Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Richard Che Bui

Resistivity Estimation Using Convolutional Neural Networks

Master's thesis in Cybernetics and Robotics
Supervisor: Damiano Varagnolo
Co-supervisor: Carl Fredrik Berg and Kurdistan Chawshin
June 2021

NTNU
Norwegian University of
Science and Technology

Richard Che Bui

Resistivity Estimation Using Convolutional Neural Networks

Master's thesis in Cybernetics and Robotics
Supervisor: Damiano Varagnolo
Co-supervisor: Carl Fredrik Berg and Kurdistan Chawshin
June 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

Preface

This thesis has been developed together with the BRU-21 group at NTNU during the spring semester of 2021. The aim has been to investigate the possibilities of predicting the resistivity of rocks using the learning structure of Convolutional Neural Networks, together with 2D cross-sections of 3D core CT-scan data. Further, this thesis is written as a contribution to digitalization in the Oil and Gas industry.

During my past two years studying at Cybernetics and Robotics at NTNU, I have felt an increase of motivation to attend Computer Science and Machine Learning courses. This has been my motivation for selecting a Machine Learning problem for my master's thesis, and I have learnt a lot during this semester.

I would like to thank my supervisor, Professor Damiano Varagnolo and co-supervisors Ph.D. student Kurdistan Chawshin and Associate Professor Carl Fredrik Berg for the opportunity to work on this project together with their valuable discussions. A special thanks to Kurdistan for providing me with the necessary code and resources, as well as her guidance. I would further like to thank Equinor for allowing me to work with their data sets.

Abstract

This thesis investigates the suitability of utilizing 2D cross-sections of 3D core CT-scan data together with Convolutional Neural Networks to create models for prediction of resistivity. One of the important roles of resistivity in the Oil and Gas industry is to determine the hydrocarbon contents from well logs. Additionally, in hydrogeology, resistivity can be used to locate water tables and estimate the intrusion of salt water into fresh water aquifers. The data used in this thesis comprises 2D cross-sections of 3D core CT-scan data over a 142 meter interval. The aim is then to extract features from the images, and use these features to predict resistivity, thus contributing to technological development in geoscience and the oil industry. By performing robust modelling of resistivity, additional information can be provided to geologists, increasing time and economical efficiency. In this thesis the Convolutional Neural Network, a state-of-the-art framework for modelling with image data is used. To tune the CNN hyperparameters, *Keras* was used, which allowed for automated machine learning by searching over a pre-defined space for optimal CNN hyperparameters. The data set itself turned out to be too small and the quality, inconsistent, thus measures for regularization was used, mostly in the form of data augmentation to improve the data set quality and size. Data augmentation with 98% overlap was used to increase the data set size, as well as flipping the images vertically and horizontally for increased robustness. However, using augmentation with overlap resulted in an issue during splitting of the data into training, validation and test sets. Therefore three different data set distributions have been proposed to emphasize the importance of splitting of data. A thorough analysis of hyperparameter tuning resulted in three optimal models, one for each data set distribution. Finally, a holdout test set was predicted by all three optimal models, where the best of the three achieved an R-squared of 0.51.

Sammendrag

Denne oppgaven undersøker egnetheten av å anvende 2D tverrsnitt av 3D kjerneprøver i form av CT-scan data sammen med konvolusjonelle nevrale nettverk(CNN) for å bygge modeller for prediksjon av resistivitet. En av de viktige rollene til resistivitet i olje- og gassindustrien er å indikere mengden av hydrokarboner fra en brønnlogg. I tillegg kan resistivitet anvendes i hydrogeologi til å lokalisere grunnvannsspeil, og estimere inntrengelsen av saltvann i grunnvannsakviferer. Dataen som er anvendt i oppgaven omfatter 2D tverrsnitt av 3D kjerneprøver i form av CT-scan data over et 142 meters intervall. Målet er da å hente bildetrekk og bildekarakteristikker fra CT-scan bildene for å estimere resistivitet, og dermed bidra til teknologisk utvikling innen geovitenskap og oljeindustrien. Ved å utføre robust modellering av resistivitet, åpnes muligheten til å tilføre ytterlig informasjon til geologer, som videre kan bidra til økonomisk og tidseffiktivitet. I oppgaven anvendes CNN, et av de mest teknisk aktuelle rammeverkene for modellering med bildedata. For å innstille hyperparameterne til CNN har biblioteket *Keras* blitt anvendt, noe som åpnet for automatisert maskinlæring ved å søke over et egendefinert søkeområde av CNN-hyperparametere. Det viste seg at størrelsen til datasettet var for liten, i tillegg til at kvaliteten var inkonsekvent. Dermed ble det anvendt regulariseringsmetoder i form av bildemodifisering for å forbedre kvaliteten og størrelsen av dataen. Bildemodifisering med 98% overlapp var anvendt for å utvide datasettstørrelsen, og i tillegg ble bildene snudd vertikalt og horisontalt for å øke robustheten. På den andre siden ble det problematisk med den overlappede dataen når datasettet skulle splittes. Dermed har vår strategi vært å bruke tre forskjellige datasettfordelinger for å fremheve viktigheten av splittelse av data. En grundig analyse av innstilling av hyperparametere resulterte i tre optimale CNN modeller, en fra hver datasettfordeling. Til slutt ble de tre optimale modellene brukt til å predikere et testsett som ble holdt utenfor treningen, hvorav den beste av de tre oppnådde en R-kvadrert på 0.51.

Contents

Figures

Tables

Acronyms

ANN Artificial Neural Network. 4, 23, 26, 27, 31, 61, 68, 72

BRU21 Better Resource Utilization in the 21st century. 6, 7

CNN Convolutional Neural Network. vi, vii, xi, 1, 2, 4, 6–9, 14, 16, 18, 23, 24, 26–29, 31, 32, 37, 38, 40–42, 49, 50, 59, 61–63, 67–70, 72, 85, 87–89, 91

CT Computed Tomography. vi, vii, 1, 2, 6–8, 13, 16, 85, 87

MSE Mean Squared Error. xi, 18–20, 32, 34, 36, 61, 62, 64, 67, 70–72, 77, 78, 80, 82, 86

ReLU Rectified Linear Unit. xi, 23, 24

Chapter 1

Introduction

Machine learning is a subject area that is widely applied to solve many problems today. Due to the evolution of technology and the large amount of data harvesting in industries, machine learning has been able to solve problems that were previously not possible. With machine learning, scientists are improving at medical diagnostics, self-driven cars are evolving, and energy production optimization is getting better. In this thesis we utilize artificial neural networks, a subset of machine learning that uses algorithms to solve problems much like the human brain, but with greater speed and with more computational complexity. This involves tasks such as discovering patterns, automating processes, and predicting future events.

This project covers the investigation and possibilities of utilizing image data for predictive modelling of resistivity in wells. The structure of the learned model is the Convolutional Neural Network, a state-of-the-art structure for modelling image data. Usually, CNN is used to perform tasks such as classification and object detection, but we want to find out if regression with CNN is compatible with using CT-scan images to predict the continuous resistivity variable. As the different hyperparameter settings of CNNs can vary a lot, we utilize Keras, a library that allows for automated machine learning during the hyperparameter tuning process. The concept around Keras involves searching over a pre-defined search space to find optimal CNN architectures.

The data we deal with consists of 2D cross-sections of 3D CT-scan of the core retrieved from Equinor's oil wells. The goal of this thesis is to create a model that learns from the 2D image data and successfully predicts the resistivity. Resistivity is a parameter among many others, e.g., permeability and porosity. The resistivity, together with other log measurement parameters are used to identify the lithological characteristics of rocks. Usually, well logs are interpreted by petrophysicists and geologists, but by performing robust modelling, one is able to provide useful information derived directly from data. This contributes to time and economical efficiency, as well as the possibility to adapt to other application areas.

To give the reader a brief overview of the thesis, we go over motivation, research goals, and the thesis outline in this chapter.

1.1 Motivation

Among many parameters, porosity and permeability are the most descriptive ones in terms of identifying rock characteristics. Resistivity is another important parameter that describes a material's ability to resist electrical current. It is related to the amount of dissolved salts in the water, and the distribution of water inside the pore space. For our application in the Oil and Gas industry, one of resistivity's main contributions is to determine the oil content and the oil-water contact, locating the separation of oil and water in a well. In addition, resistivity estimation has other applications as in hydrogeology for locating the water table, as well as providing information about the water contents and the contamination level.

By performing proper and accurate modelling with resistivity, one opens up for inference of other geological parameters such as permeability and porosity. Through this thesis we want to contribute to further the research in the geoscience and data science field, especially since there exists limited similar research on performing regression using image data.

1.2 Research goals

With the use of CT-scan images, we want to investigate the possibilities of modelling the image data to perform regression of resistivity. We will utilize the tuning library Keras, defining a suitable hyperparameter search space to find optimal architectures of CNN. Because we are dealing with a small data set, various regularization methods will be used to increase robustness and generalizability to avoid overfitting. In the end, the goal is to test the models' performances on a holdout test set for a final evaluation. With these results, we will be able to observe the reliability and confidence of deploying such models for application in real life.

1.3 Thesis outline

The structure of this thesis is divided into six main chapters. Chapter 1 has given the reader an introduction to the problem, motivation, and research goals. In Chapter 2 we delve deeper into various relevant theory regarding the thesis, such as geology, convolutional neural networks, and model validation. This chapter mainly gives the reader a background on what theory is needed for the rest of the thesis. Chapter 3 shows the application of background theory on our data set. Here, the data set is pre-processed and various CNN models are tuned and validated to observe different performances. These models are then compared, and we

then remain with a few optimal ones. Chapter 4 presents the results acquired in the process of using the optimal models to predict the holdout test set. In Chapter 5 we discuss the results of the selected models, looking at which architectures work better for the application, as well as revisiting our research goals. Chapter 6 involves concluding the thesis with accomplishments made, and what further work can be done.

Chapter 2

Background and Related Work

This chapter covers previous research related to modelling with CNN, as well as various background theory needed to perform regression of resistivity. First, we introduce some background on geology and resistivity of porous media. Then we go over three papers that we view as relevant for our thesis, covering various methodology that we apply later. Further, we go over pre-processing and preparation of the data. This section introduces the sequence of steps required for the data before we begin modelling. The learning structure used in this thesis is called CNN, a state-of-the-art methodology for modelling with image data. We go over the different types of layers, and emphasize why CNNs are effective for modelling with image data compared to the classic ANN. Further, hyperparameter tuning of CNN is covered, using Keras. Keras is a library that allows for automated machine learning, searching over a pre-defined search space of hyperparameters. Finally, model validation and selection will be covered to show the methods we use to evaluate the CNN models from tuning.

2.1 Well logs and Resistivity

In the oil and gas industry, evaluating reservoirs has an important role in the exploration of oil and gas [1]. With logging technology, geologists and petrophysicists have been able to measure the formation parameters for geological analysis. In such reservoirs, there are many rock and lithological properties that are important to investigate to tell the properties of the formations. Therefore a detailed record of measured parameters is put together, to present an overview. The well log contains several types of geological information extracted, which are categorized for the geologist to be able to navigate the desired information. For instance, a log using resistivity to determine the oil-water contact is shown in Figure 2.1. Here, the resistivity contrast, i.e., the rapid change in resistivity indicates that there is a separation between water and oil.

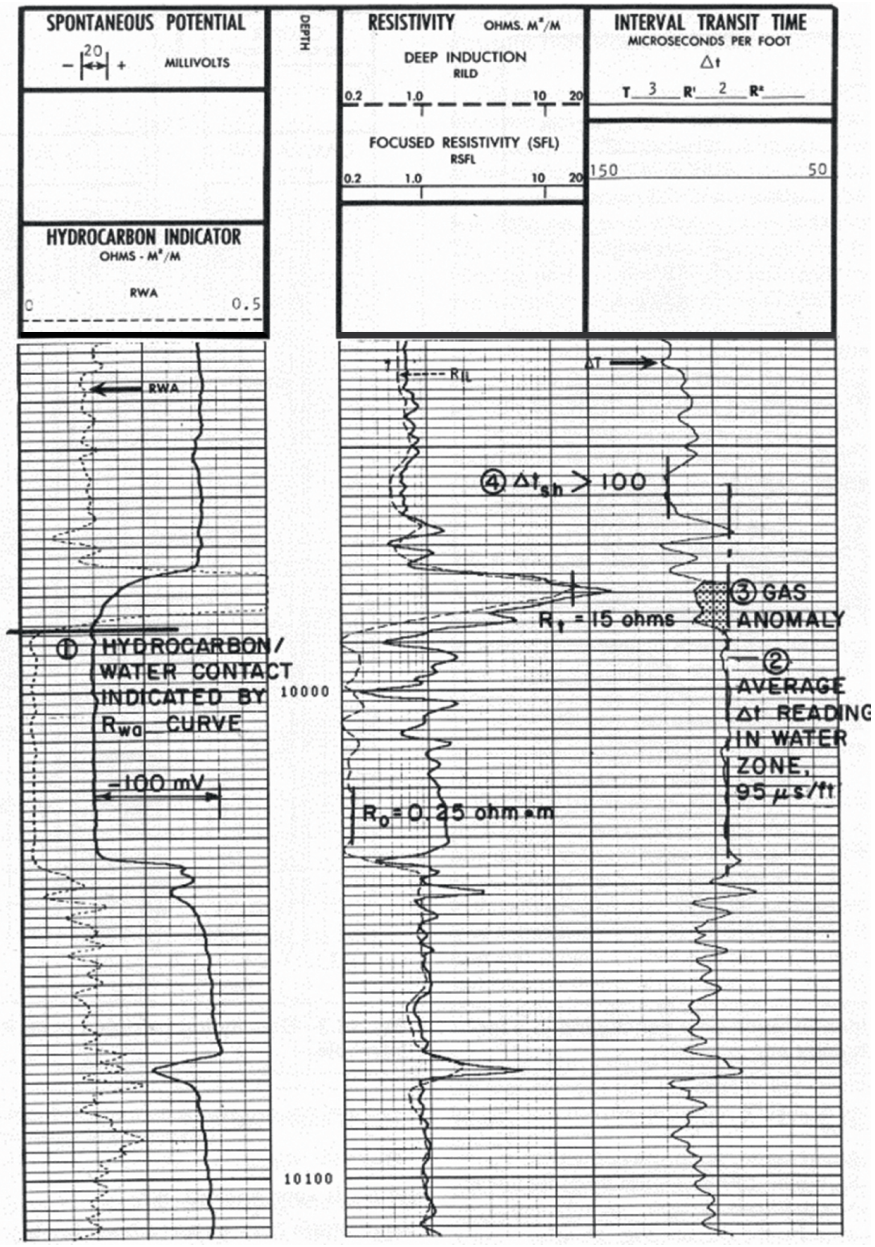


Figure 2.1: Example of a resistivity log used to locate the oil-water contact [2]

2.1.1 Resistivity log

We are most interested in electrical logs, where the resistivity parameter is the goal of our predictions. Resistivity is the property of a material to resist electrical current. This is linked to the material's porosity, where a porous rock will let the current flow through the mineral with ease, while a less porous rock makes it harder for the current to flow. The liquid enclosed in the mineral essentially acts

as a conductor, thus conducts electricity, while the mineral itself acts as an insulator. The current will travel along the path of least resistance, thus materials with low porosity have high resistivity and vice versa for high porosity materials.

This resistivity-porosity relationship can then be translated into providing information about the lithological properties of the formation. Resistivity has for instance applications such as in mineral exploration for iron and copper, geological disposal, and in hydrogeology for water-well drilling. Resistivity may for instance be an indicator of the water contents in the area, as well as providing information about the contamination level in the water. Our application area is in the oil and gas industry, where it is used for formation evaluation. One of the most useful applications of resistivity logs is to indicate the oil-water contact [3]. Since the oil contains hydrocarbons with high resistivity, and water has low resistivity, the transition between oil and water can be indicated by resistivity contrasts. This is although a major challenge as other factors than the water content, e.g. porosity, also affect the rock resistivity.

Mud invasion

A known challenge in well drilling is mud invasion, an event where drilling fluids are invading the pores in porous rocks. The invaded fluid can then displace some or all the water or hydrocarbon present. Resistivity logging in the formation then becomes disturbed, and may provide misleading information on the formation resistivity.

2.1.2 CT-scan imaging and CNN

In this thesis, we therefore want to investigate the possibilities of using CT-scan images of rock formations in a well to estimate resistivity with machine learning. Computed tomography (CT) is a tool that has been used in the oil and gas industry extensively for tasks such as imaging, characterizing lithofacies, and determining fluids in porous rocks. The idea in this thesis is to use CNN, a state-of-the-art learning method for 2D image data, to model the behaviour of resistivity.

2.2 Related work

This section involves presenting papers that are related to our research question. The first paper is from NTNU's BRU21 team that researches digitalization in the oil and gas industry. They have provided me with the data and a paper regarding 2D CT-scan imaging with CNN as well as project guidance. The second paper is about regression of the angle of digits and robotic arms, also utilizing CNN. As there is limited research on performing regression with CNN and image data, we found this paper interesting. The third paper is about data pre-processing techniques for

fault diagnosis, where generating data with overlap is presented. This is a central concept for pre-processing our data, which we will go over later.

2.2.1 Classification of rock type classes using CNN

BRU21 is NTNU's multidisciplinary program for digitalizing and developing technological contributions to the oil and gas industry in Norway. Some disciplines include Cybernetics and Robotics, Computer and Data Science, Petroleum Engineering, and Geoscience, centering around Ph.D. and PostDoc research projects.

From BRU21, Ph.D. student Kurdistan Chawshin from the Geoscience and Petroleum institute has worked on using 2D CT-scan data to perform classification of 20 rock-type classes. The paper presents a workflow of utilizing image data from an oil well to perform rock classification [4]. The paper involves applying CNN methodology together with Keras for hyperparameter tuning to find models for classification. We utilize similar methodologies of pre-processing and hyperparameter tuning, although to perform regression instead of classification. In the paper, a thorough analysis combining data science and geology is used both to perform classification with CNN and interpret the results.

2.2.2 Regression of angle for handwritten numbers with CNN

Even though CNN has been a state-of-the-art learning method for handling image data, most of the solutions revolve around classification tasks [5]. There is therefore a lack of research that utilizes deep learning for regression. Paper [5] is about regression performed with CNN, predicting a rotation angle for digits and a robotic arm. The paper performs experiments first on digits presented in Figure 2.2, then on the robotic arm data. They utilize 5000 digit data for training and 5000 digit data for testing. Further, they used 6859 data for training and 5832 for testing on the robotic arm.

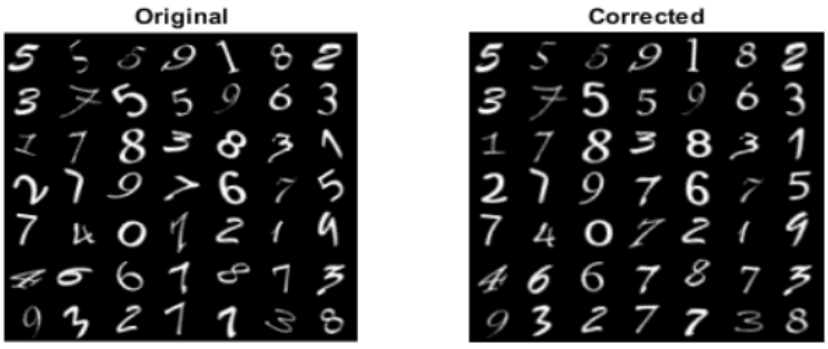


Figure 2.2: Regression of angle for handwritten numbers [5]

The paper utilizes four CNN architectures: a handcrafted CNN architecture,

and three pre-defined architectures known as LRF-ELM, H-ELM, and AlexNet [5]. The hyperparameter settings are presented in the paper and achieve good results for both regression tasks for all four proposed models.

2.2.3 Data pre-processing techniques for fault diagnosis with CNN

The paper goes over several pre-processing techniques used for intelligent fault diagnosis with CNN. The fault diagnosis is for rotating machinery, where data has been gathered from the industry. However, there is a limited amount of data because of the difficulties to obtain sufficient real fault data [6]. They then propose various pre-processing techniques where one of the motivations is to increase the data set size. One of the pre-processing methods uses data augmentation with overlap to generate more training data. This is presented in Figure 2.3, where vibration signals are augmented. In our case we will use a similar concept to generate more training data, but with image data.

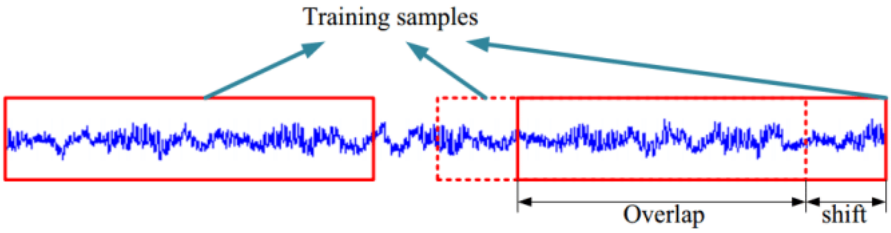


Figure 2.3: Generating fault diagnosis data using overlap [6]

2.3 Data set pre-processing for preparation of data

The data used for modelling is 2D CT-scan data of core from a well. 142 meters of core were retrieved where each image represents 1m of CT-scan imaging. The depths shown in some plots will be numbered as the respective location inside the 142 meter interval. In reality, these depths can be much deeper, but because of confidentiality purposes, we decide to use the relative scale of 142 meter interval.

Before modelling with CNN, these images had to be pre-processed to be used as input. Figure 2.4 shows an interval of 5 meters where each column represents one meter of well sample starting from the left side. From Figure 2.4 there parts with missing core, giving low-quality data that may disturb the performance. These are occurences in the data that do not provide relevant information, acting as noise, and is why pre-processing is necessary.

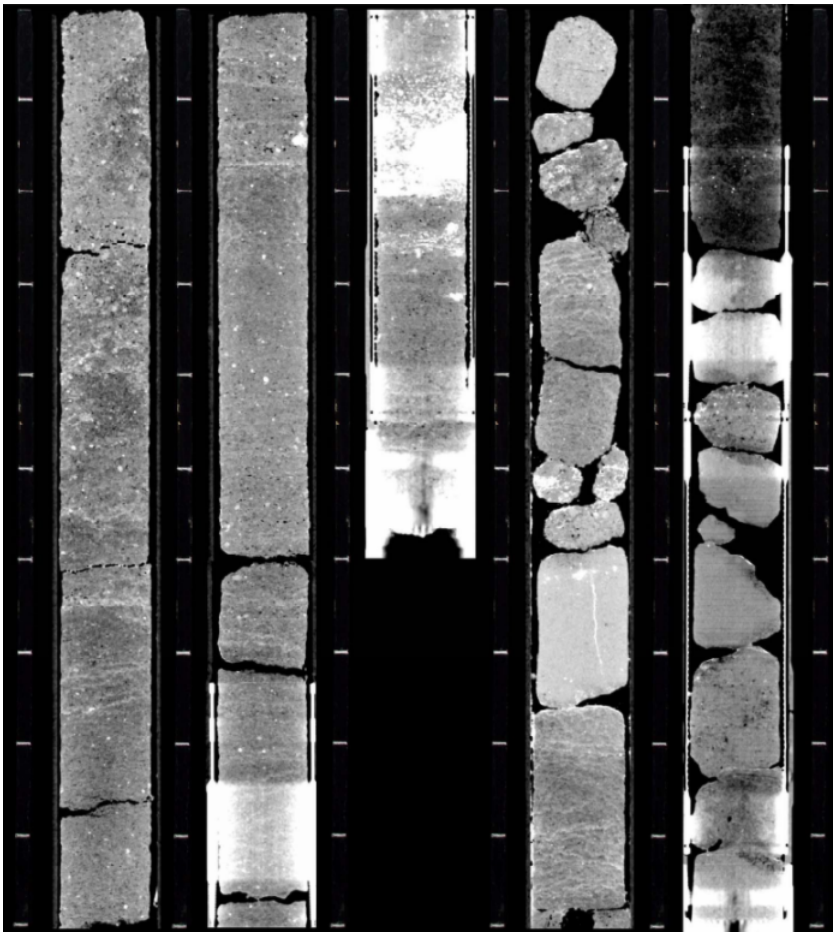


Figure 2.4: CT-scan image of well sample with five 1m sections

An artifact is to be observed in the middle column, showing a white vertical square. This is caused by core barrel couplings, having higher attenuation values [7]. Above the white rectangle there are additional areas of brightness, caused by mud invasion. A missing interval is also present from the middle to the fourth column, where core in-between these depths are not present. There are also cracks in the samples as shown in the columns on the right-hand side.

Additional to removing noise, we augment the data with overlap, generating more data, previously presented in Figure 2.3. This is because our data set size is originally small. Another augmentation method we use is flipping the images vertically and horizontally used as a regularization measure to prevent overfitting. In addition, an increase in data set size is also obtained by flipping.

The idea is to first pre-process these images, then use the cleaned and prepared data as input to the CNN model. With CNN, features are extracted from the images used to predict resistivity with regression. To give the reader a brief overview

of the pre-processing process, Figure 2.5 shows a flowchart describing the various steps required for the preparation of the image data.

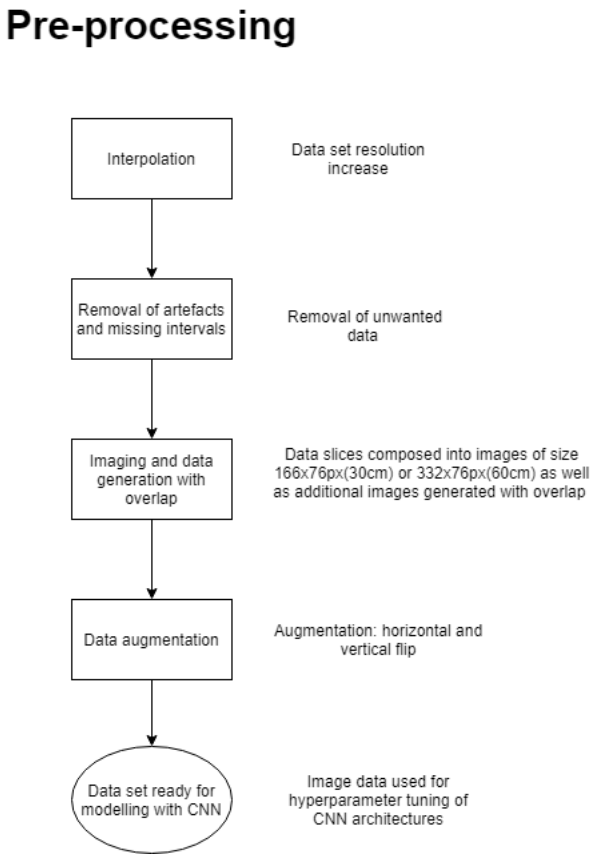


Figure 2.5: Flowchart showing different steps of data set pre-processing

2.3.1 Interpolation to increase data set resolution

As a first step of pre-processing, interpolation was performed on the data to increase the data set resolution. Interpolation involves estimating new data by constructing new data points between the ones already known. This means that between each resistivity-depth pair, additional data points are estimated and added. To interpolate, one needs a mapping function f, that is created from the original data. Figure 2.6 shows an example of linear interpolation.

