Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Engineering
Department of Geoscience and Petroleum

Dennis Adelved

Fine-Tuning of Faster Region-Based Convolutional Neural Network for Automatic Core Plug Detection in Optical Core Images

Master's thesis in Petroleum Geosciences and Engineering
Supervisor: Carl Fredrik Berg
June 2020

Dennis Adelved

Fine-Tuning of Faster Region-Based Convolutional Neural Network for Automatic Core Plug Detection in Optical Core Images

Master’s thesis in Petroleum Geosciences and Engineering
Supervisor: Carl Fredrik Berg
June 2020

Norwegian University of Science and Technology
Faculty of Engineering
Department of Geoscience and Petroleum

Summary

The work presented in this thesis outlines two approaches for the training of a faster region-based convolutional neural network (Faster R-CNN) object detection model. This model can be used for automatic detection of both CCA and non-CCA core plugs in optical core images, by fine-tuning the parameters of a pre-trained model using the `Tensorflow` object detection API. The first approach consisted of training an initial model on images from the two wells, 6406/3-2 and 6406/8-1, which are chosen based on low and high variance in the visual appearance of the core plugs, respectively. Finally, the initial model is tested on two different test sets. The first test set consists of images from only two wells, and the second test set is randomly sampled from 25 wells. The second approach deals with further fine-tuning of the parameters of the initial model to increase the model performance on a specific data set. In this second approach, the initial model was fine-tuned using a small subset of the images from the first test set and tested on the remaining images in this set. Also, in both approaches, the input images are converted to grayscale before training, which resulted in a slight increase in the model performance based on the benchmarking of different preprocessing techniques conducted in this thesis.

The result from the first approach showed that the model was able to achieve high performance with respect to the evaluation metrics on the validation set. A significant performance loss was observed on both test sets, indicating that the inter-well variance in the visual appearance of the core plugs is too large to be learned from only two wells. However, the results from the second approach showed that if the variance in visual appearance is kept relatively low, the performance can be significantly increased by fine-tuning the initial model using a small number of plug locations and with minimal training time. Employing the second approach, the precision of the model increased to an acceptable level for the considered task. Additionally, the fine-tuning approach can be used as a semi-automatic labelling tool, which can significantly reduce the amount of time required for labelling data for further training of the model and other related object detection tasks.

Sammendrag

Denne oppgaven presenterer to fremgangsmåter for å trene en Faster Region-based Convolutional Neural Network (Faster R-CNN) objektgjenkjenningsmodell. Denne modellen kan brukes til automatisk gjenkjenning av både CCA og ikke-CCA kjerneplugger i optiske kjernebilder, ved å finjustere parameterne til en ferdig trent modell ved bruke av `Tensorflows` objektgjenkjennings-API. Den første fremgangsmåten besto av å trene en opprinnelig modell på bilder fra to brønner, 6406/3-2 og 6406/8-1, som er valgt på grunnlag av sin henholdsvis lave og høye varians i det visuelle utseendet til kjernepluggene. Til slutt testes den opprinnelige modellen på to forskjellige testsett. Det første testsettet består av bilder fra kun to brønner, og det andre testsettet består av tilfeldig utvalgte bilder fra 25 brønner. Den andre fremgangsmåten omhandler ytterligere finjustering av parametrene til den opprinnelige modellen der målet var å øke ytelsen til modellen på et spesifikt datasett. I den andre fremgangsmåten ble den opprinnelige modellen finjustert ved hjelp av en liten delmengde av bildene fra det første testsettet, og testet på de gjenværende bildene i dette datasettet. I begge fremgangsmåtene ble input-bildene konvertert til gråtoner før de ble brukt til å trene modellen, noe som resulterte i en liten økning i ytelsen til modellen basert på en referansemåling av forskjellige preprosesseringsmetoder utført i denne oppaven.

Resultatet fra den første fremgangsmåten viste at modellen var i stand til å oppnå en høy ytelse med hensyn til vurderingskriteriene på valideringssettet. Et betydelig ytelsestap ble observert på begge testsettene, noe som indikerer at variansen i det visuelle utseendet til kjernepluggene for brønnene i testsettene er for stor til å kunne læres fra kun to brønner. Resultatene fra den andre fremgangsmåten viste imidlertid at hvis variansen i det visuelle utseendet er begrenset, så kan ytelsen økes betraktelig ved å finjustere den opprinnelige modellen med en liten delmengde av kjernepluggene, med minimal treningstid. Ved å bruke denne fremgangsmåten ble modellens presisjon økt til et akseptabelt nivå for den tiltenkte oppgaven. I tillegg kan finjusteringsmetoden brukes som et halvautomatisk verktøy til å generere treningsdata, noe som kan redusere tiden som kreves for å merke data for videre opplæring av modellen og til andre relaterte objektgjenkjenningsoppgaver.

Preface

The work presented in this master thesis was conducted during the spring of 2020 and concludes my MSc in Petroleum Geosciences - TPG4925 at the Norwegian University of Science and Technology (NTNU). The thesis was written at and supervised by the Department of Geoscience and Petroleum (IGP).

Trondheim, June 15, 2020
Dennis Adelved

Acknowledgements

I would like to extend my sincere thanks to my supervisor Associate Professor Carl Fredrik Berg for the continuous guidance, the insightful discussions, and the opportunity to work with such an interesting topic for my master's thesis. Further, I would like to thank PhD Candidate Kurdistan Chawshin for the valuable feedback and the discussions regarding machine learning, and machine learning terminology. Lastly, I would like to thank my family and Helene for their love and support.

Table of Contents

List of Tables

List of Figures

Listings

Abbreviations

AI	-	Artificial Intelligence
ANN	-	Artificial Neural Network
AP	-	Average Precision
API	-	Application Programming Interface
AR	-	Average Recall
CCA	-	Conventional Core Analysis
CNN	-	Convolutional Neural Network
COCO	-	Common Object in Context
DL	-	Deep Learning
FNN	-	Forward-feeding Neural Network
FC	-	Fully Connected
GD	-	Gradient Descent
GPU	-	Graphics Processing Unit
GUI	-	Graphical User Interface
HC	-	Hydrocarbon
IoU	-	Intersection over Union
mAP	-	mean Average Precision
ML	-	Machine Learning
NCS	-	Norwegian Continental Shelf
NMS	-	Non-max Suppression
R-CNN	-	Region-based Convolutional Neural Network
ReLU	-	Rectified Linear Unit
RGB	-	Red Green Blue
RoI	-	Region of Interest
SCAL	-	Special Core Analysis
SGD	-	Stochastic Gradient Descent
SSD	-	Single Shot Detection
SVM	-	Support Vector Machine
YOLO	-	You Only Look Once

Chapter 1

Introduction

Optical core images and core plugs are important data sources within the petroleum industry and are utilized across several disciplines within the field of petroleum engineering and geoscience. These data sources are often used together to enhance the information of the subsurface since they provide the closest to ground truth representation of different aspects of the subsurface (McPhee et al., 2015). From optical core images, the visual characteristics of the subsurface can be inferred, e.g. the lithology, the bedding frequency, orientation, and the texture of the rock. These characteristics are important for placing the rocks in a geological context, such as determining the depositional environment. The core plugs allow for the measurement of the petrophysical properties of the rock, which can not be obtained from simply viewing the core. These properties include but are not limited to porosity, permeability, fluid saturation, wettability, and capillary pressure (McPhee et al., 2015). The petrophysical properties of the rock provide the engineer with valuable insight into the subsurface, which can be used in both identifying and characterizing possible reservoir zones.

the core analysis workflow. This analysis includes extracting core plugs from the whole core, on which the physical measurements will be performed. The extracted core segments are slabbed along the length and photographed to display the visual characteristics of the core (McPhee et al., 2015). This practice has lead to the accumulation of a large number of wells with optical core image data and core plug measurements, which are stored in databases. One such database is maintained by the Norwegian Petroleum Directorate (NPD), which contains both core images and core sampling reports for many public wells on the Norwegian continental shelf (NCS).

However, since the core sampling reports and the optical core images are stored separately, with the optical core images stored in an especially unfavorable format, the task of manually correlating these data sources can be time-consuming. The original core image contains several approximately one-meter core segments photographed side by side. These core images need to be manually cropped into individual segments to be used in combination with other data sources such as the core analysis and well log data. Once the core images have been cropped they can be correlated with the core analysis data by read-ing through the core sampling reports, manually identifying which core plug measurement identified by measured depth corresponds to which core plug location in the optical core image in pixels. This establishes a mapping between the well depth of the measurement and the pixels in the optical core image, which can be used to connect the physical measurement and the visual characteristics of the rock. This allows the optical core images to be used in conjunction with the physical core plugs measurements, and other well related depth data such as petrophysical logs, linking core analysis, well logs and reservoir geology (McPhee et al., 2015).

In recent years, several authors have successfully applied various machine learning techniques to automate workflows associated with classifying and extracting information from core images (Gonzalez et al., 2019; Thomas et al., 2011; Prince and Shafer, 2002) and predicting petrophysical rock properties such as permeability and porosity from core analysis data (Erofeev et al., 2019). This can mainly be attributed to the significant advancements within the field of machine learning and deep learning, with respect to the algorithmic advances, the availability of data, and cheaper hardware (Chollet, 2018). Additionally open-source platforms such as `Tensorflow` have made the building and training of the machine learning models accessible for researchers within fields besides computer science. However, in most cases, the data used in these machine learning techniques needs to be manually labelled and prepared in a standard format, which can be highly time-consuming.

1.1 The Scope

The goal of this thesis is to train an object detection model, which can be used for locating and classifying both conventional core analysis (CCA) and non-CCA plugs in optical core images. The objective of such a model is to facilitate the workflow of correlating optical core images, core analysis data and petrophysical well logs. This can be done by establishing an estimated depth locations of the core plugs within optical core images, based on their pixel locations, which in turn can be used to map the geological information in the core images to the core analysis data and well logs. Further, by combining this model with the automatic core cropping model developed in the specialization project of this thesis, the two models can be integrated into a fully automatic tool to improve the existing approaches of core analysis and core image data storage. The end-goal of such a tool is that it can be used as part of a digitization workflow, where the core segments are first extracted from the optical core images. The extracted core segments are then used as input to the core plug detection model, which locates and classifies the core plugs and returns an estimated pixel-depth map of the identified cores. This pixel-depth map can then be used in correlation with other data sources, which will allow for easy querying between the different data sources.

The main objective of this thesis is the training and optimization of the machine learning model that is concerned with locating and classifying the core plugs within the optical core images. Thus, the integration of the two models to a fully automatic tool is considered outside the scope of this thesis.

The training of the object detection model is performed by fine-tuning the weights of a pre-trained model, using the `Tensorflow` object detection API. This is done using the data from two wells $6406/3-2$ and $6406/8-1$. The reason for fine-tuning a pre-trained model, rather than training a model from scratch, is that it will significantly reduce the amount of time and data needed to reach an acceptable performance. The neural network architecture used in this thesis is the **Faster Region-based Neural Network** (Faster R-CNN) architecture, with an **Inception Resnet (v2)** feature extractor (Szegedy et al., 2016). This model choice is based on its reported performance, achieving the highest mean average precision (mAP) in a comparison between modern convolution object detectors (Huang et al., 2017a).

First, an initial model is trained and evaluated on the original core images from a single well (6406/3-2). This well is chosen based on the low variance in the visual appearance of the core plugs. Further, four additional models are trained on the same data, applying a different preprocessing technique to the input of each model with the goal of increasing the performance. Secondly, the best of these five models are chosen and trained on data which includes core images from a well with greater variance in visual appearance of the core plugs (6406/3-2 and 6406/8-1). This model is then used to test different model configurations until an acceptable performance is reached. The model is tested on two data sets; A data set consisting of images from only two wells, and a data set of randomly sampled core images from the public wells from the Norwegian continental shelf. Finally, a workflow for increasing the performance of the model within specific wells will be outlined. This involves fine-tuning the trained model, using a small subset of labelled images from a test set containing images from only two wells. The model will be uploaded to the `GitHub` repository (Adelved, 2020) of this thesis, which can be downloaded and used for further training.

1.2 Outline

Chapter 2 will provide the general background for the core analysis workflow and machine learning (ML). Some sections in this chapter touches on similar concepts to that of the specialization project preceding and associated to this masters thesis. Apart from some minor adjustments, these sections remain mainly unchanged. The machine learning background will give a brief introduction to the wider field of artificial intelligence (AI), before introducing the fundamental concepts used when the training and evaluation of neural networks. Section 2.4 will outline the general background for object detection and specific theory related to the model architecture used in this thesis. Finally, a brief summary of the object detection task and the main findings of the preceding specialization project will be outlined.

Chapter 3 outlines the introduced workflow of the thesis and the methodology for fine-tuning a pre-trained model with the `Tensorflow` object detection API and labelling the data. Furthermore, the background and configuration of the preprocessing techniques used in this thesis will also be included.

In chapter 4 the results from the various models will be presented and discussed according to the workflow defined in chapter 3. In addition, both the setup and results for the fine-tuning approach mentioned in the previous section will be presented and discussed.

The conclusion and suggestions for further work are presented in chapter 5.

Chapter 2

Background

The following sections will cover the background on the core analysis workflow, the background for machine learning and the different types of methods commonly used when working with image data for classification and object detection tasks. The core analysis sections are mainly based on McPhee et al. (2015) and will be para-phrased throughout the relevant sections. Therefore, the citation will be reserved for direct or close to direct quotations.

2.1 Core Analysis

Core analysis is the only direct and quantitative measurement of the subsurface and provides the best estimation of the ground truth, which any subsurface evaluation should rest (McPhee et al., 2015). Core analysis provides information about the important rock and reservoir properties such as porosity, permeability, fluid saturation, lithology and sedimentary structures. Conventional core analysis (CCA) is the most direct way of determining reservoir properties, which are used to characterise the reservoir and establishing a relationship between well log and core data (McPhee et al., 2015). CCA is performed on the dry core samples at surface conditions and will not be fully representative of the actual reservoir. The most common reservoir properties measured in CCA is the porosity and the permeability. Special core analysis (SCAL) provides the measurements for the relative permeability. The following sections will briefly outline the fundamentals of core analysis, from the extraction of the core from the well to imaging and storage.

2.1.1 Coring Process

The most common source for the cores used in the core analysis workflow are extracted from the full diameter of the core during drilling of a well. When the designated interval is reached, drilling is halted and the drill string is pulled out of the hole. The drill bit is removed and replaced with a coring bit. The rotary coring bit consists of solid metal with diamonds or tungsten for cutting, but unlike a drill bit, a rotary coring bit has a hollow centre. This is used to extract $9 - 10$m of full-diameter core from the target formation, depending on the length of the core barrel. The core barrel consists of an inner tube and an outer tube, separated by a ball bearing. This allows the inner tube, which stores the extracted core, to stay stationary while the rotating outer tube cuts the core from the formation. The core is usually stored within a liner, which is a third tube within the inner tube. The liner allows for easy extraction of the core at the surface. After coring is finished, the outer tube and the drill string is pulled up. This exerts a force on the inner tube which stays in place, thus breaking the core from the formation. The core is then retrieved, either by pumping up the inner tube, or hoisting it up by inserting a retrieval tool from the surface (McPhee et al., 2015). At the surface, the liner and the inner tube are separated and the core can be retrieved.

Another common coring technique is the extraction of sidewall cores. Sidewall cores are taken to minimize coring costs or to obtain reservoir rock samples in an interval which has either been cored and core recovery lost, or in an interval which has not been cored conventionally (McPhee et al., 2015). Sidewall cores are considered outside the scope of this text, and therefore will not be mentioned further.

2.1.2 Core Plugging

Most of the CCA and SCAL measurements are performed on plug samples that are cut from the full diameter core. The core plugs used in CCA range from 2.5-3.8 cm in diameter and are 2.5-7.5 cm long, depending on the what tests will be performed (McPhee et al., 2015). However, the CCA plugs are usually 2.5 cm in diameter. The SCAL plugs are usually larger with a standard diameter of 3.8 cm (McPhee et al., 2015). Core plugs are usually extracted in either the horizontal and vertical direction with respect to the bedding, with the exception of dean stark plugs. These are taken from the middle of the core along the long axis of the full diameter core. In figure 2.1 the different orientations of the plugs are shown with respect to bedding and the axes of the core. Both the dean stark and

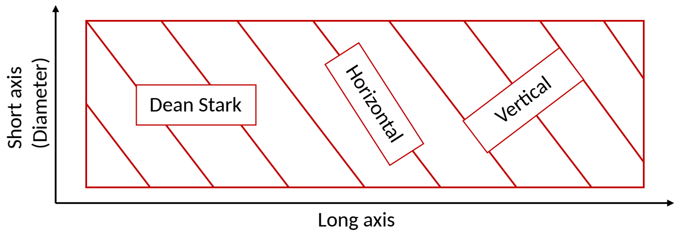


Figure 2.1: The orientations of the different types of core plugs with respect to bedding.

horizontal plugs, used to measures fluid saturation and permeability, respectively, should be taken from the centre of the core. This is done in order to minimize the effect of mud particles and filtrate on the measurements. The vertical plugs are mainly used for rock mechanics and conventional core analysis test. They are taken perpendicular to the maximum dip of the bedding. Since these plugs measure the minimum permeability, they are often cut close to the horizontal plugs so that maximum and minimum permeability can be compared (McPhee et al., 2015). The spacing of the plugs depends on the type of plug. Horizontal routine porosity-permeability measurement plugs taken in reservoir quality intervals are extracted approximately every 25 cm along the length of the core. This may vary based on lithology, where thick homogeneous intervals require fewer plugs compared to thinly bedded heterogeneous intervals (McPhee et al., 2015). SCAL plugs are taken from the preserved sections of the core. Sections that have been protected from evaporation, drying and exposure to oxygen. Thus the frequency of SCAL plugs depends on the sampling frequency of preserved sections, but are typically sampled at 1-2 m intervals. Dean stark plugs are sampled in an interval of 1 m, vertically along the long axis of the core.

2.1.3 Core Slabbing

Core slabbing is done in order to expose the sedimentological and lithological, as well as the bedding features of the core. These features are not visible on the mud invaded exterior of a core, fresh from the well. This allows for further geological examination of the core, as well as providing a clean and detailed surface for core photography (McPhee et al., 2015). The core is slabbed into three segments: A, B and C as seen in figure 2.2. First,

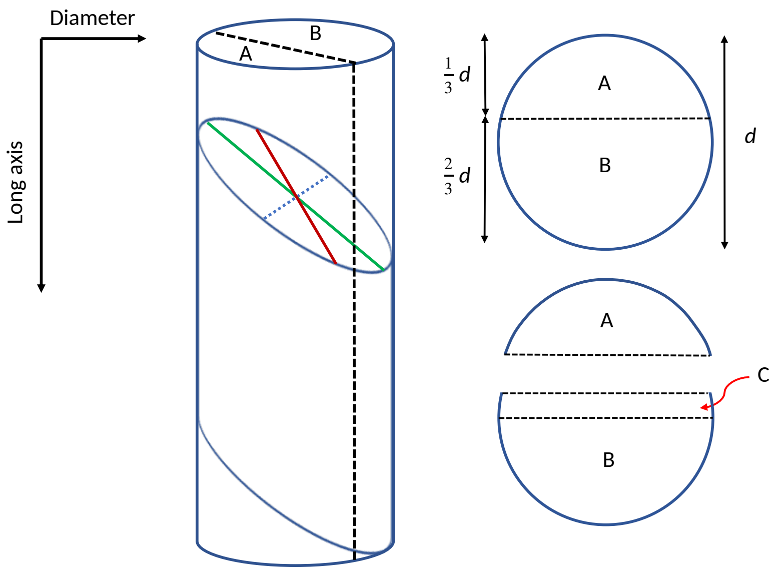
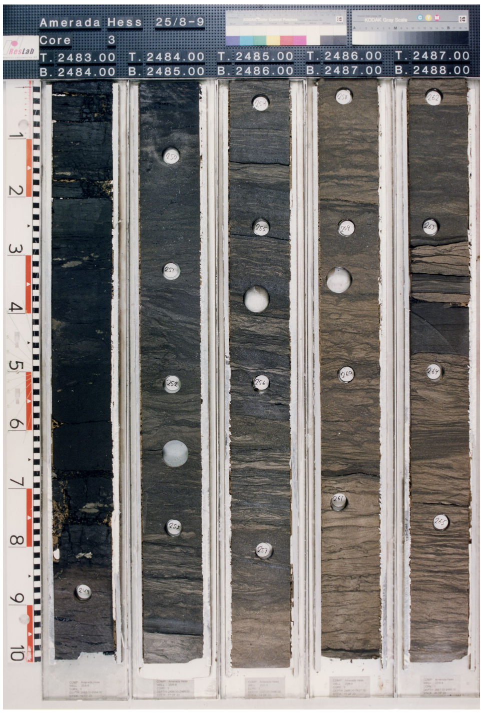


Figure 2.2: Illustration of how the cores are slabbed. The A,B and C segments are cut parallel to the maximum dip of the bedding. The maximum dip (true dip) - *green*. Apparent dip - *red*

the core is divided into A and B, with $1/3d$ and $2/3d$ of the diameter d, respectively. The core is always cut parallel to the maximum apparent dip that is visible from the outside of the core. The C segment is approximately 25 mm thick and is cut from the B segment. This is usually done by placing the B segment face-down, concave down, into a tray with transparent resin. After the resin has hardened, the top section of the core is removed by slabbing the core parallel to the resin tray (McPhee et al., 2015). This provides an easy way of displaying and preserving the information in the core, and is a particularly useful format for core photography and detailed geological examination.

2.1.4 Core Photography

Core photography is an important step in the core analysis workflow. Creating a digital record of the core allows for remote viewing of the core when the physical core is not available. Conventional imaging is the most common way of creating a digital representation of the external features of the core. In figure 2.3 a typical core image record can be seen, showing 5 C-segments from well 25/8-9. Note that in addition to the cores, the image contains information on the top and base depth of each segment in the well, a ruler for scale and a colour bar showing the intensity range of the image. Cores are normally pho-



NPD

Figure 2.3: An optical core image from well 25/8-9.

tographed under both natural light, as seen in figure 2.3, and ultraviolet light. The natural light shows the lithology and sedimentary structures. The UV-light shows the presence of hydrocarbons (HCs) since most HCs become fluorescent when exposed to UV-light, while water-saturated rocks do not.

2.2 Deep Learning

This section will give a brief introduction to the history of **artificial intelligence** (AI), as outlined by Russell and Norvig (2009). Then a short introduction to the general field of deep learning, as well as more specialized deep learning methods which are widely used in computer vision tasks such as image classification and object detection.

The field of artificial intelligence has been around since the 1950s and was in its early years, met with great enthusiasm and expectations (Russell and Norvig, 2009). Computers were programmed to solve well-defined problems that could be described by a list of formal, mathematical rules. It was believed that given a sufficiently large set of rules, a machine could achieve a human-like intelligence and solve any problem it was presented. Although, complex mathematical tasks and logic-based problems such as playing chess, could be solved by computers using this approach. The real challenge to AI proved to be solving the problems, that can not be fully described by a set of handcrafted, predetermined rules. Consider the task of recognizing an object in a picture. A task that may seem trivial to humans, and is solved intuitively without much effort but requires an immense knowledge of the world. Knowledge that can not be articulated as a finite set of formal rules, due to the limited understanding of how humans solve such intuitive tasks and the intractability of creating such a large and general set of rules.

One approach to solving these types of problems is with the use of **machine learning** (ML). Machine learning is a sub-field within AI which consists of a set of algorithms that allows computers to solve problems by learning statistical patterns from data it is presented to (Chollet, 2018). A sub-field of ML that has gained a lot of traction the last few years is **deep learning** (DL). This technique involves learning increasingly complex patterns from the input data (Chollet, 2018). Deep learning has shown great results in computer vision tasks such as image classification and object detection, and with recent AI and deep learning advances have allowed machines to surpass humans visual abilities in many image classification and object detection applications (Elgendy, 2020).

Deep learning is a sub-field of machine learning which is almost entirely based on **artificial neural networks**(ANNs) (Chollet, 2018), which will be outlined in the following sections. Additionally, the following section will outline the fundamental concepts of deep learning such as the motivation of machine learning, the architecture, training, testing and how these models are evaluated.

2.2.1 Artificial Neural Networks (ANNs) and Feedforward Neural Network (FNN)

Artificial neural networks are a set of machine learning algorithms that are inspired by the structure of the human brain, with respect to both its architecture and how it learns (Elgendy, 2020). An ANN consist of a set of neurons, which are arranged into layers. The neurons are the core processing unit of the network and are inspired by the neurons in the human brain. The neurons take a numerical input, applies some function to it and passes it to the next neuron. Each neuron in a layer is only connected to neurons in the previous and/or subsequent layers by a set of edges. Each edge has an associated weight which determines the importance of that specific connection. The value in a neuron is determined by the weighted sum of the values in the neurons that connects to it. Additionally, a bias term is added which is a constant with the value 1. The bias is added to allow the output from a neuron to be something else than 0, in cases when all the neurons that connect to it should be 0. The weighted sum and the bias are then passed through an **activation function**, to introduce non-linearity to the model. The non-linearity is added by determining which neurons should activate and pass it's output to the next layer (Elgendy, 2020). There are many types of activation function, however the default recommendation is to use the **rectified linear unit** (ReLU) (Goodfellow et al., 2016), given by equation 2.1:

$$g(z) = max\{0, z\} \tag{2.1}$$

where z is the weighted sum of the activations of the neurons in the previous layer and the weights connecting them and the current neuron plus the bias. This is illustrated in figure 2.4, where the activation a of a neuron is calculated by passing the weighted sum of the activations in the previous neurons, z, through the activation function g (Russell and Norvig, 2009). The weights and biases are referred to as the **parameters** of the network and usually denoted with $\boldsymbol{\theta}$.

The ReLU activation function activates a node, if and only if the input is above zero (positive). If the input is negative, the output is always set to zero. When the input greater than zero, it has a linear relationship with the output (Elgendy, 2020). This relationship can be seen in figure 2.5.

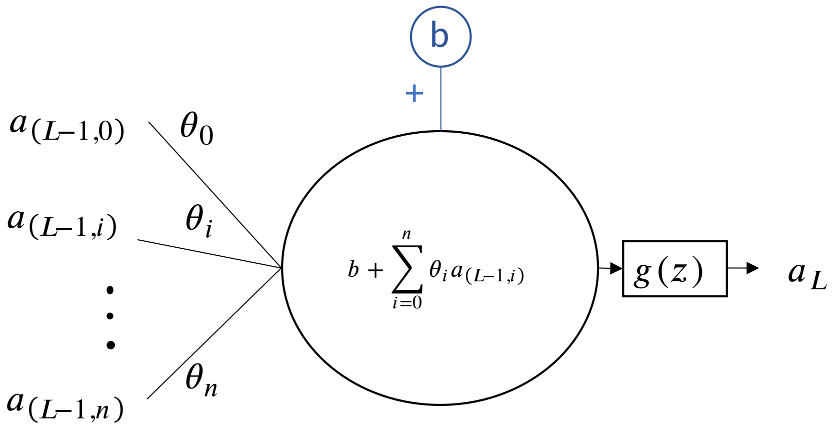


Figure 2.4: The activation for a single neuron in layer L is based on the weighted sum of the activations in the previous layer $L - 1$ and the bias passed through the activation function g.

The goal of an ANN is to approximate some function $f*$ (Goodfellow et al., 2016). For example, a classifier $y = f^*(\mathbf{x})$ maps an input to a category, which could be a model that is tasked to classify cats and dogs from an input image **x** into the correct class **y**.

There are several types of neural network architectures, each with its own set of rules for how information is communicated across the layers and with varying degrees of complexity. One of the most basic ANN architectures is the fully connected **feedforward neural network** (FNN), as seen in figure 2.6. A feedforward network defines a mapping $y = f(x; \theta)$ and learns the value of the parameters θ that result in the best function

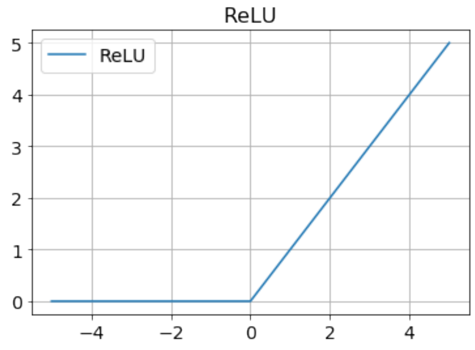


Figure 2.5: The rectified linear unit (ReLU).

approximation of $f*$ (Goodfellow et al., 2016). The information in one layer is only communicated forward to the subsequent layer in the network. Additionally, each neuron in a given layer is connected to all the neurons in the subsequent layer, hence the name fully connected feedforward (Goodfellow et al., 2016). The network consists of 3 types of layers; input layer, one or several hidden layers and an output layer as seen in figure 2.6. The

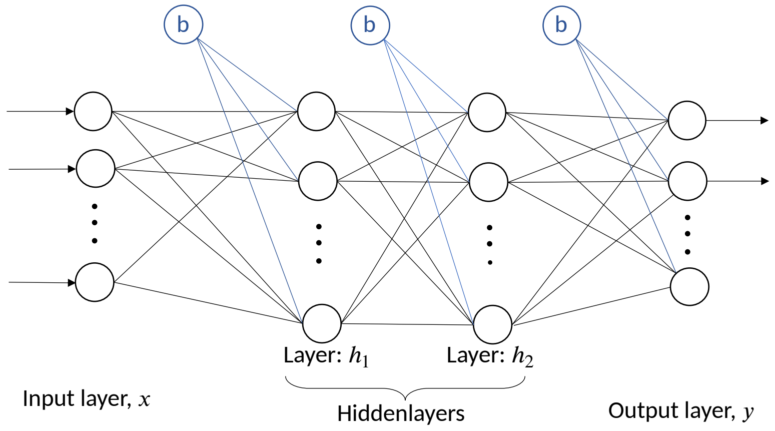


Figure 2.6: A forward feeding fully connected neural network with one input layer two hidden layers and an output layer containing both the neurons and biases. The hidden layers can be regarded as a function f that increase in complexity for every hidden layer added.

input layer is given the input data **x**, which is evaluated by the intermediate computations defined by the hidden layers h, before being classified as **y** in the output layer. The function f can be regarded as a combination of several smaller functions, one for each hidden layer in the model. Consider the network in figure 2.6, with the two hidden layers h_1 and h_2. If $f^{(1)}$ and $f^{(2)}$ corresponds to the function expression in h_1 and h_2, respectively, then f can be written as: $f(x) = f^{(2)}(f^{(1)}(x))$ (Goodfellow et al., 2016). Hence, each hidden layer adds a level of complexity to the calculations performed by the model. The number of layers in the network is referred to as the **depth**, giving rise to the name ”deep” in deep learning (Chollet, 2018).

