram for each image
chords_mat = np.zeros((images,100))

for i in np.arange(images):
    print(i)

    Chord_arr = np.array([])
    Weight_arr = np.array([])


    # Particles into bins
        #If PSSD is full of zeros -> CLD is array of zeros
    if np.sum(PSSD[i,:,:]) == 0:
        chords_mat[i,:] = np.zeros((1,100))
    else:
        emp_psd = np.random.multinomial(particles, PSSD[i,:,:,2].flatten()\
```

```
                                        /np.sum(PSSD[i,:,:,2])).
↪reshape(rows,col) # <- normalization



        # Loop through each bin
        for j in np.arange(rows):
            for k in np.arange(col):

                # Loop through each particle
                for l in np.arange(emp_psd[j,k]):

                    #%% Generate the projections

                    # Initialise the Matrix containing Feret Diameters (Max and␣
↪Min)
                    Feret = np.zeros((int(Np),2))
                    weight = np.ones((int(Np*Nc),))

                    # Initialise the Chord Matrix
                    Chord = np.zeros((Nc,Np))

                    # Projection Matrix onto X1-X2 Plane
                    Pxy = np.array([[1.,0.,0.],[0.,1.,0.]])

                    # Rotation angles, random
                    psi0   = np.random.uniform(0,2*np.pi,(1,Np))
                    theta0 = np.random.uniform(0,2*np.pi,(1,Np))
                    phi0   = np.arcsin(np.random.uniform(-1,1,(1,Np)))


                    # Particle length: sample from uniform dist defined by L1 &␣
↪L2 bin edges

                        #Check if we're at the beginning of the row/column j=␣
↪row =30
                        #k =300
                    if j == 0:
                        L1 = np.random.uniform(0, PSSD[i,j,0,0])
                    else:
                        L1 = np.random.uniform(PSSD[i,j-1,0,0], PSSD[i,j,0,0])
                    if k == 0:
                        L2 = np.random.uniform(0, PSSD[i,0,k,1])
                    else:
                        L2 = np.random.uniform(PSSD[i,0,k-1,1], PSSD[i,0,k,1])
```

```
                    #L3 is just the same as L1
                L3 = L1

                # Input list for the parallelised function
                par_arg = [((L1,L2,L3),C,(phi0[0,p],theta0[0,p],psi0[0,p]))␣
↪for p in range(0,Np)]

                # Parallelised function
                Dummy = Parallel(n_jobs=2, backend="loky")(
                    map(delayed(Ideal_CLD_generator), par_arg));
                # Convert the list output of parallel into an array
                # It contains both the chords and the weights
                Chord  = np.asarray(Dummy);
                # Define the weights vector
                weight = np.kron(Chord[:,1][:,None],np.ones((Nc,1)))
                # Eliminate the weigths from the Chord Matrix
                Chord  = Chord[:,:-1]

                # store chords and corresponding weights into Chord_arr for␣
↪a given image
                Chord_arr = np.append(Chord_arr,Chord.flatten())
                Weight_arr = np.append(Weight_arr,weight.flatten())


        Chord_hist = np.histogram(np.log10(Chord_arr),bins=100,range=(-1,3),\
                                  density=True,weights=Weight_arr)[0]
        Chord_hist = Chord_hist/np.sum(Chord_hist)
        chords_mat[i,:] = Chord_hist
```

[ ]:
```
#Save datset
np.savez('GM_needle_train_2.npz', Xtrain= PSSD,Ytrain=chords_mat)
```

D.5 Code for generating CLDs from FBRM data

[ ]:
```
"""
This code takes in the FBRM data from the experiments and generates the␣
↪experimental CLDs. Then the CLDs \
are encoded to the latent space and saved.
Contributors: John Maggioni, Solveig Sannes and George Makrygiorgos. Spring␣
↪2022
"""
```

[ ]:
```
#Importing packages
import numpy as np
import pandas as pd
import os, pdb
from pylab import *
import matplotlib.pyplot as plt
# 3D Plotting tool
from mpl_toolkits.mplot3d import Axes3D
import seaborn as sns
# from joblib import Parallel, delayed
from datetime import datetime, timedelta
from scipy.integrate import simps as simpson
from scipy.signal import savgol_filter as svf
from scipy import stats