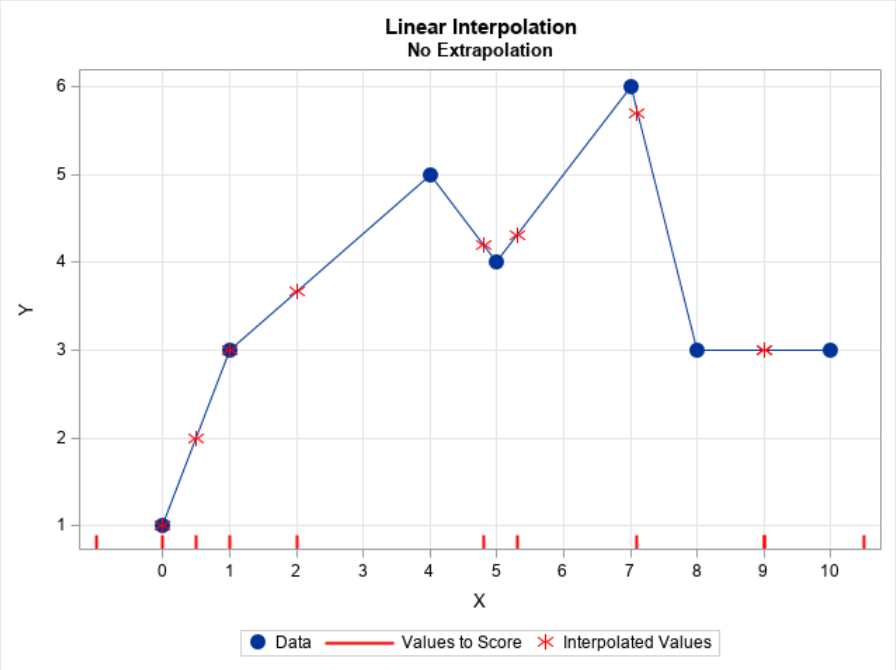


Figure 2.6: Example of linear interpolation, where the blue data points are the original ones, and the red are interpolated [8]

Interpolation was done for the 932 data points in python, creating an mapping function f that maps a given depth x_{depth}, to a new resistivity value $y_{\text{resistivity}}$.

$$f(x_{\text{depth}}) = y_{\text{resistivity}} \tag{2.1}$$

With this new estimated function f, a new given depth x would map a resistivity value.

2.3.2 Artifacts and missing intervals acting as noise

Three instances were discovered in the data set that were regarded as disturbance for modelling. These were missing data intervals, core barrel couplings and high-density areas. The missing data was most likely caused by poor core recovery, induced fractures, or rush plugs taken after retrieval of the core [4]. Another instance of unwanted data was bright areas caused by core plugs and mud invasion. These are described in Figure 2.7.

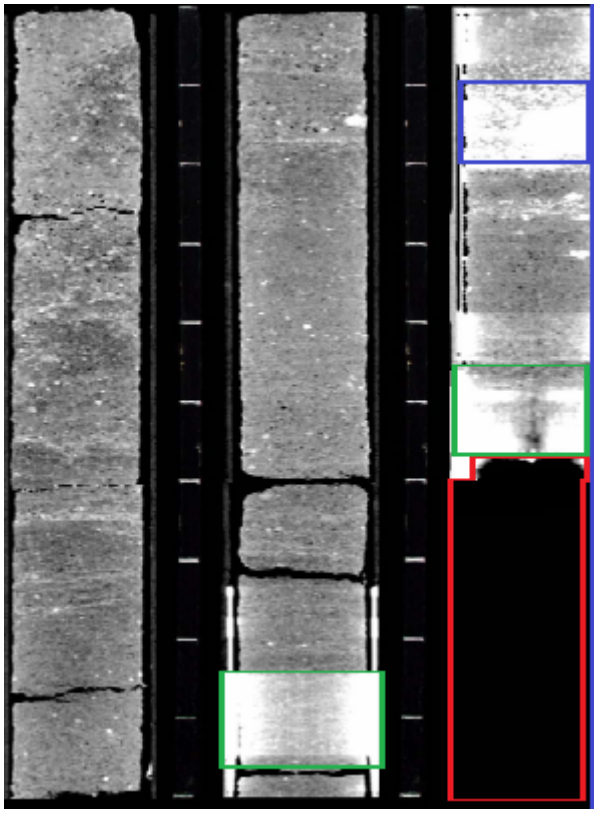


Figure 2.7: Instances of noise and disturbance: color coded in red(missing data), blue(mud invasion) and green(core barrel couplings). Inspired by [7]

These instances had to be discovered and removed before further pre-processing of data. If not dealt with, they would likely act as noise and disturb the performance of the model during tuning. Code from [4][9] was used to find the intervals with noise and remove them.

2.3.3 Data augmentation for regularization

Modelling with neural networks generally requires sufficient data to efficiently learn. Collection or generation of such data is often expensive and hard to retrieve [10]. By the use of data augmentation methods one is able to artificially generate unique data, thus increasing the data set size and the data variety. Additionally, data augmentation contributes to increased robustness during modelling by performing simple modifications to the original data.

Take for instance a data set of dogs where the majority of the dogs are facing to the left. By flipping the image horizontally, one is also able to capture dogs facing to the right, preventing the model of overfitting to only classifying left-facing dogs [11]. Data augmentation therefore acts as a regularizer, increasing the generalizability of the model to new, unseen data. For instance, the study by [10]

shows that using generic data augmentation methods such as rotating, flipping, and cropping contribute to a richer training data set with less overfitting in their object detection application.

In this thesis, we experience having a rather small data set size of 142 CT-scan images covering 1 meter each. It is therefore desired to increase the data set size using data augmentation. The augmentation methods we will use are

- **Data generation with overlap**: 98% overlapping sliding window for data generation
- **Horizontal and vertical flip**: flipping of images horizontally and vertically to introduce modified versions of the original data

Data generation with overlapping images

In this thesis, the data covered is retrieved from one well, ranging between over 142 meters interval. Each image is 1m long, thus the data set size is very small. To increase the data set, a window of 98% overlap has been slid over each image to generate more data. Each sliding of a 1m image should have resulted in 40 images with a step size of 2%. A study done by [6] shows a similar approach, only with vibration signals where using overlap with a sliding window is done to obtain a bigger data set. Figure 2.3 previously presented shows their approach. To further increase the data set size, we break down the 1m intervals into smaller image sizes. The sizes used in this thesis are either 30cm or 60cm images. Application of data generation with overlap and division into smaller images will be presented later in Chapter 3.

Horizontal and vertical flip

As a measure to increase robustness, augmentation with flipping will be done to attempt to present "unique" images to the model. The idea is to let the model see modified images that can be candidates for future predictions outside the training set. The motivation for flipping is that if you flip a whole well upside down, the sequences of rocks will be the same, just reversed, thus the resistivity also remains the same. We will test models with and without flipping, to compare performances.

To perform flipping, two augmentation methods have been used. The first method generates additional data and increases the size of the data set with the new copies. The augmentation is done before training the neural networks since it is desired to increase data set size. The other method involves creating copies "on-the-fly", meaning it does not expand the data set before training, but instead, for each training instance, during the training phase. The augmentation happens in-place, where the model sees a new augmented version of the data. This ensures that the model sees a new, unique version of the data as an attempt to prevent the same

images to occur during training. This latter augmentation method also reduces the amount of data needed to be processed overall, leading to less computational load and RAM required.

2.3.4 Normalization of data input

As a last step before the data can be used for modelling with CNN, normalization of the image data was done. In machine learning, time is an important factor, especially for real-time applications [12]. The ranges of inputs of data can be big, thus slow down the calculation processes of neural networks. Applications such as self-driving cars and speech recognition are examples of where time may be an important factor.

Normalization is about bringing the ranges of values the input can have to a more common scale. For some applications normalization can be crucial, as big input values can have more impact than smaller values in neural network computations. By normalizing the data, the idea is to prevent this bias from occurring. In this thesis, we use Min-Max scaling to map the input data to inputs between 0 and 1. This is done by dividing the original input by the maximum value of all inputs given by

$$X_{\text{norm}} = \frac{X_{\text{old}}}{X_{\text{max}}} \tag{2.2}$$

For our project, the maximum value for our image data is 255. This means we have to divide every image data input by 255.

2.4 Supervised learning

The machine learning process used in this thesis is called supervised learning. The name supervised learning comes from the learning process being directed by a supervisor [13]. Here the supervisor is the human that in advance has labeled and split the data which the algorithm learns from. The class label and data set features are therefore known in advance, and the goal is then to use this known data to train a model that maps the input x to the output y. The input x will be the image data, trained on with the learned model structure CNN to perform regression of resistivity y.

When considering images of 60cm height, the image data set consists of 2467 images. Based on the rock information, a logging tool has been previously used to used to label the resistivity of the rocks throughout the well. Machine learning with CNN is then applied to use this labeled data to learn how to predict the resistivity of future, unseen images. Figure 2.4, previously shown, shows five non-processed 1m intervals, where each interval carries a sequence of resistivity values. These resistivity values are the objective of predicting, by learning from the labeled image data.

2.4.1 Regression

Regression is the problem of finding the mathematical relationship between the continuous variable y and a set of feature variables **X**. More precisely, given a data set **X** and a target label y, the objective is to find a mapping function $f(x)$, x. By inputting a data point x, the mapping function outputs an estimate of y. In machine learning, this involves training a model on a data set **X** where the target variable y is already known. By learning from the data, the model estimates the mapping function that describes the relationship between the data and the target variable.

Simple linear regression example

A simple example is using linear regression to estimate a person's shoe size based on their height. Here, the height is the feature variable x, which is used to estimate their shoe size y. Figure 2.8 shows sample data of a set of height and shoe size pairs that are plotted.

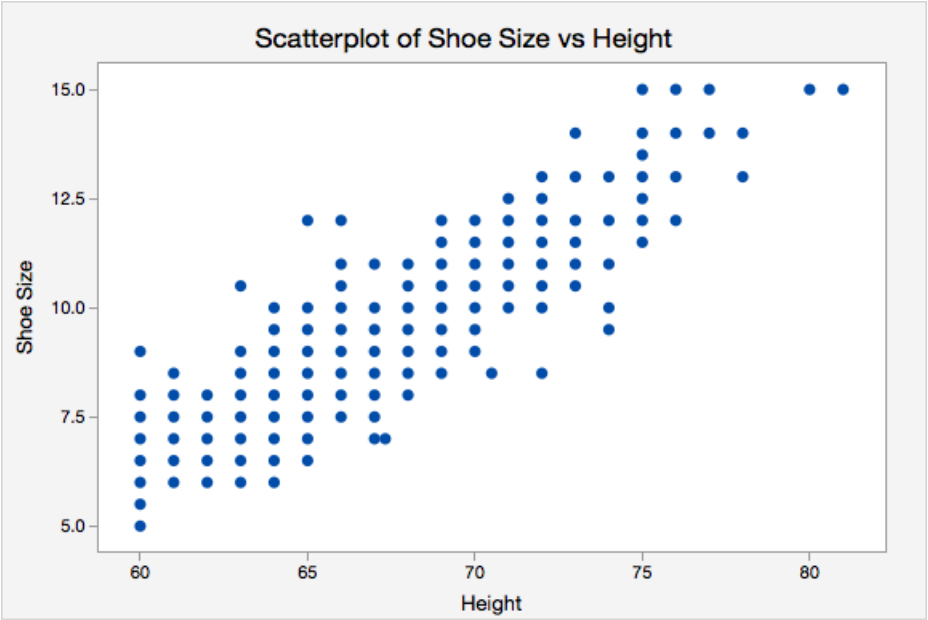


Figure 2.8: Data set example of using height(x) to infer the shoe size(y) [14]

At first glance, we observe that there is a linear relationship between the two variables. The objective is then to find the best fitting curve that describes the relationship between the data **X** and target variable y. In this example we only have one feature variable, x the height which is used to infer the target variable y, the shoe size. Since we only have one feature variable, this means to find a line

$$f(x) = \beta_1 + \beta_2 x \tag{2.3}$$

where $f(x)$ is a mapping function that predicts the shoe size, and x is the height. β_1 is the offset of the line, and β_2 is the slope. These two variables are then adjusted to fit the best line possible. To know how to adjust β_1 and β_2 a loss function is used, describing the distance from the line and our prediction, known as residual r. The residual for a given point i is given by

$$r_i = y_i - f(x_i) \tag{2.4}$$

Least squares is then the sum of the residuals squared given by

$$L = \sum_{i=1}^{n} r_i^2 \tag{2.5}$$

The goal of this example is then to adjust β_1 and β_2 to create a line that minimizes the least-squares loss function. There are several ways to solve this, where one example is iteratively by testing out different values for β_1 and β_2. The line with the smallest L is then our solution.

In this example we only had one feature variable x, so the example is pretty simple. For bigger problems where there are multiple feature variables, the problem scales in dimensionality where a simple line would no longer be a sufficient solution. This means that more complex learning methods than linear regression may be needed, as the relationship between **X** and y often becomes nonlinear.

Regression of resistivity with CNN

Generally, regression is done with numerical data, together with classical machine learning algorithms such as support vector machines, decision trees, or linear regression. In this thesis, 2D CT-scan images are used as input x to find the target value of resistivity, y. To perform regression using images as input data, the learning method CNN is used. For every 2D CT-scan image, a resistivity value is assigned. With CNN, features are extracted from the images x, and are then used to model the mapping function $\hat{f}$ that estimates resistivity y.

2.4.2 Bias variance tradeoff

The big challenge in machine learning is to create a model that is good at predicting new, unseen data. We then say the model has good generalizability. Looking at two scenarios: a nonlinear model and a linear model. The nonlinear model is often more complex and powerful, and in general achieves better accuracy than the simpler, linear model. But should we always use nonlinear classifiers? This question is answered by analyzing the bias-variance tradeoff.

Variance

Variance is known as the variation of predictions of our model, defined as the average deviation of our prediction $\hat{y}$ from the mean of our estimate $\mathrm{E}[\hat{y}]$ given

by the following equation

$$Var(\hat{y}) = E\Big[\big(\hat{y} - E[\hat{y}]\big)^2\Big] \tag{2.6}$$

In practice, this means that the model pays a lot of attention to the details when modelling the relationship between x and y. This relates to that the estimated function $\hat{f}$ has high model complexity, but has low generalizability since it is too used to model the training data. Models with high variance generally perform well on the training data as high variance is a sign of high model complexity. Although, when applying the high-variance model on new, unseen data, the performance is expected to be bad, as it is too familiar with the training data. This concept of high variance is visualized in Figure 2.9. As the variance increases, the model complexity follows, and the total error also increases. It is therefore crucial to tune the model parameters that constructs $\hat{f}$ in a way that we achieve low variance.

Bias

Bias is the difference between the average prediction of our model E[$\hat{y}$], and the ground truth y which we are trying to predict [15], given by

$$Bias(\hat{y}) = E[\hat{y}] - y \tag{2.7}$$

Bias can be seen as the simplifications and assumptions a model makes when learning the target function [16]. A model with high bias learns fast but struggles to learn complex characteristics of the data. Simple models therefore have high bias. Models with low bias are on the other hand more capable of adapting to complex behavior in a data set, but learn slower. These are categorized as more complex models. Both scenarios of high and low bias are unwanted, as they lead to high-error modelling, presented in Figure 2.9.

2.4.3 How the bias-variance tradeoff affects modelling in practice

The bias-variance tradeoff is tied to the complexity of a model. A very complex model will be good at modelling the training data set since it pays attention to the details, and often achieves good training accuracy. But when introducing the model to new, unseen data, the model will be bad as it is too "used" to modelling the training data. We then say the model is overfitted to the training set since it rather remembers the data rather than learns from the underlying semantics. These models are classified as more complex models and are known to have high variance and low bias. A very simple model will struggle at modelling the training data set as it is too simple to be able to adapt to complex patterns and characteristics of the data. Take for instance a linear model trying to model the characteristics of a highly non-linear data set. We then say the model is underfitted and has high bias and low variance. These two phenomenons are shown in figures 2.11 and 2.12.

Loss functions and metrics for scoring

In machine learning, to optimize a given mode, loss functions are used. The loss function is referred to as the objective function, where the goal is to minimize the learning error by tweaking the parameters of the learned model $\hat{f}$. In regards to this thesis where regression is used, the loss function mean squared error is one of the candidate loss functions that can be used. Given a regression problem, we have the mapping function f that is estimated by our CNN. The mapping function $\hat{f}$ outputs a resistivity prediction $\hat{y}$ dependent on the input image x. The goal is to estimate this mapping function $\hat{f}$ that describes the relationship between x and y as close to the real relationship as possible. We then evaluate the goodness of the fit of $\hat{f}$ to the data set based on MSE, i.e. the learning error. In this thesis, we decide to use mean-squared error, MSE as our main loss function. Mean squared error, MSE is a metric for the deviation of our prediction $\hat{y}$ and the ground truth y squared given by

$$MSE(\hat{y}) = E\left[(y - \hat{y})^2\right] = Var(\hat{y}) + \left(Bias(\hat{y})\right)^2 \tag{2.8}$$

MSE says how much the mean of our regression prediction $E[\hat{y}]$ deviates from the original y [15]. The goal of the regressor is to minimize this learning error, and can be achieved by changing both bias and variance, hence the bias-variance tradeoff. This tradeoff is something the supervisor has to take into account when modelling, using different techniques to balance both bias and variance, thus keep MSE low. Figure 2.9 illustrates the tradeoff.

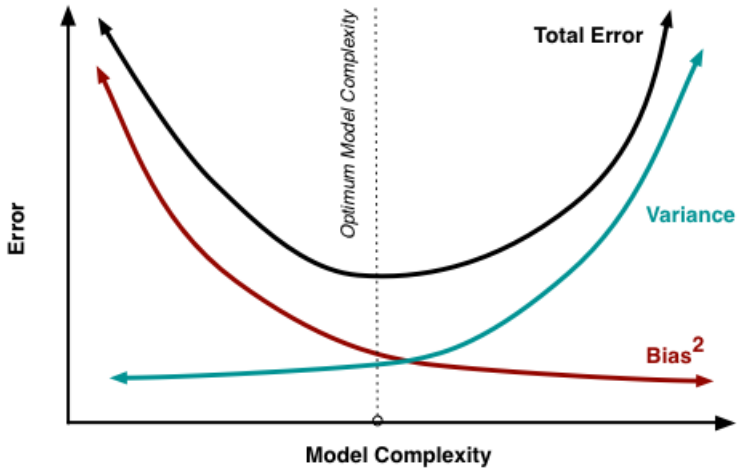


Figure 2.9: Visualization of the bias-variance tradeoff with total error, MSE [17]

As visualized, the challenge is to avoid having both high bias and variance, as this affects the learning error negatively. The sweet spot would then be to have both low bias and variance. Figure 2.10 intuitively visualizes the tradeoff with a

bullseye diagram. High variance leads to a wide spread in predictions, and high bias results in large deviations from the target value.

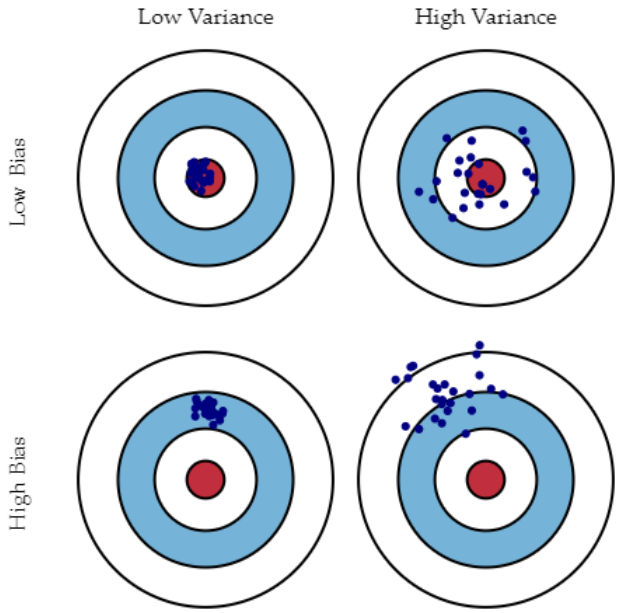


Figure 2.10: Visualization of the bias-variance tradeoff with four bullseye-diagrams [17]

Adjusting bias and variance in practice

How is low variance and bias achieved in practice? Essentially, it is achieved by implicitly adjusting the bias and variance in the model by tuning the hyperparameters. One wants to avoid having a too simple model, as bias tends to be high, and a too complex model where variance is high. When tuning, one can start with a simple model to "test out the waters", then iteratively increase model complexity to investigate the need for a more complex model. For instance take a simple neural network regression problem: starting with one hidden layer and some neurons, then adding more layers and neurons to increase model complexity. This follows the principle of Occam's razor, which says that among several competing models with similar scores, but different model complexity; pick the model with the lower complexity. Lowering the model complexity means lowering the variance, and contributes to better generalization.

Early stopping is a simple, yet effective regularization technique among neural networks. It revolves around stopping the training of a model when accuracy stops improving. As the model trains longer, the complexity of the model increases, since the weights and biases in the network increase. This causes variance and MSE to

increase as the training data is iterated over for longer than needed, thus causing the model to risk overfitting to the training data.

2.4.4 Overfitting

Overfitting and underfitting are events that are consequences of the bias-variance tradeoff. A model is overfitted if it pays much attention to the details of the data set. Such models don't learn and generalize from the data set, but rather remember the data. They often have low MSE on the training data, but when new unseen data is presented to an overfitted model, its generalizing ability is low since it is too "used" to model the training data. An underfitted model has the opposite characteristics of an overfitted model. Its model complexity is low, hence its ability to adapt to new data is low. Both of these cases are visualized in Figure 2.11 and 2.12.

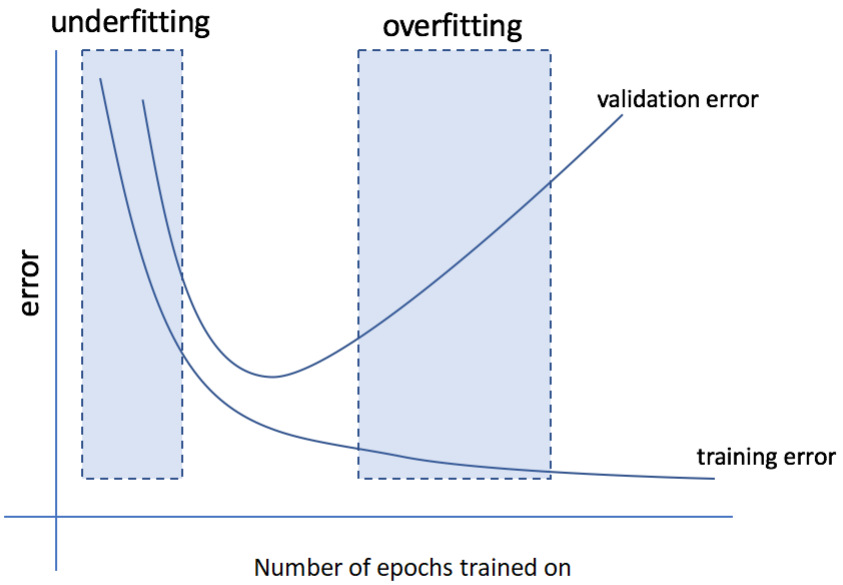


Figure 2.11: Visualization of overfitting and underfitting during the training process. Modified of [18]

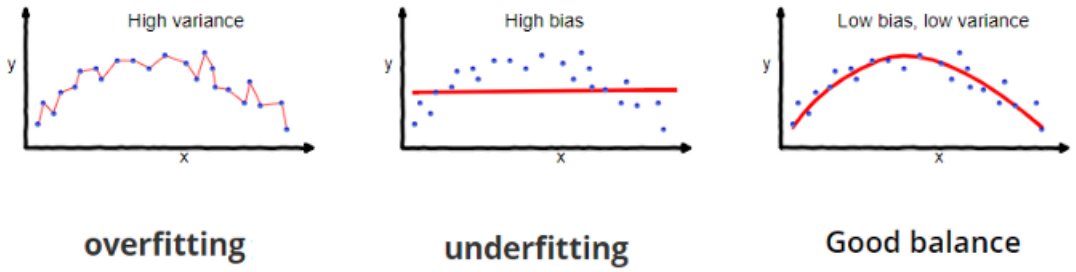


Figure 2.12: Visualization of an overfitted, an underfitted and a balanced model [19]

Data leakage

Data leakage is an important concept regarding ML that can cause models to overfit. It involves using information outside the training data to perform predictions, inserting bias into the modelling. The goal of predictive modelling is to create a model that is good at predicting unseen data. We then say the model is good at generalizing to new data. Data leakage is the event of using information from the test set to purposely improve the score of the model. We then say that test data has leaked into the training set.

In a general 3-way split, covered later in Section 2.8.2, a training set is used to train models, a validation set is used to tweak the model performances, and the test set is held outside untouched. The challenge is to use the training and validation sets to create robust models with good generalizability, and finally, when an optimal model has been found, the test set is finally predicted. By splitting it this way, we prevent test data to leak into the training set.

2.4.5 Regularization to combat overfitting

To combat overfitting, regularization methods can be used. Regularization methods are machine learning techniques used to lower the gap of generalization error. Generalization error is known as the error of predicting new instances, i.e. the test error. As introduced earlier, an overfitted model might perform well on the training set, but when introduced to new data the performance may be bad. The following regularization techniques will be used in this project.

- Early stopping
- Data augmentation
- Dropout

We have already gone over data augmentation and early stopping slightly. Dropout and early stopping will be discussed more later in this chapter.

2.5 Artificial Neural Networks

An artificial neural network is a machine learning algorithm that tries to replicate the biological behavior of the brain. It goes by several names: artificial neural networks, deep neural networks, feed-forward neural. The main goal of a neural network is to approximate a function $\hat{f}$ by learning a mapping $\mathbf{y} = f(x;\theta)$ where x is some data, and θ the model parameters. In theory, a neural network with one neuron and one hidden layer can approximate any possible function. But by increasing the number of hidden layers and neurons, allows the model for much more computational power and better adaptive ability. An example of a neural network with three hidden layers with four nodes in each layer is shown in Figure 2.13.

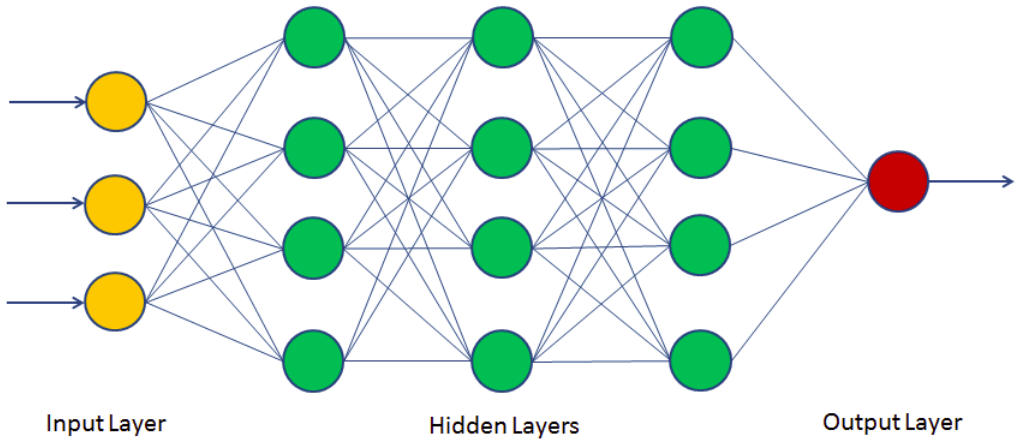


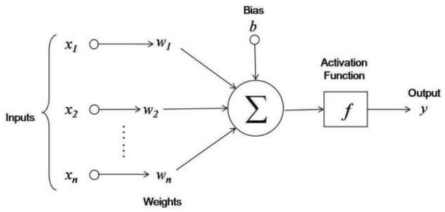
Figure 2.13: Example of feed-forward neural network with three hidden layers and four neurons in each layer [20]

2.5.1 Hidden layers and neurons

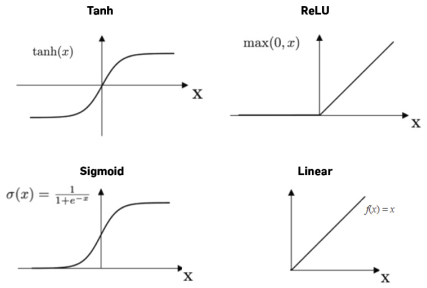
A neural network consists of an input layer, n number of hidden layers, m number of neurons in each layer, and an output layer. The neurons have connections from previous layers and form a network by having outgoing connections to the next layer. Each node processes the output from the previous layer, processing the numbers and outputting it to the next layer [21]. The processing is a linear transformation of the input with adjustable weights, a bias, and an activation function [22]. The weights are set and continually adjusted to fit the function $f(x)$ better. The activation function is a non-linearity that maps the input to the output, and the bias shifts this non-linearity. The linear transformation can be given by

$$\mathbf{y} = g(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{2.9}$$

where y is the output, g is the activation function, $\mathbf{W}$, are the weights x are the inputs and $\mathbf{b}$ are biases. Figure 2.14a shows the linear transformation of one neuron.