2.2.2 Training

The goal of the training process is to find the parameters $\boldsymbol{\theta}$ of the network that best explains the relationship between the input **x** and the desired output y in the mapping y=f(**x**;θ). Provided enough examples (training data) this parameter configuration can be learned by minimizing the difference between the prediction from the network and the labelled training data. This difference is referred to as the loss of the model and is measured by a **loss function** (Goodfellow et al., 2016). Several functions can be used to calculate the loss, one such function is the **mean-squared error** (MSE). The loss function is often denoted by $J(\boldsymbol{\theta})$ (Goodfellow et al., 2016) and using the MSE it can be written as:

$$J(\boldsymbol{\theta}) = MSE = \frac{1}{n}\sum_{j}^{n}(\hat{y}_j - y_j)^2 \tag{2.2}$$

where $\hat{y}_i$ is the predicted output from the model, y is the labelled training example from the labelled data and n is the size of the training data set. As the prediction approaches the true value y, $J(\boldsymbol{\theta}) \rightarrow 0$, the network is said to improve its performance. Thus, the objective is to minimize the error between prediction and actual value. This is equivalent to wanting to have the highest possible activation for the neuron in the output layer that corresponds to the target y. Using the cat and dog classification example, if the input x is a cat, then it is desirable to have the highest activation in the output neuron that corresponds to the cat class. Since the activation of the neurons in the output layer is the weighted response of the activation in the previous layer, these need to be adjusted in order to get a prediction closer to the target value. This is essentially an optimization problem and can be solved using a gradient-based algorithm, such as **gradient descent** (GD) (Goodfellow et al., 2016).

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \alpha\nabla_\theta J(\boldsymbol{\theta}_i) \tag{2.3}$$

The GD algorithm is given in equation 2.3, where the new value of the function parameters $\boldsymbol{\theta}$, i.e the weights, are calculated at each iteration using its current value, the gradient of the loss function with respect to the weights $\nabla_\theta J(\boldsymbol{\theta})$ and the step-size or **learning rate** α. It can be understood as; for every new value $\boldsymbol{\theta}_{i+1}$, consider its current value $\boldsymbol{\theta_i}$ and take a step of size α in the direction of the steepest gradient of the loss function $\nabla J(\boldsymbol{\theta}_i)$, and repeat until the minimum value is reached. The magnitude of each update is determined by the learning rate α, which is a positive scalar (Goodfellow et al., 2016). The learning rate should be chosen with care. A small α will take longer to converge to the minimum, and a large α might over-shoot the minimum and diverge. A popular approach for choosing the learning rate is to set α to small constant value (Goodfellow et al., 2016). In GD the loss, $J(\boldsymbol{\theta})$, is calculated based on the entire training set as seen in equation 2.3 . Therefore, it is commonly referred to as **batch gradient descent**. The drawback with this method is that every update to the weights, determined by $\nabla_\theta J(\boldsymbol{\theta})$, requires one pass over the entire training set. Thus, the larger the data set, the slower the algorithm updates the model weights (Elgendy, 2020). Therefore, a commonly used optimization algorithm when working with large data sets is the **Stochastic gradient descent** (SDG).

Instead of updating the parameters based on all training examples, a subset or **mini-batch**, of the training data is randomly selected for each iterative update. This mini-batch is then an approximation of the true gradient. This leads to faster iterations at a slight cost of convergence rate (Suvrit Sra, 2011).

The topic of optimization methods and algorithms is an extensive and complicated mathematical field and is mostly beyond the scope of this thesis. However, some basic concepts such as local and global minima will be touched upon to clarify some of the challenges encountered by optimization algorithms and how they can be mitigated. Consider a real-valued function f, which is defined in the domain S. A point x^* is the **global minimum** if $f(x^*) \leq f(x)$ for all values of $x \in S$. Similarly, the **local minimum** is defined as the point x^* if $f(x^*) \leq f(x)$ for all values $|x - x^*| < \epsilon$. For the function f seen in figure 2.7, there is only one global minimum and every iteration of the gradient descent, according to equation 2.3, will converge towards the global minimum regardless of the initialization of x. Granted an appropriate choice for the learning rate α that will not overshoot the minimum and oscillate. The equation 2.3 is a **greedy algorithm**. A greedy algorithm makes locally optimal decisions in the search for the global optimal solution (Black, 2005). Such algorithms work well if every local update of the gradient moves the solution towards the global minimum. This is rarely the case in practice and an example

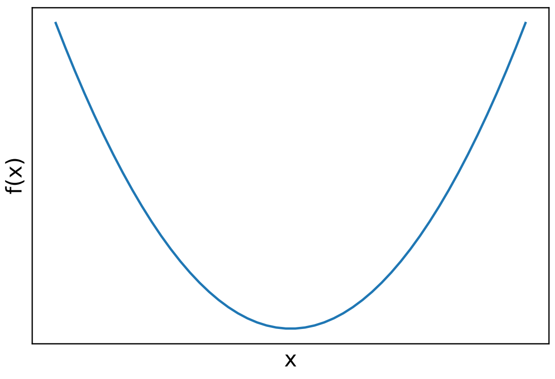


Figure 2.7: A function $f(x)$ containing one global minimum.

of this can be seen in figure 2.8. This function contains several minima that the algorithm may converge to (A, B and C), but only one global minimum (C). The solution depends on the starting position of the algorithm and can be illustrated by examining the ”peak” between A and B in figure 2.8. If the initial position is chosen slightly to the right of the peak, the algorithm will start traversing along the negative gradient and converge at A, which is a local minimum and a poor solution compared to B and C. Similarly if starting slightly left of the peak, the algorithm will converge towards B and stop before ever reaching C. Although non of these scenarios converge to the global minimum in C, the latter one is preferred since it provides an acceptable minimum compared to A. Generally there is no easy way to find the global minimum or confirm that the found minimum is the

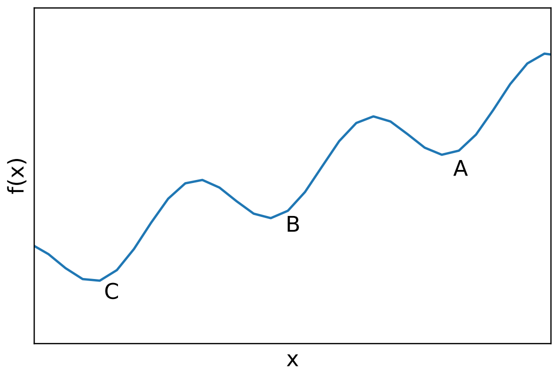


Figure 2.8: A function $f(x)$ containing 2 local minima (A and B) and one global minimum (C).

global minimum.

Although, the general field of optimization is considered outside the scope of this text it is worth mentioning that several optimization techniques have been developed to address the global-local minima problem. One popular technique is **moment optimization**, which utilizes the concept of momentum from physics to adaptively adjust the learning rate α. In the standard gradient descent in equation 2.3, the modification to the weights at each step is determined by a the learning rate α and the gradient of the loss function with respect to the weights $\nabla_\theta J(\boldsymbol{\theta})$. At each step, the momentum optimizer updates the weights similar to standard gradient descent but adds a fraction of the update from the previous step. That is, the modification of the weights at the current step depends on both the current gradient and the weight change of the previous step (Qian, 1999), and the modification to the weights at each step is given as:

$$\Delta\boldsymbol{\theta}_i = -\alpha\nabla_\theta J(\boldsymbol{\theta}_i) + \gamma\Delta\boldsymbol{\theta}_{i-1} \tag{2.4}$$

where γ is a constant value, determining the fraction of the update to the weights in the previous step to calculate the update in the current step. The value of the weight γ is usually in the range $0.5 - 0.9$ (Rumelhart et al., 1986). The last term of equation 2.4 is referred to as the momentum. As long as the gradients of the current step and the previous step points in the same direction the update to the weights increases i.e. builds momentum. This allows for faster convergence when the gradient is strictly decreasing. The momentum also makes the loss function less sensitive to small changes to the direction of the gradient. This is the most important aspect of the momentum optimizer, which reduces the risk of getting stuck in a local minimum. A commonly used analogy for illustrating the effect of adding momentum is that of a ball rolling down a hill as seen in figure 2.9. The left figure shows the standard gradient descent optimizer, which only moves the weights in the opposite direction of the gradient at the current step. In this case, the chosen learning rate α is too small, and the ball gets stuck in a local minimum. When using the same α and adding momentum, the ball is accelerating on the downhill slope since the gradients are pointing in the same direction. When the gradient changes direction, the momentum

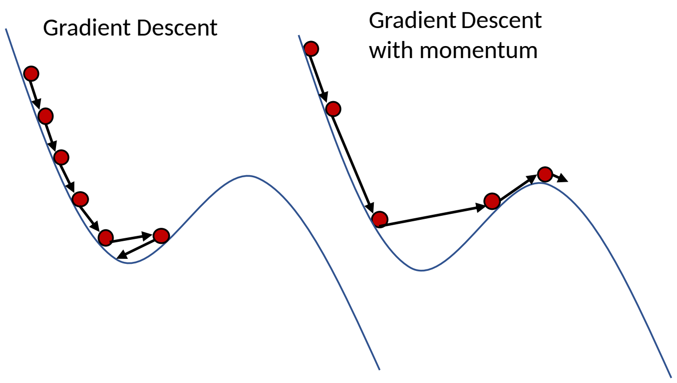


Figure 2.9: Illustrating the effect of momentum in GD, using the ball and hill analogy. *(left)* GD without momentum. *(right)* GD with momentum.

built in the downhill slope is large enough to push the ball out of the local minima. This is the essence of including the momentum and is found to increase the convergence rate significantly (Rumelhart et al., 1986).

2.2.3 Backpropagation

As outlined in the previous section, the goal of the training process is to minimize the loss function in equation 2.2 in order to bring the network prediction $\hat{y}$, the output neuron with the largest activation, closer to the desired output y. This is achieved by adjusting the weights in the network in such a manner that the loss is minimized, i.e. taking a step in the opposite direction of the steepest gradient of the loss function $\nabla_\theta J(\boldsymbol{\theta})$ with respect to θ. However, since the loss only may be calculated for the output layer, the adjustment needed for the weights earlier in the network can not be calculated directly. Therefore, the loss in the output layer must be translated backwards, and this is done by using **backpropagation**. The backpropagation algorithm computes the gradient of the loss function by applying the chain rule of calculus (Goodfellow et al., 2016) from the output to the input to identify how much each weight contributes to the error. The goal of this section is not to derive the backpropagation algorithm, but to illustrate the concept of how it works.

Consider the simple neural network consisting of the three neurons arranged in three layers L, $L-1$, $L-2$ connected by the weight θ^L and θ^{L-1} and with the activations a^L, a^{L-1} a^{L-2} as seen in figure 2.10. Since the activation in the output neuron a^L corresponds

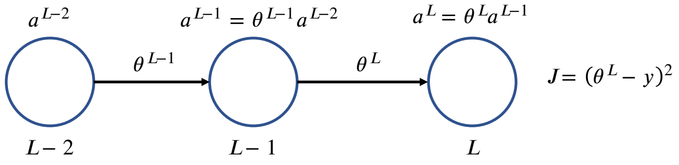


Figure 2.10: A simple network showing one output layer (L) and two hidden layers (L-1 and L-2), with their respective activation.

to the network prediction $\hat{y}$. The loss function from equation 2.2 can be written as:

$$J = (a^L - y)^2 \tag{2.5}$$

The activation in a^L is determined by the activation in the previous neuron a^{L-1} and the weight θ^L that connects them. Then a^L can be written as:

$$a^L = \theta^L a^L \tag{2.6}$$

The goal is to find the adjustment to the activation in the output a^L that will minimize the error in the loss function. However, since it is not possible to control the activation directly, it is necessary to find the adjustment to the weight θ^L that will change the activation a^L in such a manner that the loss function is minimized. This where the chain rule is applied, and can be written as:

$$\frac{\partial J}{\partial \theta^L} = \frac{\partial a^L}{\partial \theta^L} \frac{\partial J}{\partial a^L} \tag{2.7}$$

Using equation 2.5 and 2.6, equation 2.7 can be written as:

$$\frac{\partial J}{\partial \theta^L} = 2(a^L - y)a^{L-1} \tag{2.8}$$

Thus, the adjustment needed in the weight θ^L to minimize the error in the loss function can be calculated by using the activation in the last neuron a^L and and a^{L-1}. Further, the adjustment needed to a^{L-1} to minimize error in the loss function can be found by just applying the chain rule as earlier.

$$\frac{\partial J}{\partial a^{L-1}} = \frac{\partial a^{L-1}}{\partial \theta^{L-1}} \frac{\partial a^L}{\partial a^{L-1}} \frac{\partial J}{\partial a^L} \tag{2.9}$$

Thus, the adjustment needed to θ^{L-1} can be calculated:

$$\frac{\partial J}{\partial \theta_{L-1}} = 2(a^L - y)\theta^L a^{L-2} \tag{2.10}$$

This is the essence of the backpropagation algorithm. Apply the chain-rule to the calculated scalar loss, $J(\boldsymbol{\theta})$, from the output layer to calculate the gradient of the loss function $\nabla_\theta J(\boldsymbol{\theta})$ (Goodfellow et al., 2016), which is used to update an optimization function. E.g. the GD function in equation 2.3 or the momentum optimizer in equation 2.4.

2.2.4 Training, Testing and Validation Set

When the training of the ANN is finished and the parameters, the weights, are adjusted to perform well on the data it was trained on. The model has learned to map the input data to the correct targets y. The goal is to produce a model that is able to generalize well, i.e. a model that performs well on both the training data and new data it has not been previously exposed to (Chollet, 2018). This data is called the test data. In order to confirm the model's ability to generalize, it is important to never expose the model to the test set until the final evaluation of the network. It is therefore good practice to split the available data into a **training set**, a **test set** and a **validation set** prior to training. The purpose of the validation set is to keep track of the model's performance during training. The validation data is never directly used in the training of the model. However, the information provided by evaluating the model on the validation set is used to change the configuration of the model with the goal of increasing the performance on the validation set. Some examples of the configurations that can be adjusted are the number of layers in the network, the number of neurons in each layer, the learning rate or the type of optimization function. These concepts have already been outlined in the previous sections, and are often referred to as the **hyperparamters** of the model (Chollet, 2018). Since these hyperparameters are adjusted based on the performance on the validation set, the validation set is implicitly exposed to the model, which is often referred to as peeking (Russell and Norvig, 2009). Therefore, the test set, which is not used in either training or validation is required to provide an unbiased measurement of the model performance.

One method of splitting the data is the **simple hold-out validation**, where a random fraction of the data is split into the test set and withheld from training (Chollet, 2018). The remaining data is further split into the training set, which will be saved for the final evaluation and the validation set that will be used during training. An example of such a split can be seen in figure 2.11. The simple hold-out validation is often used if a lot of data

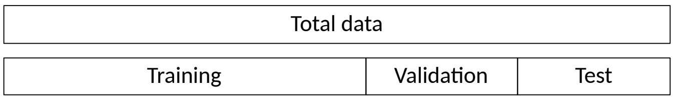


Figure 2.11: The resulting training,validation and test data using simple hold-out validation.

is available. However, in most machine learning cases, data is a limited resource. If the data set is small, such a fractional split creates a small validation and test set that may not be statistically representative of the initial data at hand. Another approach for splitting the data, that is often used when working with smaller data sets, is the **K-fold cross-validation** method. The available labelled data is split into K equal partitions, and for each partition i, train a model on the $K - 1$ remaining partitions (Chollet, 2018). The final score is the average of the K evaluations, allowing for more of the data to be used in training. Figure 2.12 illustrates an example of the K-fold cross-validation method with $K = 3$. In addition

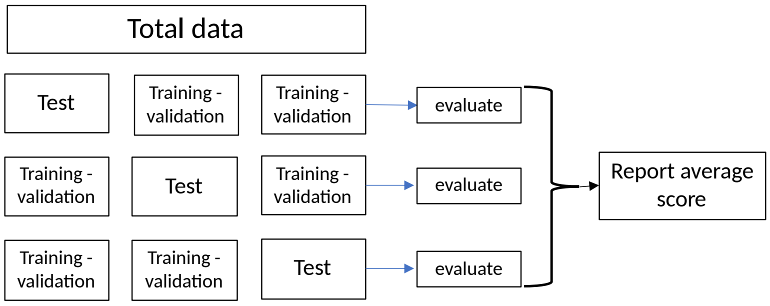


Figure 2.12: A schematic overview of K-fold cross-validation, with $K = 3$.

to increasing the amount of data that can be used in the training and validation of the model. It also reduces the risk of creating a test set that is not representative of the total data set. Consider a data set consisting of 30 images, which will be used in a classification task. Using simple hold-out validation, the data set could split into 20 images that will be used for training and validation, leaving 10 images for testing. If the 10 images in the test set, by chance, should be the easiest to classify. The model performance on the test set will be overestimated, compared to using a different split, i.e. the model performance is dependent on how the data is split. This stochastic effect of the split can be reduced by using K-fold cross-validation with $K = 3$, as seen in figure 2.12. By reserving 10 different images for testing for each run, every image can be used to evaluate the model. This will give a better estimate of the model's performance since the average of the three training-validation-testing runs is reported.

2.2.5 Evaluation Metric

There are several ways to evaluate the network performance, and this is done by choosing an appropriate **evaluation metric** to the task at hand. Some of the common evaluation metrics for classification tasks are accuracy, precision, recall and F1-score. In order to define these it is helpful to study the **confusion matrix** in table 2.1, which is a visual representation of a classification models performance (Elgendy, 2020). The accuracy, A,

	Prediction	
Ground Truth	Positive	Negative
Positive	True Positive (T_p)	False Negative (F_n)
Negative	False Positive (F_p)	True Negative (T_n)

Table 2.1: A confusion matrix showing the four possible outcomes for a prediction with respect to the ground truth.

of the model is the number of correct predictions over the total number of predictions and and can be written as:

$$A = \frac{T_p + T_n}{T_p + T_n + F_p + F_n} \tag{2.11}$$

Although, accuracy can be a useful evaluation metric, it may be quite misleading for the model's performance. Consider a data set with a large class imbalance, where the model is only able to correctly classify the majority class. This will lead to high accuracy, without being able to solve the problem. Thus, in the case of class imbalance, precision is a more suitable metric. The precision, P, is the fraction of positive predictions that are true positives, i.e. how correct are the positive predictions. Precision is given as (Elgendy, 2020):

$$P = \frac{T_p}{T_p + F_p} \tag{2.12}$$

As seen in equation 2.12, the precision metric does not account for false negatives. Thus, it is possible to achieve a high precision score, without actually identifying all the relevant ground truth positives in the data set. The model's ability to identify the relevant ground truth positives is called the recall, R, and given as (Elgendy, 2020):

$$R = \frac{T_p}{T_p + F_n} \tag{2.13}$$

Recall and precision are therefore often used in conjunction to evaluate the network performance. There is a constant trade-off between the recall and precision scores since a high recall leads to low precision and vice versa. It is important to identify whether the task requires a high recall or precision score. The importance of these concepts can be illustrated with a practical example, slightly modified after Elgendy (2020). Consider a ML model that is tasked with classifying a tumour as malignant or benign in 100 patients, where 10 are malignant. According to equation 2.12, classifying only one of the malignant tumors as malignant will yield a high precision score $P = \frac{1}{1+0} = 1$, although only being able to identify 1 out of 10. This is reflected in the recall score which will be low, since the 9 malignant tumors are wrongly classified $R = \frac{1}{1+9} = 0.1$. In this case, a higher recall score is desired. Although, this will compromise the precision score, it is better to wrongly classify a benign tumour as malignant (F_p) than the other way around (F_n). In other scenarios where the consequence of a F_n is less serious, a higher precision score may be desirable.

Although, there are some scenarios where there is a preference for either precision or recall. It is always desirable to maximize both, i.e. have both high precision and high recall scores. Therefore, both metrics are needed to fully describe the model's performance. Consider the case with two models M_1 and M_2. The precision/recall score M_1 and M_2 are 0.5/1 and 1/0.5, respectively. Without a clear preference for either precision or recall, as in the tumour example, it is hard to compare the performance of the two models. In these cases, the precision and recall score are combined into a single metric called the F1-score (Elgendy, 2020):

$$F1 = 2\frac{PR}{P + R} \tag{2.14}$$

The F1-score is a value between $0 - 1$. Using equation 2.14, model M_1 and M_2 achieves the same F1-score $F1 = 0.667$, even if they have perfect recall and precision, respectively. Since the numerator of the F1-score is the product of the two metrics, both precision and recall must be 1 in order to achieve a perfect F1-score.

2.2.6 Overfitting and Underfitting

The overall goal of machine learning is to optimize the parameters of the model on the training data in order to generalize well on the test data. By examining how the training loss and the validation loss correlate during training, the generalization power of the model can be inferred. In the early stages of training, both the training loss and validation loss will be decreasing. As long as both are decreasing, the model is said to be **underfitting** (Chollet, 2018). This implies that the model is still learning from the input and configure its parameters accordingly, i.e. both the optimization and generalization of the model is still improving. After a certain time, the validation loss will stagnate or even increase, while the training loss keeps improving. If the training loss decreases and the validation loss increase, the model is said to be **overfitting** (Chollet, 2018), as shown in figure 2.13.

That is, the model is starting to learn patterns from the training data, which are irrelevant for solving the task on the validation data (Chollet, 2018). Thus, the model optimizes its parameters on the training data, at the cost of its ability to generalize on new data. Overfitting and underfitting are undesired with respect to model performance since both

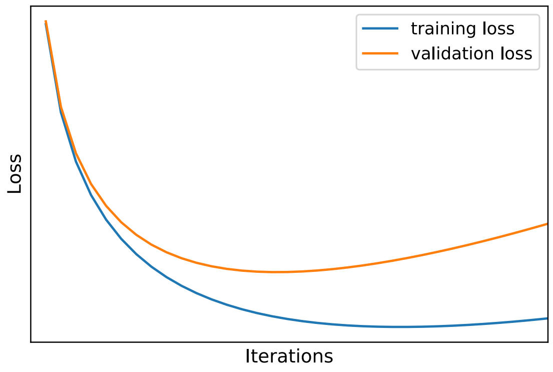


Figure 2.13: A typical sign of overfitting. Validation loss increases as training loss decreases.

lead to less accurate predictions. In the case of underfitting, where the validation loss is still decreasing, the solution is simply to continue to train the network. If the model is underfitting and the validation loss shows no further improvements the chosen model might be too simple, thus fails to capture the relevant patterns in the data. A commonly used example to illustrate the lack of complexity in the model is the fitting of a non-linear data set with a linear function. If the model is unable to learn non-linear relationships between input and output, no amount of training will increase the model's performance. In this case, the solution can be to increase the complexity of the model by adding more layers/increase layer size, i.e. increasing the number of learnable parameters (Chollet, 2018).

In the case of overfitting, the best solution is to expose the model to more training data, if available (Chollet, 2018). Otherwise, there are several techniques available to avoid overfitting, and these are often referred to as **regularization** techniques. One such technique is called **early stopping**, where the training process is terminated as soon as the model starts overfitting, as seen in figure 2.14. Other common regularization techniques include:

- Reduce network size, which in turn reduces the number of learnable parameters. This limits the model's ability to memorize a dictionary-like mapping from input to output in the training data (Chollet, 2018).

- Adding a penalty term, λ, to the cost function in equation 2.2. This penalty term scales with the complexity of the model, hence favouring less complex models with greater generalization power.

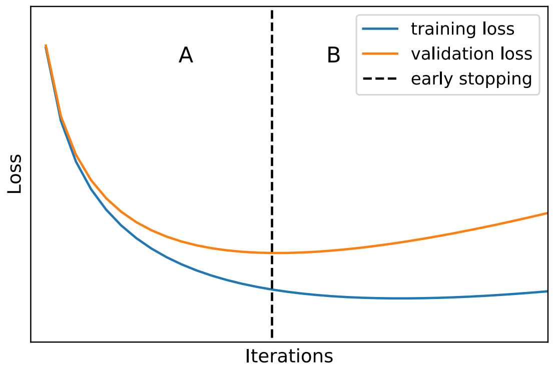


Figure 2.14: (A) The training and validation loss is still decreasing, the model is underfitting. (B) Training loss stagnates or decreases while validation loss increases, the model is overfitting. Early stopping terminates the training process as soon as the model crosses over from A to B.

- Dropping out a random fraction, usually $0.2 - 0.5$, of the output from a layer. This is done in order to break up coincidental patterns in the training data which are not significant for a general solution (Chollet, 2018).

2.3 Convolutional Neural Networks (CNN)

Convolutional neural networks (CNNs) are a type of deep neural network, which are specialized at handling data with a grid-like structure. The input data could be a 2D grid such as an image with a certain height and width or 3D grid, such as a video with the added dimension of time (Goodfellow et al., 2016). CNNs are therefore widely applied within the field of computer vision. As the name implies, the convolution operation plays a major role in how these types of networks operate. A convolutional neural network is a neural network, containing at least one convolutional layer. Each convolutional layer consists of three stages: The **convolution stage**, the **detector stage** and the **pooling stage** (Goodfellow et al., 2016), which can be seen in figure 2.15.

Each stage in the convolutional layer receives the input from the previous stage and applies an operation to said input. The first stage is the convolution stage, the input is transformed using a kernel convolution. The Detector stage adds the non-linearity, by calculating the activations using the ReLU activation function. Finally, the activations are summarized in the pooling stage, which is passed to the next convolutional layer. These three stages will be further elaborated on in the subsequent sections.

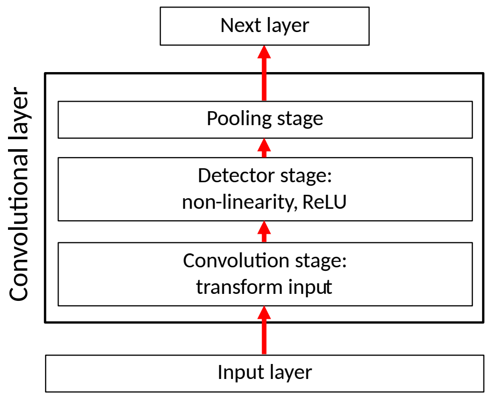


Figure 2.15: A schematic illustration of the three stages in a convolutional layer.

2.3.1 The Convolutional Operation

A convolution is a mathematical operation on two real-valued and continuous functions, f and g. Its mathematical definition is given as:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau = \int_{-\infty}^{\infty} g(\tau)f(t-\tau)d\tau \tag{2.15}$$

When working with CNNs, equation 2.15 needs to be rewritten to its discrete form:

$$(f * g)(t) = \sum_{a=-\infty}^{\infty} f(a)g(t-a) \tag{2.16}$$

where $f(t)$ is the input, $g(t)$ is the **kernel** and $(f * g)(t)$ is the output, often referred to as the **feature map** (Goodfellow et al., 2016). In practice, both the input and the kernel are usually, finite, multidimensional arrays and are often referred to as tensors. Consider a finite 2D image tensor I, with $m \times n$ pixels and a kernel K, then equation 2.16 can be rewritten as a sum over a finite number of array elements (Goodfellow et al., 2016):

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{2.17}$$

where $S(i,j)$ denotes the output from the convolution between image $I(m,n)$ and the kernel K. The (i,j) term shows the positions within the $m \times n$ image the convolution is preformed.