# suppress tensorflow compilation warnings
import os
import tensorflow as tf
from tensorflow.keras import backend as K
from tensorflow.python.framework import ops
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'

from  tensorflow.keras.models import Sequential
from  tensorflow.keras import optimizers
from  tensorflow.keras import losses
from  tensorflow.keras.layers import Dense, Dropout, Flatten, Conv2D,␣
↪MaxPool2D, Activation, LeakyReLU
from tensorflow.keras import utils
from tensorflow.python.keras.models import load_model
```

Generating the experimental CLD from FBRM data

[ ]:
```
#deine filepath
FilePath = 'FBRM-Daten csv'
```

[ ]:
```
#print summary of experimental data
data_summary_df = pd.read_excel (r'Data_Summary.xlsx')
print (data_summary_df)
exp_names = []
num_exp = len(data_summary_df["Experiment"])

for k in range(num_exp):
    exp_names.append(data_summary_df["Experiment"][k])
```

```
[ ]: #ExpName  =  [exp_names[1] + ' Default']

     # Define the CLD list
     CLD_Experimental = []
     Real_Chord = []
     for jj in range(len(exp_names)-1): #Don't get the NaN
         ExpName  =  [exp_names[jj] + ' Default']
         #print(ExpName)
         # Use panda to extract the csv files
         DataF = pd.read_csv(FilePath + '/' + ExpName[0] + '.csv',delimiter=',',␣
      ↪header=1)

         DataOriginal=DataF
         # Chords, as midpoints of the bins
         IntChords = DataF.to_numpy()[0,3:]
         IntChords = IntChords.astype(float)

         Chords = DataF.to_numpy()[1,3:-1]
         Chords = Chords.astype(float)
         Real_Chord.append(Chords)
         # CLD
         ChordDist    = DataF.to_numpy()[2:,3:-1]
         ChordDist = ChordDist.astype(float)
         ChordDist[ChordDist<=0.] = 1e-32
         ChordDist /=simpson(ChordDist,Chords)[:,None]
         # Square-weighted
         ChordDist_2 = ChordDist*(Chords**2)/simpson(ChordDist*(Chords**2),Chords)[:
      ↪,None]
         # Cube-weighted
         ChordDist_3 = ChordDist*(Chords**3)/simpson(ChordDist*(Chords**3),Chords)[:
      ↪,None]
         # Absolute time
         DateFormat   = DataF.to_numpy()[2:,0]
         TimeFormat   = DataF.to_numpy()[2:,1]
         a = [DateFormat[i] + ' ' + TimeFormat[i] for i in range(0,ChordDist.
      ↪shape[0])]
         del DataF, DateFormat, TimeFormat
         Dates = [datetime.strptime(a[i], '%d.%m.%Y %H:%M:%S') for i in␣
      ↪range(0,len(a))]
         Duration = [Dates[i]-Dates[0] for i in range(0,len(Dates))]
         Duration = [Duration[i].total_seconds() for i in range(0,len(Dates))]
         Time = np.asarray(Duration)
         del a, Dates, Duration
         plt.plot(np.log(Chords),ChordDist[-1].T);
         CLD_Experimental.append(ChordDist[-1])
```

[ ]:
```
#defining number of bins
nbins=len(CLD_Experimental[0])
nexp = len(CLD_Experimental)
```

Save the real chord length bins

[ ]:
```
#save the real chord length bins as a .csv file
np.savetxt('ch_len.csv', Real_Chord[1], delimiter=",")
ch_lenghts = Real_Chord
```

Load the autoencoder

[ ]:
```
#Load the AE, encoder & decode the CLD of each experiment

autoencoder = load_model('autoencoder.h5')

#extracting the encoder from the autoencoder
encoder = autoencoder.layers[1]
encoder.summary()

#finding the output size of encoding layer
encoder_output=autoencoder.layers[1].output.shape[1]

#creating empty array for encoded latent_CLD's
latent_RCLD=np.zeros((len(CLD_Experimental),encoder_output))
```

Encoding the experimental CLDs

[ ]:
```
#using encoder to encode the CLDs and then saving the encoded CLDS as a .npy␣
 ↪file
for i in range(len(CLD_Experimental)):
    cld_input=CLD_Experimental[i].reshape(1,100)
    latent_RCLD[i,:]=encoder(cld_input).numpy()
#save latent space experimental CLD
np.save("latent_RCLD.npy", latent_RCLD)
```

[ ]:
```
#Saving the untouched experimental CLDs
np.save("experimental_CLD.npy", CLD_Experimental)
```

D.6 Code for generating PSDs from QicPic data

[ ]:
```
"""
This code generates the PSD from the QicPic data. The PSD are then passed to␣
 ↪the CNN model to make \
predicitons about the CLD. The predicted GCLDs are then saved.
Contributors: John Maggioni, Solveig Sannes and George Makrygiorgos. Spring␣
 ↪2022