(a) Linear transformation within one neuron. Weights multiplied with inputs are summed, and fed into the activation function, resulting in an output [20]



(b) Typical activation functions: tanh, sigmoid, ReLU and linear function [23]

Figure 2.14: Linear transformation within a neuron on the left. Four typical activation functions on the right. Inspired by [22]

Modelling with neural networks has become more popular through the last decades, utilizing their ability to perform complex tasks such as classification, regression, reinforcement learning, and so on. Neural networks' capability to adapt and surpass classical ML methods has attracted more users, especially in pattern recognition. In theory, a neural network with one neuron should be able to estimate a function for prediction, but a deeper and wider network allows for more computational complexity.

This thesis revolves around utilizing 2D CT core-scan images to predict resistivity from well logs. CNN, a variant of the typical ANN is known for being state of the art for image processing and is therefore used as the machine learning algorithm. There exist a multitude of different CNN architectures for different applications, and in this thesis, tuning and investigation of different architectures are done to see if they can adapt to our image data. The tuning and searching of CNN architectures are shown later in chapter 3, Section 2.7.

2.5.2 Activation functions

The activation function is a non-parametric function that processes the node input to an output. This can be as simple as an "ON"(1) and "OFF"(0) gate, or something more complex as a continuous function [24]. Popular activation functions are for instance the sigmoid or ReLU shown in Figure 2.14b. From each node, the weights and outputs from the previous layer are multiplied and summed, then passed into the activation function. Based on some threshold, the activation function maps

the sum of inputs and generates an output.

The non-linearity can be shifted by adjusting the bias to adapt to an appropriate value range based on the inputs. It is therefore normal to normalize the data to avoid the summation reach saturation both on the lower and upper end [24]. The selection of a proper activation function is therefore an essential part of neural network modelling since it is used for computation in each node.

The activation function essentially acts as a summary of the input to the output. A node in a neural network may process hundreds or thousands of inputs from the previous layer. Therefore it is effective to have activation functions that squash these inputs into a less complex number.

In this thesis, the ReLU activation function is used extensively during modelling with CNN shown in Figure 2.14b. ReLU maps the input of the node to a number between 0 and the maximum value of all inputs received by the node. ReLU is given by equation Figure 2.15 and 2.10

$$f(x) = max(0, x) \tag{2.10}$$

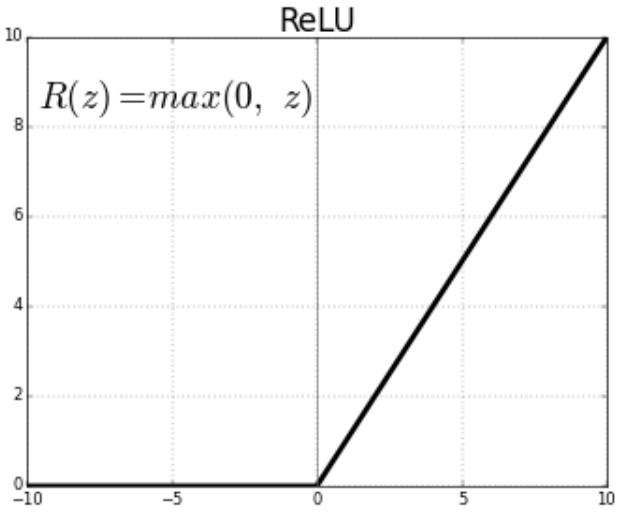


Figure 2.15: ReLU activation function, visualizing equation 2.10 [25]

where x is the input from all connections, and $f(x)$ is the output to all outgoing connections. The reason we choose to use ReLU is that the calculation of the gradient is simple, [26] shown in equation 2.11, which makes backpropagation with ReLU computationally low.

$$\frac{\partial f(x)}{\partial x} = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases} \tag{2.11}$$

2.5.3 Training neural networks

During the training of a neural network, information flows through the network and produces a final resistivity prediction. In a supervised learning problem, the network knows the ground-truth label of the prediction for training. To know how to adjust to errors, the network uses backpropagation, involving a loss function to perform gradient descent. The gradient of the prediction tells us how far away we are from the optimal solution, and allows us to adjust the weights of our neurons. For each backpropagation, the loss from the loss function gets smaller and helps to notify the model to adjust its weights. This adjustment of weights is what helps the network to produce better predictions as the network trains for longer. The name backpropagation comes from the fact that the calculation of the gradient propagates backward in the network. It starts calculating the gradient of the loss function at the output layer, propagating backward for each hidden layer, ending up on the gradient of the first layer.

During training, when weights and biases get adjusted, the variance and bias in the model get affected. To decrease the risk of overfitting during training, two regularization methods are specifically applied called early stopping and dropout.

Epochs and early stopping

One iteration of sending training data forwards and then backpropagating until the model has seen all training data once is called an epoch. For each epoch, the weights of the network are adjusted hundreds or thousands of times, depending on the size of the training data. This is the process where the network gets more "known" to the data and is where the learning happens. After a multiple number of epochs and weight adjustments, the network's loss function should converge to a minimum, as for every epoch, the weights get adjusted to the error. The idea is through this process to let the neural network learn from the underlying complexities of the data by adjusting the weights. One should therefore be mindful of selecting the number of epochs to avoid overfitting as the network's weights and biases change when training for longer. This is because the variance and bias increase for each epoch, leading to increased model complexity.

As introduced earlier, early stopping lets the user automatically stop when a neural network's loss is stagnating. Stagnating means that the network is not learning more, but rather continuing training and updating weights, increasing the model's variance. This alone can lead to overfitting, and the challenge is then to stop at the sweet spot where the network has trained enough. Early stopping can be performed manually by investigating the learning behavior of the model through graphic plots, or automatically by setting a *patience* parameter that stops the training whenever there is no decrease in loss over a set number of epochs.

Dropout

Another regularization method for neural networks that is both cheap and effective called dropout. The way dropout works is to randomly remove nodes from each hidden layer during training in each epoch, creating different neural network models for each forward pass, and backpropagation pass.

What we often experience with neural networks without dropout is that some nodes are more active than others, called co-adaptation. Co-adaptation is often the root of overfitting since it makes so that some nodes are highly dependent on others. If this independent node receives a bad input, it may affect the dependent nodes to a large degree. Dropout then lets every node have an equal chance to contribute towards the prediction, thus reducing the chance of co-adaptation.

During training, given a dropout probability p, for instance, 0.15. p is then the probability of a node in a hidden layer being dropped out. The neural network then drops the node and its in-going and outgoing connections. The active nodes receive their input and perform forward pass and backpropagation, updating their weights. This is then repeated for every epoch. Dropout is only used during training and not during testing. During training, the neural network incorporates the characteristics of the different models into the last model so that the predictive model contains all nodes.

2.5.4 Artificial neural networks and its limitations with image processing

When it comes to modelling with image data, the standard ANN struggles to deal with the computational complexity. For instance, say a 64×64 image is used as input to an ANN. The neural network could probably deal with this kind of input, but each neuron would have 4096 parameters in each neuron. With increased image size, and usually also colored images, the parameters increase exponentially regarding how many weights the ANN has to deal with. Using ANNs to perform image recognition could in theory be possible, but probably not feasible when it comes to bigger scales. One could increase the size of the network with more hidden layers and neurons to compensate for the computational requirements for image processing.

With convolutional neural networks, the number of parameters in the network decreases exponentially, as CNNs are suited for grid-structure data. By utilizing the convolutional operation, calculations are simplified, and the computational requirements are reduced significantly.

2.6 Convolutional Neural Networks

A more suitable approach to modelling with image data is with the CNN. Convolutional Neural Networks are feed-forward neural networks inspired by the visual cortex in the human brain. The visual cortex arranged in a way that they are limited to only sense sub-regions of the visual field. The neural network then works so that connections of neurons within the network make the CNN able to cover the whole visual field, thus extracting patterns and information in parallel [27]. The significant difference between ANNs and CNNs is that CNNs are more suitable for image processing due to the convolutional operation. The convolutional operation is a pixel-wise operation together with a kernel and an activation function that results in an activation [4]. When the kernel is activated over the whole image, it learns to detect patterns such as corners and edges, resulting in a feature map extracted. These features extracted hold the relevant information about the patterns that the model learns from and uses for prediction of resistivity. One kernel might find corners, resulting in a feature map highlighting corners in an image, while another kernel might learn to find vertical lines. The activation with the convolution operation and the kernels are what makes the CNN effective with grid-like structured data, hence images are effectively decomposed and interpreted by CNNs. These can be of 2D, 3D, or bigger dimensional data dependent on the application.

There are three main layers that make the architecture of a CNN: convolutional layer, pooling layer, followed by the fully-connected layer. The convolutional layer is the building block of CNNs, and is where feature maps are extracted from the image data with kernels. The pooling layer is a layer that usually comes after the convolutional layer. The pooling layer summarizes and down-samples the extracted feature maps, then sends them to the next convolutional layer, allowing for features of different scales to be extracted. Usually, there are several pairs of convolutional and pooling layers before reaching the fully-connected layer. The fully-connected layer is the standard ANN, where prediction with regression happens, consisting of hidden layers with neurons. This sequence of layers is presented in Figure 2.16. In our problem, we have a regression task at the end instead of classification, so only one neuron is used at the output layer.

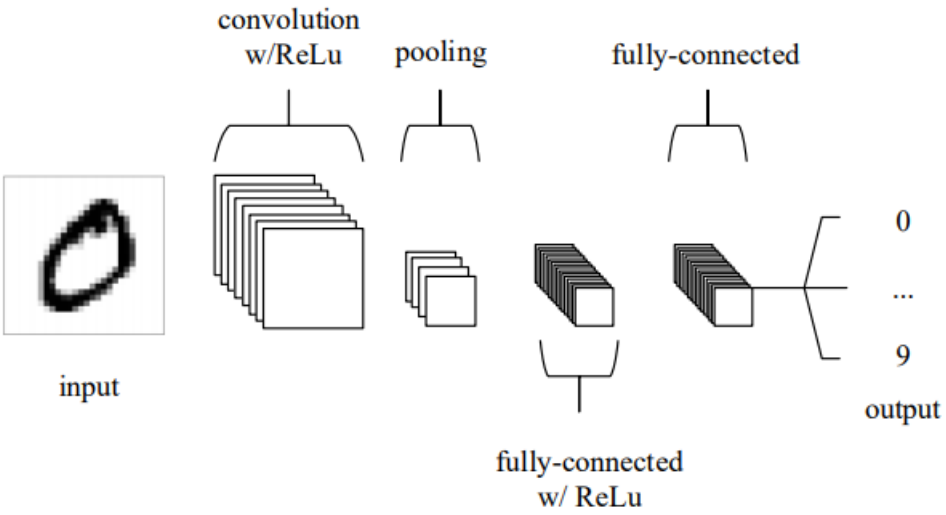


Figure 2.16: Standard architecture of the CNN involving feature extraction using the convolutional and max-pooling layer. A prediction is then produced from the fully-connected layer. In this classification example, handwritten numbers are classified from 0 to 9 [28]

2.6.1 Convolutional layers for feature extraction

The convolutional layer is the basic unit of a convolutional neural network. The convolutional layer uses kernels to extract details in images [26]. Kernels are known as the parameters of the convolutional layer, trained to learn specific patterns from the images. These patterns can be a corner, an edge, eyes, and so on. The convolutional layer utilizes these kernels together with the convolutional operation on the image, called an activation. Figure 2.17 shows an example of a kernel convolved with an image. This activation is performed over the whole image, resulting in a feature map. The feature map represents the information of which is specific to the kernel. For instance, the feature map from a kernel that detects edges will be different from the feature map from a kernel that detects eyes. Figure 2.18 shows eight kernels producing eight different feature maps over 5 sampling levels. For each level, the feature maps are downsampled.

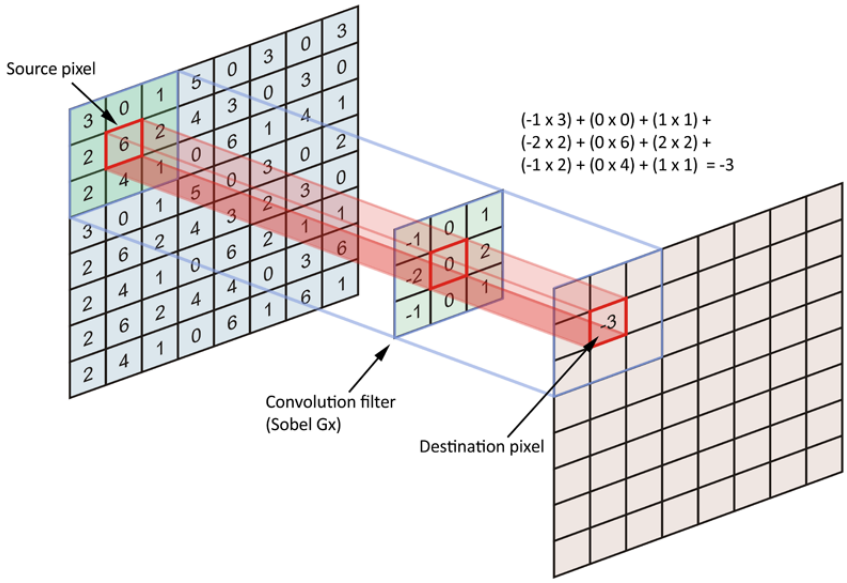


Figure 2.17: 3 × 3 kernel activated with a grid of data, producing a feature map [29].

These kernels resemble the human brain, where our visual field is limited to remembering small parts of patterns. Each kernel has its own characteristic, where for instance a line or a corner is learned. What is amazing, is that convolutional layers train multiple kernels at the same time, often hundreds of kernels, and can therefore recognize different patterns at once when seeing an image. The feature maps are stacked and sent forwards to the max-pooling layer where the feature maps are downsampled, decreasing the resolution. This results in kernels early in the network producing detailed feature maps of objects or patterns, while the kernels in the later layers produce more coarsened feature maps. This property allows the CNN to extract features from different scales regarded as low-level and high-level features. Figure 2.18 shows feature maps of an image of a cat over five convolutional layers. For each layer, the resolution gets more coarsened where each layer contains 8 kernels each, extracting different types of features.

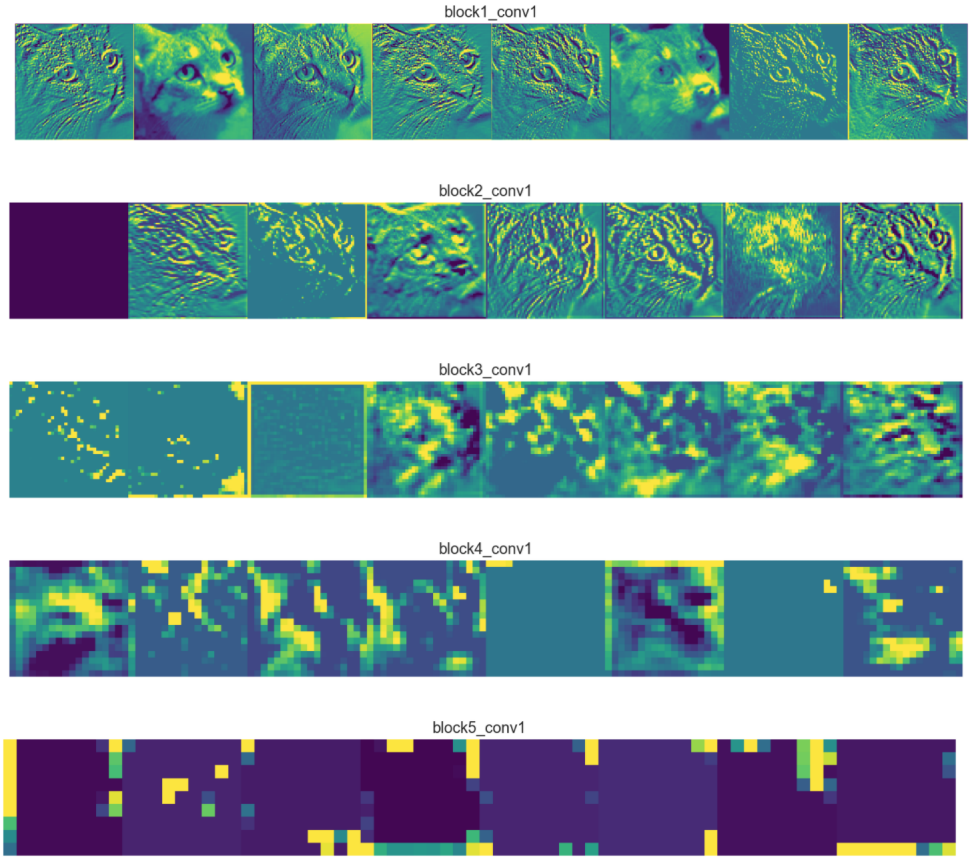


Figure 2.18: 5 levels of extracted feature maps, each row with 8 kernels. From each convolutional layer, feature maps are downsampled to capture different levels of features, represented by each row[30].

2.6.2 Pooling layer

After applying trained kernels to the input images in the convolutional layer, feature maps are generated. These feature maps are fed into the pooling layer, which acts as a "simplifier" by downsampling the feature maps. This is the main function of the pooling layer, reducing the complexity of feature maps for further layers [26]. The reduced feature maps are then used as input to the next convolutional layer, where the same process of extracting feature maps and pooling happen. Pooling helps to reduce the number of parameters trained in the network, thus reducing computational load. In this thesis, we use max pooling, one of the most popular pooling layers which extracts the most activated pixels in the images. An example of max-pooling is shown in Figure 2.19. Max-pooling works by returning only the maximum value inside of the kernel window. In this example, we have a 2×2 window size, which is commonly used. The stride is 2 which is the number of pixels we slide each time a pixel is sampled to the pooled feature map. This is also the property that causes down-sampling. To avoid down-sampling, a stride of

1×1 can be used, although is not commonly used [26]. Max-pooling can also be seen as augmenting the images, as the information loss causes coarsened or more blurred feature maps, acting as a regularizer.

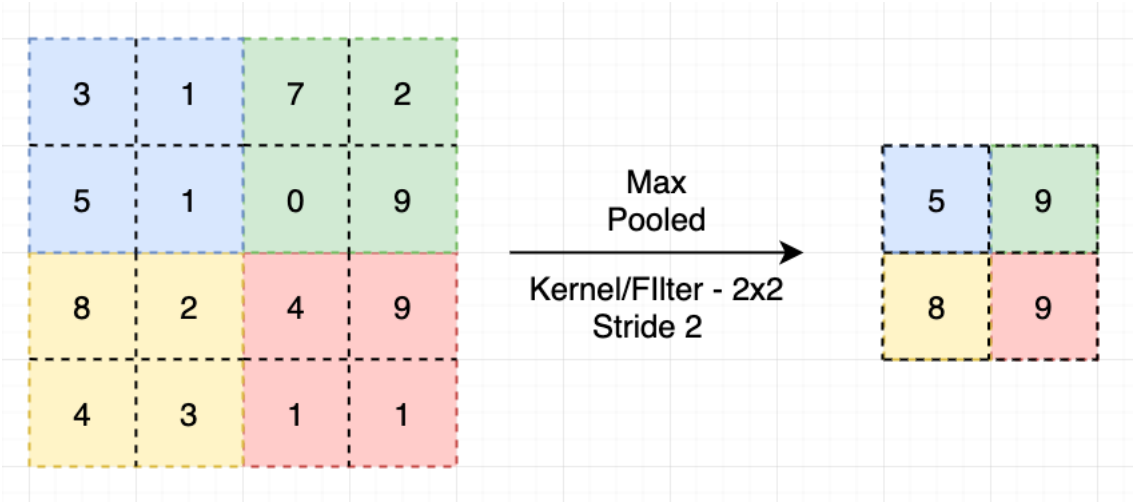


Figure 2.19: Max-pooling of feature map with window size and stride of 2 [31].

2.6.3 Fully-connected-layer: The regressor

The fully-connected layer is the classic neural network part of the CNN. After extracting feature maps and training kernels for pattern recognition, these feature maps are fed into the fully-connected layer. First, the output from the convolutional and pooling layers is flattened into a 1D vector. The reason for the flattening is so that the fully-connected layer can interpret the feature maps in a more fitting shape, instead of grid-structured data. These layers are equivalent to the classic ANN and are where regression and prediction of resistivity happens. The number of parameters in the fully-connected layer can be high and computationally exhausting. To provide regularization we use dropout in this layer, introduced in Section 2.5.3 [26]. Dropout allows us to remove nodes with their in-going and outgoing connections, reducing the number of computations. Dropout acts as a regularizer to prevent co-adaptation.

2.7 Hyperparameter tuning with Keras tuner

Automated Machine Learning (AutoML) is a wide research field that has become important with its application of ML techniques [32]. The motivation of AutoML is to enable people with less machine learning experience to use machine learning in practice more easily. Instead of hard-coding the machine learning algorithms, libraries such as *Sklearn* and *Keras* offer in-built learning algorithms that the user has to insert hyperparameters to begin modelling. Hyperparameters are the parameters of the machine learning algorithm that defines the behaviour of the model.

These hyperparameters vary dependent on which machine learning algorithm is used. For CNN used in this thesis, some hyperparameters are for instance the number of convolutional layers, the number of kernels in each layer, and the number of neurons in the fully-connected layer. In this thesis, Keras is used for tuning, where the user defines a search space of hyperparameters that Keras iterates over, exploring different CNN architectures. This allows the user for neural architecture search (NAS), which aims to search for the best neural network hyperparameters [32].

Tuning the hyperparameters of a neural network can often be exhausting as there can be a large number of factors that make the tuning process complex. As introduced in Section 2.5: an artificial neural network with one hidden layer and one neuron can model any function. So, how many hidden layers should be chosen, and how many neurons in each layer should the network have? Making the network too big may be ineffective and computationally exhausting. Making it too small may not utilize the model capacity to its fullest. A too complex network may tend to overfit, and vice versa for a simple network. This is an arbitrary problem, and considering the number of hyperparameter options in a neural network, this problem can tend to be big. Unlike loss functions during training, the hyperparameters are not differentiable. There is no correct setting of hyperparameters, and suitable ones are in general found by trial and error. The hyperparameters are essentially the settings that can be adjusted in the neural network prior to training.

To select suitable hyperparameters with Keras tuner, a search space is first defined. In this search space, we select which hyperparameters to tune, as well as the ranges of which the hyperparameters can have. The search space is essentially the ranges of values we, the user thinks are suitable and relevant for modelling. This allows us, the user to insert domain knowledge, preventing tuning of most of the bad models. For instance, the number of convolutional layers can be set from 1 to 5, and the number of kernels from 32 to 256. Each pair of settings defines a model with a certain performance where the performance is measured by a loss function, In this thesis, mean squared error(MSE) is extensively used. After searching and building different CNN architectures, Keras tuner will have a ranking of the different models by their MSE, showing the best performing models after tuning. The hyperparameters of the tuned models are then returned explicitly to the user, allowing for further development of optimal models. Application of Keras tuner and validation of the different models is shown later in Chapter 3.

2.7.1 Tuning algorithms for hyperparameter search

From Keras, we decide to use two different tuning algorithms which dictate how the hyperparameters are chosen, called Random search and Hyperband search. Each tuning algorithm chooses the hyperparameters in a certain way, which can be beneficial depending on the application. While random-search is a typical brute-

force approach to tuning, Hyperband optimization is classified as a more adaptive method, attempting to converge to the best models as fast as possible. Such adaptive methods have been optimized and developed to make the tuning experience more effective in terms of time.

Random search

Random search is a brute force approach to hyperparameter tuning where the parameters are chosen randomly for each iteration. First, a number of trials is defined, which is the number of models random search returns, each with different hyperparameters. A higher number of trials means more models are tested and evaluated. Then based on the defined search space, random search selects random values for each hyperparameter. For each iteration, this creates a unique model architecture each time, but the performance can vary a lot since there is no adaptive part of the tuning that contributes to improved model performance for each iteration. As shown in Figure 2.20 with two hyperparameters, one is labeled as important and the other as unimportant. We then rely on tuning over sufficient number of trials for convergence on the important hyperparameter.

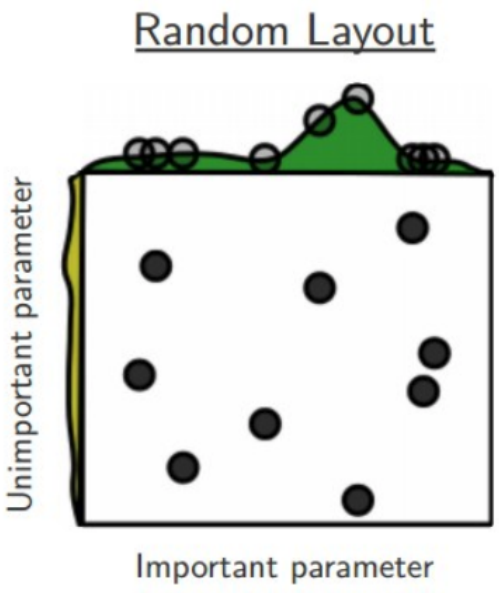


Figure 2.20: Random search algorithm for finding two optimal hyperparameters [33]

Although random search may be based on probability, using domain knowledge will increase the chance of finding good model hyperparameters. This involves choosing a search space before training that is influenced by the knowledge of the user which eliminates a big portion of bad-performing models.

Hyperband tuner

Hyperband search is an attempt to develop more efficient optimization algorithms for hyperparameter tuning. Compared to brute force approaches such as random search and grid-search that can be time-consuming, the Hyperband tuner tries to narrow down to the best models as fast as possible in an adaptive manner. The Hyperband algorithm utilizes the successive halving algorithm to make the optimization adaptive. Successive halving revolves around allocating computational resources optimally to efficiently find good model architectures. With resources, we mean RAM and computational power required to train models, as well as the time spent training the models. Below is a description of how successive halving works in practice

1. Randomly sample a set of hyperparameters and train a set of models
2. Evaluate the performances of the trained models, ranked by i.e. MSE
3. Discard the bottom half of the worst-performing models
4. Repeat 2 until one model remains

The adaptive nature of Hyperband is able to sort out the good models from the bad, thus preventing time waste on models that lead to nowhere. To prevent spending too much time and resources on training bad models, early stopping, a regularization technique is embedded into Hyperband. Early stopping makes the training stop after a certain number of training epochs. With early stopping, Hyperband is able to highlight the performances of the trained models based on only a few epochs, hence more time and computational power is allocated to the good models. To use Hyperband in practice, a library that saves the models then reuses the good ones during training is required. Creating this functionality from scratch may be time-consuming. Luckily Keras offers successive halving in their Hyperband tuner.