This process is illustrated in figure 2.16, where the input image is convolved with a kernel of width 3 centred at position f and g. The dashed red line shows the initial position of the kernel centred at position f in the input. Convolving the kernel with the input at this image location will output the weighted sum of neighbouring pixels. The number of pixels included and the weights are determined by the size and the values of the kernel, respectively. For the 3×3 kernel to fit within the 4×4 image, it can only be centred at the pixel locations f, g, j or k. Thus, the number of possible outputs from this

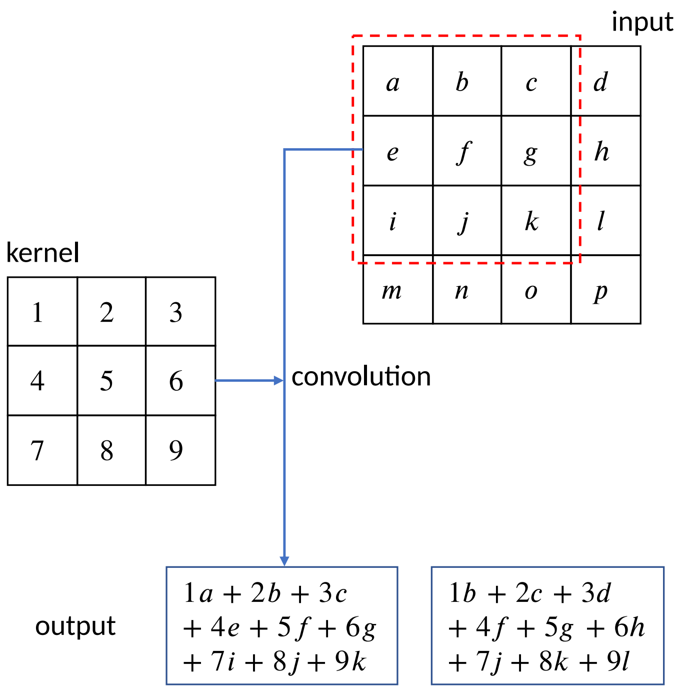

input
a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
kernel
1
2
3
4
5
6
7
8
9
convolution
output
1a + 2b + 3c + 4e + 5f + 6g + 7i + 8j + 9k
1b + 2c + 3d + 4f + 5g + 6h + 7j + 8k + 9l

Figure 2.16: An example of a two-dimensional convolution, showing the outputs when the kernel is centred at f and g. The dashed red square shows the extent of the kernel centred at f.

convolution is four, i.e. the 16 pixels in the input are reduced to 4 values in the output. This causes a reduction of 2 in both dimension, and is referred to as the **border effect** of convolution (Chollet, 2018). Hence, each convolution applied will reduce the dimensions of the feature. There are methods that can be applied to either increase or reduce this loss of dimensionality when performing the convolutional operation and are called **stride** and **padding**, respectively.

The stride determines the step size between each time the convolution is performed. If the convolution should be performed at every possible pixel location f, g, j and k in figure 2.16 the stride is said to be $s = 1 \times 1$, i.e. the spacing in spatial location, both in the horizontal and vertical direction of the image, where the convolution operation is performed is one. As the stride increases, the distance between the locations where the convolution is performed increases. This, in turn, reduces the number of outputs and increases the reduction in dimensionality.

Padding, on the other hand, is a technique used to preserve the dimensionality of the input. This is done by padding the border of the input with zeros, allowing for the convolution operation to be performed at every pixel location of the input. Padding the 4×4 input in figure 2.16 will result in a 5×5 image, which allows the kernel to be placed in every pixel location, e.g. a, which would result in the following output: $5a + 6b + 8e + 9f$.

Each layer in the network may apply several convolutions in parallel to the same input by using different kernels, which will result in different outputs. This can be seen as applying different filters or transformations to the same input to highlight different features in the input. For example, convolving the input with one kernel can result in the horizontal edges in the image are highlighted, while another kernel highlights the vertical edges. One of the most important aspects of using convolutions in neural networks are the concepts of **parameter sharing** and **sparse connectivity** (Goodfellow et al., 2016). These concepts are illustrated with a one-dimensional example in figure 2.17, with one input layer **x**, one output layer **s** and a kernel with a width of 3. When applying the convolution with the kernel centred at x_2 (kernel position 1), the output in s_2 will be the weighted sum of the inputs covered by the kernel. In this case s_2 is the weighted sum of x_1, x_2 and x_3, resulting in only three ingoing edges from the input to s_2. Similarly, several outputs are affected by the same input values. This can seen in figure 2.17, where x_3 affects s_2, s_3 and s_4 from the convolution performed at kernel positions $1, 2$ and 3, respectively. This gives rise to the name sparse connectivity and is the result of using a kernel that is smaller than the input (Goodfellow et al., 2016). Contrasting this to the fully connected feed-forward neural network in section 2.2.1, where every output has an ingoing edge from every input. The benefit of sparse connectivity is that the input can be represented with significantly fewer parameters, which both reduces the memory requirements of the model and improves its statistical efficiency (Goodfellow et al., 2016), compared to FNNs. This reduction in memory requirement and increase in efficiency by reducing the number of parameters are especially useful when working with images, which can contain several thousands of pixels.

Parameter sharing refers to using the same parameter for more than one function in a model (Goodfellow et al., 2016). Since the parameters (weights) in a CNN is defined by the values of the kernel, and the kernel is used at every position of the input (if padding is used). A change in e.g. the centre value of the kernel will affect every output. This can be illustrated with figure 2.16, where a change in the middle value of the kernel will affect the weighted sum of both outputs. This allows for the learning of only one set of parameters (Goodfellow et al., 2016), i.e. the values of the kernel. Comparing this to a FNN, where every input node has its own set of parameters, i.e. the weights of the connections from the input node to every output node in the subsequent layer. This requires that a set of parameters (weights) needs to be learned for every input (Goodfellow et al., 2016). An additional property of parameter sharing is that it makes the convolutional layer equivariant to translation, i.e. if the input changes, the output changes in the same way (Goodfellow et al., 2016). The mathematical definition of equivariance is given in equation 2.18 (Goodfellow et al., 2016):

$$f(g(x)) = g(f(x)) \tag{2.18}$$

where $f(x)$ and $g(x)$ are equivariant functions. In the context of convolution, consider an input image $I(x, y)$ where the pixels are moved a couple of units to the right using some sort of transformation function. The result after the transformation is $I'(x, y) = I(x + k, y)$, where k describes the amount of movement. Since the convolution is equivariant to translation, applying the convolution to I then transforming the output from the convolution will give the same result as applying the convolution directly to $I'(x, y)$ (Goodfellow et al., 2016). This is an important aspect of the convolution operation, which allows for the learning of local patterns in the input. This implies that the spatial location of a certain feature in the image does not affect the networks ability to detect it. For example, some local patterns in an input image, such as edges (the contrast between dark and bright pixel values), will be present almost everywhere in the image (Goodfellow et al., 2016). Therefore, it is desirable to apply the same local transformation that detects edges across the whole input, which is made possible by parameter sharing. Contrast this to densely connected layers (FNN), which are not equivariant to translation, where each pattern learned involves the whole feature space, limiting them to only learn global patterns (Chollet, 2018).

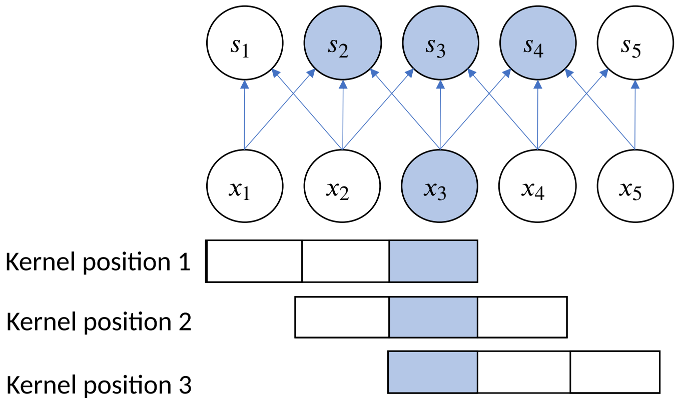


Figure 2.17: One-dimensional example illustrating the sparse connectivity of convolutional neural networks, as a result of using a smaller kernel than the input. The input neuron and the output neurons affected by it are highlighted in blue. The kernel positions illustrates the position of kernel when x_3 is included the weighted sum of s_2, s_3 and s_4.

The next stage in the convolutional layer is the detector stage. Similar to the traditional neural networks (e.g. FNNs), the output from the convolution is a linear weighted sum of the input and the kernel. In order to introduce non-linearity, the activations are calculated by passing the values in each output node through an activation function, e.g. the ReLU defined in equation 2.1. Once the activations have been calculated, they are passed as input to the final stage in the convolutional layer, the pooling stage.

2.3.2 Pooling

The pooling stage is the final stage of a convolutional layer, where a pooling function is applied. A pooling function is fed the activations from the detection stage at a certain point in the network and replaces it with a summary statistic of the nearby outputs (Goodfellow et al., 2016). There are several types of pooling functions, with the most common being the **max pooling** function. In max pooling, the activations from the detector stage are summarized by the maximum value in a rectangular neighbourhood (Goodfellow et al., 2016). In all cases the pooling helps the to make the representation approximately **translation invariant**, this means that if the input is translated by a small amount, most pooled outputs will stay the same (Goodfellow et al., 2016). An important property of translation invariance is that the will be less concerned with the specific pixel location of a certain feature in the image. Since the location of a feature that describes the same thing may vary between inputs. For example for the task of determining if an image contains a face, it is more important to know that there is an eye on the left side of the face, rather than the exact pixel location of said eye (Goodfellow et al., 2016). This property of max pooling is illustrated in the one-dimensional example in figure 2.18, where a max pooling kernel of width 3 is run over two similar activations from the detector stage. The activations in the right figure are shifted one step to the right. Although, all values in the input are changed due to the shift, only two values changes after max pooling (Goodfellow et al., 2016). An

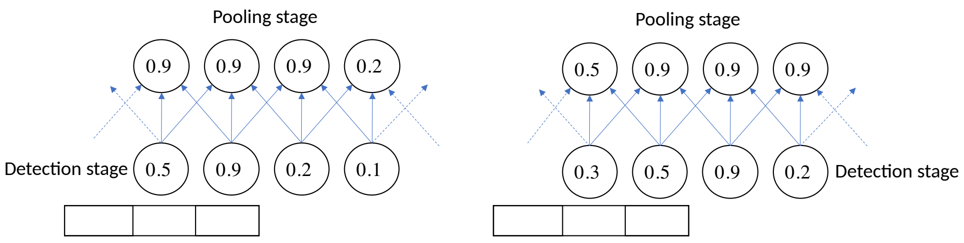


Figure 2.18: The translation invariance from pooling. The bottom row shows the activations from the ReLU in the detection stage. *(left)* The top row shows the output from max pooling with a kernel of width three and stride 1. *(right)* The result of max pooling when the input is shifted to the right by one.

additional property of the pooling operation is the downsampling of the input. Similar to the convolution operation, this downsampling is determined by the size and stride of the kernel used in the pooling. Since the number of parameters in a layer is a function of the input size, pooling will also result in improved statistical efficiency and reduced memory requirements for storing the parameters (Goodfellow et al., 2016).

2.3.3 CNN architecture

As outlined in the introduction to this section, the convolutional layer consists of the three stages: minimum one convolution stage, a detector stage and a pooling stage. A CNN usually consists of several convolutional layers with a flattened layer which is fully connected to the final output. Figure 2.19 shows a simple CNN with two convolutional layers, containing one convolution stage and one pooling stage each. The ability to detect local, translation invariant patterns allows CNNs to learn **spatial hierarchies** of the input data (Chollet, 2018). For example, the first convolutional layer in the network may learn small local features such as edges. The second layer will learn larger patterns based on the small ones found in the first layer and so on. Hence, the network will learn increasingly more complex and abstract concepts with increasing network depth (Chollet, 2018).

2.3.4 Data Augmentation

One of the main challenges with deep learning algorithms such as CNNs, is that they require a lot of training data in order to perform well. They are said to be data hungry (Elgendy, 2020). This is due to the inherent complexity of the problem space they often

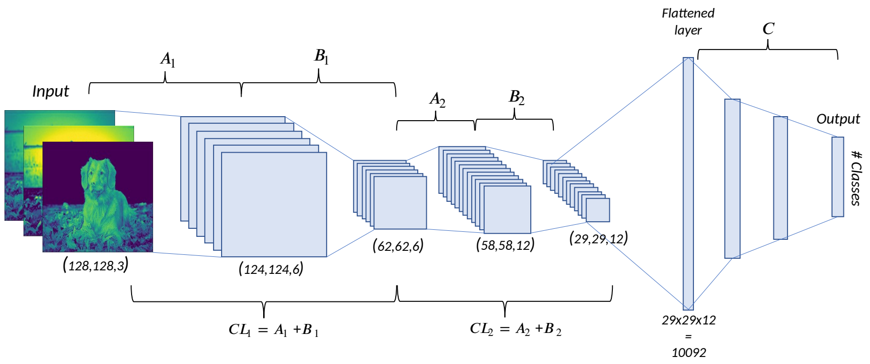


Figure 2.19: The architectural elements of a CNN. A - Convolution 5x5 kernel + Detection with activation function. B - Pooling 2x2 kernel. C-fully connected layers between flattened layer and the output. CL_x - Convolutional layer. The photography used in this and subsequent figures is licensed under the Unsplash license (Co, 2018).

are applied to. Training data often is a limited resource and the task of acquiring and labelling data is both tedious and time-consuming. **Data augmentation** is an approach for generating more training data from the existing data set, by augmenting the images with a certain number of random transformations (Chollet, 2018). Figure 2.20 shows the result of applying rotation, zooming and lateral translation at four various degrees to the original image.

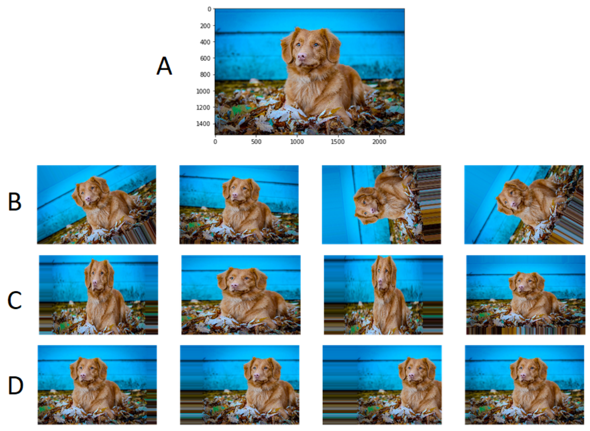


Figure 2.20: The effect of three different data augmentation techniques at 4 random degrees. A - original image, B - rotation, C - zooming and D - lateral translation.

2.3.5 Transfer Learning and Fine-Tuning

The main idea behind **transfer learning** is that a network that has learned to solve a problem in a certain domain can be transferred and applied to a different, but related problem

(Elgendy, 2020). Since training a network with good performance from scratch requires large amounts of data, transfer learning may be a highly effective approach if there is some fundamental similarity between the two problems. A common transfer learning technique is **fine-tuning**, where only the last layers of the convolutional network are retrained. Consider a network with good performance and that has been trained on a large data set such as ImageNet, a database with $1,4$ million labelled images in 1000 different categories (Chollet, 2018). The idea is that a network that has been trained on such a large and diverse data set has learned to extract general features such as edges, curves and circles in its early layers. Since these types of features are general, they should to some extent be applied to any image classification task. Due to the hierarchical structure of CNNs, the last layers are specialized to capture abstract concepts, which are highly task specific (Chollet, 2018). By retraining the last layers of the network, they can be used to capture new abstract representations that are specific to the task at hand, while preserving the general layers early in the network.

2.4 CNN for Object Detection

Apart from image classification, one of the major sub-fields of computer vision is object detection. In contrast to image classification, the task of object detection involves both localizing and classifying an unknown number of objects of varying size within an image. Another difference between classification and object detection is the labelled input data used in training. The input data for object detection tasks are usually images with labelled bounding boxes around the task objects. The bounding boxes are usually rectangles and defined by the x and y coordinates for the upper-left and lower-right corner of the box. **Region-based convolutional neural networks** (R-CNN) is a family of algorithms that have been developed to solve object detection tasks, and includes R-CNN, Fast R-CNN and Faster R-CNN (Elgendy, 2020). Additionally, there are other object detection models that are commonly used, such as YOLO and SSD. However, these are considered outside the scope of this text and will not be discussed any further. For more details regarding YOLO and SSD object detection models, the reader is encouraged to consult Redmon et al. (2015) and Liu et al. (2015), respectively. Thus, the following sections will outline the basics of R-CNN based methods and some fundamental metrics commonly used in object detection tasks.

2.4.1 Evaluation Metric for Object Detection

The goal of an object detection model is to be able to both classify and localize the object within an image. The model takes an image with ground truth bounding boxes and their associated labels as input. Then outputs a classification and spatial locations of the targets defined by bounding boxes, as seen in figure 2.21. Thus, the model needs to optimize a loss function that combines the localization and classification losses. In order to choose a evaluation metric for the model, two fundamental concepts are defined: the **confidence score** and the **intersection over union** (IoU). The confidence score also referred to the objectness loss, is a probability score in the range $[0, 1]$ describes how certain the network is that this bounding box contains an object (Elgendy, 2020). The IoU is used to determine

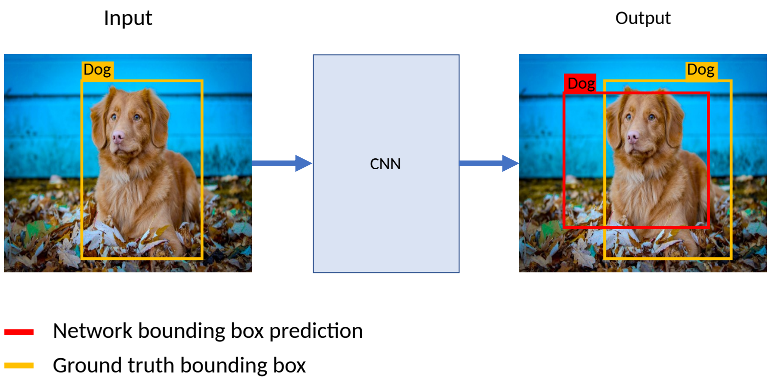


Figure 2.21: Illustration of the object detection task. Input - labelled ground truth bounding box (*yellow*). Output - A prediction for the class and the location of the object in the image, marked by a bounding box (*red*).

how well the network predicts the bounding boxes around the targets, with respect to the ground truth. The IoU is a score in the range $[0, 1]$ that expresses the amount of overlap between ground truth and prediction. This can be seen in figure 2.22 using the bounding boxes from figure 2.21. The mathematical expression for IoU is given as (Elgendy, 2020):

$$IoU(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{2.19}$$

Both the confidence score and the IoU is usually set at some threshold $t_{C/IoU} \in [0, 1]$,

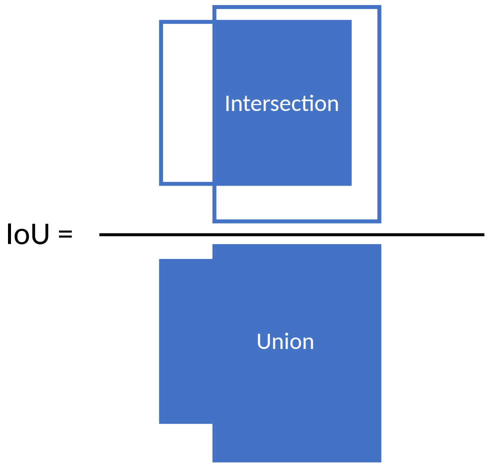


Figure 2.22: The definition of IoU

which determines what the network defines as a true positive. Predictions with a confidence score lower than t_C are rejected as false positives. The remaining predictions with an IoU higher than t_{IoU}, are kept as true positives, whilst the rest are rejected as false positives. The $t_{C/IoU}$ specifies the strictness of the policy enforced by the model for defining a true positive. The IoU and confidence score are the foundation for most evaluation metrics used for R-CNN models.

A common way of evaluating an object detection model is by studying the precision-recall curve where the precision and recall scores are calculated, as outlined in section 2.2.5, and plotted for each detection. The model is said to perform well if the precision value stays high as the recall increases. Poor object detection models will need to detect more false positives in order to retrieve all the true positives in the data set. This is shown in figure 2.23, where the good model (blue) has detected fewer false positives when all the true positives in the data set have been retrieved. Another popular evaluation metrics

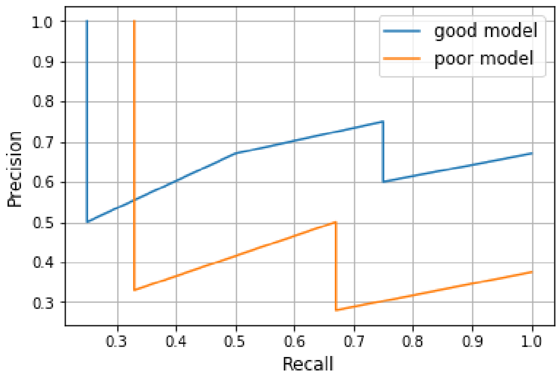


Figure 2.23: The precision-recall curve for two models. The good model (*blue*) detects fewer false positives and ends up with a higher precision than the poor model (*orange*) when all true positives have been retrieved.

for object detection tasks is the **average precision** (AP). The AP is often preferred to the precision-recall curve, since this a numeric value that describes the model performance rather than a graphical representation. The AP is calculated by first interpolating the currently highest precision value backwards until a higher precision value is encountered as seen in figure 2.24. Using this interpolated precision, P_{int} the AP can be calculated as follows:

$$AP = \sum_{i}^{n-1} (r_{i+1} - r_i) P_{int_i} \tag{2.20}$$

where $r_1, r_2, ..., r_n$ denotes n equally spaced recall levels. The AP is commonly calculated over 11 recall levels, $n = 11$. This evaluation metric is also referred to as 11-point recall (Zhang and Zhang, 2009).

Finally, when dealing with several classes in an object detection task the **mean average precision** (mAP) is the preferred evaluation metric. The mAP is simply the AP averaged over the number of classes in the data set and is given by:

$$mAP = \frac{\sum_i^K AP_i}{K} \tag{2.21}$$

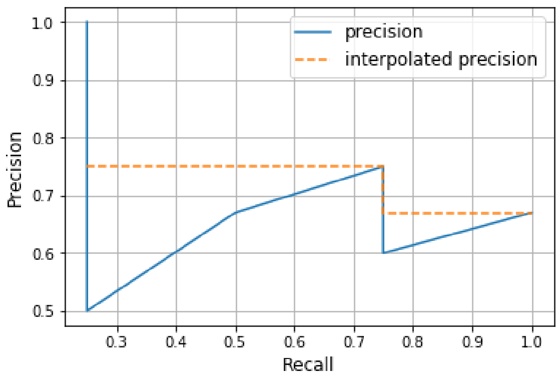


Figure 2.24: The calculated precision-recall curve and the interpolated precision using the current max precision value. The interpolation starts at the end (recall = 1) and interpolates the currently largest encountered precision value backwards.

where K is the number of classes in the data set and AP, given by equation 2.20, is the average precision for class i (Elgendy, 2020).

2.4.2 R-CNN

Consider a naive approach to solve the object detection task using a conventional CNN. In this case, several regions could be extracted from the input image and classified individually. However, since the objects in the input image may appear in varying sizes and spatial locations, a huge amount of regions needs to be extracted, rendering such a method computationally intractable. The **region convolutional neural network** (R-CNN) proposed by Girshick et al. (2014) provides a solution to this problem, by proposing a limited amount of regions, referred to as proposal, the CNN needs to classify. This is achieved by using a fixed search algorithm, such as selective search (Uijlings et al., 2013), to extract around 2000 proposals from the input. Each proposal is warped to the networks required input size and passed through the CNN. The output is used to train a linear support vector machine (SVM) classifier for each class, where each SVM is used to determine which class a proposal belongs to. Finally, the features and labelled bounding box of each proposed region are combined to train a linear regression model for improving the localization accuracy for bounding box predictions (Girshick et al., 2014). Although R-CNN achieves excellent object detection accuracy (Girshick, 2015), it suffers from several major drawbacks:

- Slow speed with respect to both training and prediction: R-CNN requires a forward pass through the network for each proposals. Around 2000 proposals per image.
- Overlapping proposals: Causes a high volume of repetitive computations.
- There is no learning involved when extracting the proposals, since selective search is a fixed algorithm.

2.4.3 Fast R-CNN

These issues are addressed in Girshick (2015) with the use of the **Fast R-CNN** algorithm. The input for the Fast R-CNN is an image and several region proposals of various sizes from a fixed search algorithm. The whole image is passed through the CNN as input to calculate the feature map and the region proposals are transformed to that feature map. Then for each proposal a **region of interest** (RoI) pooling layer extracts a fixed-sized feature map, which is mapped through two fully connected layers to a feature vector. This feature vector is fed into two sibling layers, one outputs the prediction for the classes using a **softmax** classifier and one outputs the predictions for the coordinate of the bounding box using bounding box regression (Girshick, 2015). A softmax classifier uses the activations to calculate the probabilities of the categorical distribution of the classes in the output (Goodfellow et al., 2016):

$$softmax(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_{j=1}^{K} \exp(z_j)} \tag{2.22}$$

where **z** are the numerical activation values in the fully connected layer associated with the softmax output.

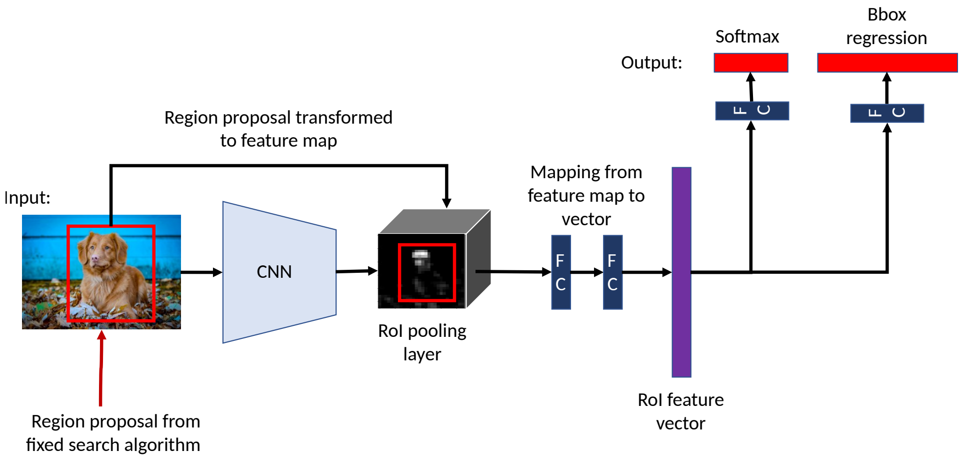

Softmax
Bbox regression
Output:
F C
F C
Region proposal transformed to feature map
Mapping from feature map to vector
Input:
CNN
F C
F C
RoI pooling layer
RoI feature vector
Region proposal from fixed search algorithm

Figure 2.25: The architecture of the Fast R-CNN model. FC - fully connected layers.

Figure 2.25 shows the architecture of the Fast R-CNN as described in (Girshick, 2015). The RoI pooling is a necessary step since the fully connected layers downstream expect input of a certain size (Elgendy, 2020). The RoI pooling is essentially a max pooling operation performed on the region of interest in order to reshape them into the correct size. Consider the RoI pooling operation in figure 2.26, with an input RGB-image, an output feature map of and a region proposal with the respective dimensions: (H_I, W_I, 3), (H_f, W_f, d) and $h \times w$. The input image is passed through the CNN, which generates an output feature map. The region proposal is then transformed from the input to the feature map. In this case, one pixel in the height and width of the feature map corresponds to H_I/H_f and W_I/W_f pixels on the input image. The transformation is parameterized as

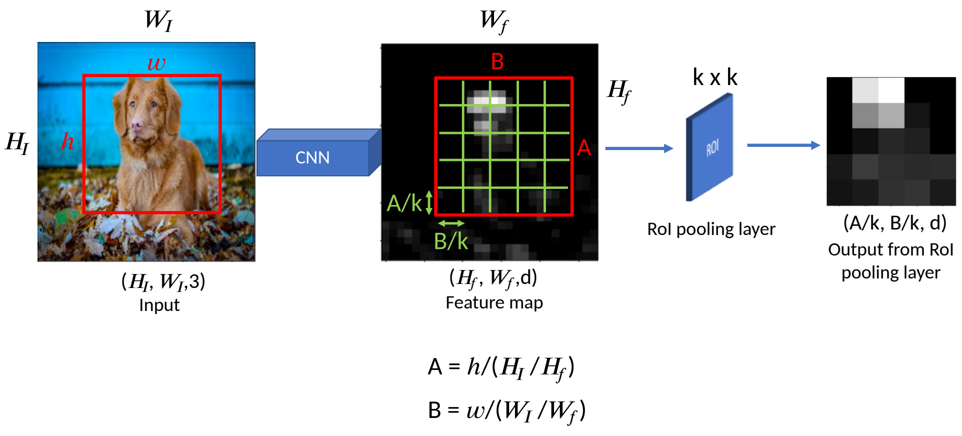


Figure 2.26: Schematic illustration of how the RoI pooling operation. The input image, feature map and region proposal with their respective dimensions (H_I, W_I, 3), (H_f, W_f, d) and $h \times w$. The transformed region proposal is parameterized by A and B, which is in turn is divided into $A/k \times B/k$ according to the fixed $k \times k$ kernel and max pooled to produce an output of the desired size.

$A = h/(H_I/H_f)$ and $B = w/(W_I/W_f)$ for the height and width of the region proposal. The transformed region proposal is divided into $A/k \times B/k$ regions, where k denotes the size of the kernel. The kernel is of a fixed size, $k \times k$, in order to produce an output with the correct dimensions, expected by the fully connected layers downstream. Finally, max pooling is performed on each of the regions resulting in an output with the desired dimensions of (A/k, B/k, d), which is mapped to the feature vector.

The loss function in Fast R-CNN is a multi-task loss, and relates to the prediction from the softmax classifier and the bounding box regression in figure 2.25. The softmax classifier outputs the normalized probability $p = p(p_0, ..., p_K)$ over the $K + 1$ classes, where the p_0 is used to classy the background. The bounding box regression outputs the bounding box prediction for each class $t^k = [t^k_x, t^k_y, t^k_w, t^k_h]$, where x, y, w, h represents the centre coordinates and width and height of the bounding box (Girshick, 2015).