"""
```

[ ]:
```
#import packages
import numpy as np
import pandas as pd
import os, pdb
from pylab import *

import matplotlib.pyplot as plt
# 3D Plotting tool
from mpl_toolkits.mplot3d import Axes3D
import seaborn as sns

# from joblib import Parallel, delayed
from datetime import datetime, timedelta
from scipy.integrate import simps as simpson

from scipy import stats


# suppress tensorflow compilation warnings
import os
import tensorflow as tf
from tensorflow.keras import backend as K
from tensorflow.python.framework import ops
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'

from  tensorflow.keras.models import Sequential
from  tensorflow.keras import optimizers
from  tensorflow.keras import losses
from  tensorflow.keras.layers import Dense, Dropout, Flatten, Conv2D,␣
 ↪MaxPool2D, Activation, LeakyReLU
from tensorflow.keras import utils
from tensorflow.python.keras.models import load_model
```

[ ]:
```
#reads and print the data summary
data_summary_df = pd.read_excel (r'Data_Summary.xlsx')
print (data_summary_df)
```

[ ]:
```
#keeping track of the number of each experiment
exp_names = []
num_exp = len(data_summary_df["Experiment"])
for k in range(num_exp-1):
    experiment_number=data_summary_df["Experiment"][k].split('-')[-1]
    if experiment_number[-1] == '0':
        exp_names.append(int(experiment_number))
    else:
        experiment_number=experiment_number.split('0')[-1]
        exp_names.append(int(experiment_number))
```

Read the PSSD and convert to correct shape for the CNN

[ ]:
```
Population_PSSD = []
vecPSSD =np.zeros((len(exp_names),1,30,300,5))

for w in range(len(exp_names)):
    print(w)
    pssd_info = []
    FilePath = r"QicPic-Daten"
    ExpName  =  data_summary_df["Population"][w]
    jj = ExpName
    QicPic_Shape = np.array([1.,2.])
    FMax_lim = 0.
    Fmin_lim = 0.
    AR_lim = 0.
    #%%
    # Depending on the Format
    try:
        QicPic_DF = pd.read_excel(FilePath + '/' + jj + '.xlsx', header=0)
    except:
        try:
            QicPic_DF = pd.read_csv(FilePath + '/' + jj + '.csv', sep=';
 ↪',header=0)
        except:
            break



    # From metres to microns
    QicPic_DF['FERET_MAX'] *= 1e6
```

```
    QicPic_DF['FERET_MIN'] *= 1e6
    QicPic_DF['ASPECT_RATIO'] = 1/QicPic_DF['ASPECT_RATIO']
    # Volume of Each Crystal
    QicPic_DF['VOLUME'] =␣
↪QicPic_DF['FERET_MAX']**QicPic_Shape[1]*QicPic_DF['FERET_MIN']**QicPic_Shape[0]

    FMax_Spacing = 50*(QicPic_DF['FERET_MAX'].max()<=400) +␣
↪250*(QicPic_DF['FERET_MAX'].max()>400)
    Fmin_Spacing = FMax_Spacing/2*((QicPic_DF['ASPECT_RATIO']).median()<=4) +␣
↪FMax_Spacing/2*((QicPic_DF['ASPECT_RATIO']).median()>4)
    AR_Spacing = 1*(QicPic_DF['ASPECT_RATIO'].median()<=4) +␣
↪2*(QicPic_DF['ASPECT_RATIO'].median()>4)

    if FMax_lim == 0.:
        FMax_lim = QicPic_DF['FERET_MAX'].max()-QicPic_DF['FERET_MAX'].
↪max()%FMax_Spacing+FMax_Spacing
    else:
        FMax_lim = FMax_lim
    if Fmin_lim == 0.:
        Fmin_lim = QicPic_DF['FERET_MIN'].max()-QicPic_DF['FERET_MIN'].
↪max()%Fmin_Spacing+Fmin_Spacing
    else:
        Fmin_lim = Fmin_lim
    if AR_lim == 0.:
        ARlim    = (QicPic_DF['ASPECT_RATIO']).
↪max()-((QicPic_DF['ASPECT_RATIO']).max())%AR_Spacing+AR_Spacing
    else:
        ARlim =AR_lim