Note that there is no guarantee that Hyperband finds the best models immediately, as the tuning algorithm is based on random search to sample the first models trained. It is therefore important to use domain knowledge to define a search space that is most probable to cover good models. Another issue might be the tradeoff between total resources versus the total number of models trained. We do not want to train beyond the total amount of disk space or memory, leading to limitations of how long and how many models we can train during one session.

2.8 Model validation and selection for evaluating and selecting optimal models

How does one know that the model parameters for a given algorithm are the most fitted for the application? For instance, how many hidden layers and neurons should a neural network have? Model validation and model selection revolves around creating different models with different hyperparameters and selecting

the model that performs best. We then say that the model is a hypothesis in the hypothesis space where a hypothesis is a "guess" on how the data is modelled. The objective is to find the best combination of hyperparameters for the model and the given application. Taken from [34], there are three main points tied to model validation and selection

1. We want to estimate the generalization accuracy, the predictive performance of a model on future (unseen) data.
2. We want to increase the predictive performance by tweaking the learning algorithm and selecting the best-performing model from a given hypothesis space.
3. We want to identify the machine learning algorithm that is best suited for the problem at hand; thus, we want to compare different algorithms, selecting the best-performing one as well as the best-performing model from the algorithm's hypothesis space.

These three points involve using different techniques to create models in the hypothesis space which we evaluate and select from. To create different models in the hypothesis space, one changes the hyperparameters in the learning algorithm, creating a hypothesis on how we think the data is modelled. By changing these hyperparameters, different models or hypotheses with different performances are constructed. The challenge is then to tune hyperparameters in a way that does not overfit or underfit on the data set meaning the model is good at generalizing to new data. After creating a set of competing models, model selection is done in the end to pick a final optimal model. In this chapter, we will go through different techniques for model validation and model selection.

2.8.1 Splitting with the Holdout method

A classic example of model validation is using the holdout method [34]. First, the labeled data set is split into two parts where a general split is often 80% of the data as training and 20% as the test set. The intuition here is to measure the performance of how the model performs to new, unseen data. A set of hyperparameters are selected, then the learning algorithm fits a model to the training set. When a model is fitted/trained, it is ready to predict the test set, indicating the performance of the model. This process is called model validation, as to validate the performance of a set of models to find the best-performing ones.

It is though important to note that when validating a model, the training set is not to be used to test on. Predicting the training set introduces optimistic bias as we already know the "answers" since we already trained on it, resulting in severe overfitting [34].

2.8.2 Splitting with the 3-way Holdout method

A problem with the 2-way holdout model is that, when validating the model on the test set, one has "used" up that test set for further tweaking and improvement of the model. The user is then exposed to a phenomenon called data leakage, as we are tuning the parameters of the model using information from the test set. Using information from the test set is interpreted as "cheating", since we are looking at the answer of the prediction when training the model. The concept of creating a good model is to have a good metric score on new, unseen data. A solution to data leakage is the 3-way holdout method, separating the data set into one training set, one validation set, and one test set. This is illustrated in Figure 2.21.

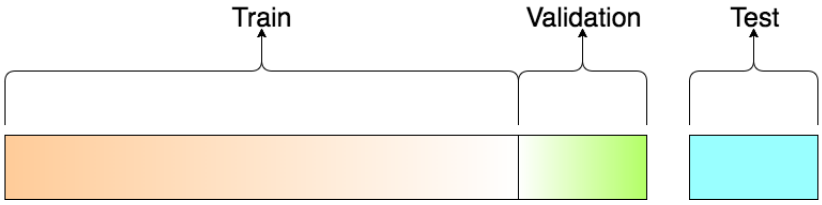


Figure 2.21: Data set splitting with the 3-way Holdout method [35]

The 3-way holdout method takes away the information about the test set, thus reducing data leakage. The test set is put away, and training and testing happen on the training and validation set. The test set is used only once we have validated several models and algorithms, and selected an optimal model. This way, one is able to tune hyperparameters and validate different models without looking at the test set.

2.8.3 Model selection

From the numbered list introduced in Section 2.8, model selection revolves around the 3rd point.

- We want to identify the machine learning algorithm that is best suited for the problem at hand; thus, we want to compare different algorithms, selecting the best-performing one as well as the best-performing model from the algorithm's hypothesis space. [34]

After training and validating with the 3-way holdout method, we are left with a set of competing models with different performances. The models referred to are the ones tuned from hyperparameter tuning with Keras tuner. These models have different numbers of kernels, numbers of convolutional layers, etc, and the most optimal one for our application is picked based on MSE.

Chapter 3

Methodology

In this chapter, we go over the process of data analysis, data pre-processing, data set splitting, and hyperparameter tuning of CNN architectures. Lastly, we will select optimal models to predict the holdout test set.

First, an introduction of the data set is done to get an overview of what type of data and materials we are dealing with. Then pre-processing of the data set is done to prepare the data for modelling with CNN, involving interpolation, removal of artifacts, data augmentation, and normalization of data. From data augmentation with overlap, there was introduced a problem with data leakage which lead to overfitting. We will therefore emphasize the importance of splitting the data properly to decrease overfitting. Further, this chapter involves searching for optimal CNN architectures through hyperparameter tuning with Keras-tuner. The hyperparameters regarded are for instance the number of convolutional layers, the number of kernels in each layer, the learning rate, and so on. Lastly, the optimal CNN architectures will be validated on different data set splits. The strategy and flow of processes are presented in Figure 3.1

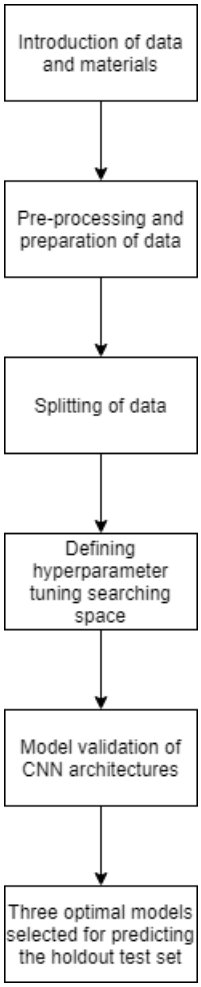


Figure 3.1: Workflow of the various steps of the methodology

3.1 Data set and materials

The data set used in this thesis consists of 2D CT scan data from 142 meter section of a well drilled by Equinor. The goal is to propose a CNN regression architecture that uses the image data to predict resistivity with confidence. The data set has already been labelled using a logging tool during drilling, mapping the resistivity continuously over the 142 meters. This is therefore known as a supervised learning problem, where the labels and features are known and then used for learning with machine learning. The goal is then to train and tune a CNN model that learns from the data, then predicts the holdout test set that was unseen by the training process. The prediction results on the holdout test set can essentially be seen as performance on future, unseen data. Thus, we want to tune on the training and validation sets, then when ready, predict the test set.

To each image, a resistivity value has been assigned by interpolation of the log values. Figure 3.2 shows a plot of resistivity over the 142 meter depth interval. Here, the resistivity is only plotted, where for each resistivity point, a 2D CT-scan image is assigned. From the plot, we can observe that there are uneven jumps in resistivity values over time. This is caused by a lack of image data, together with noisy data having to be removed after pre-processing. From an overview perspective, the resistivity values are mostly on the lower end of the range of resistivity, but there are signs of spread in the middle and higher-end as well.

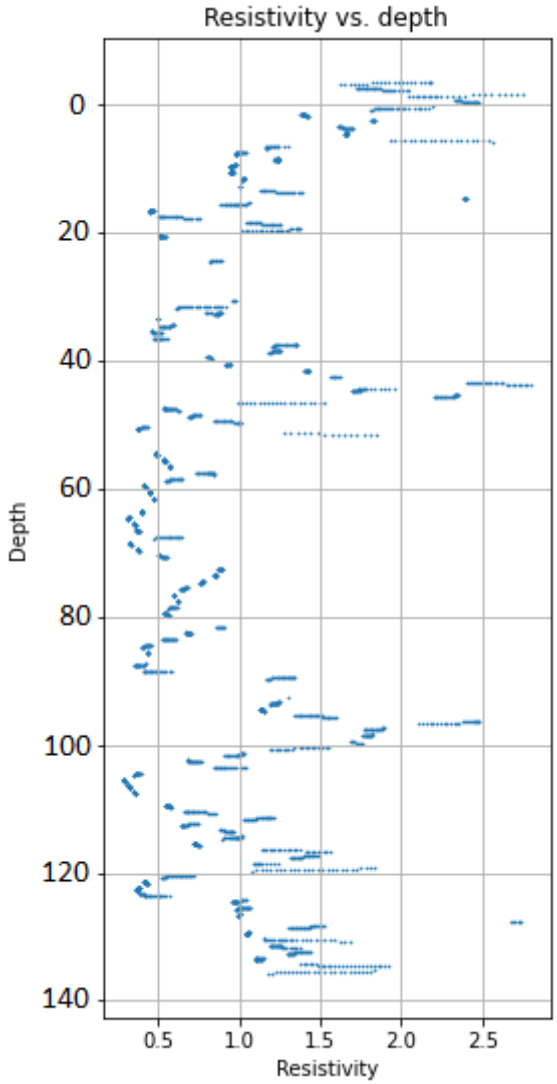


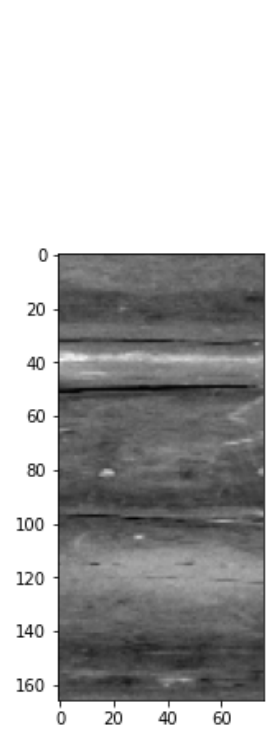
Figure 3.2: Resistivity vs depth over the whole data set

3.1.1 Three data set distributions for model validation

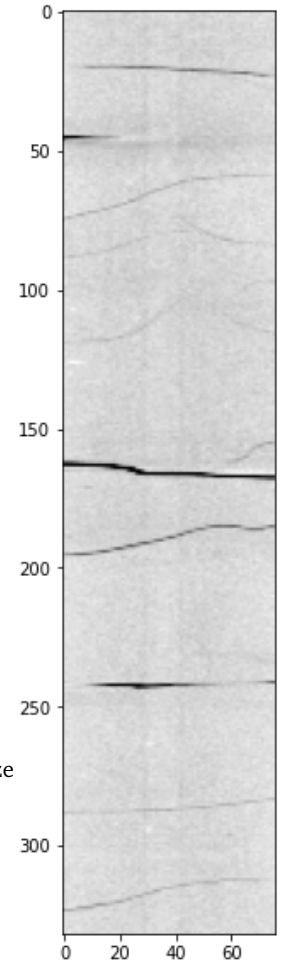
Later in this chapter, there are mainly three splits of the data set distributions that are regarded during model validation. With these three different splits, the goal is to investigate if the model performances are dependent on different distributions of resistivity and image data. For instance, a training set with many high resistivity values may perform poorly on a validation set with many low resistivity values. This is a part of the model validation process, where different model architectures are tuned and validated, and then, in the end, a final test set is predicted. The three splits considered are one random sampled, one manually split of continuous intervals, and one random sampled before data generation with overlap. Note that only training and validation sets are sampled differently, as the test set is similar in all distributions. More detail on this matter is presented in Section 3.3.

3.1.2 Testing different sizes of images: 30cm and 60cm

Additional to investigating different data set splits, different sizes of the input images are tested out. The two image sizes we consider are 30cm and 60cm. The purpose of trying out different data set sizes is to investigate see if the size of the input images affects the CNN model's capability to extract information to perform regression. Additionally, with increased image sizes, came a decrease in data set size, thus we also observe the effect of having a small and a bigger data set. With 30cm images, a data set size of 10339 is acquired, while with 60cm images, the data set size ends up at 2467. Figures 3.3a and 3.3b show the two sizes of images. For simplicity, in the following sections, we will refer to the 60cm size when showing augmentation and presenting data set distributions.



(a) Sample image of size 30cm



(b) Sample image of size 60cm

Figure 3.3: Two sample images showing the different sizes of inputs.

3.2 Pre-processing and preparation of data

Before using the data for modelling with CNN, it had to be pre-processed to remove disturbances in the data and augmented to increase robustness and data set size. In the original data, missing intervals and artifacts were present, acting as noise if not removed. These two instances were detected and removed using a script from another projects [4][9]. A consequence of the removal of artefacts and missing intervals was a loss of raw data, leaving us with a reduction in data set size. As an attempt to deal with the small data set size, data augmentation by overlapping 98% with a sliding window was done. This increased the data set size from 142 1m images to 2467 60cm images or 10339 30cm images, de-

pendent on which image size was used. Further, as an attempt to increase model robustness, vertical and horizontal flip of the images was done. As the last step of pre-processing, the image data was normalized with Max-Min normalization, bringing the input to numbers between 0 and 1. The pre-processed data could then be used for training, validation, and testing CNN models. A flowchart of pre-processing is presented in Figure 3.4.

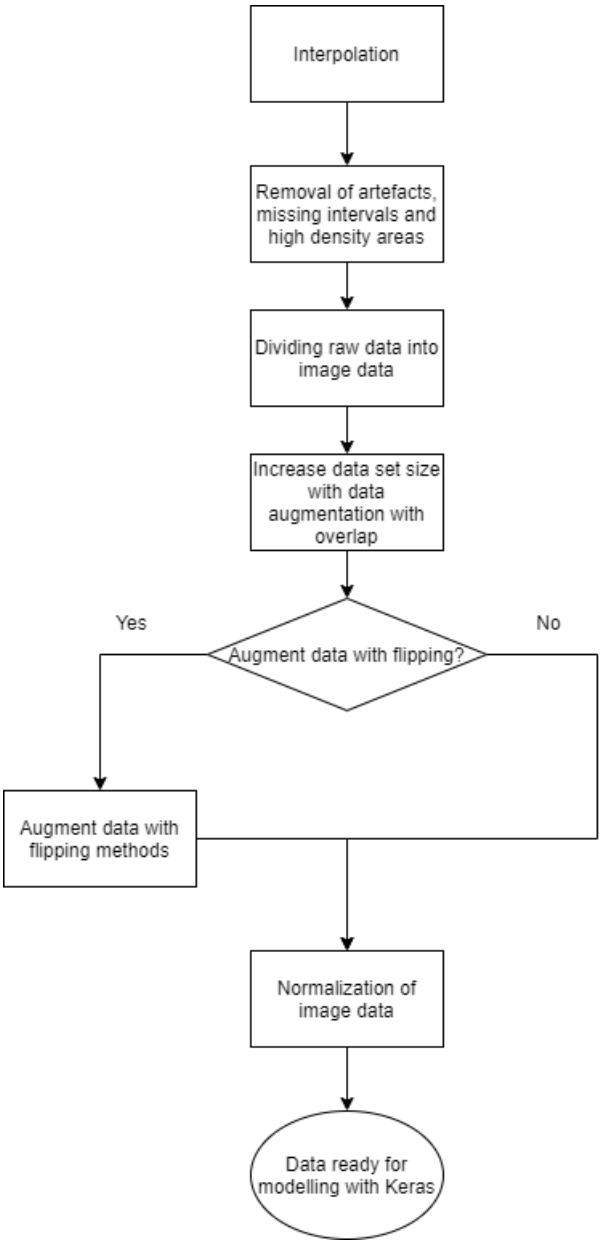


Figure 3.4: Workflow of pre-processing for preparation of the data set

3.2.1 Interpolation to increase data set resolution

At first, a pair of resistivity values and their respective depths where the rocks were located, were stored in a 932×2 matrix. This data covered depth interval of 142 meters where each 1 by 2 slice of data presented one depth and one resistivity value (resistivity, depth). To create image data out of this data set, a higher resolution version of the data set was desired. Interpolation was therefore used as a tool to increase the resolution of the 932 data points. Interpolation was done for the 932 data points in python, creating an estimation function f that maps a given depth x_{depth}, to a new resistivity value $y_{\text{resistivity}}$. The function *interp2d* from the *scipy* library was used to construct f.

$$f(x_{\text{depth}}) = y_{\text{resistivity}} \tag{3.1}$$

Using a data set with new shifted x values of the original x with size 313.924, interpolation with f, constructed new data points in-between the original ones. This resulted in an interpolated data set with higher resolution of both depths and resistivity values, from 932 to 313.924 data points. Note that the new data is not new in the sense of gathering new data, but a higher resolution data set is needed to construct images of $166 \times 76px$(30cm) or $332 \times 76px$(60cm). The new data of size 313.924 instead represents a more detailed version of the original data, used to construct the images. To demonstrate interpolation on our data set, a small interval of the data set is visualized. Figure 3.5 shows interpolation applied before and after.

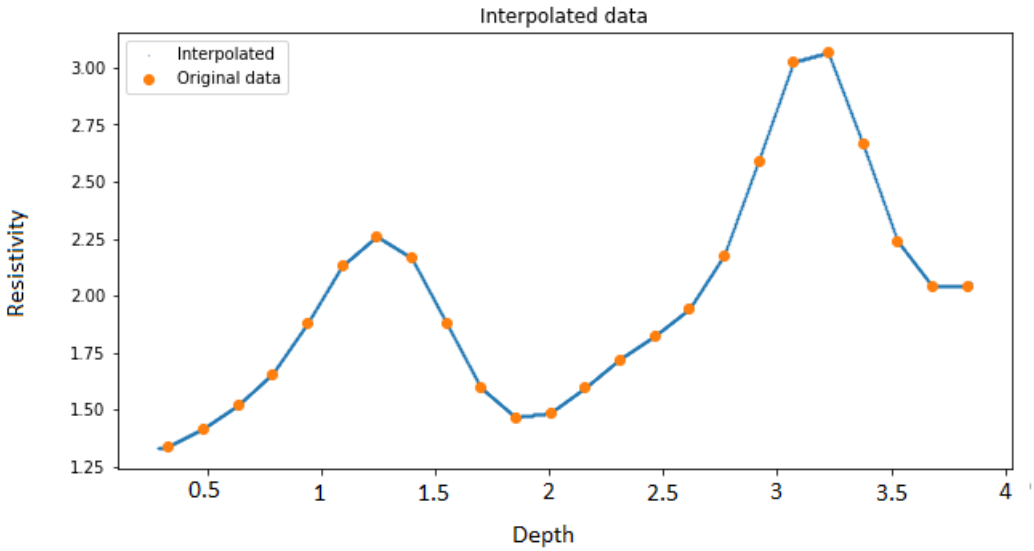


Figure 3.5: Data before and after interpolation

3.2.2 Removal of artefacts, missing intervals, and high-density areas

Three instances of noise were discovered in the data set that were regarded as disturbances for modelling if not removed. These were missing data intervals, core barrel couplings, and high-density areas. The missing data was most likely caused by poor core recovery, induced fractures, or rush plugs taken after retrieval of the core [4]. Another instance of unwanted data was bright areas caused by core plugs and mud invasion. These were previously presented and described in Figure 2.7. To remove these instances, code from projects [4] and [9] was used. The code detected the ranges of all three instances and flagged them for removal. The new cleaned data set consisted of 64.000 data points. This new data set was then used to create 166 × 76px and 332 × 76px images with data augmentation with overlap.

3.2.3 Dividing the data into 30cm and 60cm images

As presented in Figure 2.4, each column of gray-scale data consisted of 1 meter from the well. The frequency of artefacts and missing intervals in the data could vary, thus some intervals had more noise removed than others. From these intervals cleaned of noise, images could be produced, and as mentioned in Section 3.1.2, it was decided to try out both 30cm and 60cm images, previously visualized in figures 3.3a and 3.3b. This was to investigate if the image sizes affected the model performance. The 30cm images had height 166px and 60cm images had height 332px. Both sizes had width 76px.

3.2.4 Data augmentation

After the removal of artefacts, data augmentation was performed on the data set to act as a regularizer. Regularization methods are used to prevent overfitting, strengthening the model's generalizability to new, unseen images. As introduced in Section 2.3.3, data augmentation methods revolve around modifying the current data to create unique, resampled data. The motivation of using data augmentation in this thesis is therefore explained in the two following points

- **Data generation with overlap**: Increase data set size for sufficient data to train on. It is important to let the model train on sufficient data for each training iteration.
- **Horizontal and vertical flip**: Contributes to increased robustness in the model, as the model is exposed to "noisy" data that is augmented. The idea is to present unique images to the model to change the variation of images, increasing generalizability to new, unseen data.

Data generation with overlap

As a measure to compensate for the small data set size, data generation with overlap was done with code from [4] and [9]. The 64.000 data points were divided

into 1-meter intervals, then transformed into images. Using the 60cm size as a base for this example, each image consisted of a height of 332×76px. For each 1-meter interval, a new image was created, inheriting 98% of the information from the previous image. This was done by using a window that was slid down 8 pixels for each image generated. This meant that for each 60cm image, 40 additional images were generated with overlap. Because some 1-meter intervals contained more removed artefacts than others, this multiplier decreased. In total, the overlap method increased the size of the data from 142 1-meter intervals to 2467 60cm images.

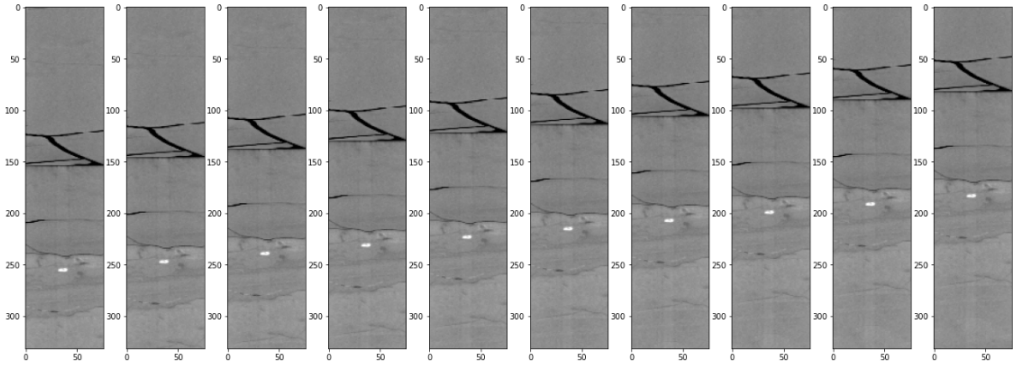


Figure 3.6: Data augmentation of 10 images using overlap

The idea of generating data this way is to have more data for training. Instead of having 142 1-meter images for training, an increase of 17.3 times was acquired or 60cm height images, and an increase of 72.8 times for the 30cm height images. Since there is some variation of 2% from last to the next image, the idea is to have CNN think that these are new images. It is although important to note that these generated images are actually not new images, but synthetic ones that have inherited 98% of the image characteristics from the previous one. Figure 3.6 shows 10 images that were generated with the use of overlap.

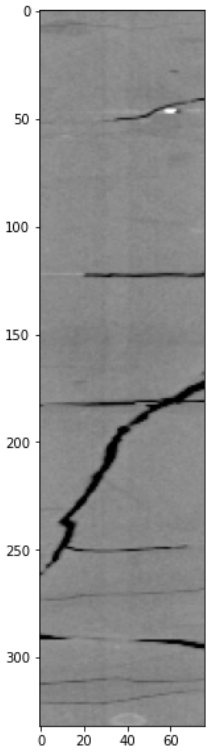
Horizontal and vertical flip

Additional to generating data with overlap, data augmentation with flipping images was performed. This method was an attempt to increase the robustness by letting the model see modified versions of the original image. Two augmentation techniques were used, one where augmentation was done before training, and one where the images were augmented during training.

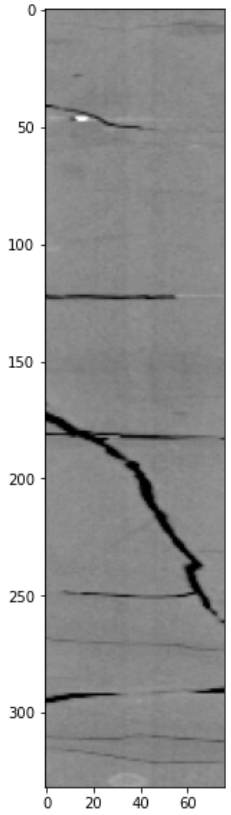
Augmenting before training: The motivation for augmenting before training was to increase the data set size further. By using four different augmentations, it was acquired an increase of 4-fold. This was done in python with *NumPy* library, using functions *flipud* and *fliplr* for flipping "up-down", and "left-right". The flipping was done first for the original image, both horizontally and vertically. Then the

horizontally and vertically flipped images were flipped once more. This way, we obtained all the combinations possible shown in Figure 3.7

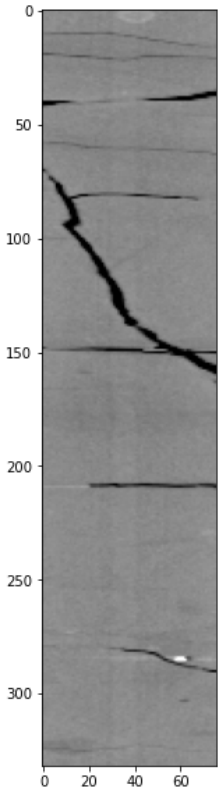
On-the-fly augmentation: The other method augments the images in real-time, so no data preparation had to be done in advance. The data set size did though not increase directly in size but rather lets the model see a unique, altered image of the original image. This method requires less memory usage and offers a convenient class configuration. Instead of storing the augmented images in an array or a folder, the images are loaded in batches, which saves memory. This method uses the Keras library with the *ImageDataGenerator* API. This API lets the user configure the desired augmentations such as zoom, flips, rotation, and more. Figure 3.8 as well as Appendix A.3 shows the class configuration for *ImageDataGenerator*. For our images, horizontal and vertical flip were the only augmentations used.



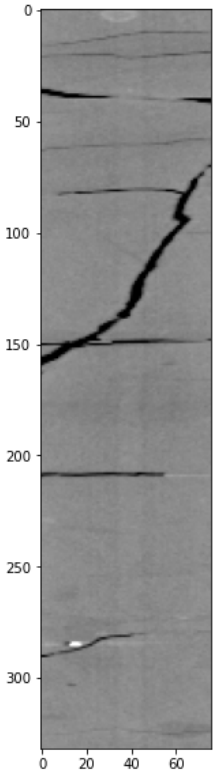
(a) No augmentation



(b) Horizontal flip



(c) Vertical flip



(d) Both horizontal and vertical flip

Figure 3.7: Data augmentation by flipping showing all four flips

```
tf.keras.preprocessing.image.ImageDataGenerator(
    featurewise_center=False,
    samplewise_center=False,
    featurewise_std_normalization=False,
    samplewise_std_normalization=False,
    zca_whitening=False,
    zca_epsilon=1e-06,
    rotation_range=0,
    width_shift_range=0.0,
    height_shift_range=0.0,
    brightness_range=None,
    shear_range=0.0,
    zoom_range=0.0,
    channel_shift_range=0.0,
    fill_mode="nearest",
    cval=0.0,
    horizontal_flip=False,
    vertical_flip=False,
    rescale=None,
    preprocessing_function=None,
    data_format=None,
    validation_split=0.0,
    dtype=None,
)
```

Figure 3.8: Arguments of *ImageDataGeneration* for data augmentation of images. For our application, *horizontal_flip* and *vertical_flip* are used [36].

3.2.5 Normalization of data input

As a last step of the pre-processing, normalization was performed on the data. The reason for performing normalization as mentioned in Section 2.3.4, was to decrease the computational load in the neural network as mentioned previously in Section 2.3.4. Normalization involved mapping the image input to numbers between 0 and 1 so that high-valued inputs were avoided. An image consists of a grid of values between 0 and 255 to represent colors. We then proceeded to divide the image data on 255 to bring the image data input to a common scale. This was done by simple division in *python*.