The multi-task loss function for Fast R-CNN given a RoI with a ground truth class g and ground truth bounding box r can be written as follows (Girshick, 2015):

$$L(p, g, t^g, r) = L_{cls}(p, g) + \lambda[g \geq 1]L_{loc}(t^g, r) \tag{2.23}$$

Here, $L_{cls}(p, g) = -log(p_g)$ denotes the classification loss, with p_g denoting the probability of class g. The negative logarithm results in smaller loss as the probability p_g increases. The localization loss, $L_{loc}(t^g, r)$, is given in equation 2.24.

$$L_{loc}(t^g, r) = \sum_{i \in x,y,w,h} smooth_{L1}(t^g_i - r_i) \tag{2.24}$$

where

$$Smooth_{L1}(x) = \begin{cases} 0.5x^2 & \text{if } |x| < 1 \\ |x| - 0.5 & \text{otherwise} \end{cases} \tag{2.25}$$

The $Smooth_{L1}$ loss is quadratic for small values and linear for large values, making the loss function less sensitive to large outliers values (Girshick, 2015). The $\lambda[g \geq 1]$ term in equation 2.23 is a regularization term. The hyperparameter λ determines balance between the L_{loc} and L_{cls}. This is usually set to $\lambda = 1$ (Girshick, 2015). Finally $[g \geq 1]$ expresses that L_{loc} is not defined for the background class. Using this multi-task loss function the Fast R-CNN model is trained using SGD and back-propagation Girshick (2015).

2.4.4 Faster R-CNN

Although, Fast R-CNN significantly improves on the simple R-CNN model, the main bottleneck is still the test-time speed. This is attributed to the use of a fixed algorithm for region proposals, such as selective search (Ren et al., 2017). This issue is addressed by Ren et al. (2017) with the Faster R-CNN model. The Faster R-CNN model consists of two modules: A **region proposal network** (RPN) used to generate region proposals, called **anchors**, and a Fast R-CNN classifier as outlined in the previous section (Ren et al., 2017). The task of the RPN is to generate good proposals for the location of the objects in the image, which will be classified by the Fast-RCNN. The input to these modules comes from a shared convolutional base that is used for feature extraction on the original input image. Figure 2.27 shows the general architecture of the model.

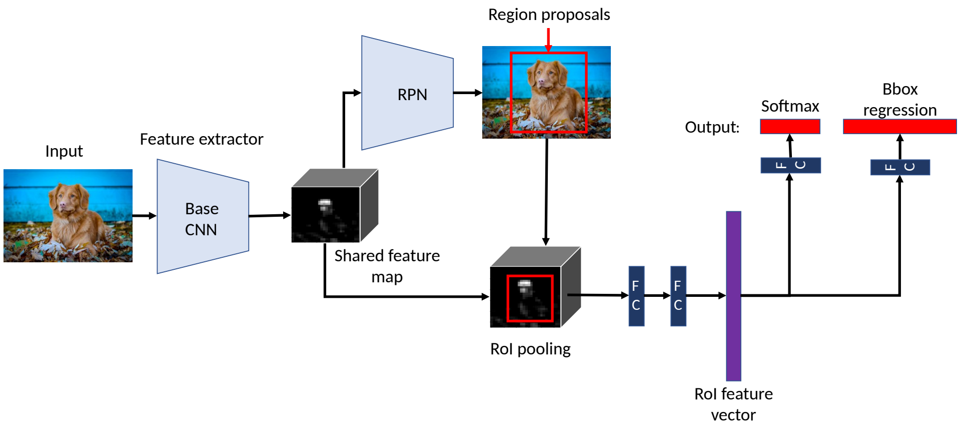


Figure 2.27: The architecture of the Faster R-CNN object detection model. Consisting of a region proposal network (RPN) and a Fast R-CNN classifier. FC - fully connected layers.

The RPN generates the anchors by taking in the feature map output from the last shared convolutional layer in the convolutional base. This feature map is passed through a small CNN by sliding a $n \times n$ window over the feature map to generate the input to the network (Ren et al., 2017). Each $n \times n$ window is mapped to a lower-dimensional feature map and the network proposes a maximum of k anchors for each input. Finally, the lower-dimensional feature map is passed to two separate fully connected layers. One box-regression layer with $4k$ outputs for the coordinates of the k anchors and a box-classification layer with $2k$ outputs determining whether or not the k anchor contains an object (Ren et al., 2017). Figure 2.28 shows the first $n \times n$ input to the RPN from the

$H \times W$ output from the base convolutional network. The RPN proposed in Ren et al.

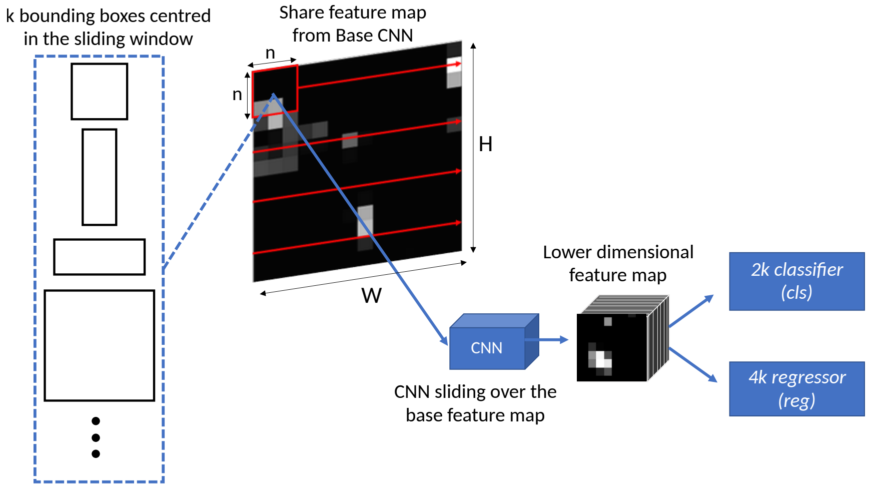

k bounding boxes centred in the sliding window
Share feature map from Base CNN
n
n
H
W
Lower dimensional feature map
2k classifier (cls)
4k regressor (reg)
CNN
CNN sliding over the base feature map

Figure 2.28: The architecture of the RPN illustrating how the anchors are generated.

(2017) uses anchors of 3 different scales and aspect ratios (1:1/2:1/1:2) yielding $k = 9$ anchors for each position of the sliding window. Hence, using the $H \times W$ feature map in figure 2.28 will yield $H \times W \times k$ anchors in total.

When training the RPN, each proposed anchor is assigned a binary class label based on containing an object or not. The label for each anchor proposal is determined by an IoU condition from equation 2.19 using the anchor A and the ground truth bounding box G_t:

$$IoU(A, G_t) = \frac{|A \cap G_t|}{|A \cup G_t|} \begin{cases} 0.7 & \text{Positive label / Object} \\ < 0.3 & \text{Negative label / Not object} \end{cases}$$

These proposals are used to optimize a loss function. The loss function that needs to be optimized is a multi-task loss function based on the output from the box-classification layer (cls) and the bounding box-regression layer (reg) given as (Ren et al., 2017):

$$L(p_i, t_i) = \frac{1}{N_{cls}} \sum_i L_{cls}(p_i, p_i^*) + \lambda \frac{1}{N_{reg}} \sum_i p_i^* L_{reg}(t_i, t_i^*) \tag{2.26}$$

where i is the index of a bounding box, p_i is the predicted probability of bounding box i containing an object and the ground truth label $p_i^* = 1$ if bounding box i is labelled positive. Otherwise, $p_i^* = 0$.

The parameterized coordinates for the predicted bounding box i are denoted as t_i, and t_i^* is the ground truth coordinates associated with t_i (Ren et al., 2017). The classification loss, L_{cls}, is the logarithmic classification loss over the two cases: object vs not object. The regression loss $L_{reg}(t_i, t_i^*) = Smooth_{L1}(t_i - t_i^*)$ is given in equation 2.25. The regression loss $L_{reg}(t_i, t_i^*)$ is only activated if the bounding box contains an object, as can be seen from the term $p_i^* L_{reg}(t_i, t_i^*)$ in equation 2.26. The bounding box regression adopts the follow parametrizations for the predicted bounding box $t = [t_x, t_y, t_w, t_h]$ and the ground truth bounding box $t^* = [t^*_x, t^*_y, t^*_w, t^*_h]$ (Ren et al., 2017):

$$\begin{aligned} t_x = (x - x_a)/w_a, t_y = (y - y_a)/h_a, t_w = log(w/w_a), t_h = log(h/h_a) \\ t^*_x = (x^* - x_a), t^*_y = (y^* - y_a)/h_a, t^*_w = log(w^*/w_a), t^*_h = log(h^*/h_a) \end{aligned} \tag{2.27}$$

where x, y, w, h denotes the centre, height and width of the bounding box. the x, x_a and x^* corresponds to the x-values for the centre of the predicted bounding box, anchor and ground truth box, respectively. The y, w and h values for the bounding box, anchor and ground truth adopts the same notation as the x values.

Finally the terms N_{cls} and N_{reg} and λ are the regularization terms, where N_{cls} is the number of training samples used for computing the loss function. $N_{reg} = H \times W \times k$, which are the number of anchor locations and λ is a constant regularization term (Ren et al., 2017).

Since both the RPN and the Fast-RCNN classifier share convolutional layers in the Faster R-CNN model, as seen in figure 2.27. In order to work as a unified network they must configure these convolutional layers in a similar manner (Ren et al., 2017). Both the RPN and Fast R-CNN, trained independently, will modify their convolutional layers in different ways (Ren et al., 2017). To avoid this, the method used to train the Faster R-CNN model outlined in Ren et al. (2017) is as follows: Use a pre-trained CNN base network to train the RPN and Fast-RCNN independently. Use the RPN to generate region proposals to fine-tune the Faster R-CNN. At this the two networks do not share convolutional layers. Use the now fine-tuned Faster R-CNN detection model to initialize the RPN training, but only fine-tuning the layers unique to the RPN. This results in a unified model where the RPN and Fast-RCNN share convolutional layers.

2.5 Brief Summary of Specialization Project

This section will give a brief summary of the work conducted in the specialization project. Since some of the same machine learning concepts that will be used in this thesis, were also used in the specialization project, this summary will also act as a practical example for some of the previously outlined machine learning concepts.

The work in the specialization project conducted in the fall of 2019 consisted of training an object detection model for detecting and extracting the core segments from the optical core images. The model was trained by fine-tuning a pre-trained Faster R-CNN using 80 randomly sampled core images from NPD, which were manually labelled using a software. In the labelling process, the bounding boxes were drawn in such a manner that the maximum number of pixels that constituted the core was included. The data was split using a simple hold-out split, where 10% of the images were split into the validation set and the remaining images were used for training. Finally, the model was tested on a new data set consisting of core images, randomly sampled from wells that were not represented in the training or validation set.

The resulting object detection model proved to be able to achieve both high precision and recall scores on the test set, and was able to accurately locate the cores in the optical core images, i.e. high IoU with the ground truth. One of the main findings was that the performance of the model with respect to the metrics used to evaluate the model could be significantly improved by increasing the weight of the localization loss, λ, in the Fast R-CNN classifier from equation 2.23. Figure 2.29 shows the predicted bounding boxes from the model on an optical core image.

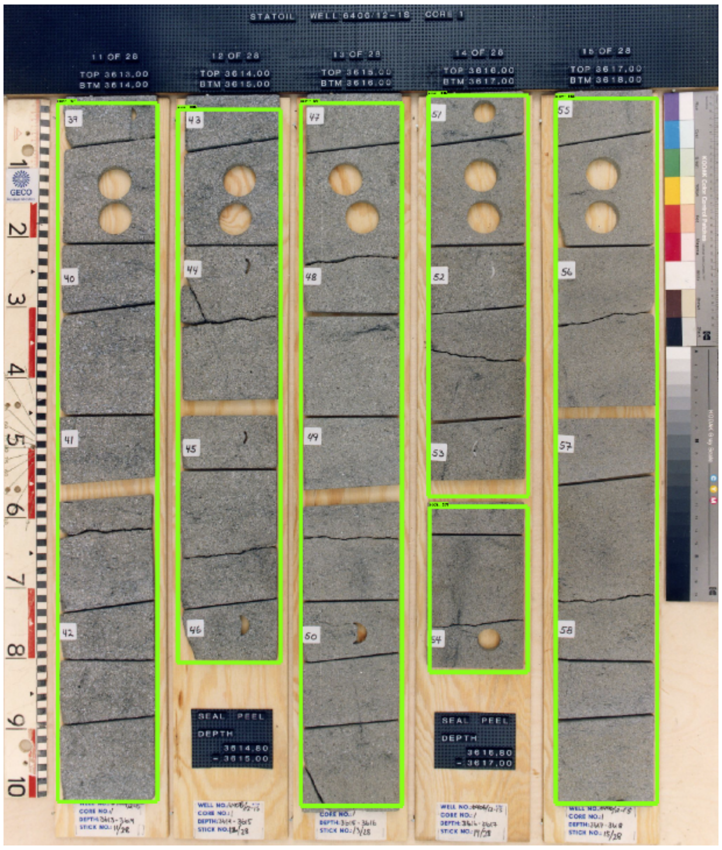


Figure 2.29: Predicted bounding boxes from the model developed in the specialization project, showing a high IoU with the ground truth. Original image (NPD).

Although, the model's ability to locate the cores is not perfect, which can be seen near the top of some of the cores in figure 2.29. In most cases, this problem could be remedied by using a naive post-processing approach, where the tops of the bounding boxes are shifted up to the bounding box with lowest y-coordinate that defines the top of the bounding box. Furthermore, in cases where the cores are defined by multiple bounding boxes, as seen in the fourth core from the left in figure 2.29, were merged into a single bounding box. In addition to automatic cropping, the model predictions can also be used for automatic label generation, which can be inspected and adjusted in the labelling software that was used and be used in further training of the model. Both the automatic labelling and cropping scripts can be found in the `GitHub` repository of the specialization project (Adelved, 2019).

Chapter 3

Methodology

The following sections will outline the methodology of fine-tuning a Faster R-CNN, with the goal of detecting and classify horizontal and vertical CCA plugs, as well as non-CCA plugs. Additionally, the background and implementation for several image preprocessing techniques applied to the data prior to training will be outlined. The methodology of this thesis can be summarized as a set of steps, which are illustrated in figure 3.1. The first steps

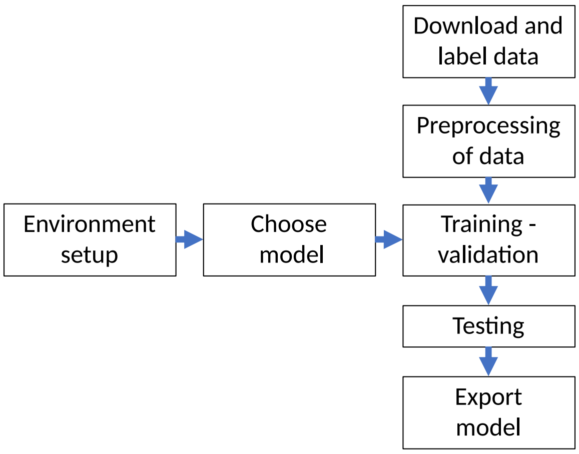


Figure 3.1: Overview of the main steps performed in the methodology of this thesis.

include setting up the environment containing the necessary structure to train the model and choosing an object detection model to train. Secondly, the data used in the training needs to be downloaded and labelled, before applying any preprocessing technique to said data. When these steps are in place, the model can be trained on the labelled data set. The final steps include testing the model on a previously unseen data set, and export the model so that it can be used outside the training environment set up at the start of the methodology.

To present these steps in a structured manner, the files and folders mentioned in this section and the data used in this thesis will be uploaded to a `GitHub` repository (Adelved, 2020), which can be consulted for more details. When referring to a file or a folder that can be found in this repository, they will be styled in *italic*. However, larger files will be stored in a `Google Cloud Storage Bucket`, which can be accessed through links in the `GitHub` repository. Further, any software, general file description, python library or GitHub repository will be styled using `texttt`. The environment setup has been conducted according to the official `Tensorflow` object detection application programming interface (API) documentation (Tensorflow, 2020c). Here a step by step guide for setting up the environment can be found, which can be consulted for further detail regarding the environment set up.

3.1 Environment Setup

In order to fine-tune an object detection model, it is necessary to establish a training environment, which contains the necessary dependencies and has the correct framework in accordance with the object detection API that will be used. Setting up the Faster R-CNN object detection pipeline was done in `Python` using the `Tensorflow` object detection API. Prior to data collection and training, it was necessary to establish a Tensorflow object detection environment with the required `Python` dependencies. This was done in accordance with the official `Tensorflow` object detection API documentation (Tensorflow, 2020c). This includes installing the necessary `Python` dependencies, installing `Protocol Buffers` (protobuf) and downloading the official `Tensorflow models` repository (Tensorflow, 2020b). This repository contains the necessary framework for running the object detection model and verifying that the environment has been set up correctly. Further, the desired object detection model was downloaded from the `Tensorflow detection model zoo` (Tensorflow, 2020a). This is a `GitHub` repository containing various pre-trained models for object detection. The model was trained locally on a `Ubunut 18.04` operating system with a `GeForce RTX 2070 SUPER®` graphic card, using `Tensorflow-1.15.0` with GPU support.

3.2 The Model

As outlined in the previous section, the `Tensorflow detection model zoo` repository contains several pre-trained object detection models that can be downloaded and fine-tuned to purpose. One of these models is the `Faster R-CNN-Inception-Resnet-V2` model. This is a Faster R-CNN, which uses the same architecture and loss functions as described in section 2.4.4. The feature extractor used in the model, i.e. the base network in figure 2.27, is the `Inception Resnet-V2` network, proposed by Szegedy et al. (2016). In this thesis, the *Faster_RCNN_Inception_Resnet_V2_atrous_COCO_2018_01_28* implementation of the `Faster R-CNN-Inception-Resnet-V2` model was used. The reason for choosing this model is due to its performance, achieving the highest accuracy in a comparison between modern state of the art object detection models (Huang et al., 2017b).

The model has been implemented in Tensorflow and trained on the `Common Objects in Context` (COCO) data set (Consortium). The files provided with the pre-trained model are:

- *frozen_inference_graph.pb*
- *checkpoint.txt*
- *model.ckpt.index*
- *model.ckpt.data*
- *model.ckpt.meta*

The *frozen_inference_graph.pb* describes the data flow, the computations performed during inference and the parameters of the model (weights and biases). The frozen part implies that the trained parameters (weights and biases) in the model have been converted to constants. Thus, the graph can only be used for inference, i.e. the model can no longer learn from new input (adjust its weights and biases), only produce a prediction based on the frozen weights. In addition to the inference graph itself, `Tensorflow` also saves the parameters and the structure of inference the graph separately. This is saved as checkpoint files in *model.ckpt.data* and *model.ckpt.meta* respectively. This means that an inference graph can be constructed using *model.ckpt.data* and *model.ckpt.meta*. In contrast to the *frozen_inference_graph.pb* the checkpoints can be fine-tuned using the new training data. During training, as the model is updated, new checkpoints will be saved. This is kept track of in the *checkpoint.txt*. Finally, the *model.ckpt.index* contains an internal mapping between *model.ckpt.data* and *model.ckpt.meta* used when constructing the new fine-tuned inference graph after training. Due to the size of these files, they will be stored in a `Google Cloud Storage Bucket`, which can be accessed through a link in the `GitHub` repository of this thesis.

3.3 Data and Labeling

The data set used in this thesis consists of RGB core images from 27 wells from the NCS, which was prepared by GeoProvider (GeoProvider, 2019) in connection with the 2019 Force Hackaton, and made publicly available under the CC-BY 4.0 license. An overview of the wells in this data set can be seen in appendix A. The core images have been cropped into segments defined by the top and base of each core segment present in the original images. This process is illustrated for the $3931 - 3933m$ interval in well 6406/3-2 in figure 3.2. The filename of each cropped core segment is labelled with the well name, together with the top and base of the interval the core represents in the well.

In the labelling process, for the sake of simplicity, the core plugs were divided into two categories: CCA plugs and SCAL. Here, the CCA category includes all plugs that are part of the routine core analysis such as horizontal and vertical plugs used to measure porosity and permeability, and dean-stark plugs used for saturation measurements. The SCAL category includes core plugs that are not part of the routine core analysis. These are often referred to as 1.5 inch plugs in the core sampling reports that can be found for

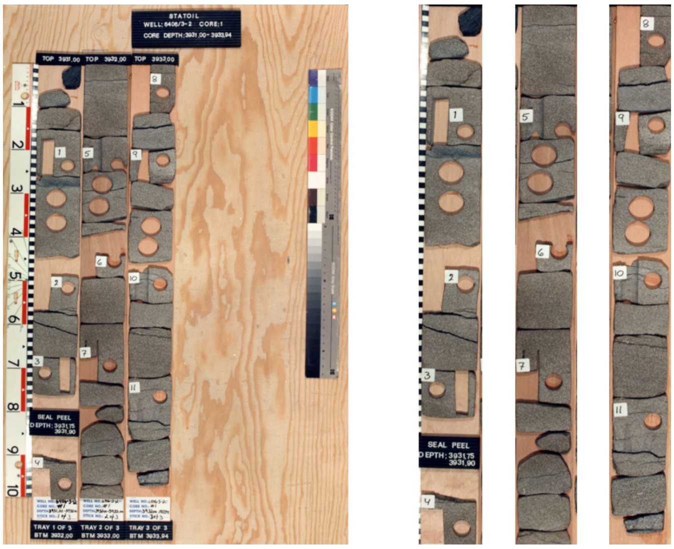


Figure 3.2: A side by side comparison of the original image from NPD *(left)* and the same image cropped by `GeoProvider`© *(right)*. Original image NPD.

the public wells on the NPD factpages (NPD). The reason for this simplified division is due to the visual appearance of the plugs in the optical core images. Based on these two categories, the plugs are divided into three classes. In the CCA category, the plugs are divided into two classes: horizontal CCA plugs and vertical CCA plugs. Based on the orientation of the dean-stark plugs, they will be assigned to either the horizontal CCA class or the vertical CCA plug class. In the SCAL category, every 1.5 inch plug is treated as the same class: SCAL.

As outlined in chapter 2, the various plugs have several features that contribute to the classification of a core plug into one of the mentioned classes. The main distinctional feature between a horizontal and vertical plug is the plug orientation with respect to the bedding of the core. The horizontal and vertical plugs are extracted parallel and perpendicular to the bedding, respectively. Since the core is usually slabbed parallel to the maximum dip of the bedding (McPhee et al., 2015), the horizontal and vertical plugs express a distinct visual characteristic in image view, as seen in figure 3.3. Thus, both the orientation with respect to bedding and the difference in visual appearance is used when classifying the cores plugs. The SCAL and horizontal plugs display a similar visual appearance since both plugs are extracted parallel to the bedding plane. However, they can be distinguished by two main features: plug frequency and diameter. As outlined in section 3.3, horizontal and vertical CCA plugs are often extracted pairwise at the same depth point. This is done in order to measure the maximum and minimum permeability at each sampling point (McPhee et al., 2015). Using these features, summarized in table 3.1, the core plugs were categorized into the appropriate plug classes. In addition to visual appearance, the core sampling report for both the CCA and non-CCA are provided for some wells on NPD factpages. These reports were used when the class of a core could not be determined from

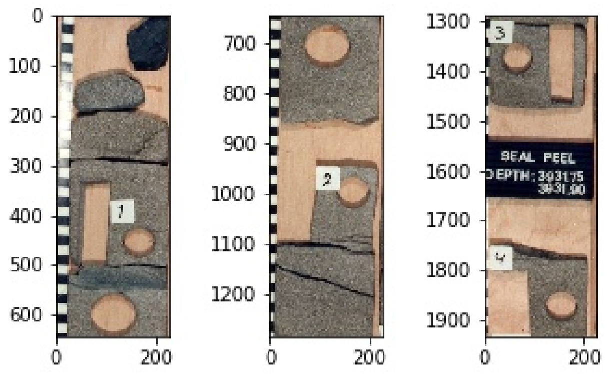


Figure 3.3: The $3391 - 3392m$ interval from well $6406/3 - 2$ showing four horizontal and vertical CCA plug pairs and two SCAL plugs. In this view the vertical CCA plugs are rectangular, and both the horizontal CCA and SCAL plugs are circular. The axes are displayed in pixels

Plug type	Diamter	Orientation w.r.t bedding	Samp. frequency
Horizontal CCA	2.5 - 3.81 cm	parallel	25 cm
Vertical CCA	2.5 - 3.81 cm	perpendicular	25 cm
SCAL	$\geq$ 3.81 cm	parallel	1-2 m

Table 3.1: Summary of the classification criteria for the core plugs.

the visual appearance alone, e.g. where no clear bedding is present.

Although, horizontal and SCAL plugs usually have a circular appearance in image view, it is not always the case. Since they are extracted parallel to bedding, they may have any number of orientation parallel to the xy-plane as seen in figure 3.4. Recall from

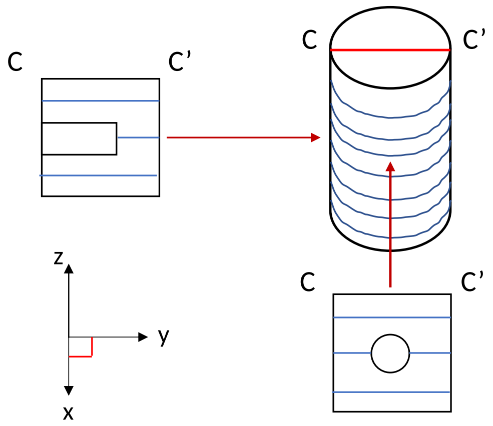


Figure 3.4: The resulting shape in image view when extracting bedding parallel cores strictly parallel to the x *(circular shape)* and y-axis *(rectangular shape)*

section 2.1.3, the core is slabbed into three segments: A, B and C, where the C segment is preserved and photographed, as seen in figure 2.2. In figure 3.4 the $C - C'$ cross-section represents the image view of the cores after slabbing. Extracting the core parallel to the x-axis will give a circular shape in image view and extracting parallel to the y-axis will give a rectangular shape. Additionally, any direction in between will give rise to a slightly different visual appearance in image view. It is believed that this will introduce a significant amount of variance to the classes where the plugs are extracted parallel to bedding. Similarly, the appearance of the vertical plugs may be affected by the direction of extraction. However, bedding is rarely parallel to the z-axis, the variation in visual appearance is often less than for the bedding parallel plugs with respect to the direction of extraction.

The main visual variance in the vertical plugs relates to the degree they penetrate the yz-plane parallel to the $C - C'$ cross-section. The degree and angle of penetration will affect both the shape and relief of the core plug in image view on the $C - C'$ cross-section. For example, if a vertical plug does not penetrate the $C - C'$ cross-section, it will not be visible in the optical core image. This causes some discrepancy between the number of plugs seen on the core and the number of plugs in the core sample report. However, the variance of the visual appearance of the vertical plugs is expected to be less than that of the bedding parallel plugs.

The most important features used in the labelling process when classifying the core plugs were their visual appearance and their orientation with respect to bedding. However, as outlined above, there can be large variations in the visual appearance of the core plugs, especially in the bedding parallel core plugs. In order to account for the variance in bedding parallel core plugs, two data sets were made. The first data set from well 4606/3-2, where the horizontal plugs have mainly been extracted parallel to the x-axis. Here, the horizontal and SCAL plugs will mainly appear circular in image view. The second data set is from well 4606/8-1 where the extracted plugs have a more arbitrary orientation on the xy-plane, where the bedding parallel plugs have a variety of shapes. Well 4606/3-2 and 4606/8-1 can be regarded as the low variance and high variance data set, respectively. The two data sets are summarized in table 3.2.

Data set	Well	Bedding parallel plug shape	Samples
Low Variance	4606/3-2	mainly circular to sub-circular	289
High Variance	4606/8-1	rectangular to circular	136

Table 3.2: Summary of the low and high variance data set, with respect to the shape of the bedding parallel core plugs in image view.

The labelling process consists of drawing bounding boxes around each core plug present in the images processed by `GeoProvider`© (GeoProvider, 2019). This was done using `LabelImg`, which is is a open-source `Python` application for labelling data for object detection tasks (Tzutalin, 2015). Each image can contain several bounding boxes, and each image has a corresponding `.xml` file in which this information is stored. The `.xml` file describes the path to the corresponding image, the dimensions of said image and the coordinates of the bounding boxes that enclose the various types of core plugs that are present in the image. When drawing the bounding boxes, in addition to enclose the core plug itself, a small part of the surrounding core was included. The motivation was to include the context of the plug orientation with respect to bedding. This is illustrated in figure 3.5 for the three plug classes in the data set.

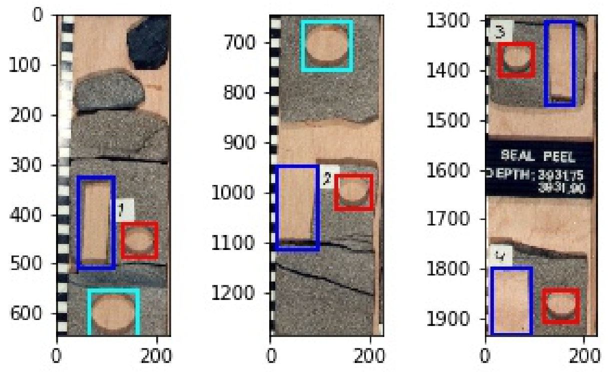


Figure 3.5: The three plug types in the data set, a small region of the surrounding core is included in the bounding box to include the bedding, if present. Here the horizontal CCA, vertical CCA and SCAL plugs are displayed in *red*, *blue* and *cyan*, respectively.