    # Bin Sizes
    # Computed according to:
    # Scott, D. 1979.
    # On optimal and data-based histograms.
    # Biometrika, 66:605-610
    W_FMax = 3.5*QicPic_DF['FERET_MAX'].std()*QicPic_DF['FERET_MAX'].size**(-1/
↪3)
    W_Fmin = 3.5*QicPic_DF['FERET_MIN'].std()*QicPic_DF['FERET_MIN'].size**(-1/
↪3)
    W_AR   = 3.5*QicPic_DF['ASPECT_RATIO'].std()*QicPic_DF['ASPECT_RATIO'].
↪size**(-1/3)


    FMaxbins=np.linspace(0,FMax_lim,301)
    Fminbins = np.linspace(0,Fmin_lim,31)
```

```
    FM = FMaxbins[:]+W_FMax
    Fm = Fminbins[:]+W_Fmin

    V = np.kron(FM[:,None]**QicPic_Shape[1],Fm[None,:]**QicPic_Shape[0])
    PSSD,_,_ = np.histogram2d(QicPic_DF['FERET_MAX'],␣
↪,QicPic_DF['FERET_MIN'],bins=[FMaxbins,Fminbins],density=True )


    #%% Plot the Results
    values = np.vstack([QicPic_DF['FERET_MAX'], QicPic_DF['FERET_MIN']])
    kernel = stats.gaussian_kde(values)

    xmin = QicPic_DF['FERET_MAX'].min()
    xmax = QicPic_DF['FERET_MAX'].max()
    ymin = QicPic_DF['FERET_MIN'].min()
    ymax = QicPic_DF['FERET_MIN'].max()
    X, Y = np.mgrid[xmin:xmax:300j, ymin:ymax:300j]
    positions = np.vstack([X.ravel(), Y.ravel()])
    Z = np.reshape(kernel(positions).T, X.shape)


    plt.contourf(FM[:-1],Fm[:-1],(PSSD).T)

    ##########################################################
    #Create Sample
    n_channels=5
    sample = np.zeros((1,len(Fm),len(FM),n_channels))
    """

    Channel 1: L2 (Fm)
    Channel 2: L1 (FM)
    Channel 3: PSSD Value
    Channel 4: DL2 #Width of bin
    Channel 5: DL1

    """
    #creating a vector for the samples with 5 cahnnels
    sample = np.zeros((1,len(Fm)-1,len(FM)-1,n_channels))

    for i in range(len(Fm)-1):
        for j in range(len(FM)-1):
            sample[0,i,j,0] = Fm[i]/1000
            sample[0,i,j,1] = FM[j]/1000
            sample[0,i,j,2] = PSSD.T[i,j]
```

```
            sample[0,i,j,3] = (Fm[i+1]-Fm[i])/1000
            sample[0,i,j,4] = (FM[j+1]-FM[j])/1000

    #Saving the sample PSSD
    vecPSSD[w] = sample

    #Store the pssd as input for the CNN and the name in order to match with␣
↪the data_summary
    psd_info=[sample,data_summary_df["Population"][w]]
    Population_PSSD.append(psd_info)
    last_iteration=w
```

[ ]:
```
#Saving the normalized experimental PSSD
vecPSSD=vecPSSD.reshape(26,30,300,5)
np.save('experimentalPSSD.npy' ,vecPSSD)
```

Loading the CNN to make predictions about the GCLD

[ ]:
```
#loading the trained CNN model
cnn_model=load_model('cnn_model.h5')

latent_dim = 5

#Creating a vector for predicitons
latent_GCLD_pred = np.zeros((len(vecPSSD),latent_dim))

#Make predicitons for the GCLD from the expeirmental PSSD
for i in range(len(vecPSSD)):
    latent_GCLD_pred[i]=cnn_model.predict(vecPSSD[i].reshape(1,30,300,5))[0]
```

[ ]:
```
#Save the predicitons in a .npy fil
np.save("latent_GCLD_pred.npy", latent_GCLD_pred)
```

Solveig Sannes

A machine learning framework for predicting chord length distributions

NTNU
Norwegian University of
Science and Technology