3.3 Splitting of data set for various data set distributions

Before tuning CNN architectures with the 3-way holdout method, splitting the data set into training, validation, and test sets were done. As introduced in Section 2.8, the training and validation sets are used during hyperparameter tuning before choosing an optimal model. The test sets are held outside this process since we want an unbiased data set that we have not seen to evaluate the final performance of our chosen models. This section will go over the process of splitting where we propose three different splits. Figure 3.9 shows the flow of processes for splitting the three distributions.

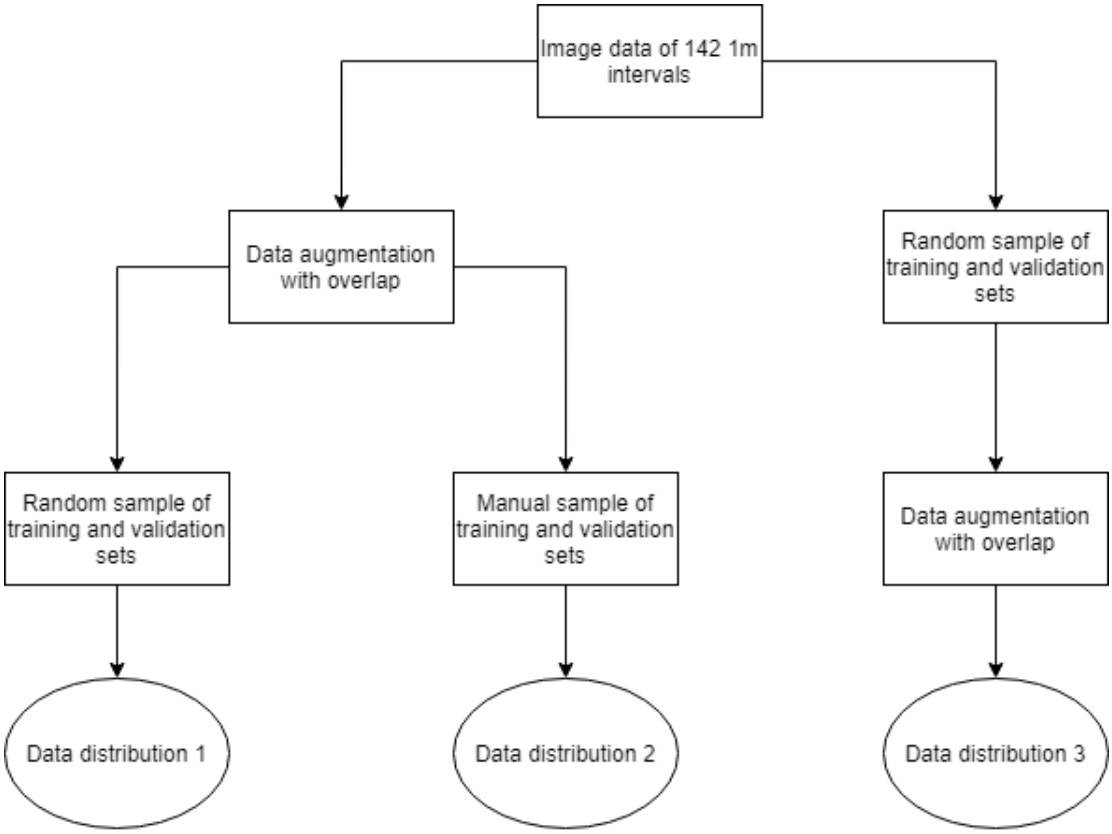


Figure 3.9: Workflow of splitting process into three different distributions

3.3.1 The issue with data augmentation with overlap

In this project, the process of splitting data was especially important. Since the data was augmented using 98% overlap over images, this introduced the risk of data leakage when using random sampling. Usually, when splitting, random sampling of validation and test sets is often used. This means that the validation and test sets are randomly picked from the whole data set distribution. The advantage of random sampling is that the picking of data is unbiased to the learning

method since it is not handpicked by the programmer. Additionally, using random sampling often ensures that the validation and test sets represent a wide range of different data values. Although when using random sampling on our overlap-generated data, the issue with data leakage occurred. As introduced in Section 2.4.4, data leakage is the event of having information outside the training set when modelling. We essentially experienced having validation samples in the training set because of random sampling while having data generated with overlap. This is known as inserting bias into the training since we are essentially training on the samples that we are supposed to predict. As an attempt to visualize this issue, Figure 3.10 is presented.

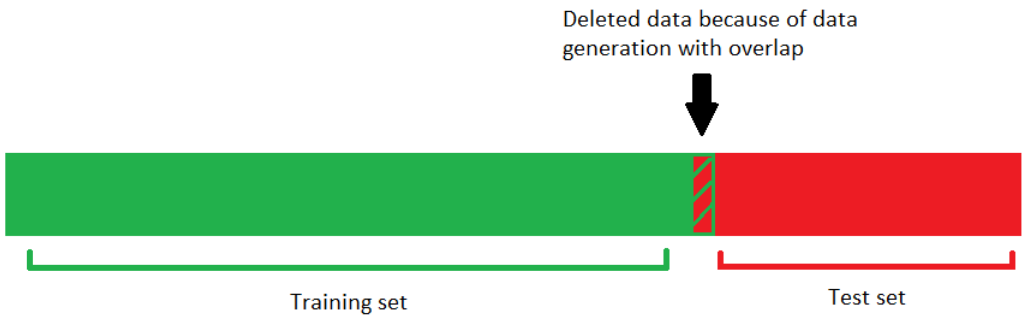


Figure 3.10: Consequence of data augmentation with overlap visualization

To show the consequences of the overlap-generated data set, we show a data set split with random sampling and data leakage, then we propose two additional data set splits to show our solution of handling the problem. These splits are used further in this thesis during training of CNN models, and model validation. The idea is to investigate different performances based on their respective data set distributions. Note that all the test sets are similar, just to ensure that we measure the same test data for the final evaluation. These test sets are held outside so that we prevent having data leakage from the model validation phase.

3.3.2 Random sampling of training and validation sets after data generation with overlap

The overlap issue was observed when splitting the data set with random sampling, after data generation with overlap. Usually, this a common technique for splitting

in machine learning applications, where the validation and test sets consist of about 10% each of the total data set size. By random sampling the validation set, there is no bias towards handpicking the "right" data for later prediction and validation. Another advantage with random sampling is that the variation of resistivity values usually covers a wide range, resulting in good, representative distributions for model validation. Figure 3.11 shows a simple example of random sampling.

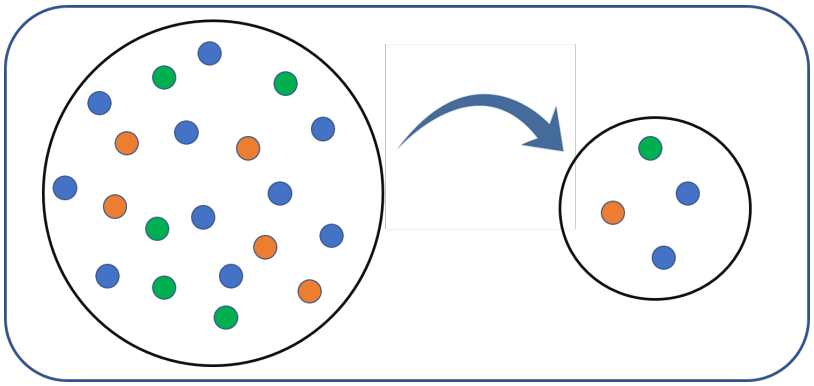


Figure 3.11: Simple example of random sampling [37].

For our problem, random sampling became an issue because of data augmentation of overlap. Since the overlapped images contained up to 98% similarity of some images, this caused the validation set to contain many images that were almost identical to images in the training set. This is known as an instance of data leakage where the performance on the validation set might be good, but not the test set. Note that the test set was manually sampled to not contain leaked data. Figure 3.12 shows the training and validation set retrieved by random sampling and the continuous sample of the test set. Further, distributions of the three data set splits are shown in figures 3.13, 3.14 and 3.15.

This overlap issue was detected progressively as the project was worked on, and we therefore committed to using two additional splits, covered in the next subsections as our attempted solutions.

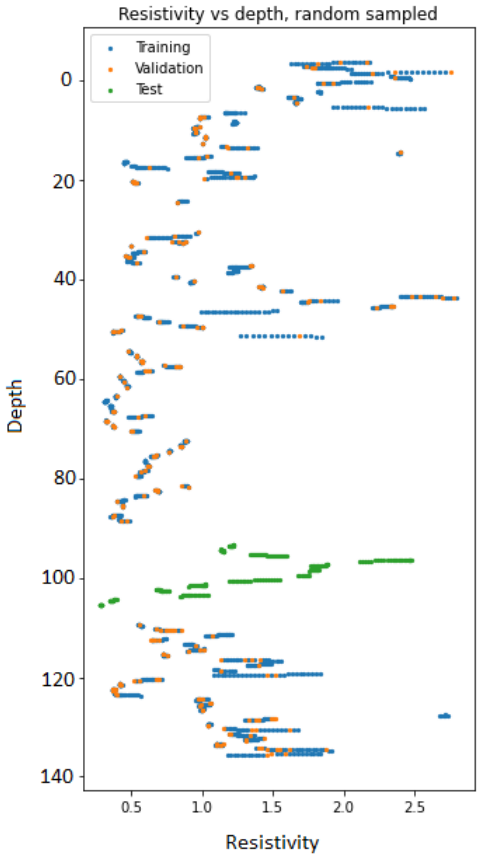


Figure 3.12: Plot of sampling intervals of resistivity distribution 1 for training, validation and test sets

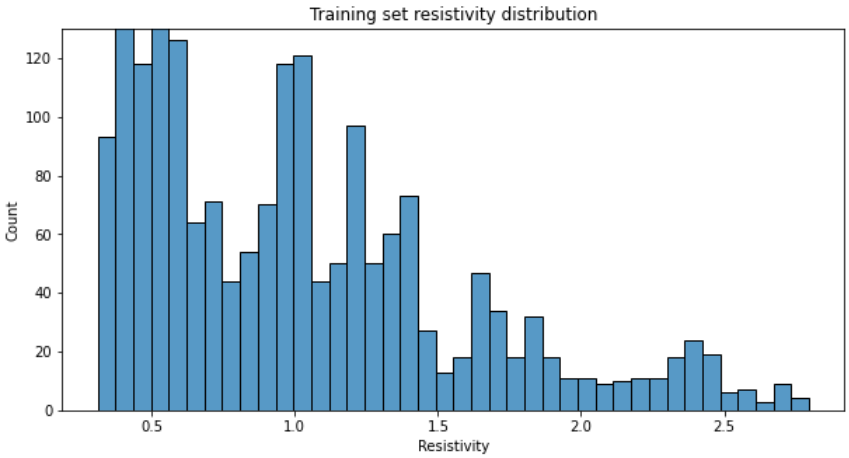


Figure 3.13: Resistivity distribution 1, training data

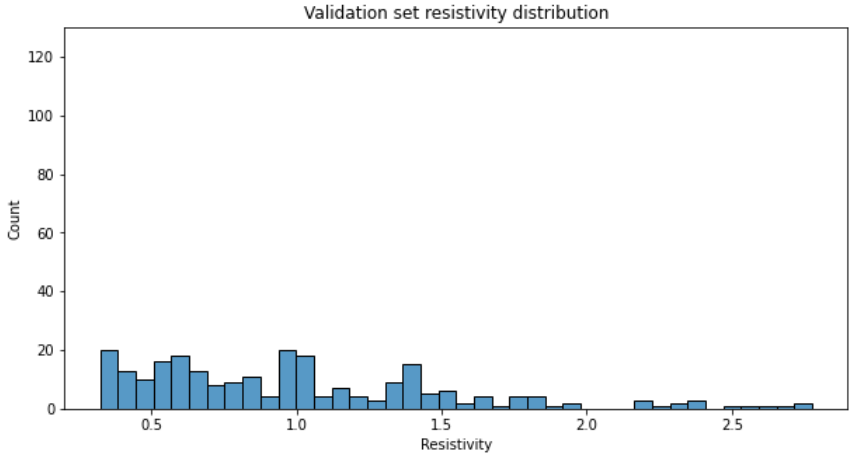


Figure 3.14: Resistivity distribution 1, validation data

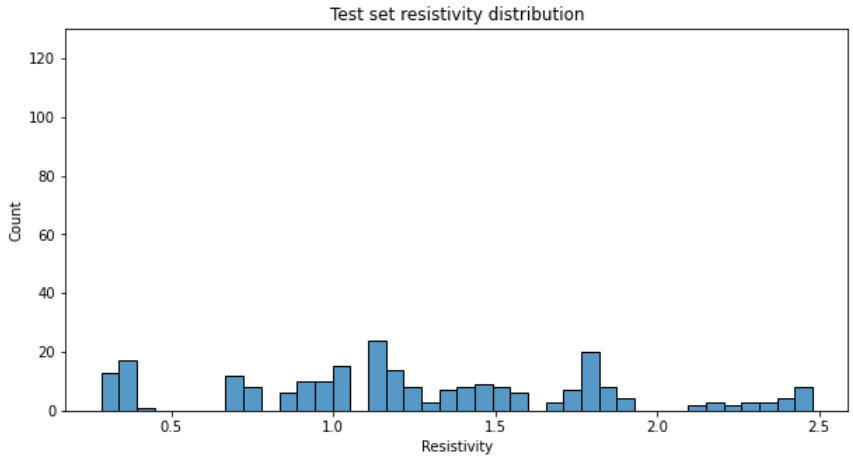


Figure 3.15: Resistivity distribution 1, test data

3.3.3 Manual sampling of training and validation sets after data generation with overlap

A solution to prevent data leakage of the validation set when splitting was to sample continuous intervals of both validation and test sets. The challenge would then be to select continuous validation intervals that contained a good spread of resistivity values. Sampling this way was risky, as there was a risk of missing out on important resistivity values or sampling too many outliers. An outlier is a data point that holds a very uncommon value compared to the rest of the data set. Often they can disturb the prediction process since the trained model is not used to see such data. It is therefore desired to have validation sets that are representative of the training set.

A plot of the three sampling intervals is shown in Figure 3.16. As mentioned, it was important for both test and validation sets to contain a good spread of resistivity data points. This is visualized further in figures 3.17, 3.18, 3.19 to emphasize our point. With this method, overlap still had to be dealt with. This was done by deleting images in front and behind each sampling interval to prevent data leakage. Two intervals for test and validation sets were sampled, resulting in the loss of 168 images.

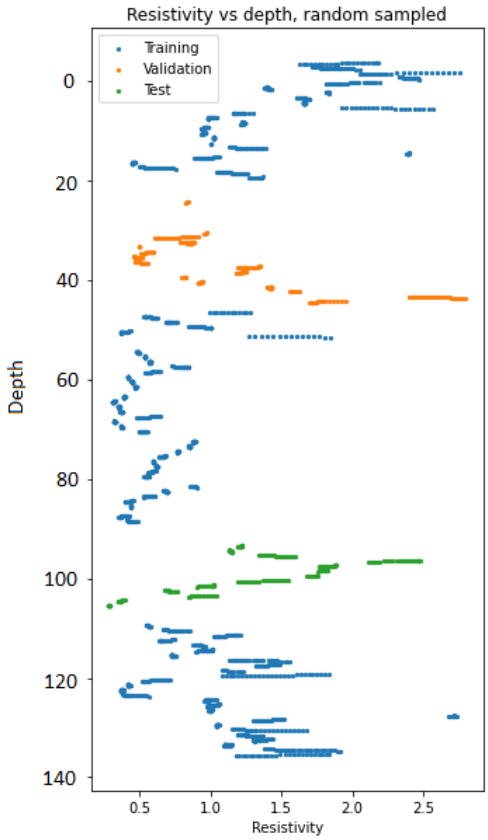


Figure 3.16: Plot of sampling intervals of resistivity distribution 2 for training, validation and test sets

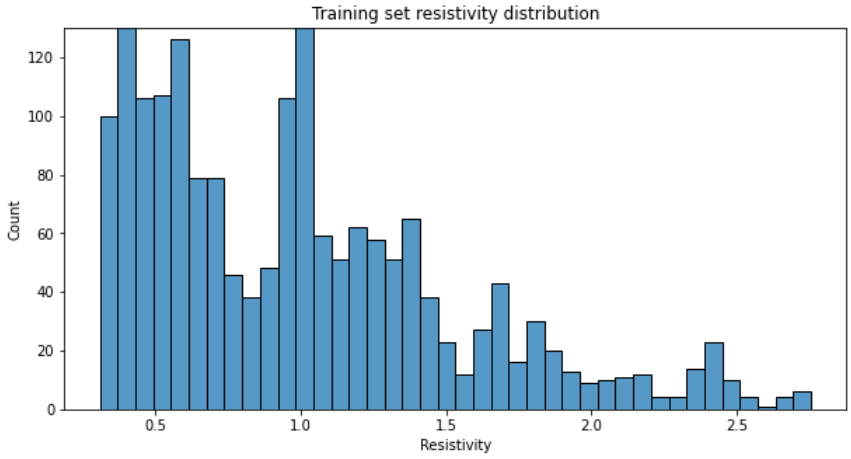


Figure 3.17: Resistivity distribution 2, training data

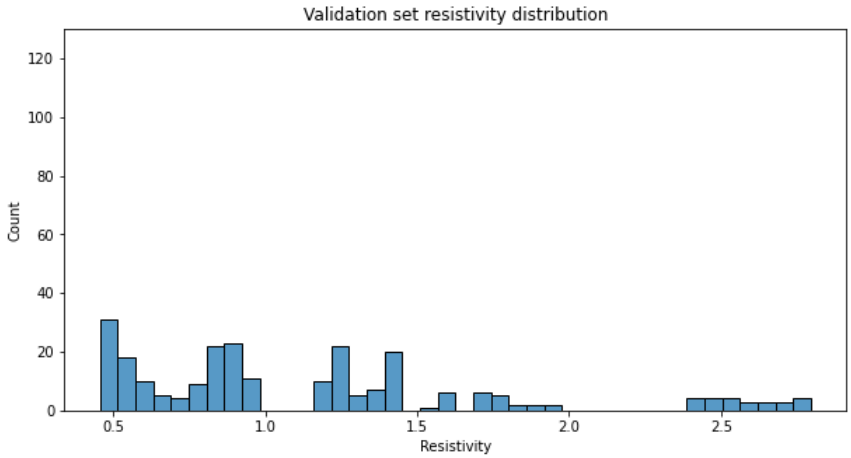


Figure 3.18: Resistivity distribution 2, validation data

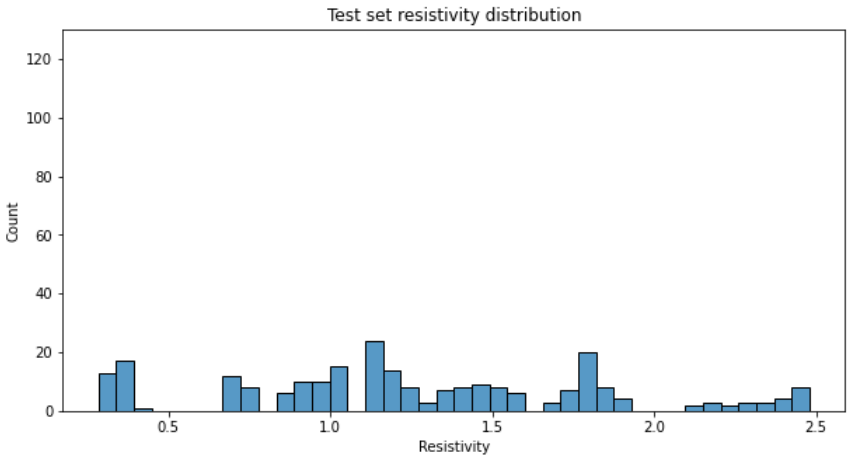


Figure 3.19: Resistivity distribution 2, test data

3.3.4 Random sampling of training and validation sets before data generation with overlap

The other proposed solution to prevent data leakage was to sample the data before data augmentation with overlap. This method involved random sampling the 1m intervals into training validation and test sets, before augmenting with overlap to increase the data set sizes. By random sampling before overlap, the 40 next images would not contain parts of any other image in the data set, unlike the first data distribution. Figures 3.21, 3.22 and 3.23 shows the distribution of resistivity samples over all three data sets. As presented, the training and validation sets are randomly sampled, while the test set is continuously sampled.

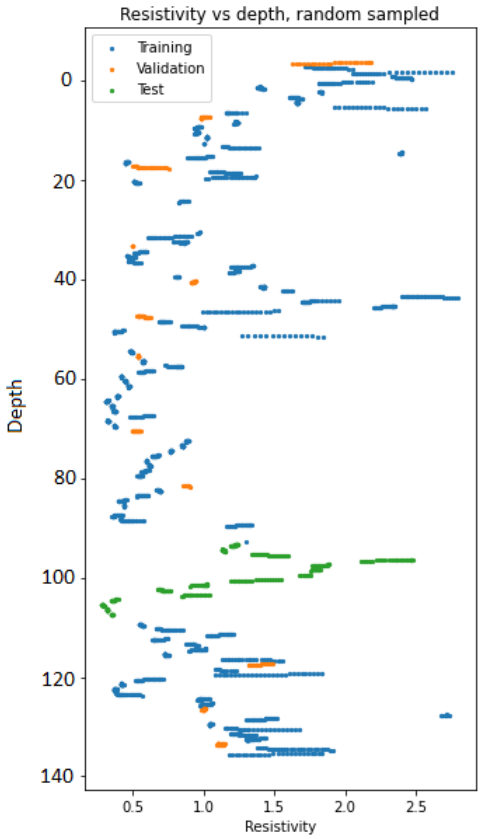


Figure 3.20: Plot of sampling intervals of resistivity distribution 3 for training, validation and test sets

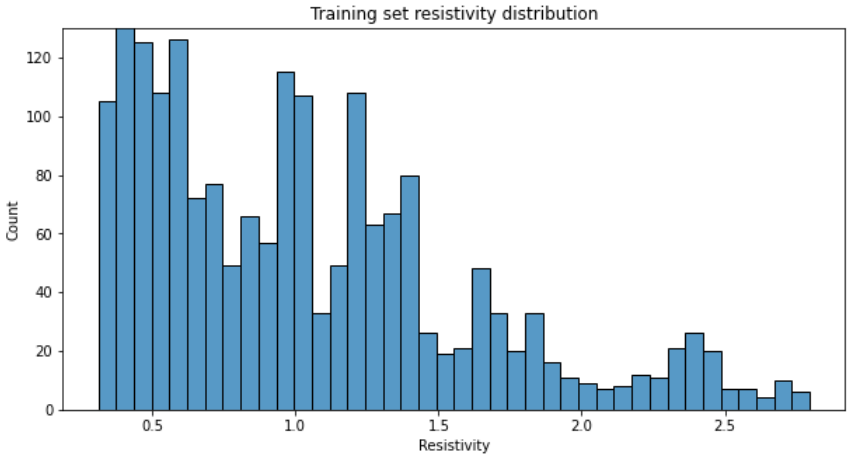


Figure 3.21: Resistivity distribution 3, training data

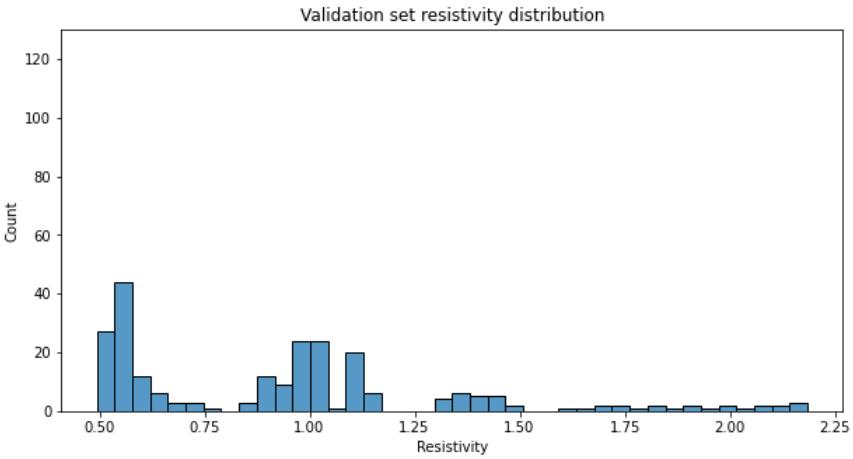


Figure 3.22: Resistivity distribution 3, validation data

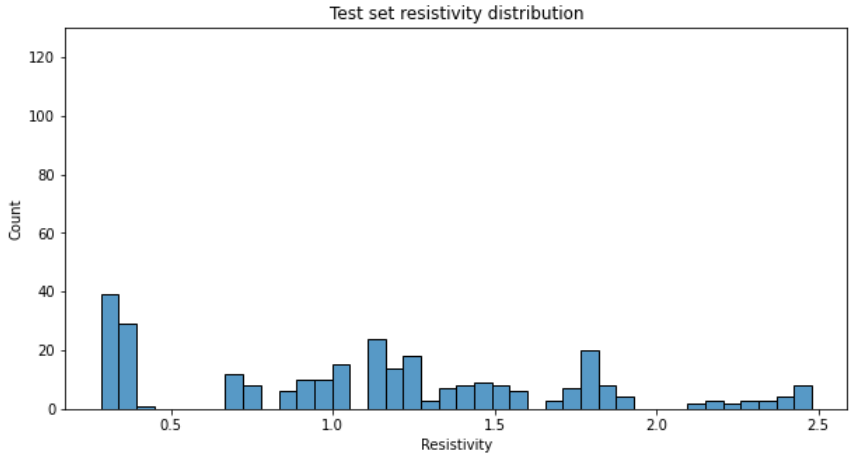


Figure 3.23: Resistivity distribution 3, test data

3.3.5 Prediction and further validation

These three pairs of data set splits are the ones that are going to be used to tune different models further in this chapter. By analyzing different distributions and variants of splittings, the idea is to discover characteristics of splits and regularization methods that work and do not work. When optimal models for each distribution are tuned, testing on the holdout test set is performed for final evaluations. The results of the test set will be presented in chapter 4.

3.4 Training and Hyperparameter tuning of CNN Architectures

In this section, we propose a strategy for training and tuning various CNN architectures. From the bullet-point list from Section 2.8, this section revolves around the second point.

- We want to increase the predictive performance by tweaking the learning algorithm and selecting the best-performing model from a given hypothesis space.

To search for optimal models, we use the Keras library which allows us to automate the searching process. A pre-defined search space over hyperparameters will be presented, as well as how we evaluate the models.

In neural network modelling, there are two types of parameters that affect the model performance: hyperparameters that the user sets, and parameters during training. The hyperparameters are adjusted before training, generating different hypothesises of how we think the data is modelled. On the other hand, we have the inner model parameters that are adjusted during training, involving the weights and biases [38]. Both the hyperparameters and the trainable parameters affect the bias and variance in the model. Thus we have to keep in mind the model complexity to avoid overfitting. For hyperparameters, this involves assigning suitable values such as the number of convolutional layers, the number of kernels, the number of neurons in the hidden layer, and so on. During training, we have to think about things as how long the model should train with number of epochs and the batch size. These are things that affect the model complexity, thus affecting the bias and variance in the model. The goal is to tune the model complexity so that it is able to predict resistivity, and not just remember the training data, avoiding overfitting. On the other hand, we want sufficient computational complexity in the model so that it is able to learn, avoiding underfitting. Figure 3.24 shows the hyperparameter tuning process.