As outlined in the previous section, the images used for training and validation of the model have already been cropped by `GeoProvider`© (GeoProvider, 2019). The two main reasons for using the preprocessed core images are to reduce the size and complexity of the input data. The images have been cropped as follows: If the number of cores in the image is described by n, and each core has a width and height denoted by w and h, respectively. Further, the original image has the width and height denoted by W and H, respectively. After cropping, the original image of size $W \times H$ is reduced to n images of size $w \times h$. In this case, as seen in figure 3.2, the original input with the dimension $1538 \times 2410 \times 3$ is reduced to 3 RGB images with the approximate dimensions of $220 \times 1900 \times 3$. This significant reduction in size will greatly decrease the training and validation time of the model. Further, this will reduce the complexity of the input by removing the part of the image that does not contain any core plugs. These areas can be regarded as noise in relation to this specific task, and are excluded in order to prevent them from influencing the model.

3.4 Data Preprocessing

In this thesis, several image processing techniques were tested on the cropped images in the original data set. This was done in an effort to counteract undesirable model behaviour, which was discovered during the course of the training. The main issues were related to overfitting, i.e. decreasing training loss and increasing validation loss. An image preprocessing technique is defined as a transformation that is applied to the pixels of the data, prior to training the model (Prince, 2012). As outlined in chapter 2, overfitting may arise from the model learning features in the training data that are not relevant to solve the task in a general manner and may be a sign of redundant complexity in the input or the model architecture (Chollet, 2018). A common way to reduce the complexity of the input is to convert the RGB images to grayscale. Since many objects do not require the colour information to be recognized in an image (Elgendy, 2020). Thus, reducing the complexity from RGB to grayscale as a preprocessing step for the input would make the model less sensitive to the colour of the input images, which depends on several factors such as the lighting conditions or the camera model used to take the image, and possibly lead to a model with a greater ability to generalize. However, a preprocessing technique may also remove some features in the image that are important to solving the task. If this is the case, applying the preprocessing step may reduce the performance of the network.

This prompted the training of several models, applying a different preprocessing technique to the input of each model. The goal of the preprocessing was to investigate if the overfitting was related to the input, rather than the model itself and if it could be mitigated by applying some transformations to the pixel data prior to training. These models will be referred to as the candidate models, one model for each preprocessing technique. These models will be further elaborated on in section 3.5. The preprocessing technique applied to the input of these candidate models is summarized in table 3.3. The following subsections

Model name	Preprocessing technique
RGB	No preprocessing
gray	RGB to Grayscale conversion
sobel	Edge detection with Sobel filter
canny	Edge detection with the Canny edge detection algorithm
wavelet	Convolving input images with the Ricker wavelet

Table 3.3: Summary of the candidate models, with their name and the preprocessing technique applied to the input.

will outline the background and methodology for the image preprocessing techniques used in this thesis.

3.4.1 Grayscale

In the RGB colour model, each colour appears in its primary spectral components of red, green and blue, which is based in the Cartesian coordinate system where the colour subspace is defined by a cube (Gonzalez and Woods, 2018), as seen in figure 3.6. The coordinate system has its origin $(0, 0, 0)$ and denotes the colour black, with the red, green and blue (R, G, and B) colours defined along the axes of the coordinate system. The colour white can be found at $(1, 1, 1)$, the point where every colour component holds its maximum value. Using this cube, every colour can be represented using a certain combination of the colour components. The grayscale is the vector extending from the origin to the point $(1, 1, 1)$, as seen in figure 3.6. This line extends through all the points in the cube where the three colour components hold the same value and represent the various shades of gray between black and white (Gonzalez and Woods, 2018). Since the grayscale is a

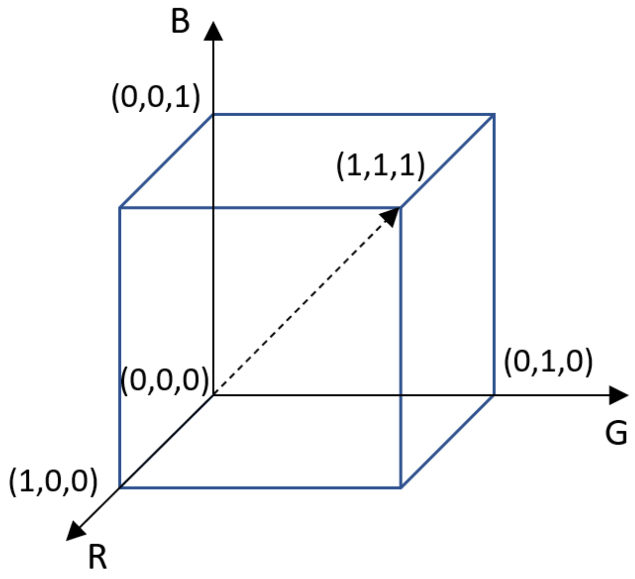


Figure 3.6: The RGB color model illustrated as a cube on the Cartesian coordinate system. Each axis is assigned a primary colour R - red, G - green and B - blue. The dashed vector from the origin $(0, 0, 0)$ to $(1, 1, 1)$ shows the grayscale from black to white.

one-dimensional vector, a grayscale image only requires one value to describe each pixel in the image.

The images in the data set were converted to grayscale using the `Python Image Library` (PIL) package according to the following equation:

$$L = 0.299R + 0.587G + 0.114B \tag{3.1}$$

Here, L denotes the grayscale pixel value, and R, G and B the three colour component values in the original image. Each converted grayscale pixel is the weighted sum of the RGB channels, using the standardized weights established by the international telecommunication union (International Telecommunication Union, 2001) for converting RGB to grayscale.

3.4.2 Edge Detection

As outlined in section 3.3, due to the way the length of the core is slabbed, the most applicable feature for classifying the core plugs was their visual appearance in the images. Both the horizontal CCA plugs and the SCAL usually appear as circles, which can be differentiated based on diameter and the vertical plugs appear as rectangles. Thus, using a preprocessing technique which enhances this structural information could be beneficial to the performance of the model. This has already been successfully attempted by Abdillah et al. (2018), where both Canny edge detection and the Sobel filter was employed as a preprocessing step prior to training a model for vehicle classification on images from a traffic camera. A similar approach was employed in this thesis for extracting the edges from the images, using both Canny and Sobel edge detection, which are classical edge detection algorithms. Additionally, an experimental processing technique was tested, which is inspired by the method used to generate synthetic seismic.

3.4.2.1 The Edge and Digital Images

An edge is characterized by a transition between two intensity levels between neighbouring pixels (Gonzalez and Woods, 2018). This could be either an abrupt change across one pixel or a gradual change that spans multiple pixels. Commonly occurring edges in digital images are the step, roof and ramp edge (Gonzalez and Woods, 2018). These are illustrated in figure 3.7, with their corresponding intensity level curves and gradients shown in blue and red, respectively.

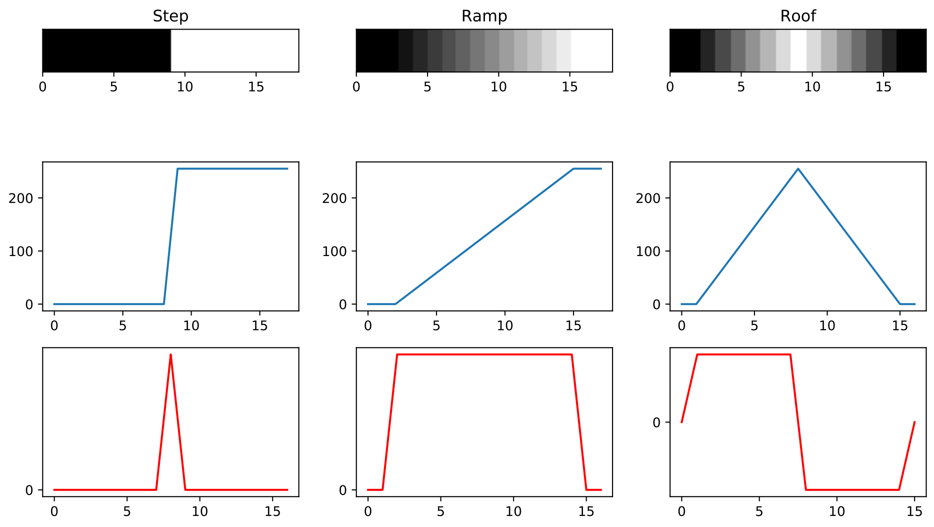


Figure 3.7: A set of commonly occurring edges in digital images. The two-dimensional representation of the intensity values (top), the intensity values ($middle$) and the gradient ($bottom$).

The ideal edge occurs over the distance of one pixel, as seen in the step edge in figure 3.7, where the edge location is marked by the abrupt increase in the magnitude of the gradient. However, due to the limitations of the focusing mechanism of the lens in a camera (Gonzalez and Woods, 2018), it is rare that an edge can be imaged across a single pixel. In practice digital images have blurred edges (Gonzalez and Woods, 2018), meaning the edge is marked by a gradual change in intensity values. This can be seen in the ramp edge in figure 3.7, where the resulting gradient produces a wider edge than in the step edge. Consequently, the edge is no longer determined by a single point. Instead, an edge point is any point contained within the ramp of the intensity profile (Gonzalez and Woods, 2018).

Digital images often contain noise, which will further impact the gradients of the image. Signal and noise analysis is an extensive scientific field. Thus, the various types of image noise and their causes are considered outside the scope of this thesis. However, it is worth mentioning the presence of noise may adversely impact the quality of the gradient-based edge detection algorithms. A common source of noise is random additive noise generated by the electronics in the camera (Gonzalez and Woods, 2018). This type of noise manifests itself as randomly increased or decreased intensity values in the digital image, which may be interpreted as edges when using the gradient. Figure 3.8 shows the intensity profile and gradient when adding random Gaussian noise to the edges in figure 3.7. In addition to the signal of the edge of interest, several noise generated edges will be

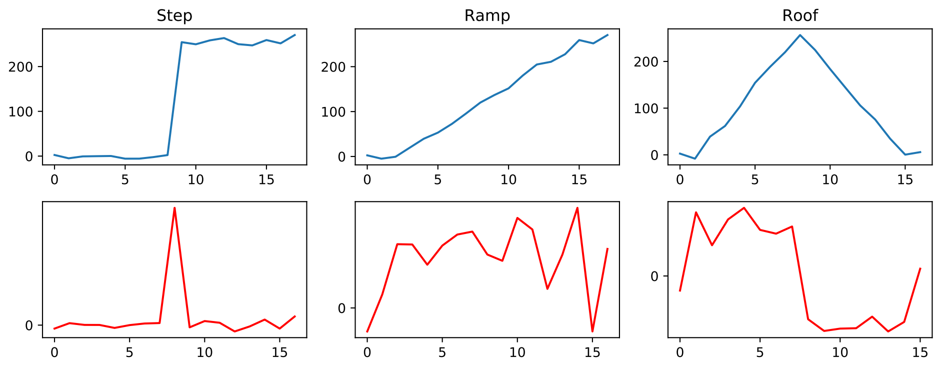


Figure 3.8: The intensity values and the gradient with the addition of additive Gaussian noise.

detected. Notice the gradients sensitivity to noise compared to the intensity profile, even in cases of moderate noise. By further increasing the signal to noise ratio in the image, the actual edge of interest might be obscured by the noise generated edges, rendering the use of the gradient futile for edge detection. Thus, it is common practice to suppress this noise with a smoothing algorithm, such as a Gaussian filter, prior to edge detection, increasing the signal to noise ratio (Gonzalez and Woods, 2018).

3.4.2.2 The Sobel Filter

This section will outline the fundamental background for calculating the gradients of an image and one of the basic edge detection algorithms, the Sobel filter. The Sobel filter will be used as one of the preprocessing techniques. The following will outline the derivation of the Sobel kernel as outlined by Sobel and Feldman (1973).

For the sake of simplicity consider the 3×3 two-dimensional grayscale image in figure 3.9. The gradient at (x, y), can be expressed as (Gonzalez and Woods, 2018):

$$\nabla f(x,y) = \begin{bmatrix} g_x(x,y) \\ g_y(x,y) \end{bmatrix} = \begin{bmatrix} \frac{\partial f(x,y)}{\partial x} \\ \frac{\partial f(x,y)}{\partial y} \end{bmatrix}. \tag{3.2}$$

where ∇f is the gradient, and g_x and g_y are the gradient operators in the x and y direction, respectively. Since ∇f is a vector, the direction of ∇f points in the direction of the greatest

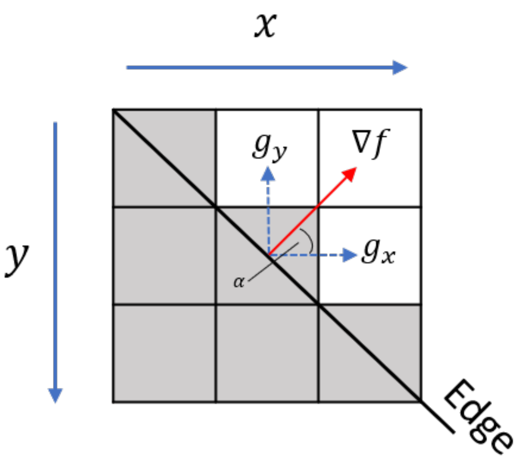


Figure 3.9: A 3×3 image, showing the gradient vector, ∇f, normal to the edge with its directional derivatives g_x and g_y. The direction of ∇f is given by the angle α.

rate of change of ∇f at (x, y) (Gonzalez and Woods, 2018). The direction of the gradient is given by the angle, α:

$$\alpha = tan^{-1}\left(\frac{g_y}{g_x}\right) \tag{3.3}$$

Further, the magnitude, $M(x, y)$, of ∇f is defined by its directional components (Gonzalez and Woods, 2018):

$$M(x, y) = ||\nabla f(x, y)|| = \sqrt{g_x^2(x, y) + g_y^2(x, y)} \tag{3.4}$$

Using these definitions, the gradient can be calculated for each (x, y) position in the image. It is common to refer to $M(x, y)$ or $||\nabla f(x, y)||$ as the gradient image, or simply the gradient (Gonzalez and Woods, 2018). Since an image consists of a set of distinct pixels, the outlined equations must be implemented as a discrete approximations of their analytical counterparts, i.e. the gradients need to be calculated at each pixel point. In practice, this is performed as a kernel convolution between the source image and a kernel. There are several kernels that can be used to compute the gradient. One of the more popular is the Sobel filter. The Sobel filter utilizes a kernel that is symmetric about its centre (Sobel and Feldman, 1973). Thus, the smallest kernel size is a 3×3 kernel, where the gradient at any location within the image is calculated based on its $n = 3 * 3 - 1 = 8$ neighbouring pixels.

Consider the 3×3 image in figure 3.10 defined on a Cartesian coordinate system, with its centre pixel as the origin. The letters $a - i$ denotes the intensity value of the given pixel. The gradient, G, in e is determined by the average of the sum of the magnitudes of the eight directional derivatives, multiplied with their direction. The eight directional derivatives are shown in figure 3.10 as red vectors. Each vector shows the direction of

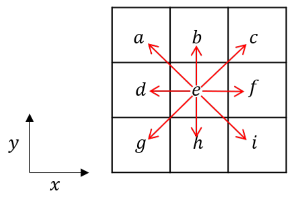


Figure 3.10: A 3×3 image with the intensity values $a - i$. The gradient of e is determined by averaged sum of the magnitudes of the directional derivatives to its neighbouring pixels.

the derivative, and they are defined by the eight possible pixel pairs e can make with its neighbours. The magnitude of each directional derivative is given as the difference in intensity value between the pixel pairs, divided by the distance between them (Sobel and Feldman, 1973). Summing the directional derives pairwise, where the head of the vectors in the pair are antipodal points, will result in the cancelling of the e terms (Sobel and Feldman, 1973). This has been shown for the antipodal pair $((b, e), (h, e))$ in equation 3.5.

$$g_{b,h} = \frac{b-e}{d_e(b,e)}\hat{\mathbf{i}} + \frac{h-e}{d_e(h,e)}[-\hat{\mathbf{i}}] = \frac{b-h}{d_e(b,h)}\hat{\mathbf{i}} \tag{3.5}$$

Here, $\hat{\mathbf{i}}, \hat{\mathbf{j}}$ denotes the direction of the unit vector and d_e denotes the euclidean distance between the pixel pairs. The euclidean distance between two arbitrary points, $p = (p_1, p_2)$ and $q = (q_1, q_2)$, can be expressed as the following equation:

$$d_e(p,q) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2} \tag{3.6}$$

Using the euclidean distance the expressions in equation 3.5 be written as:

$$g_{b,h} = \frac{b-e}{1}\hat{\mathbf{j}} + \frac{h-e}{1}[-\hat{\mathbf{j}}] = (b-h)\hat{\mathbf{j}} \tag{3.7}$$

In the work outlined by Sobel and Feldman (1973) the average of the sum of the vectors, defined by the antipodal pixel pairs, is used to calculate the gradient. Thus, the $g_{b,h}$ is the average sum of the two vertical gradients (b-e and h-e) and $g_{f,d}$ is the average sum of the two horizontal gradients (f-e and d-e):

$$g_{b,h} = \frac{1}{2}(b-h)\hat{\mathbf{j}} \tag{3.8a}$$

$$g_{f,d} = \frac{1}{2}(f-d)\hat{\mathbf{i}} \tag{3.8b}$$

The four diagonal vectors which originates in e, can be decomposed into a horizontal and vertical direction. Using (c, e) as an example, and that $cos(\frac{\pi}{4}) = sin(\frac{\pi}{4}) = \frac{1}{\sqrt{2}}$, $g_{c,e}$ can be rewritten as:

$$g_{c,e} = \frac{1}{\sqrt{2}}[(c-e)cos(\frac{\pi}{4})\hat{\mathbf{i}} + (c-e)sin(\frac{\pi}{4})\hat{\mathbf{j}}] = \frac{1}{2}(c-e)[\hat{\mathbf{i}}, \hat{\mathbf{j}}] \tag{3.9}$$

Using equation 3.9, the averaged sum of the antipodal diagonal pairs $((c, e), (g, e))$ and $((a, e), (i, e))$ can be written as:

$$g_{c,g} = \frac{1}{4}(c - e)[\hat{\mathbf{i}}, \hat{\mathbf{j}}] + (g - e)[-\hat{\mathbf{i}}, -\hat{\mathbf{j}}] \tag{3.10a}$$

$$g_{a,i} = \frac{1}{4}(a - e)[-\hat{\mathbf{i}}, \hat{\mathbf{j}}] + (i - e)[\hat{\mathbf{i}}, -\hat{\mathbf{j}}] \tag{3.10b}$$

Finally, the gradient in e can be obtained by summing the gradients of the antipodal pairs from equations 3.8a, 3.8b, 3.10a and 3.10b:

$$G = g_x + g_y = \left[\frac{1}{2}(f - d) + \frac{1}{4}(-a + c - g + i)\right]\hat{\mathbf{i}} + \left[\frac{1}{2}(b - h) + \frac{1}{4}(a + c - g - i)\right]\hat{\mathbf{j}} \tag{3.11}$$

where g_x and g_y are the components of the gradient in the x and y direction in the image. To make equation 3.11 an expression for the average of gradients in e, it should be divided by four. However, for the sake of convenience, the expression in equation 3.11 is scaled by a factor of four (Sobel and Feldman, 1973). This results in an estimate for the gradient that is 16 times larger than the actual average of the gradients in e and is given as:

$$G = g_x + g_y = [2(f - d) + (-a + c - g + i)]\,\hat{\mathbf{i}} + [2(b - h) + (a + c - g - i)]\,\hat{\mathbf{j}} \tag{3.12}$$

In equation 3.13a and 3.13b, g_x and g_y from equation 3.12 are implemented as kernels.

$$g_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \tag{3.13a}$$

$$g_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \tag{3.13b}$$

In this thesis the data set was preprocessed with the 3×3 Sobel kernels in equation 3.13a and 3.13b, using the `OpenCV` python library implementation. First, the gradients in the x and y directions were separately extracted and squared to obtain the magnitudes. These were then added together to represent both the horizontal and vertical edges in the image. Figure 3.11 illustrates the steps used for the Sobel edge detection preprocessing technique, where the bottom right image $(||g_x(x, y)|| + ||g_y(x, y)||)$ is used as input for training the model.

3.4.2.3 Canny Edge Detection

The Sobel filter provides a good representation of the main structures in the image. However, as seen in figure 3.11. In addition to the main structural features, the texture of the rock is modelled as edges. Thus, it is desirable to investigate if further refinement of the image, using a more sophisticated edge detection model, will yield a better model performance.

The Canny edge detection algorithm is a multi-stage algorithm which strives to further improve edge detection with respect to the following criteria, as outlined by Canny (1986):

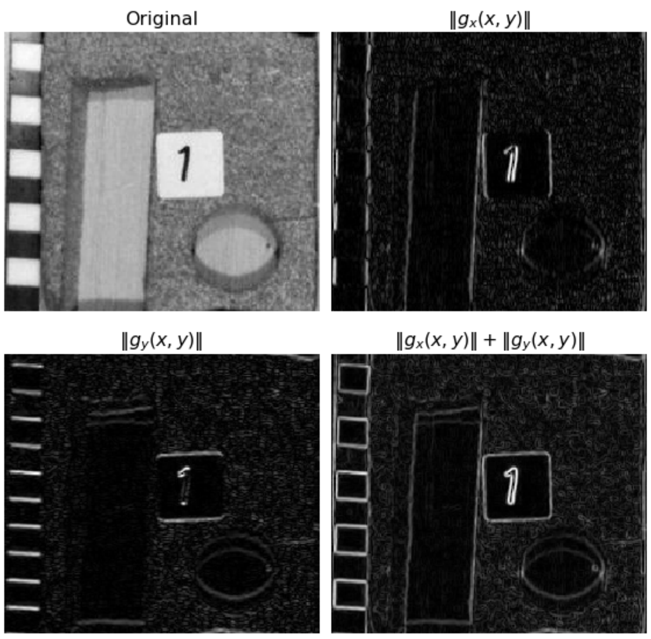


Figure 3.11: The magnitude of the Sobel filter applied in the x-direction $(||g_x(x, y)||)$, y-direction $(||g_y(x, y)||)$, and the sum of both directional components $(||g_x(x, y)|| + ||g_y(x, y)||)$.

1. Good detection, i.e. low probability of not detecting the edge (false negative) and low probability of detecting a non-edge as an edge (false positive). Both these probabilities are monotonically decreasing functions of the output signal-to-noise ratio, this criterion corresponds to maximizing signal-to-noise ratio (Canny, 1986).
2. Good localization. The points marked as edge points by the operator should be as close as possible to the centre of the true edge (Canny, 1986).
3. Only one response to a single edge. This is implicitly captured in the first criterion since when there are two responses to the same edge, one of them must be considered false (Canny, 1986).

These points are directly addressing the issues that arise in the simpler first-derivative based edge detection methods when introducing Gaussian noise. Recall that the location of the edges on the gradients in figure 3.7, became increasingly ambiguous with the addition of noise. The work outlined in Canny (1986) starts with mathematically expressing the three mentioned criteria, before deriving an optimal solution to these expressions through numerical optimization on a one-dimensional series, similar to the ones presented in figure 3.8. This is done in order to find an optimal operator for detecting a step edge.

The expressions and the full derivation of the numerical optimized operator are considered outside the scope of this thesis, and the curious reader is encouraged to consult the original paper for more information. However, the fact that the derived operator can be approximated as the first derivative of the Gaussian function (Canny, 1986), will be discussed.

The derivative of the one-dimensional Gaussian can be written as:

$$\frac{\partial}{\partial x}G(x) = \frac{\partial}{\partial x}e^{-\frac{x^2}{\sigma^2}} = \frac{-x}{\sigma^2}e^{-\frac{x^2}{2\sigma^2}} \tag{3.14}$$

where $G(x)$ denotes the one-dimensional Gaussian function and σ denotes the standard deviation. Figure 3.12 shows the Gaussian, $G(x)$ and the derivative, $G'(x)$ with three different standard deviations. The Gaussian is symmetric about the origin and the amplitude of $G'(x)$ is inversely proportional with the standard deviation, σ, and proportional with the width of the impulse. The main reason for choosing the Gaussian approximation rather than the optimal one-dimensional operator derived by Canny (1986) is due to the effectiveness of calculating the two-dimensional extension of an operator if it can be represented as some derivative of the Gaussian (Canny, 1986). Further, the Gaussian approximation only performed 20% worse than the optimal numerical operator. A difference in magnitude that is visually hard to detect (Canny, 1986).

Consider a convolution between a series,$t(x)$, and the Gaussian. Every point in the output will be the weighted sum the n nearest points in $t(x)$, where n is determined by the width of the Gaussian. The weight assigned to each neighbouring point is determined by the amplitude of the Gaussian at the given distance from the point being evaluated. By increasing σ, the number of points used to calculate the centre value will be increased. However, the weight assigned to the neighbours closer to the centre will be reduced. I.e. using a larger σ will reduce the significance of the points closer to the point that is currently being evaluated, in order to include a larger portion of the surrounding points in said evaluation. Using a Gaussian with a large σ will have a greater smoothing effect on the output.

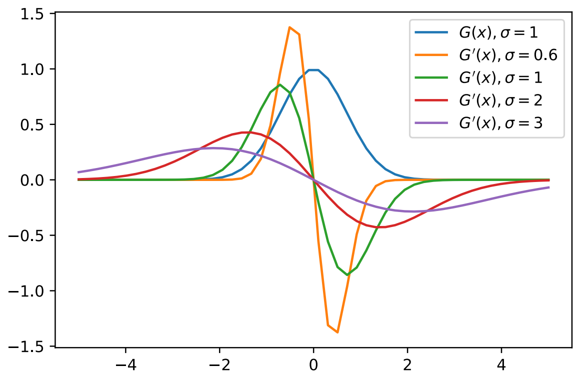


Figure 3.12: The Gaussian $G(x)$, with its first derivatives, $G'(x)$ with $\sigma = \{0.6, 1, 2, 3\}$. The value of σ controls the width of the Gaussian.

The smoothing effect is a desirable attribute, since the main problems related to the image gradient is its sensitivity to noise, as outlined in the previous section. The Gaussian

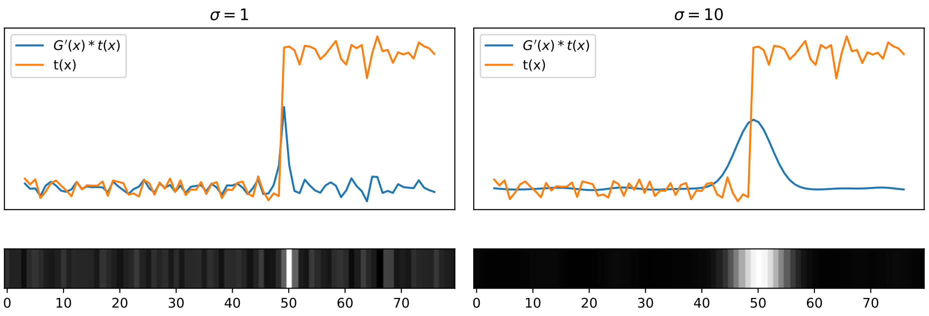


Figure 3.13: The effect of smoothing on a step edge with additive Gaussian noise convolved with the first derivative of the Gaussian, with $\sigma = 1$ $(left)$ and $\sigma = 10$ $(right)$. The amount of smoothing, i.e. noise suppression, is proportional with σ, and gives rise to the ridge profile seen in the right figure.

operator has the added benefit of increasing the signal to noise ratio, which is controlled by σ. This is illustrated in figure 3.13, where the convolution between a noisy step edge, $t(x)$, located at $x = 50$, and $G'(x)$ using $\sigma = 1$ and $\sigma = 10$. Using $\sigma = 1$ will yield a more accurate localization of the edge of interest at $x = 50$, but not enough noise suppression. This can be seen as the numerous noise generated edges that occur at $x < 50$ and $x > 50$. Increasing σ increases the smoothing. The numerous noise generated edges are suppressed, whilst still being able to reasonably locate the edge of interest at $x = 50$.

Notice that the edge is defined by the ridge created by the smoothed gradient. The removal of noise may come at the cost of accurate localization of the edge of interest. An additional risk of noise suppression is the removal of the signal itself. Choosing a σ that is too large will limit the magnitude of the intensity contrast that can be detected. Consider that some of the intensity contrasts in $t(x)$ in figure 3.13 were part of the actual signal. Then using $\sigma = 10$ would suppress these edges, losing valuable information. Smoothing can be regarded as a trade-off between noise suppression, localization accuracy, and detection accuracy. Notice that this trade-off echoes the points 1) and 2) from earlier in this section, as outlined by Canny (1986). Thus, the choice of σ is determined by the contents of the input image.

The two-dimensional Gaussian can be written as:

$$G(x, y) = e^{-\frac{x^2+y^2}{2\sigma}} \tag{3.15}$$

Recall that the magnitude of the edge is greatest perpendicular to the edge. In a two-dimensional image, $I(x, y)$, there is no prior knowledge of the direction of the edges. Contrast this to the one-dimensional series in figure 3.13, where only one orientation of the edge is possible. Namely, normal to x. A convolution between the series $t(x)$ and $G'(x)$, $G'(x)$ is guaranteed to align perpendicular to the edge. Generalizing this approach in two-dimensions would require the one-dimensional gradient to be calculated in all possible directions in the image. In order to make this process less computational exhaustive, this operation can be approximated by convolving the source image with the two-dimensional Gaussian from equation 3.15 to form a smoothed image, $I_s(x, y)$, then differentiating perpendicular to the edge normal (Canny, 1986):

$$I_s(x, y) = G(x, y) * I(x, y) \tag{3.16}$$

Then both the direction and magnitude of the gradient can be calculated, applying the equation 3.4 and 3.3, (Gonzalez and Woods, 2018):

$$M_s(x, y) = ||\nabla I_s(x, y)|| = \sqrt{g_x^2(x, y) + g_y^2(x, y)} \tag{3.17}$$

and

$$\alpha(x, y) = tan^{-1} \left[\frac{g_y(x, y)}{g_x(x, y)} \right] \tag{3.18}$$

where the directional derivatives g_x and g_y can be calculated using e.g. the Sobel kernel from the previous section. Due to the smoothing effect of the Gaussian, as illustrated in figure 3.13. The smoothed image $||\nabla I_s(x, y)||$ will contain ridges around the local maxima. To improve the localization of the edge, these ridges are often thinned using *non-maximum suppression* (NMS) (Gonzalez and Woods, 2018). This is done by specifying the number of orientations the gradient vector is allowed to have.