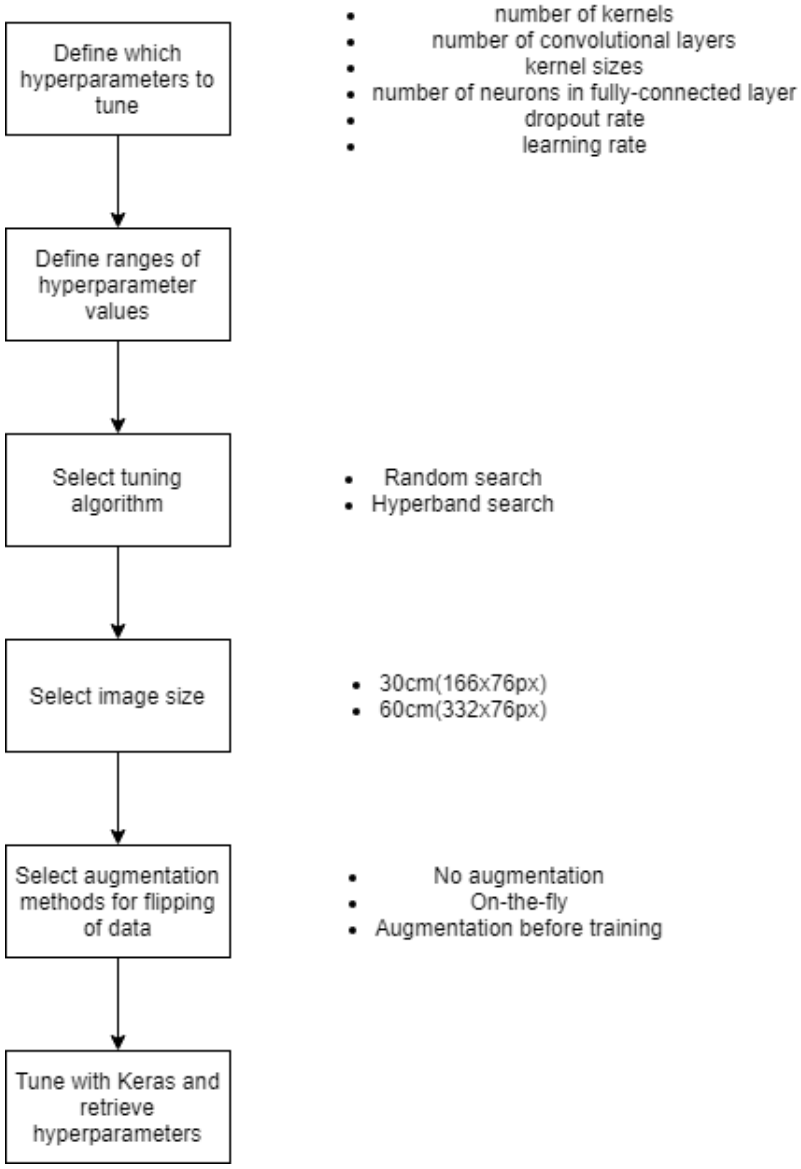


Figure 3.24: Workflow of tuning hyperparameters for finding optimal CNN architectures

3.4.1 Training phase and trainable model parameters

During the training phase, the inner model parameters are adjusted during training. As the model trains on more data, it gets more familiar with data patterns and characteristics and adjusts these parameters using backpropagation. The parameters are divided into two: kernels from the convolutional component, and weights and biases from the fully-connected layer [4]. The kernels are essentially learning different patterns such as edges and shapes. The weights and biases are tied to

the classic ANN part of the CNN, performing the actual regression.

Inner parameter optimization using loss functions and backpropagation

During the training phase, the CNN is fed with training data and its weights and biases are set randomly to define a starting point. The model begins to predict the training data, where these predictions $\hat{y}$ are evaluated to the ground truth labels y. This produces a prediction error y-$\hat{y}$. The neural network then uses gradient methods together with a loss function to calculate the right "direction" to adjust the weights. A classic gradient method is gradient descent, where the gradient of the loss function is iteratively calculated backwards in the network after each forward pass of data. This is called backpropagation and is essentially where the learning of the trainable parameters happens. Since we are dealing with a regression problem, MSE, a popular loss function is used. MSE is given by

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{3.2}$$

where y is the ground truth, $\hat{y}$ is our prediction, and n is the number of prediction samples. MSE calculates the difference between the prediction of the training data, versus the actual value of the data. As the model trains for longer, the CNN gets better at predicting the training data. This involves that the convolutional kernels improve at detecting patterns, and the weights in the fully-connected layer are adjusted properly, producing better resistivity predictions.

Epochs and batch sizes

The pass of the data forward to acquire the prediction loss, then backwards with backpropagation to adjust the weights is called an iteration, introduced in Section 2.5.3. During this process, the weights get adjusted, and generally, hundreds or thousands of such iterations are required for convergence. When the whole training data set has been passed forwards, then backwards, it is called an epoch. The number of epochs required to train depends on the model and the data set. When calculating the gradient, there are different methods for deciding how many training samples are used. In this project, mini-batch gradient descent is used, meaning the gradient is calculated based on a number of data points, called the batch size. Usually, the batch size depends on the size of the training data set, so we use 32 if the training data is not augmented, while 128 if the data is augmented.

3.4.2 Hyperparameter tuning with Keras tuner

The hyperparameters are the second type of parameters that the users sets before training CNN architectures. Each different setting produces a new hypothesis in the hypothesis space. A hypothesis is a guess on how we, the user think the data is modelled. The hyperparameters are essentially the parameters that define a model's behaviour. In our case, this relates among hyperparameters such as

- Number of kernels in convolutional layers
- Number of convolutional layers
- Kernel sizes
- Number of neurons in the fully-connected layer
- Dropout rate
- Learning rate

Selecting the proper CNN architecture is not a trivial problem, and is often solved by trial and error. Compared to loss functions that can be differentiated and solved using gradient descent or other numerical methods, model hyperparameters do not have that characteristic. There is no simple and easy way to select the proper hyperparameters, but rather testing out different hyperparameter settings. This can be done manually, but in this thesis, Keras-tuner is used to automate hyperparameter tuning.

With Keras, we define the general structure of the network, then in each part of the network, we define what ranges of hyperparameter values the different components can have. Keras tuner essentially allows us to define a search space of hyperparameters that are iterated over, creating and testing out different CNN models. The models are ranked based on MSE on the validation data, so we can separate bad architectures from good ones.

Before tuning, a hypermodel has to be defined, which is the search space we have referred to. The architecture of the CNN consists of three main layers: the convolutional layer where feature extraction of feature maps are produced, the pooling layer where feature maps extracted are summarized and stacked, then lastly the fully-connected layer where regression of resistivity happens.

Input layer

To begin with, we define the input layer of which the image data is received to the model shown in Figure 3.25

```
model.add(Conv2D(filters=hp.Int('conv2d layer',
          min_value=32,
          max_value= 256,
          step=32),
       kernel_size= hp.Choice('kernel_size', (3,3), (5,5)),
       padding='same',
       activation='relu',
       input_shape=(332,76,1)))
model.add(MaxPooling2D(pool_size=(2, 2),padding='same'))
```

Figure 3.25: Input layer of hypermodel

This input layer consists of a convolutional layer defined by *Conv2D* and a max-pooling layer *MaxPooling2D*. To iterate over numerical values, we define a hypermodel parameter as *hp.Int*, where in the example, kernels are set to. The minimum number of kernels the convolutional layer can have is set to 32, and the maximum to 256 kernels. For each iteration of different models, we want to search with a step size of 32. This means that for each new model, the number of kernels is randomly selected between 32 and 256 with a magnitude of 32. We also search over the kernel size, which is defined with the hyperparameter *hp.Choice*. This means that from all the possible choices defined, one of them is selected. Here we search over kernel sizes 3×3 and 5×5. We use the activation function ReLU and do not search over other functions. Since it is the input layer, we include the input shape of the image. In this case, the input size is 60cm images, consisting of an image of 332×76 pixels. The max-pooling layer has a fixed kernel size of 2×2.

Looping the number of convolutional layers

Further, we define a *for loop* which loops over the code inside of it to a defined number of times, shown in Figure 3.26. Inside the for loop, we define a pair of convolutional and max-pooling layers. The reason is to search over a different number of layers in the CNN architecture. Here, the iterator i is set to be a numerical hypermodel parameter since we want to search over CNN architectures with different numbers of layers. One model might have 3 convolutional layers, and another 5. Here we add the same code as in Figure 3.25, without having to define the input shape.

```
for i in range(hp.Int('conv_layers', 1,5)):
  model.add(Conv2D(
      hp.Int('number of conv layers',
             min_value=32,
             max_value=256,
             step=32),
             kernel_size=hp.Choice('kernel_size_2', (3,3), (5,5)),
             activation = 'relu',
             padding='same'))
  model.add(MaxPooling2D(pool_size=(2, 2),padding='same'))
```

Figure 3.26: Looping convolutional and max-pooling layers of hypermodel

Fully-connected layer

The last part of the CNN is the fully-connected layer where the regression happens. These layers consist of one flattening layer, one hidden layer, and one regression layer. After the for loop of extracting features with the *for loop* of convolutional

```
model.add(Flatten())
model.add(Dense(units=hp.Int(
          'dense_units',
          min_value=32,
          max_value=256,
          step=32),
      activation='relu'))
model.add(Dropout(rate=hp.Float('dropout_3',
                  min_value=0.0,
                  max_value=0.2,
                  step=0.05)))
model.add(Dense(1, activation='linear'))
```

Figure 3.27: Fully connected layer of hypermodel

layers, the feature maps are flattened with *Flatten()* to a 1D vector as input to the fully-connected layer. Here, the hidden and regression layer is added as the *Dense()* layers. The neurons number of neurons in the hidden layer is defined as a hypermodel parameter *hp.Int*, where the minimum and maximum values for the number of neurons are set to 32 and 256 respectively. Activation is set to ReLU. In this layer, *Dropout()* is added as a regularizer, with a dropout probability between 0 and 0.2 and step size 0.05. Finally, the regression layer is a dense layer with one neuron. The activation is a linear activation function since we are performing regression.

Optimizer and loss function

As a finishing touch, we compile the model. We then have to select an optimizer and a loss function. Here, we have used Adam, a popular optimizer with a learning rate between 0.0001 and 0.01. The loss function is set to be MSE since we want to measure the distance between our prediction and the ground truth.

Adam stands for adaptive moment estimation and is based on stochastic optimization. Unlike the classic gradient descent, Adam stores a history of the past gradients. These gradients averaged by a decaying exponential. This property of Adam leads to its adaptive characteristic when updating the gradient direction, and is one of the most popular optimizers today.

How the network decides to adjust the weights depends on the optimizer and the learning rate. The learning rate is the size of the step of which we move towards the minima in the loss function. With bigger steps, the loss function is converged faster but might miss the actual minima. With a smaller learning rate, the convergence is slower but might have a higher probability of finding the local or global minima.

Hypermodel summary and sample models

To visualize the whole hypermodel, Figure 3.28 and Table 3.1 are presented. Additional code for hyperparameter tuning with Keras is attached in Appendix A.2.

Hyperparameter	**Min value**	**Max value**	**Step size**
Number of kernels in each convolutional layer	32	256	32
Kernel sizes in each convolutional layer	(3,3)	(5,5)	2
Number of convolutional layers	1	5	1
Number of nodes in the fully connected layer	32	256	32
Dropout rate	0	0.2	0.05
Learning rate	0.0001	0.01	0.001

Table 3.1: Hyperparameter search space for CNN using Keras

```
def build_model(hp):
  model=Sequential()

  #Input convolutional layer
  model.add(Conv2D(filters=hp.Int('conv input layer',
            min_value=32,
            max_value= 256,
            step=32),
         kernel_size= hp.Choice('kernel_size', (3,3), (5,5)),
         padding='same',
         activation='relu',
         input_shape=(height,width,1)))

  #Maxpooling layer
  model.add(MaxPooling2D(pool_size=(2, 2),padding='same'))

  #Searching over 1 to 5 conv layers
  for i in range(hp.Int('conv_layers', 1,5)):
    model.add(Conv2D(
        hp.Int('number of conv layers',
              min_value=32,
              max_value=256,
              step=32),
              kernel_size=hp.Choice('kernel_size_2', (3,3), (5,5)),
              activation = 'relu',
              padding='same'))
    model.add(MaxPooling2D(pool_size=(2, 2),padding='same'))

  #Flattening of feature maps
  model.add(Flatten())

  #Fully-connected layer
  model.add(Dense(units=hp.Int(
            'dense_units',
            min_value=32,
            max_value=256,
            step=32),
        activation='relu'))
  #Dropout in fully-connected layer
  model.add(Dropout(rate=hp.Float('dropout_3',
                    min_value=0.0,
                    max_value=0.2,
                    step=0.05)))

  #Regression layer
  model.add(Dense(1, activation='linear'))
  model.compile(optimizer=Adam(hp.Float('learning_rate',
                min_value=1e-4,
                max_value=1e-2,
                sampling='LOG',
                default=1e-3)),
        loss='mean_squared_error',
        metrics=['mean_squared_error'])
  return model
```

Figure 3.28: Hypermodel for CNN hyperparameter tuning

Note that these are the hyperparameters that we, the user think are a suitable guess on what type of architecture that will give a good performance for prediction. There therefore exist other search spaces that weigh the hyperparameters differently and produce other types of architectures. The following image shows the code of the defined hyperparameter search space. Additionally, the ranges of the hyperparameters were also limited to how much GPU power was available.

With this search space, the tuning algorithm selects a combination of values inside the defined hyperparameter ranges. The idea is to test out enough hyperparameter settings until we find optimal ones. In this thesis, mainly two tuning algorithms are used: brute force with random search and adaptive tuning with the Hyperband tuner. Below are four sample models found with random search to demonstrate results from Keras tuner. As shown in Table 3.2, the found hyperparameters are explicitly presented to the user, together with the score(MSE), indicating which models performed the best. In this example case, 20 models were tuned, and Keras tuner found these four models to perform the best.

Hyperparameter	**Model 1**	**Model 2**	**Model 3**	**Model 4**
Number of kernels in each convolutional layer	256	160	256	128
Kernel sizes in each convolutional layer	(3,3)	(3,3)	(3,3)	(5,5)
Number of convolutional layers	5	3	2	2
Number of nodes in the fully connected layer	256	160	160	96
Dropout rate	0.1	0.05	0.25	0.45
Learning rate	0.005	0.003	0.008	0.008
MSE	0.4275	0.4463	0.4530	0.4670

Table 3.2: Four sample models from tuning

3.4.3 General model architecture description

Since we are dealing with three different data set splits, we want to search for one optimal model of each split. We then present a generalized figure for hyperparameter tuning. The following figure attempts to visualize the search space of CNN architectures.

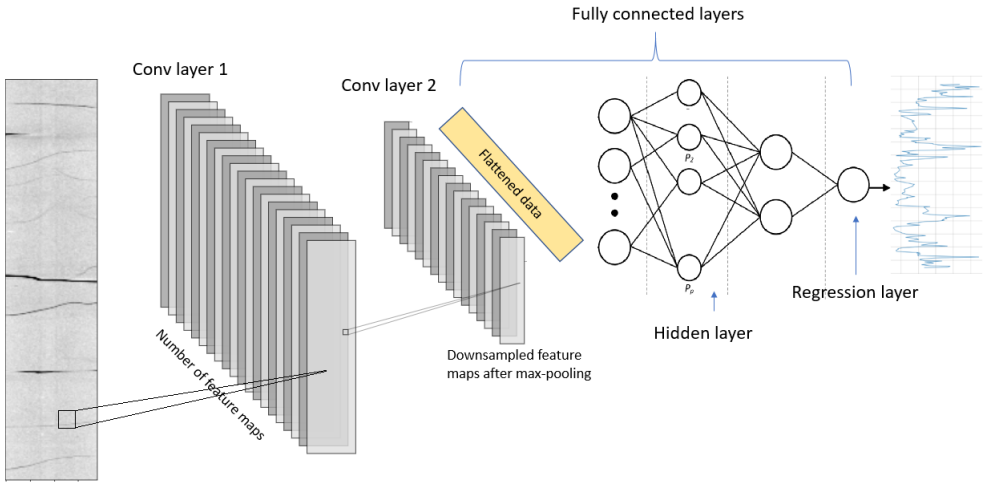


Figure 3.29: General CNN architecture description, inspired by [4]

The first convolutional layer takes the raw input, where the convolutional layer has an N number of kernels. N is a number between 32 and 256 that the tuner decides. The kernel sizes are set to either be (3x3) or (5x5) as we want to test out both kernel sizes. Going above these two selected would increase the loss of information so we limit the tuner to only try these two. The kernels are convolved and activated with the images, where we use the activation function ReLU. Each convolutional layer produces feature maps that are stacked dependent on the number of kernels. After each convolutional layer, the feature maps are max-pooled with kernels of (2x2), decreasing the image resolution by a factor of 2. This results in decrease of computational complexity by reducing the number of trainable model parameters. This process with convolutional and max-pooling layers continues depending on the number of such pairs. The number of pairs are searched over by the tuner, where we have set the number of convolutional and max-pooling pairs to be between 1 and 6. In the image, it is only shown 2 pairs for simplicity. With an increased number of convolutional and max-pooling layers, the model will be able to extract more abstract features from the images. This is something to keep in mind when tuning, as it affects the bias and variance of the model. We therefore want to keep an eye on these hyperparameters to avoid overfitting.

From the last max-pooling layer, the feature maps are flattened to be fed into the fully-connected layers. These layers are known as the classic ANN structure of the CNN. In this layer, we have one hidden layer with N numbers of nodes in the layer. The number of nodes is searched over by the tuner, choosing a number between 32 and 256. These are also set to be activated with the ReLU function. In the hidden layer, dropout is also used to drop out nodes as a regularization measurement. This makes it so that for each epoch, the hidden layer drops out nodes with a probability of p, the dropout rate. This lets more nodes of the hidden

layer contribute towards prediction, rather than having most of the computational load on a few neurons. Lastly, we have the last layer of the fully-connected layers, where the regression happens. Since we are dealing with a regression problem, this layer contains one node with a linear activation function, producing the prediction output. The details of the search space were previously presented in Table 3.1. Code for creating a general CNN model is also added in Appendix A.1.

3.4.4 Tuning algorithms

Random search

Random search is known as a brute-force tuning algorithm. With enough trials, the chance of finding a decent model will be high with the cost of time. For big model architectures where tuning is computationally costly, random search might not be the best alternative as other tuning algorithms converge to "good" models faster. Starting out with tuning in this thesis, Random search was frequently used to tune and test out simple CNN architectures. The main motivation with this take was to confirm if the code worked. Below are the arguments the tuning algorithm needed.

```
tuner=RandomSearch(
    build_model,
    objective='val_mean_squared_error',
    seed=42,
    max_trials=20,
    directory='tuner_4'
)
```

Figure 3.30: Random search configuration for hyperparameter tuning

For Random search, a number of trials had to be set for how many models were to be tuned. A number of epochs, i.e. how long the model would be trained for also had to be set. Objective was set to "validation mean squared error" to rank the models based on validation loss. Seed was set to make the tuning re-instantiate for each compilation, preventing the same models to appear. Below is a figure of the random search parameters used

Tuning with Hyperband

The Hyperband tuner bases its optimization on the successive halving algorithm which tries to allocate the resources optimally to find model architectures. By experience through this project, Hyperband was an efficient tuning algorithm. Instead of tuning for a big number of epochs, it tuned up to 3 epochs before starting on a new iteration, saving the score of the last one. It kept iterating over

again, discarding the bad ones before continuing training on 7 epochs this time. This halving process kept on until the *max_epochs* epoch was met. The arguments for Hyperband tuner are shown in 3.31. *max_epochs* is set as the ceiling of where the tuner stops tuning. The parameter *factor* decides how many of the total models are reduced for each iteration, as well as how big the increase of resources for the remaining models.

```
tuner=Hyperband(
    build_model,
    objective='val_mean_squared_error',
    seed=131,
    factor=3,
    max_epochs=20,
    directory='tuner_5',
    hyperband_iterations=1
    )
```

Figure 3.31: Hyperband tuner configuration for hyperparameter tuning

3.5 Model validation and selection of Convolutional Neural Network architectures

Now that a search space has been defined, we want to evaluate use this tuning strategy to find the three optimal models. This involves applying regularization techniques such as data augmentation, dropout, early stopping to find architectures for all three splits. These models will be evaluated based on their MSE score, until three optimal model architectures, one for each split, has been found. From the bullet-point list of model validation and selection, this involves the third point, as mentioned in Section 2.8.3

- We want to identify the machine learning algorithm that is best suited for the problem at hand; thus, we want to compare different algorithms, selecting the best-performing one as well as the best-performing model from the algorithm's hypothesis space.

Tuning was done in the previous section, consisting of using the 3-way holdout method together with Keras-tuner. This allowed us to validate various CNN architectures without looking at the test set. From the tuned models, Keras ranked each model with an MSE score, and from the list of tuned models, we selected the best fitting ones from each data distribution. Additional to MSE, R-squared will be used to evaluate the correlation between our predicted resistivity, and the actual validation resistivity labels, where an R-squared of 1 indicates perfect correlation.

In this section, we will present the performances of the three optimal models on

the validation set. Since we have three types of data sets, together with augmented data(horizontal and vertical flip), and different sizes of data(30cm and 60cm images), we will present validation results of the combination of all the settings: two image sizes, as well as the different augmentation techniques: no augmentation, augmentation before training and augmentation during training. In the end, one combination of image size and augmentation method used will be picked based on validation MSE and R-squared to test the holdout test set. The test results will be presented in the next chapter.

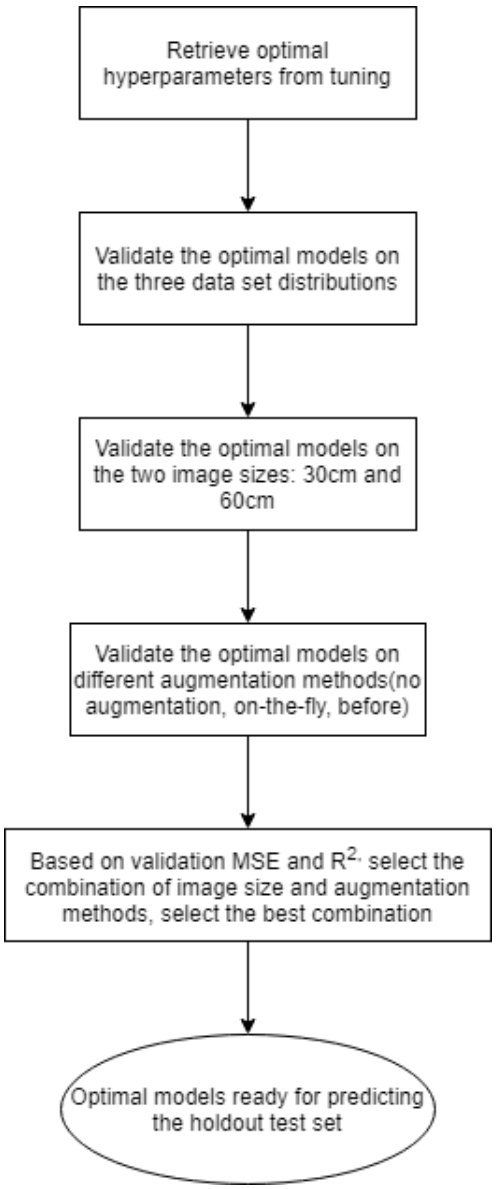


Figure 3.32: Flowchart for model validation

3.5.1 Model validation of random sampled split after data augmentation with overlap

As the randomly sampled validation set after data generation with overlap, we refer to the training and validation data distributions previously presented in Section 3.3.2 respectively. Tables of MSE vs. validation MSE will be shown to evaluate the performances. Additionally, both augmented and non-augmented validation will be done as well as validation on both 30cm and 60cm images.

Tuning with Keras tuner, the optimal model for this data set split was found with Hyperband search. Figure 3.33 visualizes the CNN, while Table 3.3 shows the details of the model. The optimal model contains five pairs of convolutional and max-pooling layers for feature extraction. The feature maps are downsampled for each layer pair, allowing the CNN to extract detailed information from the input images. After feature extraction, the feature maps are flattened and fed into the fully connected layer. Here, an ANN with one hidden layer containing 224 nodes computes the flattened data before the final regression layer with one node predicts resistivity.

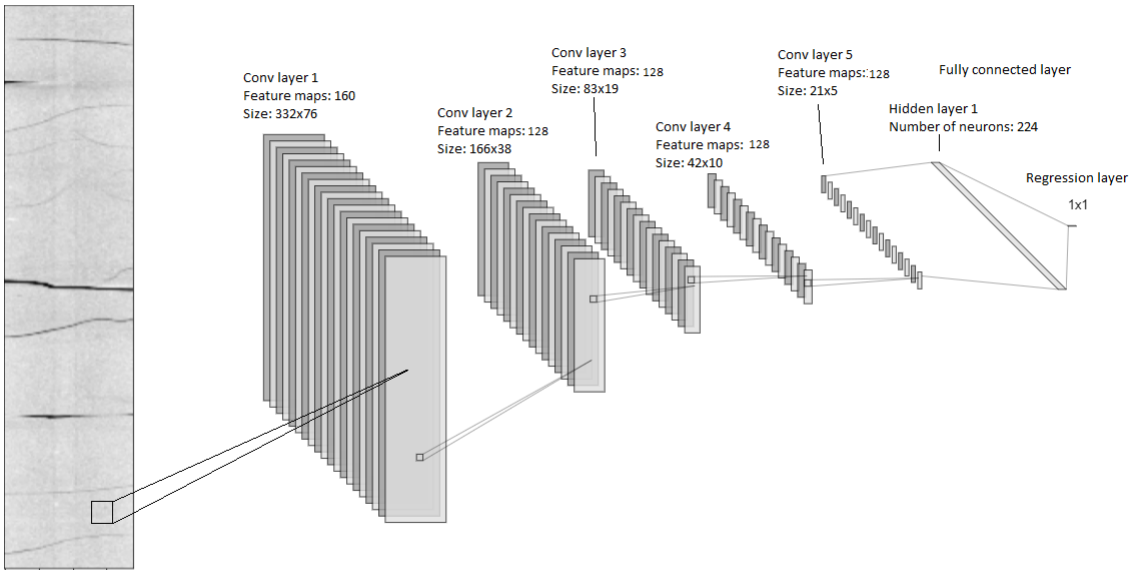


Figure 3.33: Visualization of CNN architecture for random sampling after overlap

Component	Layer type	Kernels/Nodes	Kernel size	Activation function
Convolutional component (Feature extractor)	Conv2D	160	(3x3)	ReLU
	MaxPool2D	160	(2x2)	-
	Conv2D	128	(3x3)	ReLU
	MaxPool2D	128	(2x2)	-
	Conv2D	128	(3x3)	ReLU
	MaxPool2D	128	(2x2)	-
	Conv2D	128	(3x3)	ReLU
	MaxPool2D	128	(2x2)	-
	Conv2D	128	(3x3)	ReLU
	MaxPool2D	128	(2x2)	-
Fully connected layer (Regressor)	Flatten	-	-	-
	Dense	224	-	ReLU
	Dense	1	-	Linear

Table 3.3: Details of CNN architecture of optimal model 1

This model was validated on the two proposed sizes: 30cm images and 60cm images. For each of the image sizes, validation was further done for no augmentation, augmentation before training the neural network, and after. Validation of all of the combinations proposed is presented in Table 3.4.

Size of images	Augmentation	Training MSE	Validation MSE	R-squared
30cm	No augmentation	0.0044	0.0056	0.9864
	Augmentation before training	0.0010	0.0451	0.8662
	Augmentation during training	0.0050	0.0067	0.9645
60cm	No augmentation	0.0021	0.0075	0.9687
	Augmentation before training	0.0038	0.0048	0.9868
	Augmentation during training	0.0076	0.0056	0.9487

Table 3.4: Model validation results for random sampled split after overlap

3.5.2 Model validation of continuous split after data augmentation with overlap

The model found for this data set distribution was tuned on the manually sampled data presented previously in Section 3.3.3. This data set split was our first proposed solution to the data leakage problem with overlapped data. The same procedure with model validation is done for all combinations of image sizes and augmentation methods presented in Table 3.6. Visualization of the optimal model found for this distribution is presented in Figure 3.34, and the details in Table 3.5

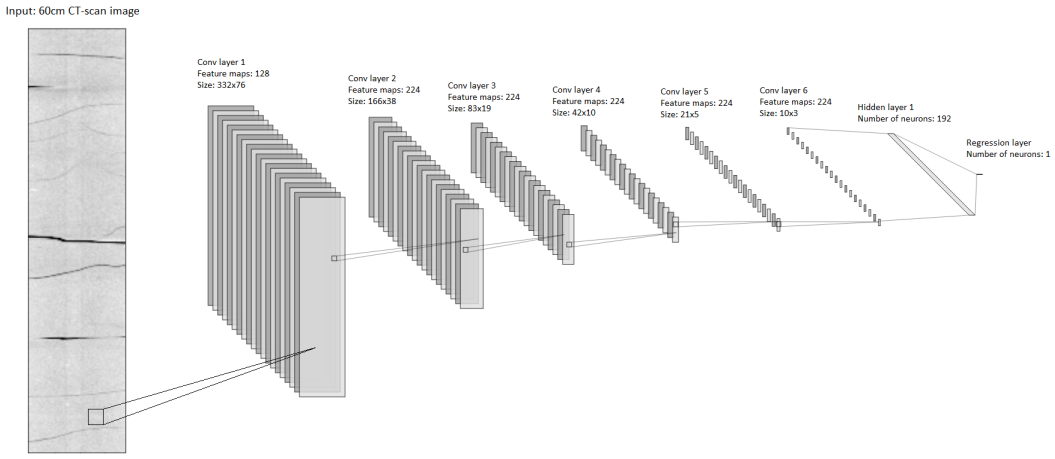


Figure 3.34: Visualization of CNN architecture for manual sampling after overlap

Component	Layer type	Kernels/Nodes	Kernel size	Activation function
Convolutional component (Feature extractor)	Conv2D	128	(3x3)	ReLU
	MaxPool2D	128	(2x2)	-
	Conv2D	224	(3x3)	ReLU
	MaxPool2D	224	(2x2)	-
	Conv2D	224	(3x3)	ReLU
	MaxPool2D	224	(2x2)	-
	Conv2D	224	(3x3)	ReLU
	MaxPool2D	224	(2x2)	-
	Conv2D	224	(3x3)	ReLU
	MaxPool2D	224	(2x2)	-
	Conv2D	224	(3x3)	ReLU
	MaxPool2D	224	(2x2)	-
Fully connected layer (Regressor)	Flatten	-	-	-
	Dense	192	-	ReLU
	Dense	1	-	Linear

Table 3.5: Details of CNN architecture of optimal model 2

Size of images	Augmentation	Training MSE	Validation MSE	R-squared
30cm	No augmentation	0.0036	0.3732	0.0532
	Augmentation before training	0.0021	0.4290	0.0885
	Augmentation during training	0.0114	0.4107	0.0421
60cm	No augmentation	0.0082	0.3725	0.0132
	Augmentation betbfore training	0.0044	0.2994	0.2507
	Augmentation during training	0.0092	0.4844	0.2831

Table 3.6: Model validation results for manually sampled split after overlap

3.5.3 Model validation of random sampled split before data augmentation with overlap

The last optimal model found by Keras was tuned on our second proposed solution. The data was randomly sampled before overlap, allowing us to random sample as well as avoiding data leakage, presented previously in Section 3.3.4. The same procedure as the last two optimal models were done, where model validation results for this optimal model are presented in Table 3.8. Visualization of the model and its details are presented in Figure 3.35 and Table 3.7.

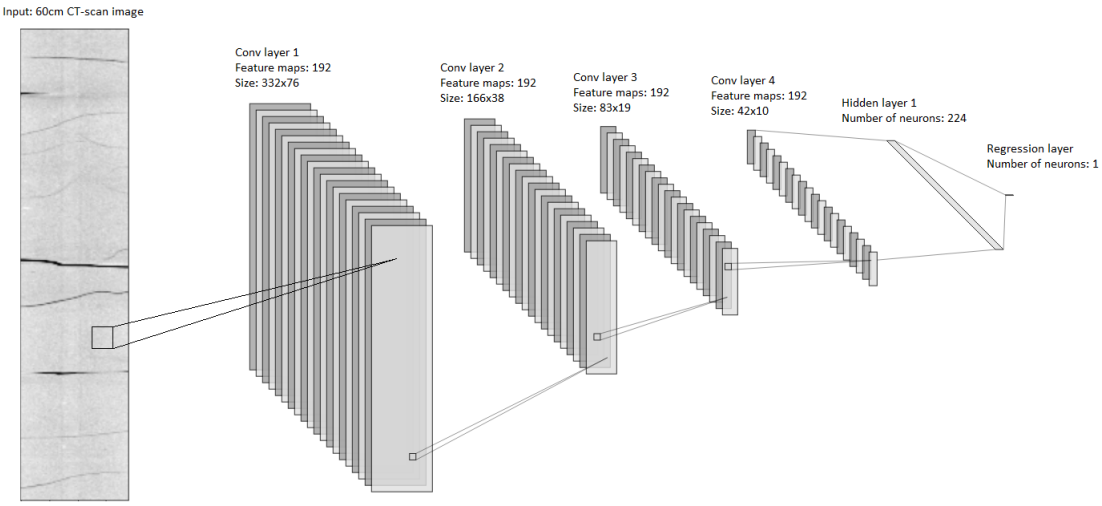


Figure 3.35: Visualization of CNN architecture for random sampling before overlap

Component	Layer type	Kernels/Nodes	Kernel size	Activation function
Convolutional component (Feature extractor)	Conv2D	192	(3x3)	ReLU
	MaxPool2D	192	(2x2)	-
	Conv2D	192	(3x3)	ReLU
	MaxPool2D	192	(2x2)	-
	Conv2D	192	(3x3)	ReLU
	MaxPool2D	192	(2x2)	-
	Conv2D	192	(3x3)	ReLU
	MaxPool2D	192	(2x2)	-
Fully-connected layer (Regressor)	Flatten	-	-	-
	Dense	224	-	ReLU
	Dense	1	-	Linear

Table 3.7: Details of CNN architecture of optimal model 3

Size of images	Augmentation	Training MSE	Validation MSE	R-squared
30cm	No augmentation	0.1549	0.4007	0.5069
	Augmentation before training	0.0083	0.3041	0.4841
	Augmentation during training	0.1418	0.3759	0.4762
60cm	No augmentation	0.1518	0.1935	0.5354
	Augmentation before training	0.0097	0.2909	0.4510
	Augmentation during training	0.1401	0.2462	0.3960

Table 3.8: Model validation results for random sampled split before overlap

3.5.4 Predicting the holdout test set

We have now validated three optimal models on the different variants of data sets, involving image sizes 30cm and 60cm, and also augmentation with and without. This has allowed us to observe how different data set distributions, image sizes, and data augmentation affect the performances measured with MSE. Further, we want to pick the best-performing instance from each of the three optimal models and predict the holdout test set. This will be presented in the next section.

Chapter 4

Results

In this section, the holdout test set is predicted with the three proposed CNN architectures found by model validation. One optimal model was constructed from the three different data set distributions: random sampled after overlap, manually sampled, and randomly sampled before overlap. The test set is considered as unseen data to our models, where validation of optimal models has been performed on training and validation sets. In this chapter, we will present the holdout test set, and predict it with the three optimal models. Cross-plots of predicted test set vs actual test set predictions will be shown to visualize the three models' performances. In these cross-plots, an optimal regression line and a regression line derived from our predictions will be presented for comparison measurements. The orange regression line represents the R-squared of our predictions. Prediction plots will also be shown to observe if the models are able to successful in modelling a trajectory similar to the test set. The models trained for each distribution are based on the model validation done in Section 3.5, where we choose the combination of augmentation and image size based on the validation MSE.

4.1 Prediction on the holdout test set for the three optimal models

4.1.1 Description of holdout test set

The holdout test set is continuously sampled, allowing us to cover the same depth interval, and the same resistivity values for all three data set distributions. A plot of the test set's resistivity values over the corresponding depths is plotted in Figure 4.1 over 13 meters. The depth in the plot will be numbered as the respective depth interval inside the 142 meter interval because of confidentiality. From the plot, we observe that there are clusters of data points over the trajectory of the test set. The density of data points in these linear clusters are caused by data generation with overlap. There are also gaps between the clusters which are caused by removal of data from pre-processing. With each optimal model found and their combination of data augmentation and image size, the holdout test set is predicted.

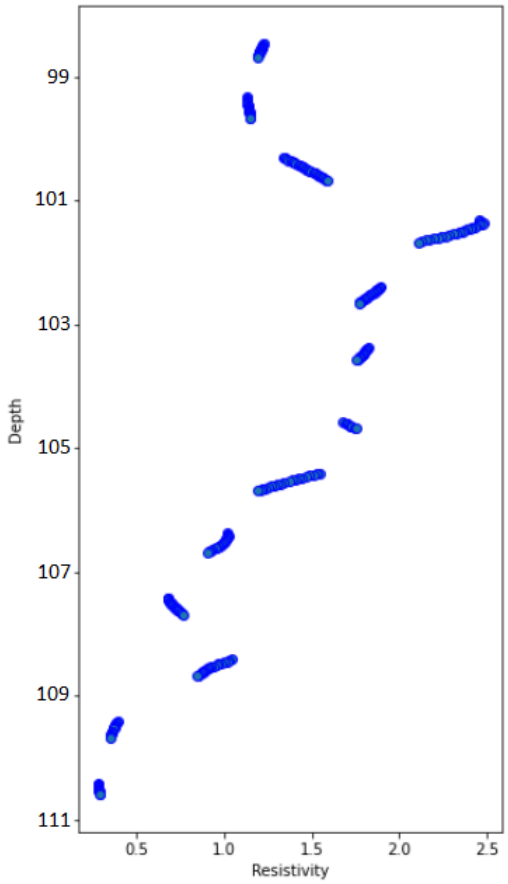


Figure 4.1: Visualization of test set: resistivity vs. depth

4.1.2 Optimal model 1: Tuned from the randomly split data set after overlap

The validation results of this distribution showed very good results both of training MSE and validation MSE from Table 3.4. The R-squared value showed almost perfect correlation between validation predictions and the actual validation labels. This was although expected since the random sampling was done after overlap, leading to data leakage of training samples in the validation set. In this subsection we will predict the holdout test set to observe the performance on unseen data.

From Table 3.4 from model validation and selection, it was observed that the combination of 60cm image size and augmentation before training resulted in the best validation results. The optimal model architecture from Figure 3.33 was then trained on with 60cm images and augmentation before training as regularization. This meant that the training data set consisted of 8000 images with image height of 60cm. To visualize the test performances, we present a prediction plot

presented in Figure 4.2 and a cross plot in Figure 4.3. The red dotted regression line is the optimal, desired line of predictions we want, whereas the orange dotted regression line represents the R-squared of our predictions.

Image size	Augmentation	Test MSE	Test R-squared
60cm	Before training	0.4022	-0.2377

Table 4.1: Test prediction MSE and R-squared from optimal model 1

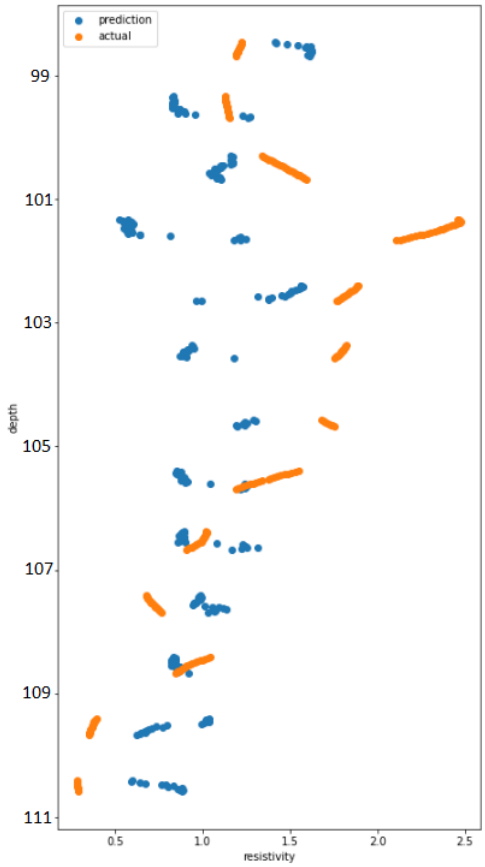


Figure 4.2: Prediction plot: test predictions vs. actual test resistivity from optimal model 1

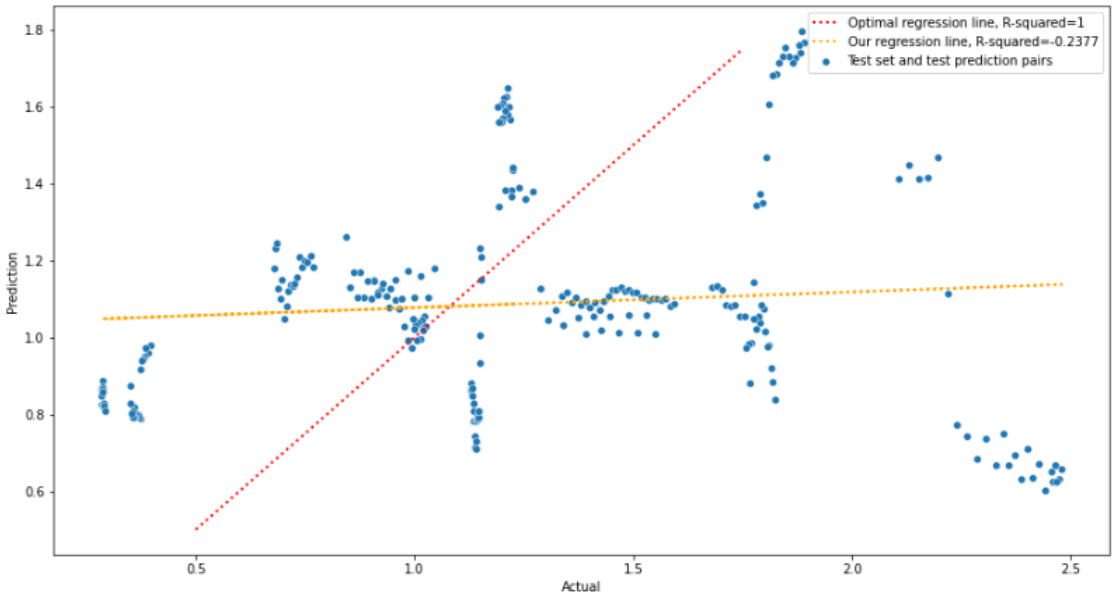


Figure 4.3: Crossplot of test predictions vs actual test resistivity from optimal model 1. The red dotted line and orange dotted line represent the optimal prediction trajectory versus our prediction trajectory.

4.1.3 Optimal model 2: Tuned from the continuously split data set

The same procedure as the first optimal model was done for the model found from the manually split data distribution. Figure 3.34 shows the optimal model we refer to. This was our first attempt to avoid data leakage by manually sampling the validation set. The test set for this split is though approximately the same as the test set from the previous subsection. Table 3.6 shows the MSE and R-squared from predicting the test set. Further, figures 4.4 and 4.5 show the prediction plot as well as a cross plot.

Image size	Augmentation	Test MSE	Test R-squared
60cm	Before training	0.3047	0.061

Table 4.2: Test prediction MSE and R-squared from optimal model 2

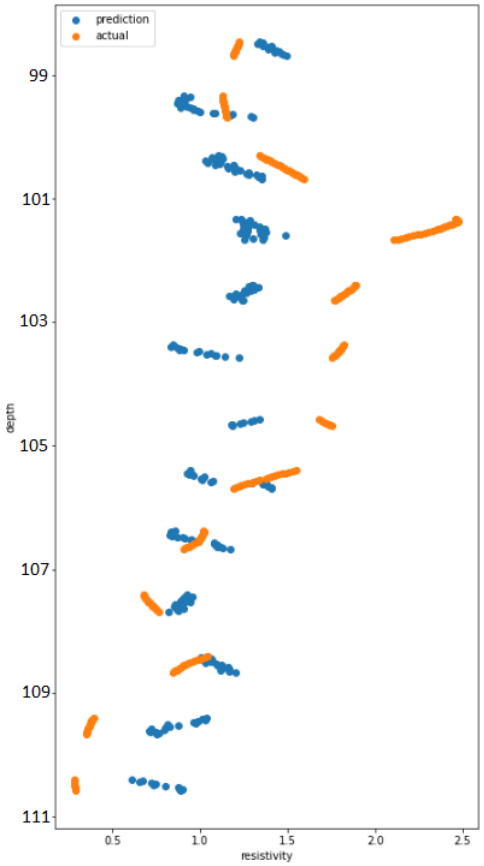


Figure 4.4: Prediction plot: test predictions vs. actual test resistivity from optimal model 2

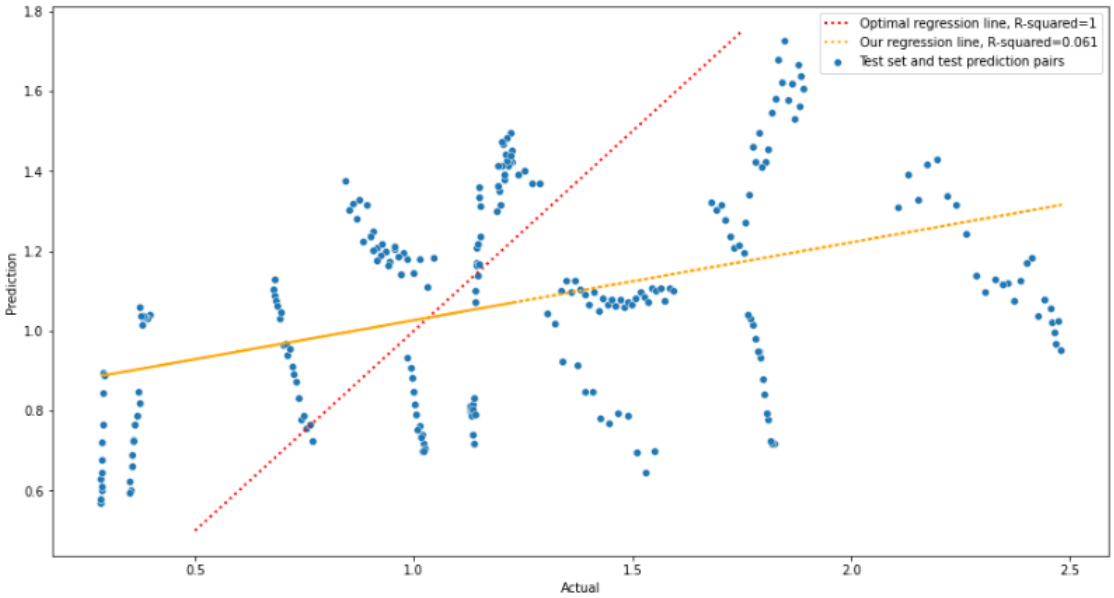


Figure 4.5: Crossplot of test predictions vs actual test resistivity from optimal model 2. The red dotted line and orange dotted line represent the optimal prediction trajectory versus our prediction trajectory.

4.1.4 Optimal model 3: Tuned from the randomly split data set before overlap

The last optimal model was tuned from the randomly split data set before overlap. This was our second solution as an attempt to deal with data leakage since we were able to random sample without having to deal with the overlap issue. The optimal model regarded for this split was previously shown in Figure 3.35 and Table 4.3 shows the test MSE and test R-squared for this model. Again, a plot of test predictions vs actual resistivity labels is shown in Figure 4.6 and the cross plot in Figure 4.7.

Image size	Augmentation	Test MSE	Test R-squared
60cm	No augmentation	0.1754	0.5117

Table 4.3: Test prediction MSE and R-squared from optimal model 3

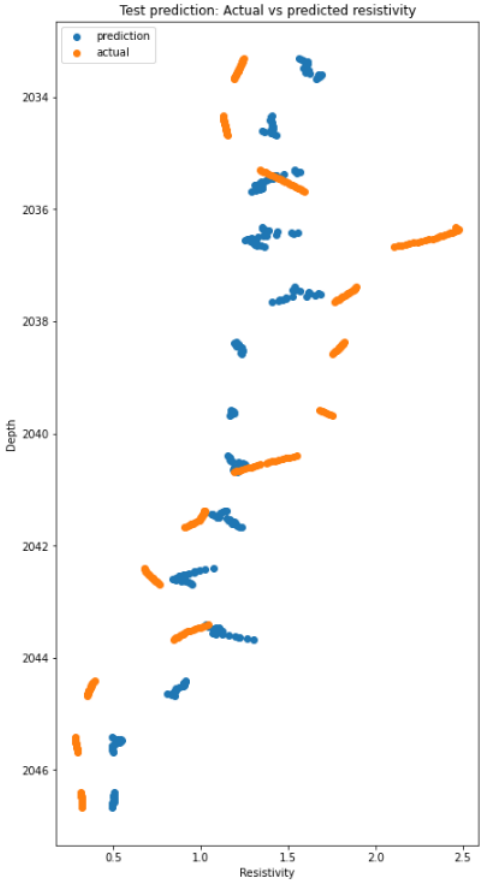


Figure 4.6: Prediction plot: test predictions vs. actual test resistivity from optimal model 3

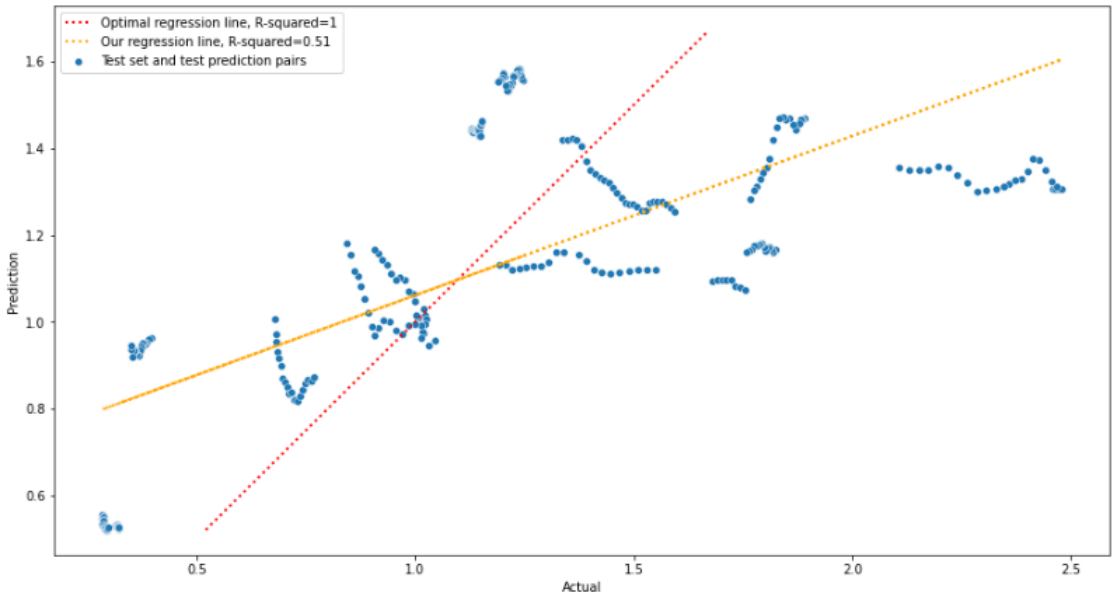


Figure 4.7: Crossplot of test predictions vs actual test resistivity for optimal model 3. The red dotted line and orange dotted line represent the optimal prediction trajectory versus our prediction trajectory.

Chapter 5

Discussion

We have now presented a strategy of predicting resistivity using 2D core CT-scan images of well formations, involving pre-processing, hyperparameter tuning, and model validation and selection. The data set quality and data set size have been two central factors, contributing to difficulties during modelling. Removal of disturbances in the data has been done, as well as regularization in form of augmentation as an attempt to both increase the data set sizes and prevent overfitting on the training data. With Keras, numerous CNN architectures have been searched over to find models that are fitting to model the lithofacies image data. In this section we want to discuss the test performances of three optimal models from each data distribution in light of pre-processing, hyperparameter tuning, and model validation. Various limitations of the three processes will continuously be highlighted and discussed.

5.1 Model performances on the holdout test set

Optimal models from three data distributions have been tested on the mutual holdout test set, and performances have been presented in Tables 4.1, 4.2 and 4.3. Different sizes of input images (30cm and 60cm), as well as augmentation with and without flipping have been utilized to train the models. The same Tables previously mentioned also show the chosen combinations of image sizes and augmentation method used to train the final model. As observed from cross plots in figures 4.3, 4.5 and 4.7, there is no clear correlation between our predicted resistivity versus the actual resistivity labels. One would desire to get an affine map throughout the plot, indicating that there is high correlation between both the actual test resistivity and our test prediction. From all three cross plots, the predictions are though very choppy, since we can observe linear clusters and gaps of the predictions. The cause of this might be because the test set in itself has gaps and is not very continuous by nature. More on the data set and pre-processing is discussed further in this chapter.

The first optimal model, trained on data distribution 1 showed to have very good

performance during model validation from table 3.4. This outcome was expected as we desired to show how data leakage affected our validation vs. test results. On the other hand, the test results on this data set split showed to have a rather high test MSE of 0.4022 and a test R-squared of -0.2377, showing weak correlation between the model's test predictions and the actual test resistivities. This is a big sign of overfitting, as the validation and training results are overly optimistic. We see the test predictions, which were held outside training and validation, are not comparable to the validation results looking at the validation R-squared of 0.98 and test R-squared of -0.2377. Having that big of a difference in performance between validation results and test results is a clear sign of overfitting, which we have attempted to emphasize with the three data set splits.

The second optimal model, validated on the continuous validation set performed better than the previous with lower test MSE and R-squared of 0.3047 and 0.061 respectively. However, the overall performance is still classified as bad, as both the prediction and cross plot show big spread in resistivity and struggles to find a similar trend to the original test resistivity curve presented in Figure 4.5.

The third optimal model was validated on the randomly sampled validation set before data generation with overlap. From both the cross- and prediction plots of the test set in Figures 4.6 and 4.7, we see large improvement in predictions. The MSE and R-squared from this model are also showing improvement, with a test MSE of 0.1754 and over 0.51 in R-squared score. The predictions have improved, and the spread is not as high as the two previous ones, which can be an effect of the random sample before overlap. Unlike the two previous models, the training data was not augmented which might have had an impact on the test performance. From table 3.8 of model validation for the third optimal model, we see that for both 30cm and 60cm image sizes, *no augmentation* had the highest R-squared, although not the highest validation MSE. However comparing the validation results to the two previous models, this third model performed the best on all points except for training MSE. This could mean that the type of training and validation distribution trained on might have had more impact on the test performances rather than regularization and augmentation methods applied.

At this point, looking solely at the test results, the three performances are not close to the kind of performance required for practical use in the industry. As also commented in the conclusions section, very likely more work on the data in form of analysis or data gathering, and probably more detailed modelling, should be performed to arrive at high technology readiness levels. In the two following sections, we then discuss the results from the pre-processing and the hyperparameter tuning steps.