In this thesis, the `OpenCV` implementation is used, where the gradient vector is only allowed to oriented horizontally ($[\hat{\mathbf{i}}, 0] \vee [-\hat{\mathbf{i}}, 0]$), vertically ($[0, \hat{\mathbf{j}}] \vee [0, -\hat{\mathbf{j}}]$) or along the two diagonals ($[\frac{\sqrt{2}}{2}\hat{\mathbf{i}}, \frac{\sqrt{2}}{2}\hat{\mathbf{j}}] \vee [\frac{-\sqrt{2}}{2}\hat{\mathbf{i}}, \frac{-\sqrt{2}}{2}\hat{\mathbf{j}}] \wedge [\frac{-\sqrt{2}}{2}\hat{\mathbf{i}}, \frac{\sqrt{2}}{2}\hat{\mathbf{j}}] \vee [\frac{\sqrt{2}}{2}\hat{\mathbf{i}}, \frac{-\sqrt{2}}{2}\hat{\mathbf{j}}]$) in the image. Any α in between is rounded to the closest pre-defined orientation (OpenCV). When the directions of the gradient vectors have been determined, each point (x, y) in $||\nabla I_s(x, y)||$ is evaluated with respect to its two neighbours along the gradient vector. If the magnitude of the gradient of one or both the neighbours are greater than the point currently being evaluated, the point is discarded.

This ensures that the edges in the output will be a local maxima. The effect of the NMS is illustrated in figure 3.14, using the one-dimensional series $t(x)$ with $\sigma = 10$ from figure 3.13. The edges have been effectively thinned by removing edges within the ridge, with gradient magnitudes smaller than the local maxima. This yields a lower localization error. This example can be viewed as the vector gradient of a sub-vertical edge that has been rounded to the horizontal ($[\hat{\mathbf{i}}, 0] \vee [-\hat{\mathbf{i}}, 0]$), before performing NMS.

The last step in the Canny edge detection algorithm is a thresholding step. The essence of this step is to separate the noise generated gradients from the signal. Since the amplitudes of the additive noise can be modelled as a Gaussian distribution, while the step edge response will be composed of large values occurring very infrequently (Canny, 1986). In a histogram representation, the noise energy is expected to be confined to the lower percentiles of the histogram (e.g. less than 80 percent) (Canny, 1986) and the signal to the upper. However, simply applying this threshold may introduce unwanted effects on the gradient image $||I_s(x, y)||$. If only one threshold, T, is chosen, any pixel below this value will be removed. The weakness of any thresholding algorithm is that the value of T needs to be set correctly. Setting it too high increases the chance of removing the weaker part of the signal - false positive. Setting it too low will preserve the stronger part of the noise - false negative (Gonzalez and Woods, 2018). Thus, the threshold value must be close to

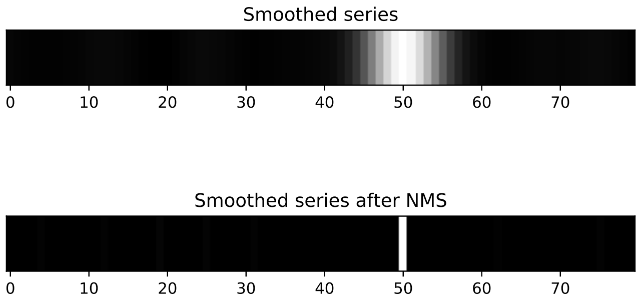


Figure 3.14: The effect of NMS applied along the direction of the gradient vector. The ridge profile in the original image after smoothing (top) and the output after NMS $(bottom)$.

optimal to be effective. For the edge to be preserved when using a single threshold, every single point that constitutes an edge needs to be greater than T, which is usually not the case. Applying the threshold T may lead to breaking up of the edge where the values of the points along the edge fall below T.

The Canny edge algorithm attempts to account for this by defining two thresholds, T_U and T_L, which defines an upper and lower threshold value. Any value above $T_U > ||I_s(x, y)||$ is kept as an edge and any value below $T_L > ||I_s(x, y)||$ is discarded as a non-edge. To determine if the values in between, $T_L < ||I_s(x, y)|| < T_U$, constitutes an edge are classified based on connectivity. Consider a point (p_0, q_0), where $p_{0:i:n} \in x$ and $q_{0:j:n} \in y$. If $||I_s(p_0, q_0)||$ is such that $T_L < ||I_s(p_0, q_0)|| < T_U$ and is connected to a point (p_i, q_j) with a gradient value $T_U < ||I_s(p_i, q_j)||$. Then $||I_s(p_0, q_0)||$ is regarded as part of the edge, although it having a lower value than T_U. If there is no connection to a point with a gradient value larger than T_U, (p_0, q_0) is classified as a non-edge. This method is called *hysteresis thresholding*, and significantly reduces the probability of keeping a false negative or discarding a false positive (Canny, 1986).

To summarize the Canny edge detection algorithm improves both the detection and localization of the gradients that constitute the edges in the image. This is achieved by:

- Reducing noise by smoothing, using the first derivative of the two-dimensional Gaussian.
- Extracting the directional gradients
- Thinning the gradients along a finite set of discrete directions, using non-maximum suppression
- Thresholding based on connectivity, using hysteresis thresholding

In this thesis, the `OpenCV` implementation of the Canny edge detection algorithm is used. The input is first smoothed using a 5×5 Gaussian filter, before the directional gradient,

g_x and g_y, are calculated using the Sobel kernels from equation 3.13a and 3.13b. Then the edges are thinned using NMS with the four pre-defined directions, as outlined earlier, and thresholded according to a user-defined T_U and T_L. Due to the variety of images in both the training and test set, it is desirable to choose these threshold values adaptively. As outlined, the noise energy is expected to be confined to the lower percentiles of the gradient image. Further, the ratio of the high to the low threshold in the implementation is in the range two or three to one (Canny, 1986). Figure 3.15 shows the output from the Canny edge detection algorithm using three different percentiles, $P_i = \{80, 85, 90\}$, of the intensity values of the input as T_U. The lower threshold is given as $T_L = \frac{1}{2}T_U$, in order to achieve the $1 : 2$ ratio between $T_U : T_L$. Using the upper percentiles seems to be a

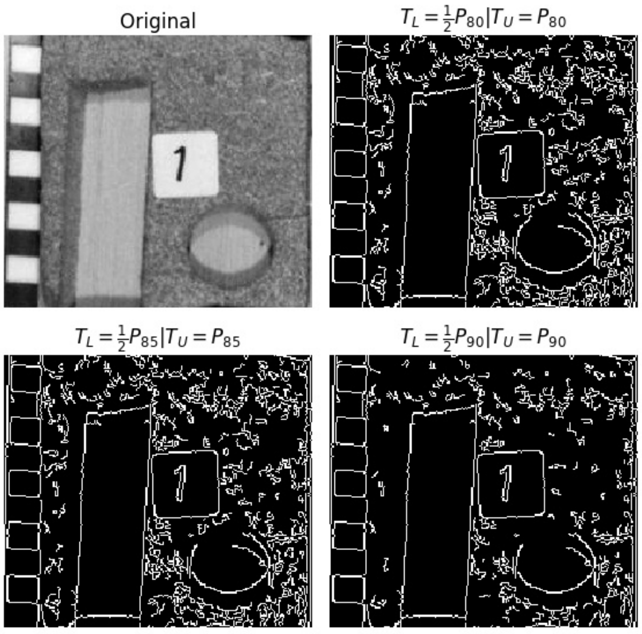

Original
$T_L = \frac{1}{2}P_{80}|T_U = P_{80}$
$T_L = \frac{1}{2}P_{85}|T_U = P_{85}$
$T_L = \frac{1}{2}P_{90}|T_U = P_{90}$

Figure 3.15: The output from the Canny edge detection algorithm using the $P_i = \{80, 85, 90\}$ percentile of the intensity values as the upper threshold, T_U, and the lower threshold, $T_L = \frac{1}{2}T_U$ in the hysteresis thresholding.

reasonable choice of T_L and T_U for preserving the main structural features of the input. In the implementation for generating training data, for each image, $P_i = \{80 \vee 85 \vee 90 \vee 95\}$ is randomly chosen as T_U.

3.4.2.4 Wavelet Convolution

The final preprocessing technique applied to the data is inspired by the convolutional model used for creating synthetic seismic in wells. The idea behind the convolutional model is that the seismic trace can be modelled as the convolution between the seismic pulse, the wavelet, and the reflection coefficient of a boundary between two rock layers in the well (Simm and Bacon, 2014). The wavelet describes how the amplitude of the seismic pulse changes with time, and the reflection coefficient can be regarded as the contrast between two layers. The idea is to apply the same concept to the image data to identify the contrasts between the pixel values to better visualize the structural information of the core images. Note that both seismic modelling and the subject of the reflection coefficient are extensive topics, outside the scope of this thesis. The goal of this section is not to outline these, but rather draw inspiration from the processes used when modelling synthetic seismic.

The reflection coefficient is given as (Simm and Bacon, 2014):

$$R_c = \frac{v_2\rho_2 - v_1\rho_1}{v_2\rho_2 + v_1\rho_1} = \frac{AI_2 - AI_1}{AI_2 + AI_1} \tag{3.19}$$

where AI_1 and AI_2 are the product of the bulk density ρ and the velocity of the medium v. Further, AI_1 denotes the acoustic impedance on the incident side and AI_2 is the acoustic impedance of the transmitted side of the boundary. Equation 3.19 is the simplest form of the reflection coefficient, and it only holds if the wave is propagated perpendicular to the boundary. Using equation 3.19, the reflection coefficient of the rock boundaries can be calculated to form a reflection series. The synthetic seismogram is then created by convolving the reflection series with a the seismic pulse (Simm and Bacon, 2014). As mentioned, the wavelet describes how the amplitude of a seismic pulse, $w(t)$, changes with time. There are a number of types of idealized wavelets that are in common use, for example, to make well synthetics when the exact wavelet is not known (Simm and Bacon, 2014). One of the most popular, due to its simplicity, is the Ricker wavelet. The amplitude, $w(t)$, of the Ricker wavelet at time t is given as (Ryan, 1994):

$$w(t) = (1 - 2\pi^2 f^2 t^2)e^{-\pi^2 f^2 t^2} \tag{3.20}$$

where f denotes the peak frequency in the frequency spectrum of the wavelet, i.e. the dominant frequency. The wavelet is symmetric around its centre, defined by the peak frequency with two side-lobes. Further, the width of the wavelet,w_w is inversely proportional with the peak frequency, given as (Ryan, 1994):

$$w_w = \frac{\sqrt{6}}{\pi f} \tag{3.21}$$

Figure 3.16 shows the characteristic shape of the Ricker wavelet, with plotted with the peak frequencies $f = [100Hz, 50Hz, 25Hz]$. Convolving the reflection series with the wavelet, generates the synthetic seismic.

Treating each intensity values (x, y) of the input image as the reflection coefficient, a set of reflection series can be created along the height and width of the image. Consider a two-dimensional image array where the height and width are represented by the number of rows and columns in the array, $R \times C$. Since each intensity value represents the reflection coefficient at that point, a reflection series can be made from each column, using the row values associated with that column. Similarly, each row can be a reflection series, using the column values associated with the row. Convolving each reflection series with the Ricker

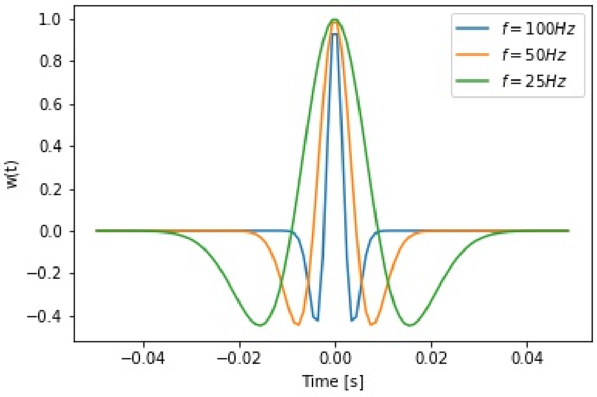


Figure 3.16: The Ricker wavelet, with three different values for the peak frequency.

wavelet will generate a ”synthetic seismogram” representation of the image. Figure 3.17 shows the resulting images after convolving along the width (lower row) and height (upper row) of the image, using the Ricker wavelet with the peak frequencies defined in figure 3.16. Comparing these images to the previous edge detection techniques in figure 3.11 and 3.15, each edge is marked by a transition from a positive value (blue) to a negative value (red), or vice versa. The edge itself is marked by the zero-crossing between the positive and negative value. Thus, the convolved images show both the location and the sign of the contrast, i.e. the change in the gradient of the intensity values.

Due to the inverse proportionality of wavelet width and peak frequency, the lower frequencies increases the amount of smoothing of the reflection series. This can be seen in figure 3.17, where the visibility of the internal texture of the core is drastically affected by the use of a wider wavelet. In order to make an unbiased choice of the wavelet width, for each image, the frequency randomly selected to be $f = [30 : 10 : 100]Hz$. The assumption is that there are benefits of using both a wide and narrow wavelet, and frequencies between 30 and 100 seemed to yield a good balance between smoothing and resolution. The narrow wavelet increases the number of small features that can be imaged (resolution), while the wide wavelet increases the smoothing and reduces the noise and complexity of the image. In the preprocessing step each image was convolved with a Ricker wavelet of random frequency from f, along its horizontal and vertical axis, creating two output images per input.

3.5 Training-Validation Pipeline

Before training the object detection model, the labelled RGB images were split into a training set and a validation set. The images in the low variance set consists of 289 images from well 6406/3 - 2 and the images in the high variance set from well 6406/8 - 1 consists of 136 images. As outlined in section 3.3, this is done to account for the variance in the

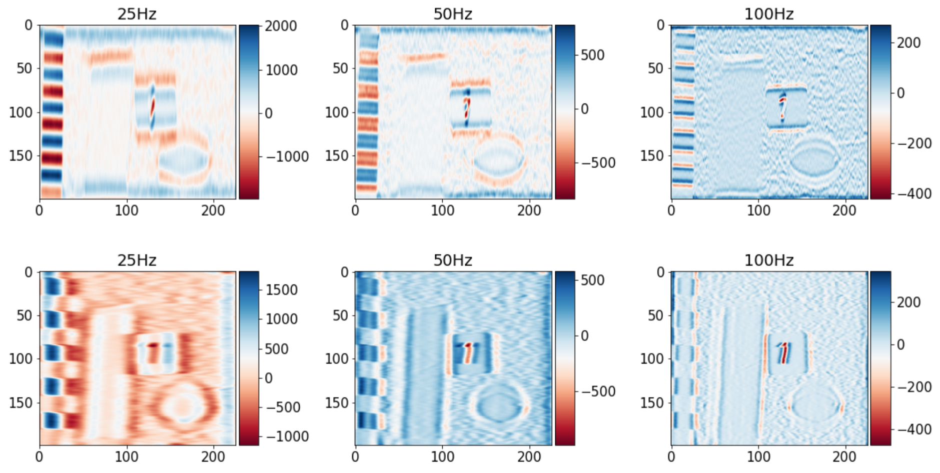


Figure 3.17: The output from convolving the Ricker wavelet, with three values for the peak frequency vertically (top) and horizontally $(bottom)$.

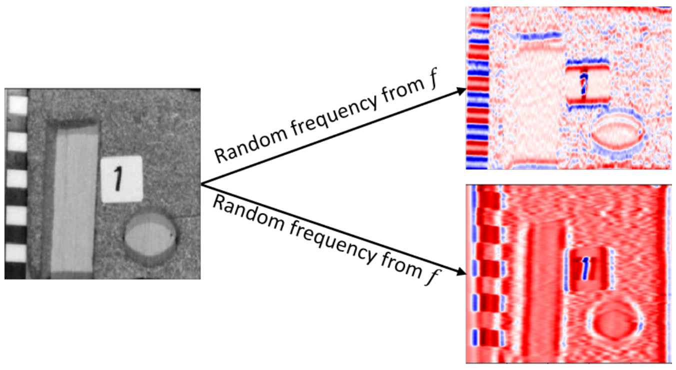


Figure 3.18: Showing the process where each image is convolved in the horizontal and vertical direction using a random frequency drawn from $f = [30 : 10 : 100]Hz$.

visual appearance of bedding parallel plugs have on the model.

The training-validation split was the same for both data sets and was created using a simple-holdout split, where 20% of the images were randomly split into the validation set. The testing of the preprocessing techniques was only performed on the low variance data set from well 6406/3-2. The assumption is that if a preprocessing technique does not increase the performance in the low variance set, it would fare no better in the high variance set. Therefore, the preprocessing techniques were tested on the low variance data set first. Figure 3.19 shows the setup for the experiment, and can be summarized as follows:

1. Two data sets: high variance and low variance data set from well 4606/8-1 and

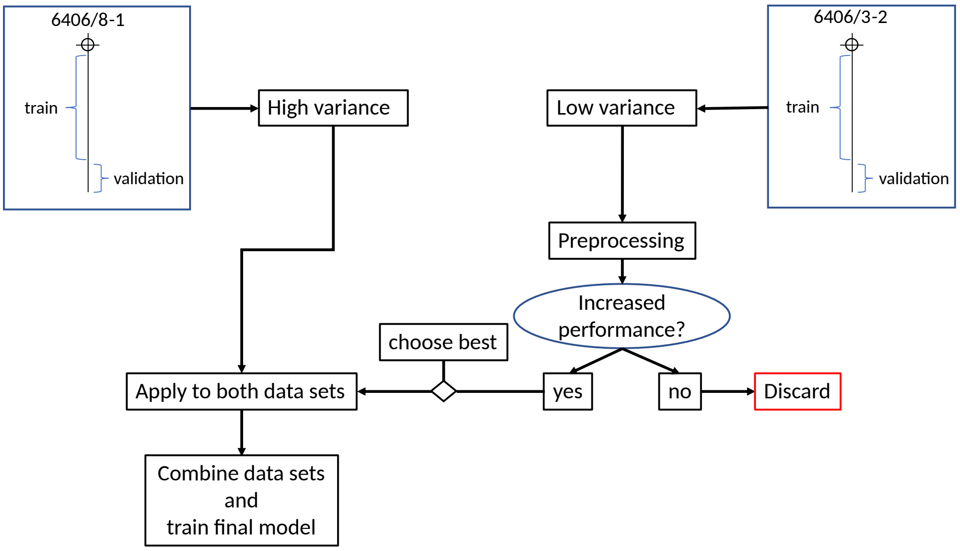


Figure 3.19: The setup for the experiment. The preprocessing techniques are tested on the low variance data set to verify their effectiveness with respect to model performance on the evaluation metrics. If performance is increased on the low variance data set, the preprocessing technique is applied to the high variance data set as well. Otherwise it is discarded. The final model is trained using both data sets, with the best performing preprocessing technique.

4606/3-2, respectively.

2. Test the 4 preprocessing techniques outlined in section 3.4 on the low variance data set.
3. Choose the best performing preprocessing technique and apply to both data sets.
4. Combine the data sets, with the applied preprocessing technique, and train a final model.

The first preprocessing technique applied to the training and validation set was the conversion to grayscale and any further preprocessing steps were applied to this new grayscale version of the training and validation set. The python functions for each preprocessing technique can be found in appendix H.

The setup for the input and preprocessing pipeline for the model is illustrated in figure 3.20. The result of preprocessing are 5 different training-validation set pairs, summarized in table 3.4. Each training-evaluation pair is then fed to the network for training. The result of this training will be 5 models, trained on the same data, using a different preprocessing step. These will be referred to as the candidate models. Based on the training results, one of these models will be chosen for further training according to figure 3.19. These models will be further discussed and elaborated on in chapter 4. From this point on, the training-validation pipeline will be described in a general manner. This is done to avoid repetition when describing the methodology. Thus, unless otherwise specified, any operation from

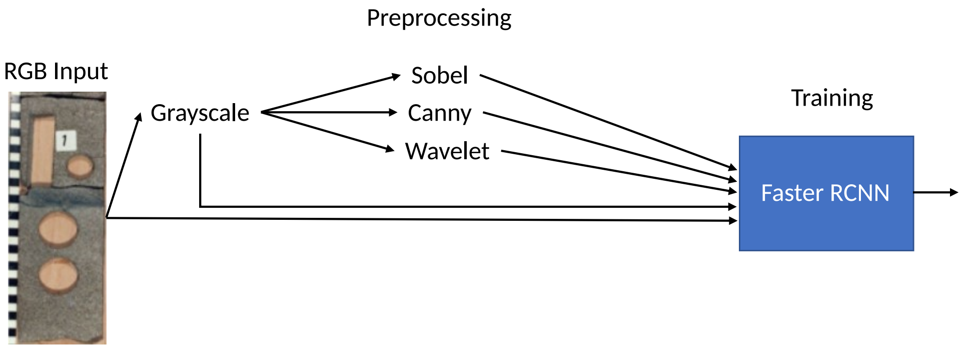


Figure 3.20: The set up for the preprocessing in the training-evaluation pipeline. Effectively training 5 models with 5 training-validation set, where a unique preprocessing task has been applied.

Name	Type of augmentation	# samples (train/valid)
RGB	Original RGB images	232/57
grayscale	Converted to grayscale	232/57
Sobel	Sobel filter	232/57
Canny	Canny edge detection algorithm	232/57
Wavelet Convolution	Wavelet convolution horizontal + vertical	464/114

Table 3.4: Overview of the 5 training-validation sets with their name, type of augmentation and the number of samples in the training and validation set for the low variance data set.

this point on that relates to the training-validation pipeline will apply to the candidate models in table 3.4, as well as any other model that will be presented in the chapter 4. Further, some of the subsequent sections will only relate to the final model. This will be made clear in the introduction of these sections.

As outlined in section 3.3, each image is saved with its corresponding `.xml` file, which denotes the location of each plug type and their location within the image. In order to read the data in an efficient manner the data is converted to a `Tensorflow Record(tfrecord)`, which is a `Tensorflow` binary storage format. This allows the model to parse both the images and the annotated bounding boxes in a serialized manner. One `tfrecord` is generated for the training data and one for the validation data, by running *generate_tfrecords.py* as seen in listing 3.1. However, since *generate_tfrecords.py* expects the image and labels in a `.csv` format, the `.xml` files are converted using *xml_to_csv.py*.

```
$ python path/to/generate_tfrecord.py --csv_input=path/to/csv/file --
 output_path=path/to/save/directory/train.record
```

Listing 3.1: Running the *generate_tfrecords.py* in terminal for the training data. Specifying the path to the `.csv` file containing the training data and the output path for the tfrecord file.

The two `tfrecord` files, *train.record* and *valid.record* contains the image data and the bounding box coordinates for the training set and the validation set respectively. As can be seen in *generate_tfrecords.py*, each label is converted to a numeric value. In this case the core plugs class to numeric value mapping is as follows: horizontal plug (hplug) = 1, vertical plug (vplug) = 2, special core analysis plug (scal) = 3. This relationship between label name and numeric value is saved in the file *labelmap.pbtxt*.

As mentioned in the environment setup in section 3.1, the `Tensorflow` object detection API uses `Protocol Buffer` (`Protobuf`) to configure the training-evaluation. `Protobuf` is a language and platform-neutral way of storing and serializing structured data and is therefore needed when setting up a `Tensorflow` object detection environment. The `protobuf` files are called from a configuration file that defines the training-evaluation pipeline. The configuration file can be divided into the following segments:

- model: Defines the meta-architecture and hyperparameters of the model.
- training_config: Defines training hyperparameters such as batch size, optimizer function, learning rate and type of data augmentation.
- train_input_reader: Provides the model with the training data
- eval_config: Determines the evaluation metrics that are reported and how they are reported
- eval_input_reader: Provides the model with the evaluation data

The initial hyperparameters chosen were based on the hyperparameter provided in the default configuration file *faster_rcnn_inception_resnet_v2.config*, provided with the pre-trained model. This configuration describes the implementation of the Faster R-CNN model, which is based on the implementation of the Faster R-CNN with an Inception Resnet feature extractor as outlined in Huang et al. (2017a). This default configuration file can be found in appendix B and for the sake of simplicity, this will be the configuration file that will be presented in this section. However, changes to the hyperparameters may occur in between training of the various models. These changes and which models they relate to will be made clear when presenting and discussing the results of the models in chapter 4

The model-segment of the configuration file defines the object detection model that is used. In this case a Faster R-CNN network as outlined in section 2.4.4, with an `Inception_resnet_v2` CNN (Szegedy et al., 2016) as the feature extractor. Additionally, the number of classes in the data set and the dimensions the input is reshaped to before it is fed to the network, as seen in listing 3.2. The original input is reshaped into a shape of $min\_dimension \times max\_dimension$, where $min\_dimension$ and $max\_dimension$ corresponds to the short and long dimension of the input, respectively (Huang et al., 2017a). Furthermore, the model-segment is divided into two stages which are marked with the following prefixes: $first\_stage$ and $second\_stage$. These correspond to the region proposal network (RPN) and the Fast R-CNN classifier module in the Faster R-CNN in figure 2.27, respectively. Under each stage in the configuration file, the hyperparameters of the RPN and the classifier can be modified. For the $first\_stage$ (RPN) the hyperparameters for the anchor scales, aspect ratios and stride can be modified changing the values for the scales and $aspect\_ratios$ and $height/width\_stride$ under the $first\_stage\_anchor\_generator$ as seen line 1013-1020 in listing 3.2. From these settings, there will be generated 12 anchors at ever anchor location in the image. These anchor locations have a spacing of 8 pixels along the height and width of the image.

```
model {
  faster_rcnn {
    num_classes: 3
    image_resizer {
      keep_aspect_ratio_resizer {
        min_dimension: 600
        max_dimension: 1200
      }
    }
    feature_extractor {
      type: ”faster_rcnn_inception_resnet_v2”
      first_stage_features_stride: 8
    }
    first_stage_anchor_generator {
      grid_anchor_generator {
        height_stride: 8
        width_stride: 8
        scales: 0.25
        scales: 0.5
        scales: 1.0
        scales: 2.0
        aspect_ratios: 0.5
        aspect_ratios: 1.0
        aspect_ratios: 2.0
      }
    }
```

Listing 3.2: A Faster R-CNN model with 3 classes ('hplug'—'vplug'—'scal') and the dimensions the input is resized to. The feature extractor is set to the 'faster_rcnn_inception_resnet_v2'.

The IoU and the confidence score thresholds in the RPN can be modified as seen in line 1000-1001 of listing 3.3. Recall the IoU concept in figure 2.22, which is a measure of the overlap between the ground truth bounding box and the prediction from the network according to equation 2.19. The confidence score is a probability score in the range $[0, 1]$, which describes how certain the network is that a predicted bounding box contains an object. These values define the thresholds for what region proposals that will be retained (positive anchors). This is determined by using an algorithm called **non-max suppression** (NMS). The NMS algorithm takes all the proposals with their corresponding confidence scores as input and discards any anchor with a confidence score lower than the *nms_score_threshold*. The remaining anchors are then sorted by the confidence score. The anchor with the highest confidence score, A_{high}, is removed and placed in a separate list, $L_{retained}$. The A_{high} is then used to calculate the IoU with the anchors in the original list A_i. Any anchor with an $IoU(A_{high}, A_i) > nms\_iou\_threshold$ are discarded. This process is repeated until $L_{retained}$ contains no two anchors with an IoU score greater than nms_iou_score. Additionally, the maximum number of anchors retained in $L_{retained}$ after applying the NMS algorithm can also be specified in line 1002.

The two last lines of listing 3.3 specifies the weight of the terms used to balance the multi-task loss function of the RPN. This Faster R-CNN model uses a slight variation of the loss function from equation 2.26. This implementation uses two constant values to balance the localization and classification loss, compared to the loss function in 2.26. Here the $first\_stage\_localization\_loss\_weight$ and $first\_stage\_objectness\_loss\_weight$ corresponds to the weight placed on the localization loss and classification loss, respectively.

```
    first_stage_nms_score_threshold: 0.0
    first_stage_nms_iou_threshold: 0.699999988079
    first_stage_max_proposals: 300
    first_stage_localization_loss_weight: 2.0
    first_stage_objectness_loss_weight: 1.0
```

Listing 3.3: The thresholds for the IoU and confidence score used by the RPN and the maximum number of proposals output after applying these thresholds.