5.2 Data set and pre-processing

Pre-processing has been a central part of the thesis, both for constructing the images with code from BRU-21 and augmenting the images to increase the data set size. Using data generation with 98% overlap, we were in theory able to increase the data set size by a 40-fold. However, having this much synthetic data may have limited the learning performance of our models as we have augmented the data to the point where a big majority of the data was synthesized. Additionally, there are several gaps in the resistivity channel because of the removal of missing intervals, artefacts, and high density areas. A major solution to these two issues would be to gather more field data so that the training data becomes cleaner (preventing the need for augmentation), and also so that the missing gaps "fill". This is also the cause of the jump in resistivity predictions from the cross plots previously shown in figures 4.3, 4.5 and 4.7. Having enough data is usually a requirement for proper modelling, something we do not experience in this thesis, having to use regularization to fix the data set issue.

Another reason why CNN might have difficulties modelling the image data could be the lack of distinct patterns in some images. Looking back at Figure 3.3b showing the 60cm image, there are some cracks and the color might be distinct, but there are no corners, edges, or patterns that stand out. The 30cm image in Figure 3.3a might although have more distinct features that are more meaningful to the CNN, making it easier to recognize other images with the same features. This leads to another point about the difficulties of using CNN for regression. CNN has lately shown to be very effective with object detection, for instance with self-driven cars: detecting humans, traffic light, and so on. In such image frames, there might be more distinct features involving humans, bikes, etc, making CNN more compatible with classification. Such distinct features are something we might lack in our application, leading to difficulties performing regression of resistivity.

Additionally, two images from totally different depths can have the same resistivity, but the CT-scan images may visually be different from each other in sense of color and cracks, etc. This may be caused by two different-looking rock types having the same resistivity, or the lack of image quality during retrieval of the rock samples. This then requires more detailed feature engineering, analyzing the visual features of the images.

5.3 Hyperparameter tuning and regularization

Tuning with Keras allowed us to search over the hyperparameters and the value ranges that we thought were suitable for modelling the image data. For instance, the range of the number of kernels and number of convolutional layers was probably sufficient in terms of model complexity. The kernel size was varied between sizes of 3 and 5 for the convolutional layers, but it seemed like the smaller kernel

size tended to perform better as it was chosen for all three optimal models. The number of neurons was also varied between a rather big range from 32 to 256. When defining the search space, bias and variance was taken into consideration, resulting in not too complex and also not too simple models. Hyperband tuner was used much more frequently than Random search for searching for CNN architectures. We experienced that Hyperband was more reliable and consistent in finding good models as random search was more probabilistic.

On the other side, other hyperparameters may have been tuned to create a more dynamic search space. For instance, tuning with different activation functions other than ReLU, although we focused in our thesis on ReLU, given its fame for being a function that typically leads to good performance. The same applies to the choice of the optimizer, that also could have been tuned instead of using only Adam. We could also have used additional layers such as batch normalization to add regularization. Unfortunately the constrained amount of time given to M.Sc. thesis implied we had to make some choices, and a-posteriori maybe we could have done differently (a-priori, though, the choices we made were in our opinion sufficiently well-thought).

Regularization techniques such as early stopping, dropout and data augmentation were used as attempts to avoid overfitting. However, it could be that adding more regularization and other techniques might not have been sufficient enough for this application. As mentioned in the previous section, there is an internal problem involving the data set quality and continuity of data set samples. Working more on gathering data would probably increase the performances greater than adding more regularization methods and tweaking hyperparameters. It could also be that modelling this application is not feasible enough to arrive at field deployments.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this thesis, we have analyzed and presented a strategy of applying the state-of-the-art learning method CNN for predicting CT-scan image data for regression of resistivity. In overview, the whole process involved tasks of data analysis, data pre-processing, and hyperparameter tuning of CNN architectures. Three optimal models were found from tuning, and finally, prediction on a holdout test set was done to evaluate the performance on unseen data. All three models however performed pretty poorly on predicting the test set, showing signs of incapability of learning the underlying characteristics from the images. Revisiting our research goal from the Section 1.2, i.e., "*we want to investigate the possibilities of modelling the image data to perform regression of resistivity*", our results did not quite meet the requirements of models that are sufficiently well-performing to be used in real life operations. The next paragraphs discusses on why we think this happened, and potential solutions to the issue.

Because of the small data set size, data augmentation has been a central part of the thesis for expanding the size with data augmentation with overlap and acting as a regularizer with flipping. The overlapped data was experienced to be troublesome, as splitting by random sampling lead to data leakage. We proposed three data set splits to first show an example of severe overfitting, then our two following solutions as attempts to deal with the overlap issue. The two latter distributions were presented as our solution to the overlap issue, where the model from random sampling before overlap performed best with an R-squared of 0.51. From this, we learned that using too much synthetic data became troublesome. Augmentation essentially cannot alone be used to the degree that the data set becomes good in terms of data set size and quality. Augmentation acts as a regularizer, and is limited on how much increase of performance it can provide. The starting point regarding the data set size and quality is still a bottleneck for this specific application where we have to process images that are very rich in features and possible looks even if belonging to the same rock type.

With Keras, a pre-defined search space over relevant hyperparameters we desired to tune was constructed. Using Hyperband search, we were able to effectively find efficient models due to the searching algorithm's adaptive nature. For model validation, two image input sizes were considered: 30cm and 60cm, and augmentation with and without flipping was also performed as regularization. It was observed that the 60cm image input performed best, however, there were mixed results with augmentation methods used. Using Hyperband we learned that we saved a lot of time because it converges very fast, as each iteration in most cases guarantees to return some good performing models. Compared to random search, Hyperband returns better models and the adaptive nature of the algorithm proves efficient (i.e., its additional computational requirements per step are effective in reducing the number of steps required, thus leading overall to a smaller computational requirement). We believe that this algorithm works well on our specific learning problem, likely because the hyperparameters range we defined is quite broad (this corresponding to a small prior information derived from domain expertise) and thus random search has a large field to search, causing it to be much less efficient than Hyperband.

In any case, by looking at a combination of validation results from Section 3.5 and of test results from Section 4.1, we conclude that the performances were more dependent on the data distribution trained on, rather than on accurately tuning the hyperparameters or applying specific regularization methods. Since the same tuning strategy was used when finding all three optimal models, this fact highlights that the poor results problem lies in the data set size and quality, which has contributed to difficulties in this modelling process. More work on gathering data and a more detailed modelling process is probably needed before deploying models in practice for this application.

Finally, we want to discuss the case of cross-validation on another well: after modelling the CT-scan image data for regression of resistivity, we desired to test the performance on a different, but similar well. Unfortunately, our modelling did not produce good enough results, thus cross-well validation on another well was not performed. Discussing with geologists, this may be because the formations at two different wells may be quite different, i.e., the same rock types in two different wells may present very different features (e.g., veins, strata, etc.). It is expected that cross-testing wells should perform better when these wells are geographically close, i.e. selecting a well close to the original one. It would therefore be interesting to see if our model would be able to adapt to unseen, but similar data from another well. Interestingly, different companies may want to pool their images so to enable better learning; however images from the own wells are considered very important proprietary information, and such information sharing mechanisms are unlikely to be implemented in the foreseen future.

6.2 Future work

To improve modelling performance, we propose a list of tasks that we think are worth investigating further for improvement of regression of resistivity.

Gathering more data: Gathering more data might be the most influential process, considering the state of our application. With enough data, the need for data generation with overlap should not be needed to the same degree, or not needed at all. This should allow us to observe a more realistic performance since the degree of synthetic data will decrease. The gaps of resistivity shown earlier in the data set should also be lesser, reducing the clusters of data. Gathering enough data should also open up for other powerful model validation techniques such as K-fold cross-validation, which we did not attempt because of the difficulties revolving around overlapped data. Note though that we do not know at the moment how much more data would be necessary to collect to achieve this. Very likely it is more about collecting enough diverse images corresponding to the same rock types, increasing the degree of seeing enough features that a rock type may present.

Pre-processing: A big part of the project was using pre-processing to remove disturbances from the data and turning the raw data into images. This has resulted in loss of data and inconsistencies in the continuity of resistivity labels. There are probably other pre-processing methods that can enhance our data so that the learning capabilities of CNN increases.

Tune more hyperparameters: In this thesis, there was a big focus on the number of convolutional and max-pooling pairs, as well as the number of neurons in the fully-connected layer. We focused on tuning these parameters because we thought they had the most influence on feature extraction and modelling of the image data. Testing out other hyperparameters such as other optimizers, different activation functions, and layer-types may contribute to better performance.

Testing other data augmentation methods: The augmentation method used directly on the images in this thesis was mainly flipping the images both vertically and horizontally. We also obtained some augmentation through max-pooling due to downsampling, thus blurring the feature maps. The reasoning of only using vertical and horizontal flip was due to our assumption that for instance rotating a well would interfere with the underlying characteristic of a well. Although, other image augmentation techniques probably could be used, such as zooming, rotating, or other creative methods.

Bibliography

[1] Y. Wu, B. Lu, W. Zhang, Y. Jiang, B. Wang and Z. Huang, 'A new logging-while-drilling method for resistivity measurement in oil-based mud,' *Sensors*, vol. 20, no. 4, p. 1075, Feb. 2020. DOI: 10.3390/s20041075. [Online]. Available: https://doi.org/10.3390/s20041075.

[2] Z. Bassiouni, 'Well logging,' in *Geophysics and Geosequestration*, T. L. Davis, M. Landrø and M. Wilson, Eds. Cambridge University Press, 2019, pp. 181–194. DOI: 10.1017/9781316480724.012.

[3] *Nmr Radial Saturation Profiling For Delineating Oil-Water Contact In A High-Resistivity Low-Contrast Formation Drilled With Oil-Based Mud*, vol. All Days, SPWLA Annual Logging Symposium, SPWLA-2008-Y, May 2008. eprint: https://onepetro.org/SPWLAALS/proceedings-pdf/SPWLA08/All-SPWLA08/SPWLA-2008-Y/1799356/spwla-2008-y.pdf.

[4] K. Chawshin, C. F. Berg, D. Varagnolo and O. Lopez, 'Lithology classification of whole core ct scans using convolutional neural networks,' *SN Applied Sciences*, vol. 3, no. 6, pp. 1–21, 2021.

[5] N. Aldahoul and Z. Zaw, 'Benchmarking different deep regression models for predicting image rotation angle and robot's end effector's position,' Oct. 2019, pp. 1–6. DOI: 10.1109/ICOM47790.2019.8952047.

[6] S. Tang, S. Yuan and Y. Zhu, 'Data preprocessing techniques in convolutional neural network based on fault diagnosis towards rotating machinery,' *IEEE Access*, vol. 8, pp. 149 487–149 496, 2020. DOI: 10.1109/access.2020.3012182. [Online]. Available: https://doi.org/10.1109/access.2020.3012182.

[7] K. Chawshin, A. Gonzalez, C. F. Berg, D. Varagnolo, Z. Heidari and O. Lopez, 'Classifying lithofacies from textural features in whole core ct-scan images,' *SPE Reservoir Evaluation & Engineering*, vol. 24, no. 02, pp. 341–357, 2021.

[8] R. Wicklin. (2020). 'Linear interpolation in sas,' [Online]. Available: http://proc-x.com/2020/05/linear-interpolation-in-sas-2/. (accessed: 28.04.2021).

[9] A. Gonzalez, L. Kanyan, Z. Heidari, O. Lopez *et al.*, 'Integrated multi-physics workflow for automatic rock classification and formation evaluation using multi-scale image analysis and conventional well logs,' in *SPWLA 60th Annual Logging Symposium*, Society of Petrophysicists and Well-Log Analysts, 2019.

[10] L. Taylor and G. Nitschke, 'Improving deep learning with generic data augmentation,' in *2018 IEEE Symposium Series on Computational Intelligence (SSCI)*, IEEE, Nov. 2018. DOI: 10.1109/ssci.2018.8628742. [Online]. Available: https://doi.org/10.1109/ssci.2018.8628742.

[11] N. Tomar. (2020). 'Data augmentation for semantic segmentation — deep learning — idiot developer,' [Online]. Available: https://nikhilroxtomar.medium.com/data-augmentation-for-semantic-segmentation-deep-learning-idiot-developer-e2b58ef5232f. (accessed: 22.04.2021).

[12] JoJun-Mo, 'Effectiveness of normalization pre-processing of big data to the machine learning performance,' vol. 14, no. 3, pp. 547–552, Jun. 2019.

[13] C. D. Manning, P. Raghavan and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press, 2008. DOI: 10.1017/CBO9780511809071.

[14] T. P. S. University. (2020). 'Lesson 3: Describing data, part 2,' [Online]. Available: https://online.stat.psu.edu/stat200/book/export/html/61. (accessed: 03.05.2021).

[15] B. Ghojogh and M. Crowley, 'The theory behind overfitting, cross validation, regularization, bagging, and boosting: Tutorial,' *arXiv preprint arXiv:1905.12787*, 2019.

[16] J. Brownlee. (2019). 'Gentle introduction to the bias-variance trade-off in machine learning,' [Online]. Available: https://machinelearningmastery.com/gentle-introduction-to-the-bias-variance-trade-off-in-machine-learning/. (accessed: 10.12.2020).

[17] S. Fortmann-Roe, 'Understanding the bias-variance tradeoff.,' 2012. [Online]. Available: http://scott.fortmann-roe.com/docs/BiasVariance.html.

[18] J. Jordan. (2017). 'Evaluating a machine learning model.,' [Online]. Available: https://www.jeremyjordan.me/evaluating-a-machine-learning-model/. (accessed: 30.10.2020).

[19] S. Seema. (2018). 'Understanding the bias-variance tradeoff,' [Online]. Available: https://towardsdatascience.com/understanding-the-bias-variance-tradeoff-165e6942b229. (accessed: 05.11.2020).

[20] A. Navlani. (2019). 'Neural network models in r,' [Online]. Available: https://www.datacamp.com/community/tutorials/neural-network-models-r. (accessed: 18.11.2020).

[21] E. Alpaydin, *Introduction to Machine Learning, third edition*, ser. Adaptive Computation and Machine Learning series. MIT Press, 2014, ISBN: 9780262325752. [Online]. Available: https://books.google.no/books?id=7f5bBAAAQBAJ.

[22] A. T. Henriksen, 'Domain adaptation for maritime instance segmentation: From synthetic data to the real-world.,' 2019. DOI: http://hdl.handle.net/11250/2631161.

[23] P. Singh. (2020). 'Neural network from scratch,' [Online]. Available: https://medium.com/analytics-vidhya/neural-network-from-scratch-ed75e5e14cd. (accessed: 19.11.2020).

[24] B. Müller, J. Reinhardt and M. T. Strickland, *Neural networks: an introduction*. Springer Science & Business Media, 2012.

[25] A. Suman. (2020). 'Activation function,' [Online]. Available: https://medium.com/analytics-vidhya/activation-function-c762b22fd4da. (accessed: 06.05.2021).

[26] S. Albawi, T. A. Mohammed and S. Al-Zawi, 'Understanding of a convolutional neural network,' in *2017 International Conference on Engineering and Technology (ICET)*, 2017, pp. 1–6. DOI: 10.1109/ICEngTechnol.2017.8308186.

[27] F. La Rosa, 'A deep learning approach to bone segmentation in ct scans,' 2017.

[28] K. O'Shea and R. Nash, 'An introduction to convolutional neural networks,' *ArXiv e-prints*, Nov. 2015.

[29] D. Cornelisse. (2018). 'An intuitive guide to convolutional neural networks,' [Online]. Available: https://www.freecodecamp.org/news/an-intuitive-guide-to-convolutional-neural-networks-260c2de0a050/. (accessed: 12.04.2021).

[30] A. Dertat. (2017). 'Applied deep learning - part 4: Convolutional neural networks,' [Online]. Available: https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2. (accessed: 17.04.2021).

[31] R. Hassan and A. Mohsin Abdulazeez, 'Deep learning convolutional neural network for face recognition: A review,' Jan. 2021. DOI: 10.5281/zenodo.4471013.

[32] 'Auto-keras: An efficient neural architecture search system,' 2020. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/3292500.3330648.

[33] D. Baskan. (2020). 'An introduction to bayesian hyperparameter optimisation for discrete and categorical features,' [Online]. Available: https://medium.com/analytics-vidhya/bayesian-hyperparameter-optimisation-for-discrete-and-categorical-features-a26454f77ab2. (accessed: 20.04.2021).

[34] R. Sebastian, 'Model Evaluation, Model Selection, and Algorithm Selection in Machine Learning,' 2018. DOI: https://arxiv.org/abs/1811.12808.

[35] Y. Dfertin. (2019). 'Create a multi-label classification ai: Train our ai [part 2],' [Online]. Available: https://towardsdatascience.com/create-a-multi-label-classification-ai-train-our-ai-part-2-85064466d55a. (accessed: 10.11.2020).

[36] F. Chollet *et al.*, *Keras*, https://keras.io/api/preprocessing/image/, 2015.

[37] S. Thatte. (2019). 'Importance of sampling in the era of big data,' [Online]. Available: https://towardsdatascience.com/importance-of-sampling-in-the-era-of-big-data-d2cf83e06c6a. (accessed: 14.04.2021).

[38] L. Yang and A. Shami, 'On hyperparameter optimization of machine learning algorithms: Theory and practice,' *Neurocomputing*, vol. 415, pp. 295–316, Nov. 2020. DOI: 10.1016/j.neucom.2020.07.061. [Online]. Available: https://doi.org/10.1016/j.neucom.2020.07.061.

Appendix A

Code Listings

A.1 General code for construction of CNN model and performing predictions

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras import optimizers


height = 332
width = 76


#Load training, validation and test sets
x_train = np.load('x_train.npy', allow_pickle=True)
y_train = np.load('y_train.npy')
x_val = np.load('x_val.npy', allow_pickle=True)
y_val = np.load('y_val.npy')
x_test = np.load('x_test.npy', allow_pickle=True)
y_test = np.load('y_test.npy')


#Extract image from x_train(depth, image)
x_train_depth = []
x_train_imdata = []
for i in range(0, len(x_train)):
    x_train_depth.append(x_train[i][0])
    x_train_imdata.append(x_train[i][1])
x_train_depth = np.array(x_train_depth)
x_train_imdata = np.array(x_train_imdata)

x_val_depth = []
x_val_imdata = []
for i in range(0, len(x_val)):
    x_val_depth.append(x_val[i][0])
    x_val_imdata.append(x_val[i][1])
x_val_depth = np.array(x_val_depth)
x_val_imdata = np.array(x_val_imdata)
```

```
x_test_depth = []
x_test_imdata = []
for i in range(0, len(x_test)):
    x_test_depth.append(x_test[i][0])
    x_test_imdata.append(x_test[i][1])
x_test_depth = np.array(x_test_depth)
x_test_imdata = np.array(x_test_imdata)

#Assign training data to images, reshape input and rescale
x_train = x_train_imdata
x_val = x_val_imdata
x_test = x_test_imdata
#reshape input
x_train = x_train.reshape(-1, height, width, 1)
x_val = x_val.reshape(-1, height, width, 1)
x_test = x_test.reshape(-1, height, width, 1)

x_val = x_val/255
x_train = x_train/255
x_test = x_test/255


#Define CNN model, here a random model has been used. General template for
#constructing model from known hyperparameters
def CNN_model():
  Regressor = Sequential()
  Regressor.add(Conv2D(192, kernel_size=(3, 3), activation = 'relu',
                       padding='same', input_shape=(height,width, 1)))
  Regressor.add(MaxPooling2D(pool_size=(2, 2),padding='same'))
  for i in range(1,4):
    Regressor.add(Conv2D(224, kernel_size=(3, 3), activation = 'relu', padding='same'))
    Regressor.add(MaxPooling2D(pool_size=(2, 2),padding='same'))
  Regressor.add(Flatten())
  Regressor.add(Dense(96, activation='relu'))
  Regressor.add(Dropout(0.05))
  Regressor.add(Dense(1, activation='linear'))
  Adam= optimizers.Adam(lr=0.0013)
  Regressor.compile(loss = 'mean_squared_error', optimizer=Adam, metrics=['mean_squared_error'])
  return Regressor

#Initiate CNN model and show details of model
CNN_model = CNN_model()
CNN_model.summary()

#Fit CNN_model to the training data as well as validating on the validation data
history = CNN_model.fit(x_train, y_train, batch_size=32, epochs=40,
                        validation_data=(x_val, y_val), shuffle=True, verbose=1)
```

A.2 Keras module for hyperparameter tuning

```
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras.optimizers import Adam
from kerastuner.tuners import Hyperband

#Size of input image
```

```
height = 332 #166
width = 76

#Define search space(hypermodel)
def build_model(hp):
    model = Sequential()

    # Input convolutional layer
    model.add(Conv2D(filters=hp.Int('conv␣input␣layer',
                        min_value=32,
                        max_value=256,
                        step=32),
                        kernel_size=hp.Choice('kernel_size', (3, 3), (5, 5)),
                        padding='same',
                        activation='relu',
                        input_shape=(height, width, 1)))

    # Maxpooling layer
    model.add(MaxPooling2D(pool_size=(2, 2), padding='same'))

    # Searching over 1 to 5 conv layers
    for i in range(hp.Int('conv_layers', 1, 5)):
        model.add(Conv2D(
            hp.Int('number␣of␣conv␣layers',
                   min_value=32,
                   max_value=256,
                   step=32),
            kernel_size=hp.Choice('kernel_size_2', (3, 3), (5, 5)),
            activation='relu',
            padding='same'))
        model.add(MaxPooling2D(pool_size=(2, 2), padding='same'))

    # Flattening of feature maps
    model.add(Flatten())

    # Fully-connected layer
    model.add(Dense(units=hp.Int(
        'dense_units',
        min_value=32,
        max_value=256,
        step=32),
        activation='relu'))
    # Dropout in fully-connected layer
    model.add(Dropout(rate=hp.Float('dropout_3',
                                    min_value=0.0,
                                    max_value=0.2,
                                    step=0.05)))

    # Regression layer
    model.add(Dense(1, activation='linear'))
    model.compile(optimizer=Adam(hp.Float('learning_rate',
                                          min_value=1e-4,
                                          max_value=1e-2,
                                          sampling='LOG',
                                          default=1e-3)),
                  loss='mean_squared_error',
                  metrics=['mean_squared_error'])
    return model

#Define hyperband search parameters
```

```
tuner=Hyperband(
    build_model,
    objective='val_mean_squared_error',
    seed=137,
    factor=3,
    max_epochs=20,
    directory='tuner_9',
    hyperband_iterations=1
    )
#Search over defined search space(hypermodel)
tuner.search(x_train, y_train, batch_size = 32, validation_data=(x_val,y_val),
                                            shuffle=True, verbose=1)

#Print 10 best models found
tuner.results_summary()

#Retrieve the best model from tuning
best_hps = tuner.get_best_hyperparameters(num_trials = 1)[0]

#Print details of CNN architecture(layer types, number of kernels in each layer, details of fully-connected-la
model_best_hps.summary()

#Tune the optimal model found over an increased number of epochs
history = model_best_hps.fit(x_train, y_train, batch_size=32, epochs=50,
                             validation_data=(x_val, y_val), shuffle=True, verbose=1 )
```

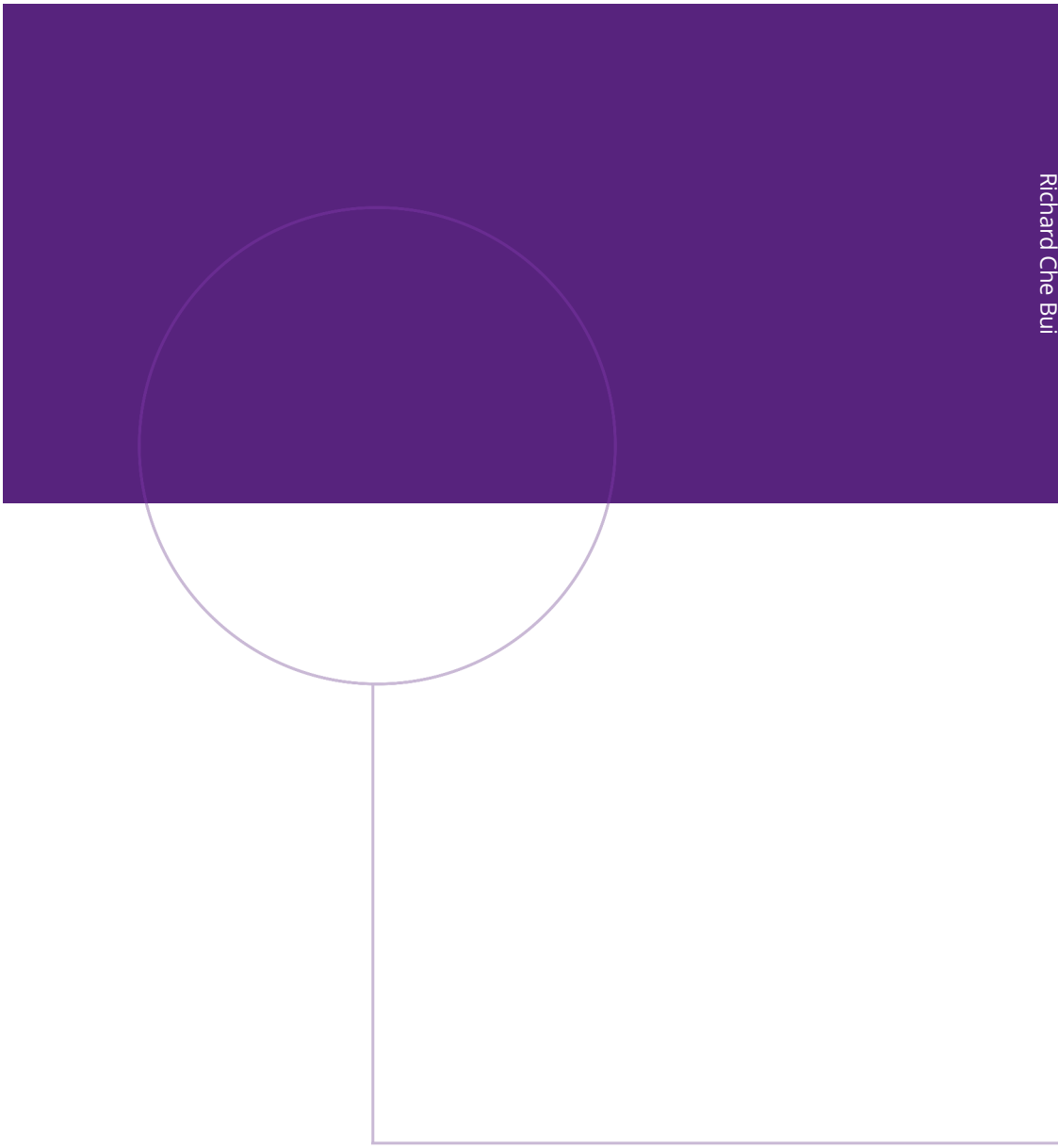
A.3 Augmentation on-the-fly

```
from keras.preprocessing.image import ImageDataGenerator

#Initiate ImageDataGenerator for augmentation with flipping
datagen = ImageDataGenerator(
    horizontal_flip=True,
    vertical_flip=True,
    fill_mode = 'constant'
    )

#Augments training data respective to the settings in ImageDataGenerator
augment = datagen.flow(x_train, y_train, batch_size=32)

#Fit the CNN_model to the augmented data
history = CNN_model.fit_generator(augment, steps_per_epoch=len(x_train)/32,
                epochs=50, validation_data=(x_val, y_val), shuffle=True, verbose=1 )
```



Richard Che Bui

NTNU
Norwegian University of
Science and Technology