For the second module (Fast R-CNN classifier) the IoU and confidence thresholds, as well as the maximum number of proposals retained after NMS can be specified as in the first stage. Similarly, this implementation of the Faster R-CNN uses two weights to balance the classification and localization loss in the multi-task for the Fast R-CNN classifier in equation 2.23. The λ in equation 2.23 corresponds to the $second\_stage\_localization\_loss\_weight$ and the $second\_stage\_classification\_loss\_weight$ can be used to increase the weight of the classification loss. Additionally, a constraint for the maximum number of predictions per class and the type of classifier used is also specified as seen in listing 3.4.

```
    second_stage_post_processing {
      batch_non_max_suppression {
        score_threshold: 0.300000011921
        iou_threshold: 0.600000023842
        max_detections_per_class: 100
        max_total_detections: 100
      }
      score_converter: SOFTMAX
    }
    second_stage_localization_loss_weight: 2.0
    second_stage_classification_loss_weight: 1.0
    }
```

Listing 3.4: The thresholds for the IoU and confidence score and maximum number of anchors retained per class and in total used by the NMS in the Fast R-CNN classifier. Additionally, the type of classifier and the relative weights of the multi-class loss can be specified.

Under $train\_config$ the hyperparameters for the training process are specified. Hyperparameters such as the batch size, the optimizer function, the learning rate, number of training steps, and the type of data augmentations to apply. A compressed version of the training configurations can be seen in listing 3.5, which shows the settings for the default model. Since model uses the momentum optimizer, the value of the weight γ from equation 2.4 can be specified under $momentum\_optimizer\_value$.

```
    train_config {
  batch_size: 1
  data_augmentation_options {
    random_horizontal_flip {
    }
  }
  optimizer {
    momentum_optimizer {
      learning_rate {
        manual_step_learning_rate {
          initial_learning_rate: 0.000300000014249
          schedule {
            step: 0
            learning_rate: 0.000300000014249
          }
      momentum_optimizer_value: 0.899999976158
    }
    use_moving_average: false
  }
  gradient_clipping_by_norm: 10.0
  fine_tune_checkpoint: "/path/to/pre/trained/model/checkpoints"
  from_detection_checkpoint: true
  num_steps: 200000
```

Listing 3.5: The train_config where batch size, type of data augmentation, optimizer function, learning rate and the number of steps to train the model can be specified. Further, a learning schedule and any parameters related to the chosen optimizer function can be specified. Additionally, the path to the checkpoints that will be fine-tuned can be provided.

Additionally, the path to the checkpoints that will be fine-tuned in the training process must be specified. This path is set to the pre-trained model's checkpoints. In this case, this path is directed to the checkpoints in the *Faster_RCNN_Inception_resnet_V2_COCO_2018_01_28* directory as seen in listing 3.6.

```
fine_tune_checkpoint:"/path/to/
    faster_rcnn_inception_resnet_v2_atrous_coco_2018_01_28/model.ckpt"
```

Listing 3.6: The specified path to the pre-trained models checkpoint. This absolute path needs to be changed according to the file structure of the project.

The modifications $train\_input\_reader$ and $eval\_input\_reader$ segments relates to specifying the path to the *train.record* and *valid.record*, respectively. Additionally, the path to the label map, *labelmap.pbtxt*, needs to be provided for both segments. The $eval\_config$ segments is shown in listing 3.7 and specifies the type of metric used to evaluate the model.

```
eval_config: {
  num_examples: 8000
  metrics_set: "coco_detection_metrics"
}
```

Listing 3.7: The evaluation metric used is the COCO detection metrics. The num_examples refers to the number of examples to process during evaluation.

The models are evaluated using the COCO evaluation metrics, which will be further explained in section 3.5.1.

After the configuration file has been modified to the desired configuration, the training can be initialized by running `train.py` from the terminal as seen in listing 3.8.

```
  $ python path/to/train.py --logtostderr --train_dir=training/directory
    --pipeline_config_path=path/to/configuration/file
```

Listing 3.8: Running train.py from the terminal specifying where to log any error messages from the training process (–logtostderr) the directory where the training checkpoints will be saved (–train_dir) and the path to the configuration file (–pipeline_config_path).

Additionally, as seen in listing 3.9, the `model_main.py` is ran in order to keep track of the models performance on the validation set with respect to the COCO evaluation metrics during the training process.

```
$ python path/to/model_main.py --alsologtostderr --pipeline_config_path=
   path/to/configuration/file --checkpoint_dir=training/directory --
   model_dir=evaluation/directory
```

Listing 3.9: Running `model_main.py` from the terminal specifying where to log any error messages from the evaluation process (–alsologtostderr) the path to the configuration file (–pipeline_config_path) the directory where the trained checkpoints used for evaluation are saved(–checkpoint_dir) and where to save the output form the evaluation (–model_dir).

Finally, both the training and evaluation processes are monitored using tensorboard by specifying the directories where the output from the training and evaluation are saved as defined in listing 3.8 and 3.9, as seen in listing 3.10.

```
$ tensorboard --logdir=training/directory
$ tensorboard --logdir=evalutaion/directory
```

Listing 3.10: Logging both the training evaluation in tensorboard by specifying the path of their respective directories.

3.5.1 COCO evaluation metric

The metrics used to evaluate the model performance are the COCO evaluation metrics as specified in listing 3.7. The COCO evaluation metric consists of 12 metrics that characterize the performance of the object detector with respect to the IoU between the predicted bounding box and the ground truth. The IoU, as outlined in section 2.4.4, is used to define whether the detected object is counted as a true positive or false negative by using a threshold. A higher IoU threshold represents a stricter metric since the overlap between prediction and ground truth must be higher to be counted.

There are 6 metrics that relates to average precision (AP) and 6 relates to **average recall** (AR). The COCO evaluation metric makes no distinction between AP and the mAP outlined in section 2.4.1 (Consortium, 2019). The mAP in equation 2.21 is equivalent with the AP in the COCO evaluation metric. Additionally, both the AP and AR are averaged over 10 IoU thresholds in the range $[0.5 : 0.05 : 0.95]$ unless otherwise specified (Consortium, 2019). The AR is calculated in a similar manner to mAP in equation 2.21. However, equation 2.20 is summed over the y-axis of the precision-recall curve instead of the x-axis. In order to avoid confusion, the following notation for the AP and AR metrics are adopted:

- $AP^{IoU=0.5:0.05:0.95} = mAP$: The precision is calculated at 10 IoU threshold from 0.5 to 0.95 with a step size of 0.05. The reported metric is the averaged precision across the different thresholds and classes in the data set. This is the main evaluation metric and will be denoted as mAP from this point on.

- $AP^{IoU=0.5}$: This is the most lenient threshold evaluation metric, where predicted bounding boxes with an IoU $\geq$ 0.5 are counted as true positives. This reported precision is averaged precision across the classes in the data set at a fixed IoU thresholds, $t_{IoU} = 0.5$ (not averaged across the 10 IoU thresholds).

- $AP^{IoU=0.75}$: This is the strict threshold evaluation metric, where predicted bounding boxes with an IoU ≥ 0.75 are counted as true positives. The reported precision is averaged precision across the classes in the data set at a fixed IoU thresholds, $t_{IoU} = 0.75$(not averaged across the 10 IoU thresholds).

Figure 3.21 shows a visual representation of the IoU thresholds outlined above. The AP

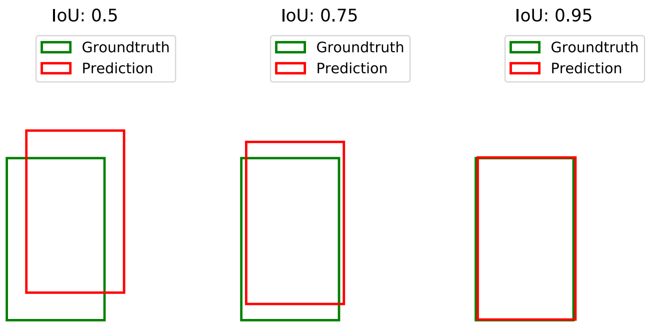


Figure 3.21: A visual representation of the IoU thresholds used to define a true postive.

for the scales are summarized below and are used to evaluate how accurately the model detects objects of various sizes in the image.

- AP^{small}: The average precision of the bounding boxes that have an area $< 36^2$ pixels.
- AP^{medium}: The average precision of the bounding boxes that have an area $> 36^2$ and $< 96^2$ pixels.
- AP^{large}: The average precision of the bounding boxes that have an area $> 96^2$ pixels.

The AR is calculated by averaging the recall values, for each class separately, over a range of IoU thresholds $[0.5 : 1]$. The AR metric is also reported for the 3 area-scales mentioned above. The 3 last AR metrics reports the maximum AR given a fixed number of detections, and give as follows:

- AR^1: The maximum AR given 1 detection per image
- AR^{10}: The maximum AR given 10 detections per image
- AR^{100}: The maximum AR given 100 detections per image

3.6 Testing

Since the training and validation set is split from a set of core images from the same wells, it is important to evaluate the model performance on a data set that the model has not been exposed to. This is done to verify if the model can generalize on previously unseen data. This step is only performed for the final model, once the desired performance has been achieved on the validation set. Since the test set is supposed to provide an unbiased evaluation of the model's performance, it can only be used once. In this thesis, two test sets were created. One test set from the two wells $6406/3-3$ and $6406/1-3$, which are relatively similar to the training and validation data, with respect to the visual appearance of the core plugs. The second test set is randomly sampled from the remaining 25 wells when $6406/3-2$ and $6406/8-1$ are excluded. These test sets will be elaborated further on in the results. The following describes the general methodology for making a test set, which applies to both test sets made in this thesis.

A new set of core images are selected from a set of wells that are not present in the original data from well $6406/3-2$ and $6406/8-1$. These are labelled and converted to a `tfrecord`, *test.record*, as previously outlined. A copy of the configuration file is made and named *faster_rcnn_inception_v2_testing.config*, where the *eval_input_reader* path is changed from *valid.record* to *test.record*. Finally the `model_main.py` is ran from the terminal as seen in listing 3.11, using the new configuration file and saving the output in a new directory testing directory.

```
$ python path/to/model_main.py --alsologtostderr --run_once --
    pipeline_config_path=path/to/configuration/file --checkpoint_dir=
    training/directory --model_dir=testing/directory
```

Listing 3.11: Running *model_main.py* from the terminal specifying where to log any error messages from the evaluation process (–alsologtostderr) the path to the configuration file (–pipeline_config_path) the directory where the training checkpoints used for evaluation are saved (–checkpoint_dir) and where to save the output form the evaluation (–model_dir). The –run_once option specifies that the model is only evaluated once using the latest checkpoint and not continuously logging the checkpoint directory.

3.7 Export Inference Graph

This step is only performed for the final model. After the training-validation process is finished. The latest fine-tuned checkpoints are exported as a frozen inference graph using `export_inference_graph.py`. This allows the model to be used for inference of cores in new images, without further updating the weights. This is a necessary step in order to use the model to generate predictions outside the training-evaluation pipeline. The inference graph can be exported according to listing 3.12

```
$ python export_inference_graph.py --input_type image_tensor --
    pipeline_config_path path/to/configuration/file --
    trained_checkpoint_prefix path/to/model.ckpt-0000 --output_directory
    path/to/output/directory.
```

Listing 3.12: Running $export\_inference\_graph.py$ from terminal specifying the input data type (–input_type) path to config file (–pipeline_config_path) path to the desired fine-tuned checkpoint (–trained_checkpoint_prefix) and where to save the frozen inference graph (–output_directory). The 0000 in model.ckpt-0000 checkpoint refers to the step the model was saved and in this case would be before training.

Chapter 4

Results and Discussion

In the following sections, the achieved results following the training-validation process, outlined in chapter 3, will be presented and discussed. Based on the results and discussion, the decision of which candidate model that will be chosen as the nominee will be made, as shown in figure 3.19. The preprocessing technique used in the nominee model will be applied to both the high and low variance data set which will be used to train the final model. Several hyperparameter configurations will be proposed and tested, to increase the performance of the model. The hyperparameter configuration that yields the best performing model will be used in the training of the final model. Lastly, the final model is tested on two test sets of different difficulty, followed by a discussion of the results and a proposed workflow for increasing the performance of the model.

First, the candidate models will be presented, which were trained and validated on the low variance data set consisting of 289 core images from well $6406/3-2$. As outlined in section 3.5 the data set was split into a training and validation set using an $80/20$ simple hold-out validation split. These models will be used to verify if the preprocessing techniques, outlined in chapter 3, can improve the performance of the model. The preprocessing technique used in the best performing model, the nominee, was applied to the high variance data from well $4606/8-1$ as well. This data set consists of 136 images and was split using the same simple hold-out validation split. A new model will be trained using the data from both well preprocessed with the preprocessing technique from the nominee model. A learning rate analysis will be performed on this new model, to find an optimal learning rate, which will be used to train the final model.

The results from the training and the performance of the final model with respect to two different test sets will be presented and discussed, before presenting a workflow that involves the fine-tuning of the final model using a small subset of one of the test sets. Lastly, this chapter will be concluded with a summary of the results and a proposed use case involving both the model developed in this thesis and the model from the specialization project presented in section 2.5.

4.1 The Candidate Models

In this section, the results from the 5 candidate models and the hyperparameters used will be presented. First, the results from the training and validation of the *RGB* model and its performance on the COCO evaluation metrics will be presented. These results will act as a benchmark, which the remaining candidate models will be compared to. As outlined in section 3.5, several hyperparameters can be adjusted to improve the model performance with respect to the task at hand. Here, in the initial benchmarking of the preprocessing techniques, the hyperparameters are kept as they were provided with the pre-trained model, which is based on the configuration used in Huang et al. (2017a). However, the localization term in the second stage of the Faster R-CNN is changed to $\lambda_2 = 100$. The reason for setting $\lambda_2 = 100$ is to reward better localization of the core plugs, i.e. increase the IoU of the detected core plugs. As mentioned in section 2.5 increasing the localization loss weight of the Fast R-CNN classifier showed to increase the performance with respect to the COCO-evaluation metrics. In the specialization project the most notable increase occurred in the mean average precision, mAP as defined in section 3.5.1, which increased from $mAP = 0.77$ to $mAP = 0.86$. Therefore, it is desirable to investigate if similar results can be achieved for detecting and classifying the different core plug types.

For the models to be comparable, the hyperparameters are kept constant across the 5 candidate models. The hyperparameter configuration can be found in the configuration file of each model, which will be included in the `Github` repository of this thesis. The main hyperparameters for the models are summarized in table 4.1. Figure 4.1 shows the

Optimization function	Momentum optimizer
Momentum optimizer value	0.9
Learning rate (constant), α	0.0003
Second stage localization loss weight λ_2	100
Second stage classification loss weight	1
Data augmentation	random horizontal flip

Table 4.1: Summary of the main hyperparameters used in the training of the 5 candidate models.

setup for this initial testing of the preprocessing techniques outlined in chapter 3. The candidate models are trained and evaluated on the same 289 images, randomly sampled from well 6406/3-2, using an 80/20 simple hold-out validation split. Based on the model performance, from these candidate models, one nominee will be chosen.

In order to present the models in a structured manner, each model will be named according to the notation adopted in table 3.3 and styled in *italic*. Further, for the core plugs in the three classes from table 3.1, the following notation will be adopted:

- Horizontal CCA - *hplug*
- Vertical CCA - *vplug*
- SCAL - *scal*

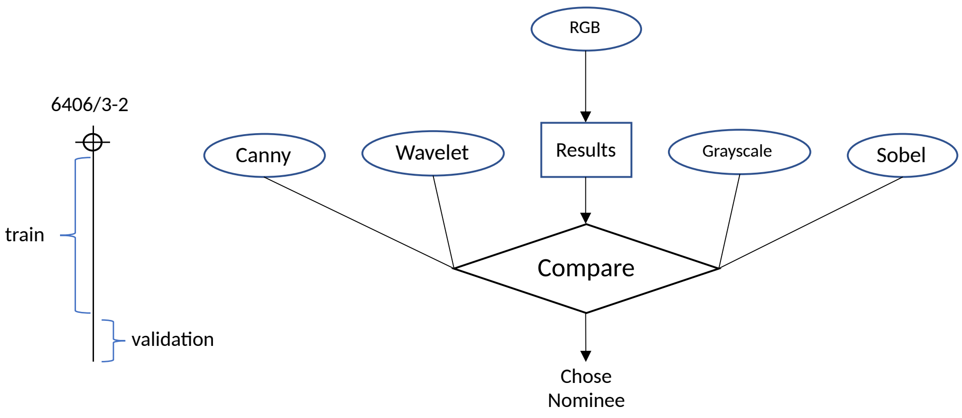


Figure 4.1: The set up for the initial benchmarking of the candidate models. The input to each candidate model is preprocessed using a different technique, from which one will be chosen as the nominee. The input to each candidate model comes from a 80/20 simple hold-out validation split of 289 images from well 6406/3-2.

4.1.1 RGB Model Evaluation

The first model is the *RGB* model, which was trained using the original RGB images. The *RGB* model was trained for 150k steps, and the training-validation loss can be seen in figure 4.2. Here, the training and validation loss refers to the total loss. The total loss is the sum of the losses in the RPN and Faster R-CNN classifier. The figure shows that

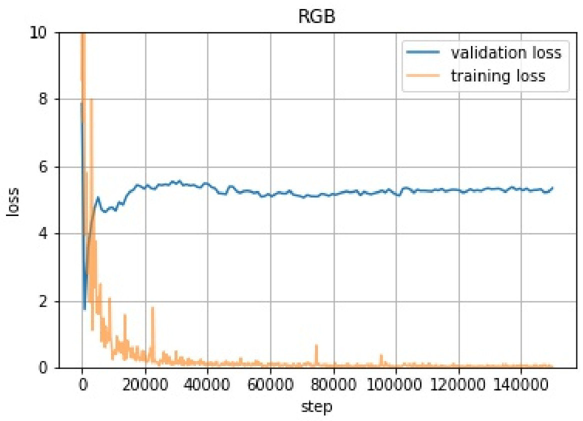


Figure 4.2: The training-validation loss for the *RGB* model.

the training loss is steadily decreasing until around step 60k. Past step 60k, the training loss shows little to no improvement. Evaluating the validation on the same interval, shows almost an immediate increase in the validation loss on the [0,20k] interval, before plateauing and converging towards a stable value, past step 60k. This is a clear sign of the model overfitting, where the loss on the training set is decreasing, while the loss on the validation set is increasing, i.e. the model has learned to map the core plugs in the training set, without the ability to generalize on the validation set. The training-validation loss for the *RGB* model shows that the model is initially moving towards a good fit, start overfitting almost immediately, and converges towards a stable high loss value.

The performance of the *RGB* model with respect to the 12 COCO evaluation metrics are summarized in figure 4.3. This figure shows the AP and the AR of the *RGB* model

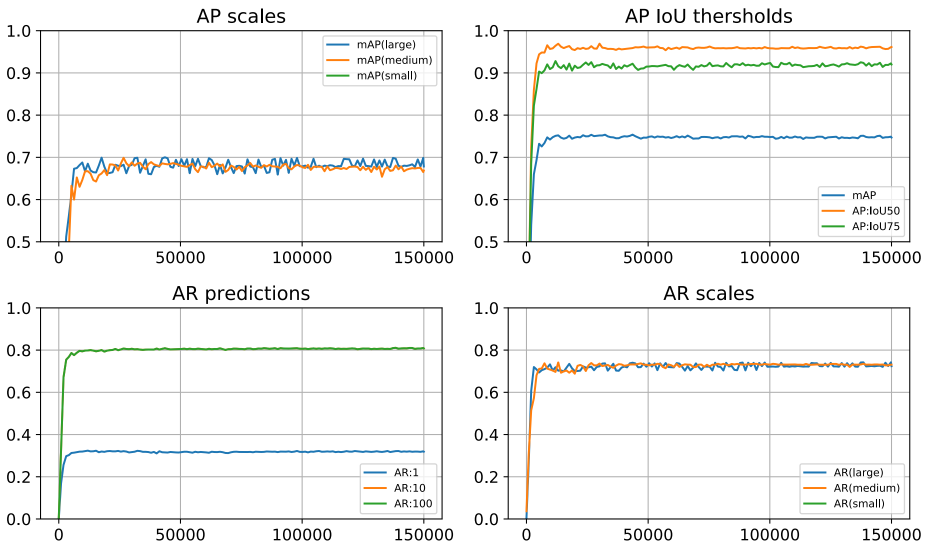


Figure 4.3: The COCO evaluation metrics for the *RGB* model.

on the validation set, according to the COCO evaluation metrics outlined in section 3.5.1. The AP IoU thresholds plot shows the average precision over the interval $IoU = [0.5 : 0.05 : 0.95]$, denoted mAP, and the average precision for the $AP^{IoU=50}$ and $AP^{IoU=75}$. The AR predictions plot shows the maximum average recall value given a fixed number of predictions. The AP and AR over the object size areas are given in the AP scales and AR scales plots, respectively. From the plots in figure 4.3, it is evident that there is a significant improvement occurring in the first 25k training steps with respect to the evaluation metrics. The evaluation metrics increases rapidly around [0,25k] interval, before reaching a value close to its maximum value on the entire training interval. This can be seen as the curves of the evaluation metrics flatten after step 25k, showing little to no improvement past this step.

The $AP^{IoU=50}$ is the highest scoring AP metric, followed by $AP^{IoU=75}$ and mAP in descending order with respect to their score. These scores at end of training (step 150k) are summarized in table 4.2. Since $AP^{IoU=50}$ has the most lenient IoU threshold for defining a true positive, any $IoU \geq 0.5$ is included in this metric. When the definition of a true positive is set at IoU=50, the model reaches an average precision $AP^{IoU=50} = 0.958$.

Metric	AP @ end of training
mAP	0.747
$AP^{IoU=75}$	0.920
$AP^{IoU=50}$	0.961

Table 4.2: The AP scores for the 2 IoU thresholds and the mAP for the *RGB* model at end of training.

This indicates that at this threshold, there are very few false positives being detected. As the threshold increases, a larger part of the detected core plugs will be defined as false positives, which is reflected in the lower AP score for $AP^{IoU=75}$ and mAP. This indicates that, of the detected core plugs, only a small portion of the false positives can not be accounted for in the evaluation metrics.

The average recall AR metrics for a set of a fixed number of detections are summarized in table 4.3 The average recall is a measurement of how well the model can detect the

Metric	AR @ end of training
AR^1	0.319
AR^{10}	0.808
AR^{100}	0.808

Table 4.3: The AR scores given a fixed number of predictions for the *RGB* model at end of training.

core plugs in the data set, i.e. the portion of the relevant objects in the data set that are identified as a true positive. The AR^{10} and AR^{100} values are identical, which indicates that the number of core plugs that can be detected in an image is not limited by the number of predictions the network makes, past AR^{10}. Further, AR^1 shows that allowing only a single detection is not sufficient to identify the core plugs in the image. This is to be expected since it is known that most of the core images contain more than 1 core plug. Allowing only one detection will leave out a significant portion of the relevant objects. Thus, AR^{10} and AR^{100} represent the model's average recall better and show that 80.8% of the core plug objects in the data set can be identified by the model.

The AP-scales and AR-scales plots in figure 4.3 show that the size of the objects in the data set falls into the medium and large object category according to the area sizes outlined in section 3.5.1. The metrics $AP(small)$ and $AR(small)$ have a score of -1 (not visible in the plots), which implies that no object in the data set falls into this size category. Since the variance of the size of the core plugs in a certain class is not expected to be large. These 6 object size metrics, AP scales, and AR scales in figure 4.3, are not as relevant to the task at hand. Therefore, they will only be introduced here and not be further elaborated on.

The COCO-evaluation metrics only show the model performance for precision and recall averaged across all classes in the data set. To better understand the model performance on the data set it is useful to also look at the performance in each class. The confusion matrices in figure 4.4 shows the class by class breakdown of the RGB model's predictions on the validation set for two IoU thresholds, $IoU = 0.5$ and $IoU = 0.95$, at confidence score

= 0.5. These matrices are called IoU50 and IoU95. The results build on the concept of IoU and confidence score as defined in section 2.4.1. The values in the confusion matrices have been generated by first accepting any predictions from the network with a confidence score ≥ 0.5. From these accepted predictions, if they have an IoU with the ground truth $IoU \geq 0.5$ they are counted as a true positive in both matrices. If the $IoU < 0.95$ they are counted as a false positive in IoU95, and as a true positive in IoU50. Any predictions with $IoU \leq 0.5$ will be regarded as a false positive in both matrices. The three first rows

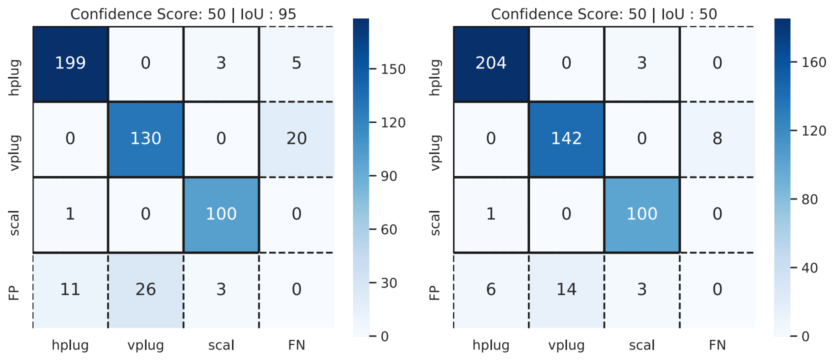


Figure 4.4: The confusion matrices for the *RGB* model at IoU50 and IoU95

and columns (solid line) show the number of predictions per class, where the number of correct predictions per class lies along the diagonal. The last column shows the number of plugs that were not detected in each class (false negatives) and the last row shows the number of detections that were not part of any class (false positives). The main difference between the matrices for IoU50 and IoU95 occurs in the *vplug* class. This shows that the *vplugs* are the most challenging class. This can be seen in the FP row in the IoU50 matrix. Here, 14 detections have been classified as false positives. This is to some degree related to the detected plugs having an $IoU < 0.5$, and are rejected as false positives. Further, some of the detected *vplugs* have an $0.5 \geq IoU < 0.95$ with the ground truth. This can be seen as the difference in false positives between IoU50 and IoU95, where 12 additional *vplugs* are counted as false positives in IoU95. The FN column shows the number of plugs in each class that was not detected by the model, i.e. the number of plugs that are defined as ground truth but not detected by the model. Similarly, the *vplug* class has the highest amount of false negatives, which is to be expected. If a plug in the ground truth is rejected and counted as a false positive due to low IoU, it will also be counted as a false negative. Using this information, it is clear that the false positives in the *vplug* class are not solely related to low IoU score, since some of the false positives can not be accounted for as detection in the other classes or as false negatives. This shows that the model detects 6 ”*vplugs*”, which are not in the ground truth. This could be related to either mislabeling, where some vertical plugs have been overlooked in the labelling process or that there are some natural features in the core, e.g. fractures or broken segments that are visually similar to the vertical core plugs in the data set.

From the values in the confusion matrices the precision and recall at both IoU thresholds are calculated according to equation 2.12 and 2.13 and are summarized in table 4.4. As indicated by the confusion matrices, the precision and recall values shows that the

	IoU50		IoU95	
	P	R	P	R
hplug	0.967	0.986	0.943	0.986
vplug	0.910	0.947	0.833	0.867
scal	0.943	0.990	0.943	0.990

Table 4.4: Summary of the precision-recall scores of the *RGB* model at end of training for each plug class at IoU50 and IoU95.

model seems to perform best on the *scal* and *hplugs*, where 99% of the *scal* plugs in the data set are identified with 94.3% precision at both IoU thresholds. The confusion matrices show that only two *scal* plugs in the data set have a $IoU < 0.5$ with the ground truth, and that all *scal* plugs are recalled. However, one of the *scal* plugs are misclassified as a *hplug* and three *hplugs* are classified as *scal* at both IoU thresholds.

In the *hplug* class, the model's precision/recall values are $96.7/98.6\%$ and $94.3/98.6\%$ at the low and high IoU thresholds, respectively. Note that this class, similar to the *vplug* class, also has a number of false positives that can not be accounted for as misclassifications or in the false negatives. This indicates that the *hplug* class may suffer from the same symptoms as the ones in the *vplug* class. However, due to the low variance with respect to the shape of the *hplugs* in the data set, this is not very likely. It is believed that the false positives in this category can mainly be attributed to cropping issues, similar to the case seen in figure 4.5. Since some cores are not aligned parallel to the height and width

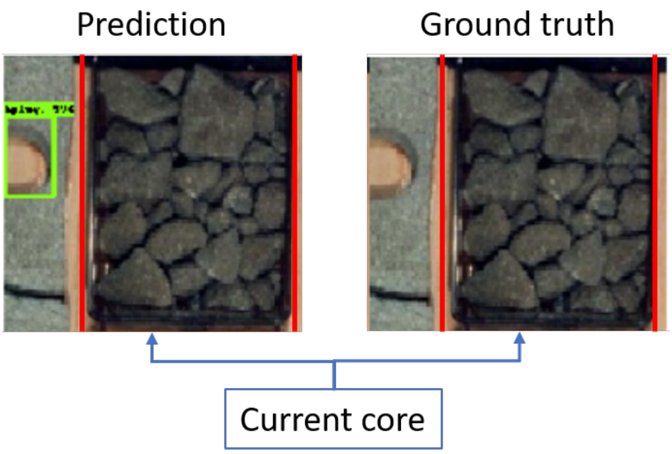


Figure 4.5: The prediction and ground truth from the *RGB* model for a segment of core in the validation set. The red lines show the boundaries of the current core, which contains no core plugs. The detected core plug (*green*) in the image belongs to the previous core.

of the image, there is an offset between the top and base of the core. In order to include the whole image within the crop, some of the neighbouring core might be included in the image of the current core. Since the core plugs of the neighbour core are not labelled, they are counted as false positives when detected by the model. Thus, the model might perform better on the *hplugs* than the precision score suggests, i.e. fewer false positives. The same concept could also apply to the *vplugs*, where the offset of the core brings the plugs of the neighbouring cores into the frame. However, due to the larger amount of variance in the *vplug* class in this data set, as outlined in 3.3. Some of the false positives may be attributed to both mislabeling and resemblance in visual appearance to non-plug features in the core as well.

In summary, the *RGB* model has the worst performance on the *vplug* class. This can be seen in the larger drop in precision and recall between the IoU50 and IoU95, relative to the other classes. As outlined in section 3.3, there is some variance to be expected in the visual appearance of the vertical core plugs, but this variance was expected to be lower than the variance in the *hplug* and *scal* class. However, since the variance in the *hplug* and *scal* class is low in this data set, it is believed that variance in the *vplug* class becomes more ”visible”. Thus, the performance on the *vplugs* will be worse, relative to the two other classes. These results show that the model is able to almost perfectly detect both the *scal* and *hplugs* in the low variance data set, and performs only slightly worse on the *vplugs*. However, from figure 4.2 it is evident that the model is overfitting, and it is believed that the performance will be significantly reduced as more variance is introduced, and the model is exposed to previously unseen data. This assumption is based on the low variance both in and between the visual appearance of the *hplug* and *scal* plugs in the training and validation data. And the low variance in the visual appearance between the *hplugs* in the training and validation data. The similarity between the core plugs in the training and validation set is attributed to the fact that the model is trained and validated on core images from the same well.

4.1.2 Comparing Candidates

Since the *RGB* model is overfitting, the performance is expected to drop significantly when introducing more variance in the data, and exposing it to previously unseen data (test set). Therefore, it was desirable to investigate if the overfitting behaviour could be counteracted by reducing the complexity of the input by applying the preprocessing techniques discussed in chapter 3 which gives rise to the four remaining candidate models.

The training-validation loss for these candidate models are shown in figure 4.6. Similar to the *RGB* model, after the initial steps, the four candidate models start overfitting before converging towards a stable high loss value. As seen in figure 4.6 the *canny*, *sobel* and *wavelet* model have been trained for fewer steps than the both the *gray* and *RGB* model. Due to the higher degree of overfitting, the training of these models was terminated earlier. This indicates that the preprocessing technique used in these models adversely affects the model’s ability to generalize on the validation set, i.e. removed some important features in the data. The validation loss of the *gray* and RGB models are essentially similar, and fluctuate about each other, before the *gray* model achieves a slightly lower loss than the *RGB* model. This could indicate that the conversion to grayscale may have successfully

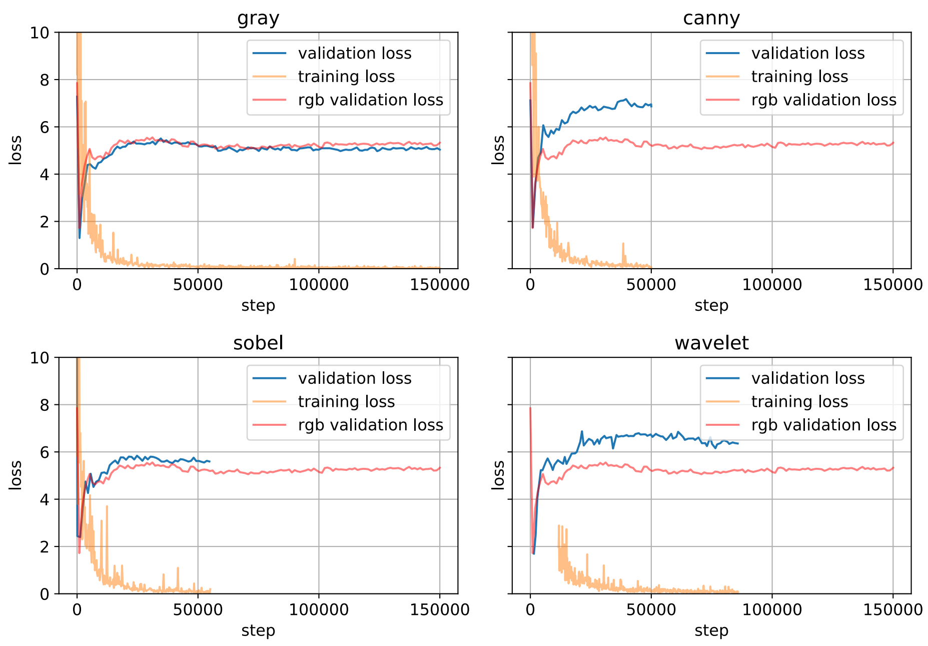


Figure 4.6: Training-validation loss for the *canny*, *wavelet*, *sobel* and *gray* candidate models.

removed some redundant complexity in the input, which has lead to increased generalization.

The 5 candidate models performance with respect to the COCO evaluation metrics plotted in figure 4.7. Here, only the main precision metric, mAP, and the recall metric AR^{100} are shown. Both the mAP and the AR^{100} shows that the *canny*, *sobel* and *wavelet*

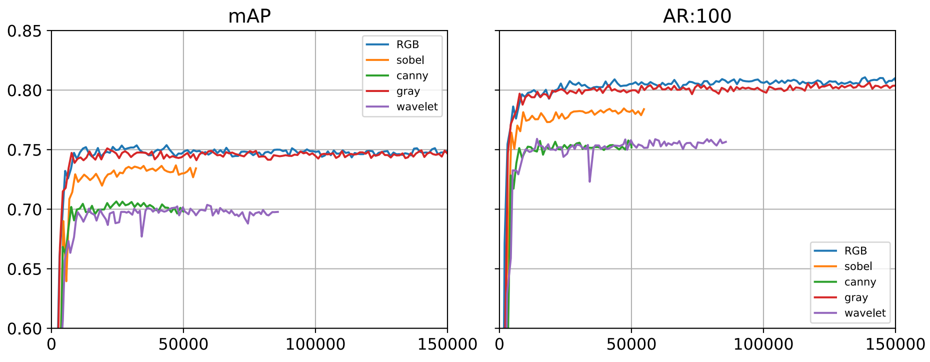


Figure 4.7: The mAP and AR^{100} for the 5 candidate models

models are performing significantly worse than the *RGB* model. The *gray* and RGB models seems to perform equally good on most of the interval. However, the *RGB* model has a slightly better performance with respect to AR^{100} on the entire interval and on the [0,50k] interval with respect to mAP. However, as the validation loss of the *gray* model becomes lower than the validation loss of the *RGB* model, the mAP of these models become similar.

The class by class breakdown of the 5 candidate models at IoU50 and IoU95 have been summarized in the scatter plot in figure 4.8. The confusion matrices for each model and the full numerical precision and recall scores for each model can be found in appendix C. The precision-recall plots show that the five candidate models performs relatively well in

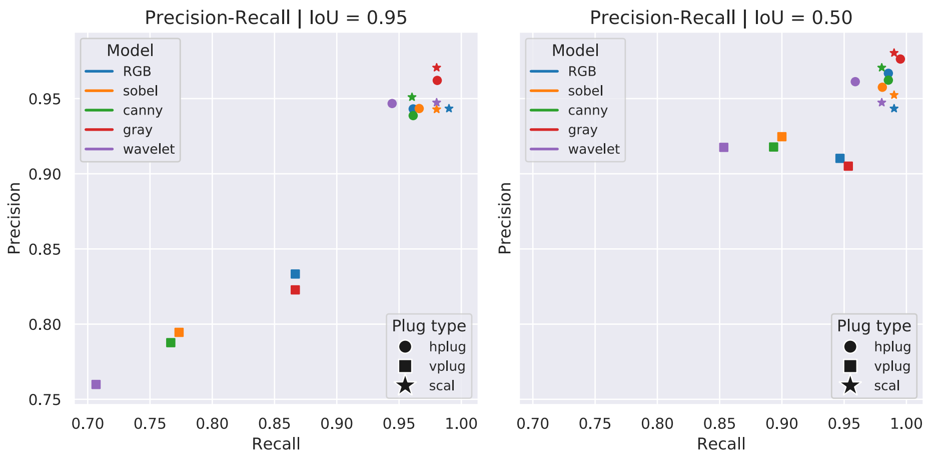


Figure 4.8: Precision-recall plot for the 5 candidate models.

both the *hplug* and *scal* category. This can be seen as the clustering of the *hplug* and *scal* categories for each model in the upper right corner. In some cases, models with a lower mAP even outperforms the *RGB* model, with respect to precision, recall or both. The *canny*, *sobel* and *wavelet* model achieve a higher precision, i.e. fewer false positives, in the *scal* class at IoU50. However, the precision score of the *canny* and *wavelet* models can not be directly compared to the *RGB* model due to their lower recall score. This reason is that the precision scores for these two models have been calculated over fewer detections.

The *gray* model is the only model that out-performs the *RGB* model with respect to both precision and recall at IoU=50 for the *hplug* class, and achieves a higher precision for the *scal* class at identical recall scores. At IoU95, the *gray* model achieves both a higher precision and recall score than the *RGB* model. This shows that the *gray* model is able to predict the *hplugs* with a better IoU with respect to the ground truth. Since fewer predictions in the *gray* model are discarded as false positives due to having an $IoU < 0.95$. For the *vplug* class it is clear that the *canny*, *sobel* and *wavelet* models are underperforming compared to the *gray* and *RGB* model. The difference in performance between the *gray* and *RGB* model, with respect to the *vplug* class is less obvious. At IoU50 the *gray* model has a lower precision score, but at a higher recall. However, at IoU95 the *RGB* model has a higher precision score than the *gray* model, at the same recall. This shows that the *gray* model may have a better performance at IoU50, but the additional *vplugs* detected at this threshold have an $IoU < 0.95$ with the ground truth and are not included at IoU95.

In order to better compare the performance of the models for each class at different precision-recall levels, the F1-score is calculated, at both IoU thresholds, according to equation 2.14 and plotted in figure 4.9. From these plots, the candidate models performance on the plug classes are much clearer. As expected from the precision-recall plots in figure 4.8, the *gray* model performs better than the four other models with respect to the *hplug* and *scal* class. Further, the *gray* model performs slightly better than the *RGB* model in the *vplug* category at IoU50, and slightly worse at IoU95. Although, the *canny*, *sobel* and *wavelet* model achieves a higher precision score in the *scal* category than the *RGB* model at both IoU thresholds, as seen in figure 4.8. The F1 score shows that only the *canny* and *sobel* model performs better at IoU50, and that the *RGB* model performs better than all three at IoU95.

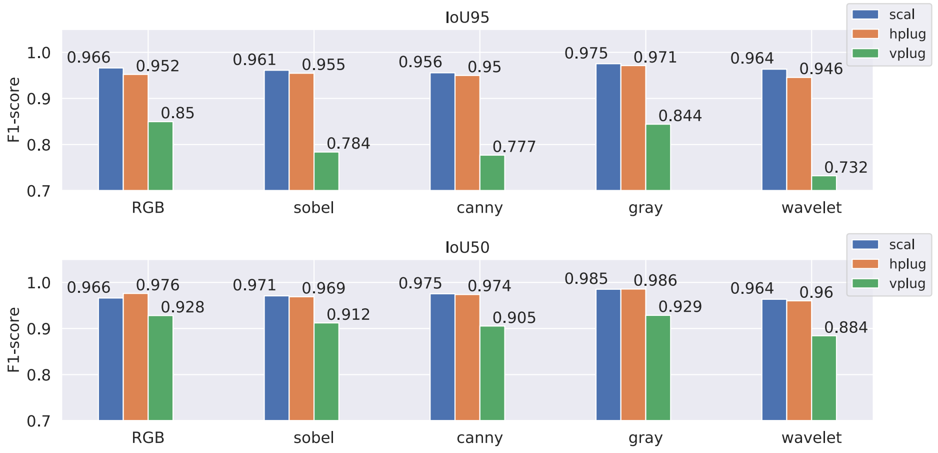


Figure 4.9: The calculated F1-score for the 5 candidate models.

From these results, it is evident that the *canny*, *sobel* and *wavelet* model significantly under-performs with respect to AR^{100} and mAP. Although, some of these models achieve a higher or similar F1 score compared to the *RGB* model in the *hplug* and *scal* class. The under-performance of these models in the *vplug* class is so large, that it significantly reduces both the AR^{100} and mAP of the models. This can be seen in figure 4.8, where the recall values show that a larger portion of the *vplugs* in the validation set can not be identified, i.e. more false negatives, even at IoU50. This under-performance could be attributed to several factors, but the most evident factor are the low relief core plugs.

If vertical core plugs do not sufficiently penetrate the $C - C'$ cross-section, as outlined in section 3.3, the relief of the vertical core plug in image view will be low. Low relief vertical core plugs are problematic in all 5 candidate models. However, both the gradient-based edge detection preprocessing techniques and the Ricker *wavelet* convolution struggled the most with the low relief plugs. Figure 4.10 shows a false negative *vplug* due to low relief in both the RGB and *gray* model. The red bounding box represents the

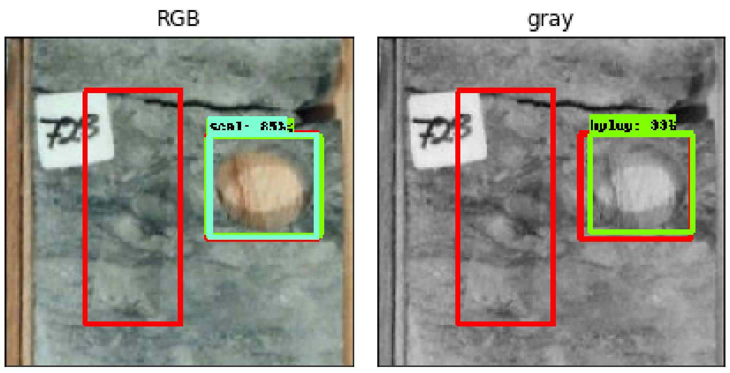


Figure 4.10: False negative due to a low relief vertical core plug in the RGB and *gray* model. Additionally, the *hplug* is classified as both the *scal* and *hplug* class in the *RGB* model, leading to a false positive as well. Bounding box colours: red - ground truth, green - *hplug*, turquoise - *scal*.

ground truth bounding box, i.e. the label. The turquoise and green bounding box corresponds to the models prediction of the *scal* and *hplug* class, respectively. In addition to not being able to classify the *vplug* (false negative), the *RGB* model classifies the horizontal core plug into both the *hplug* and *scal* class. This generates both a true positive for the *hplug* class and a false positive for the *scal* class, as can be seen in the confusion matrix in figure 4.4. However, this is an extreme case of a low relief vertical core plug, which can be easily missed even during the labelling process, unless the CCA report is consulted. Furthermore, in these extreme cases when the RGB and *gray* model failed, the *canny*, *sobel* and *wavelet* model would usually fail to detect the plug as well. Although, there were some rare cases where either the *canny*, *sobel* or *wavelet* model was able to detect a low relief vertical core plug that the *gray* and *RGB* model could not detect. The more common case was that the other way around.

Figure 4.11 shows a false negative in the *vplug* class for the *canny*, *sobel* and *wavelet* model from three different cores where the *gray* model have successfully detected said *vplug*. These examples are taken from different cores since there were no observed cases where all three edge-based preprocessing techniques would fail unless the RGB and *gray* would fail as well. However, the examples in figure 4.11 suggests that the false negatives could be related to the preprocessing technique removing or blurring of the vertical edge of the vertical core plug. In the *canny* example, the vertical edge is masked by the edges generated by the coarse grains within the core, which indicates lack of smoothing by the 5×5 Gaussian kernel and bad choice of thresholds in the hysteresis thresholding. The *sobel* example suggests that the pixel contrast between the interior of the vertical core plug and the outside is too low to generate an edge with large enough magnitude. Lastly, the

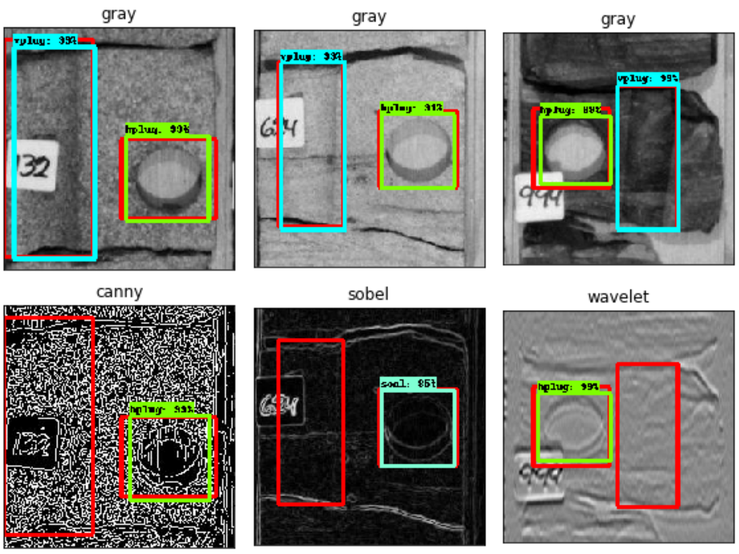


Figure 4.11: False negatives for the *vplug* class in the edge based preprocessing models that are not present in the *gray* model due to removal of horizontal edge. Bounding box colours: red - ground truth, *hplug* - green, *scal* - turquoise, cyan - *vplug*.

vertical edge is not visible in the *wavelet* example, since convolution has been applied in the vertical image direction using a low-frequency *wavelet* which blurs the slanted part of the core plug. As outlined in section 3.4, the output from these preprocessing techniques relies, to some extent, on some predetermined user-defined parameters. e.g. in the Canny edge detection algorithm, both the kernel size and the hysteresis thresholds are defined by the user. These parameters control the degree of smoothing applied to the image and the separation between the real edges and noise generated edges in the image, respectively. The number of false negative in these models suggest that these predetermined parameters are not general enough to have the desired effect across all training and validation examples.

Further, the vertical core plugs that can be identified in these models have a lower IoU with the ground truth than both the *gray* and *RGB* model. This can be seen by the larger difference in the precision-recall scores at the two IoU thresholds. Since a significant portion of the predictions have an $IoU < 0.95$, and are rejected as false positives. These models are now classifying true positives as false positives due to their poor overlap with the ground truth and are counted as false negatives. Thus, the number of false positives and false negatives increase, which in turn reduces the precision and recall, respectively. This is believed to be related to the importance assigned to an edge when determining the top and base of the vertical core plugs in these models. Figure 4.12 illustrates a very specific case where the edge based methods detects the vertical core plug with a lower IoU than the *gray* model. The elongated black clast below the core generates a large magnitude edge which is confused with the base of vertical core plug in the edge-based models. This

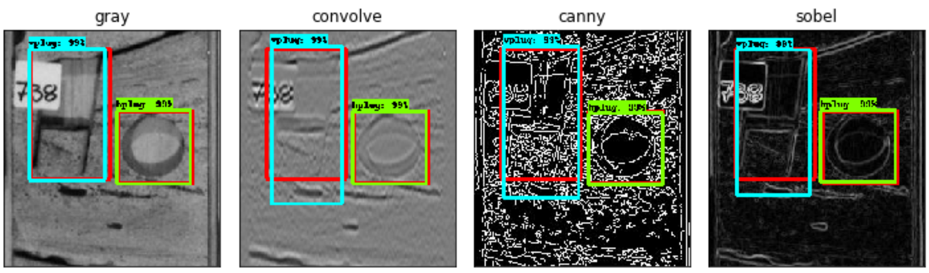


Figure 4.12: The effect of strong edges close to the base of the vertical core plugs on the edge based preprocessing techniques. Resulting in a poorer IoU with ground truth. Bounding box colours: red - ground truth, *hplug* - green, cyan - *vplug*.

leads to a lower IoU with the ground truth, and expect for in the *gray* model, the detected vertical core plugs in figure 4.12 will be rejected at IoU95. This is a very specific example, but it illustrates the presence of a stronger edge near the base of the vertical core plug will reduce the precision of the edge based models at higher IoU thresholds.

Additionally, these edge-based methods have lead to the detection of some false positives that are not present in either the *gray* or *RGB* model. Figure 4.13 shows an example for each preprocessing technique where a large grain or a clast has been misclassified as a horizontal core plug. This shows that an overemphasis on the shape, without consideration

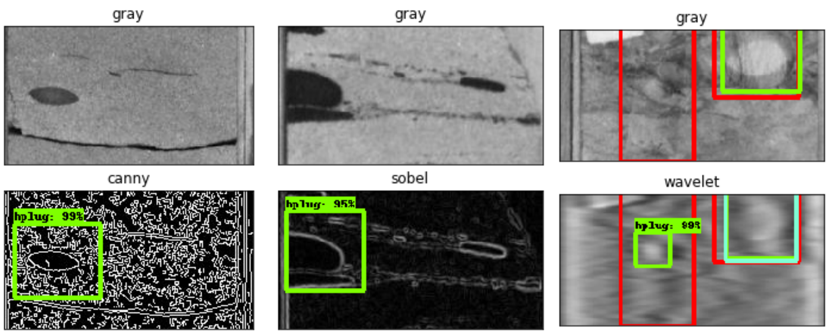


Figure 4.13: False positives in the *hplug* class related to the edge based preprocessing techniques. Clasts are misclassified as core plugs. Bounding box colours: red - ground truth, *hplug* - green.

for the context of its surroundings, can be a source of false positives in these models. In the *canny* and *sobel* examples the false positives are generated due to dark clasts that sets up a circular gradient. In this data set, the plywood background the core is placed on is often visible for the horizontal core plugs, which often has a higher pixel value than the core. Thus, the ingoing gradient will be positive and the outgoing gradient will be negative. The Sobel filter and the Canny edge detection algorithm does not account for the sign of the gradient, which could increase the risk of false positives when encountering circular shapes. The false positives generated by the *sobel* and *canny* model in figure 4.13 is not present in the *wavelet* model, which can be seen in appendix D. This is most likely related to that the *wavelet* model takes into account the sign of the gradient. However, false positives occur in this model as well as seen in the lower right image in figure 4.13, where a light clast generates a circular gradient with the same signs as the horizontal core plugs.

In summary, the edge-based preprocessing techniques show that there could some slight benefits in extracting the structural patterns in the core images, as can be seen in the slight increase in F1-score for the *scal* class in both the *canny* and *sobel* model at IoU50. However, it is clear from the results and the discussion above that this overemphasis on the edges of the core plugs, can be misleading and increases the risk of generating both false negatives and false positives. If the edges are too weak to be clearly imaged, or the noise generated edges can not be removed by the preprocessing technique with the current pre-defined parameters, they will not be detected by the model (false negatives), leading to lower recall as seen in figure 4.11. If the preprocessing techniques generate a core like shape as seen in figure 4.13, it may be misclassified as a plug (false positive) and reduce the precision. Further, this over-reliance on the edges to locate the core plugs causes strong edges close to the core plugs to be misinterpreted as the boundaries of the cores, as seen in figure 4.12, leading to a lower IoU with the ground truth. A combination of these effects is believed to be the reason why the increase in performance seen in (Abdillah et al., 2018), is not observed in this thesis. Since the images of the vehicles used in the classification task were taken from above by a traffic camera, resulting in a consistent background of black tarmac, making it easier to choose the pre-defined parameters for the preprocessing techniques which are general. Further, work outlined in Abdillah et al. (2018) is a classification task, which is not concerned with locating the objects. Thus, the performance is not affected by the noise generated edges being mistaken as object boundaries, leading to poorer IoU with the ground truth (figure 4.12) and false positives (figure 4.13). Based on these results the preprocessing techniques used in the *canny*, *sobel* and *wavelet* models are discarded.

The *gray* model out-performs the *RGB* model in both the *hplug* and *scal* class with respect to the precision-recall and F1 score at both IoU thresholds. Further, the performance on the *vplug* class is comparable between the two models, where the *RGB* model performs slightly better at IoU95. However, these effects seem to balance each other, since both models achieve almost identical mAP scores. Although, the *RGB* model has a slightly higher AR^{100} score it might, to some degree, be attributed to the overfitting. This can be seen in the confusion matrices for the *RGB* model in figure 4.4, where the three *hplugs* have been misclassified as *scal* plugs. Thus, the recall score will be higher due to the lack of false negatives, but at the cost of precision since three false positives are introduced. One of these false positives can be seen in figure 4.10. This does not occur in the *gray* model, which can be seen in the confusion matrices of the *gray* model in in appendix C. Here, two of the three *hplugs* are correctly classified and the third is not misclassified and kept as false negatives. This will lead to a lower recall, but not at the cost of precision. Since the AP^{100} is calculated at each IoU threshold in $IoU = [0.5 : 0.05 : 0.95]$ it is reasonable to assume that this occurs to some extent at each IoU threshold, leading to the higher $AR = 100$ for the *RGB* model. Further, the number of false positives that can not be accounted for in the ground truth increases by two for the *vplug* class in the *gray* model. This can explain the difference between the precision scores at IoU95.

The obtained results show that the conversion to grayscale may have removed some redundant complexity in the input, which could allow for better generalization, i.e. lower validation loss as seen in figure 4.6. This in turn may be the cause of the increased performance on the *hplug* and *scal* class in the validation set, and a comparable performance on the *vplug* class on this data set. Although, the *gray* model performs slightly better on this data set, the difference in performance between the *gray* model and *RGB* model is marginal. As seen in figure 4.7, the mAP for the *gray* and *RGB* models are almost identical, and are fluctuating about each other past step 100k. Since the class by class breakdown of the two models are based on their respective performance at the end of training, at step 150k. It is possible that the increase in performance in the *gray* model could be attributed to luck, i.e. if the *gray* model is evaluated at an upswing in mAP and the *RGB* is evaluated at a downswing. Further, the increase in performance in the *gray* model could be specific to this data set.

In order to verify this, the two models should be trained and evaluated on several data sets before any conclusions can be drawn. This test is not conducted in this thesis since both the manual labelling of new data and the training of several models are time-consuming tasks. This decision is based on the marginal improvement to the performance the conversion to grayscale showed on this data set, and the time it would take to prove if the increase to performance is in fact general. Therefore, it can not be concluded with any certainty that the conversion to grayscale improves the model's ability to detect core plugs in optical core images. However, the similarity between the *RGB* and *gray* model suggests either can be chosen as the nominee, without compromising the performance of the final model. Since the results suggest that the *gray* model may have removed some redundant complexity in the input, which allows for a lower validation loss, it is chosen as the nominee.

4.2 Hyperparameters Optimization

As outlined, the *gray* model showed to have the best performance with respect to the training-validation loss and the evaluation metrics of the 5 candidate models on the low variance data set, therefore it is chosen as the nominee. However, the *gray* model only performs better relative to the other models. The training-validation loss in figure 4.6 shows that the model is still severely overfitting early in the training interval, before converging towards a high validation loss. This shows that the model is still not able to generalize on the validation set. In this section, the results from the hyperparameter adjustments that were tried to reduce the overfitting will be presented. The considered hyperparameters are the second stage localization loss, λ_2, and the learning rate α. The experimentation with localization loss, λ_2, was performed on the *RGB* model, using the low variance data set. Of the two mentioned hyperparameters, it is believed that the λ_2 is the main cause of the overfitting. This assumption is based on the significant increase made to this weight, resulting in a (classification:localization)-loss ratio of (1:100). This is 50 times that of the ratio used by Huang et al. (2017a) (1:2), on which the default configuration file is based on.

Since the same overfitting pattern is present in all of the candidate models, and the 5 models were trained using the same λ_2, it is assumed that any improvement to the validation loss gained by adjusting this hyperparameter in one model can be translated to the other models. If successful, this hyperparameter will be changed in the nominee model (*gray*) as well.

The experimentation with the learning rate was performed on the *gray* model trained on both the high and low variance data sets combined.

4.2.1 Localization Loss

This section will investigate the effect of the localization loss weight on the model performance, which showed to increase the performance of the model developed in the specialization project, with respect to the COCO-evaluation metrics. Therefore, this section will mention some of these results from the specialization project, which can be found in appendix I.

As outlined in section 4.1, the candidate models were trained using a large weight in the second stage localization loss, λ_2, due to the performance gain with respect to the mAP and AR^{100} it showed to have in the specialization project. Figure 4.14 shows the training-validation loss, mAP and AR^{100}, for the initial ***RGB*** model presented in section 4.1 and a model trained on the same data set using the same hyperparameter setup from table 4.1, except in the latter model the weight of the localization loss is set to its default configuration $\lambda_2 = 2$. In order to distinguish between these models, the following notation will be adopted in this section:

- $RGB_{\lambda_2 100}$: the original ***RGB*** model trained using the hyperparameter configuration in table 4.1.

- item $RGB_{\lambda_2 2}$: the original ***RGB*** model trained using the hyperparameter configuration in table 4.1, with $\lambda_2 = 2$.

From these results, as seen in figure 4.14, it is evident that the main cause of the overfitting behaviour in the $RGB_{\lambda_2 100}$ model can be attributed to the high localization loss weight. In the $RGB_{\lambda_2 2}$ model, the severe overfitting occurring in the initial steps of the training interval have been resolved. The validation loss of this model is steadily decreasing on the [0,20k], before slightly increasing, and converging towards a much lower loss value. This shows that the $RGB_{\lambda_2 2}$ model have a higher generalization capability. As seen in the mAP and AR^{100} plots in figure 4.14, both the mAP and AR^{100} scores suggest that the $RGB_{\lambda_2 100}$ model is performing better across the entire training interval. However, it reaches the highest mAP on the [20k,60k] interval, which coincides with the highest validation loss. As the validation loss decreases, on the [60K,80K] interval, the mAP decreases as well. This indicates that the increase in the evaluation metrics in the $RGB_{\lambda_2 100}$ model is mainly related to overfitting with respect to the localization of the core plugs. This can be seen in figure 4.15, where the $AP^{IoU=50}$ and $AP^{IoU=75}$ for both models are plotted for the [0,80k] interval.

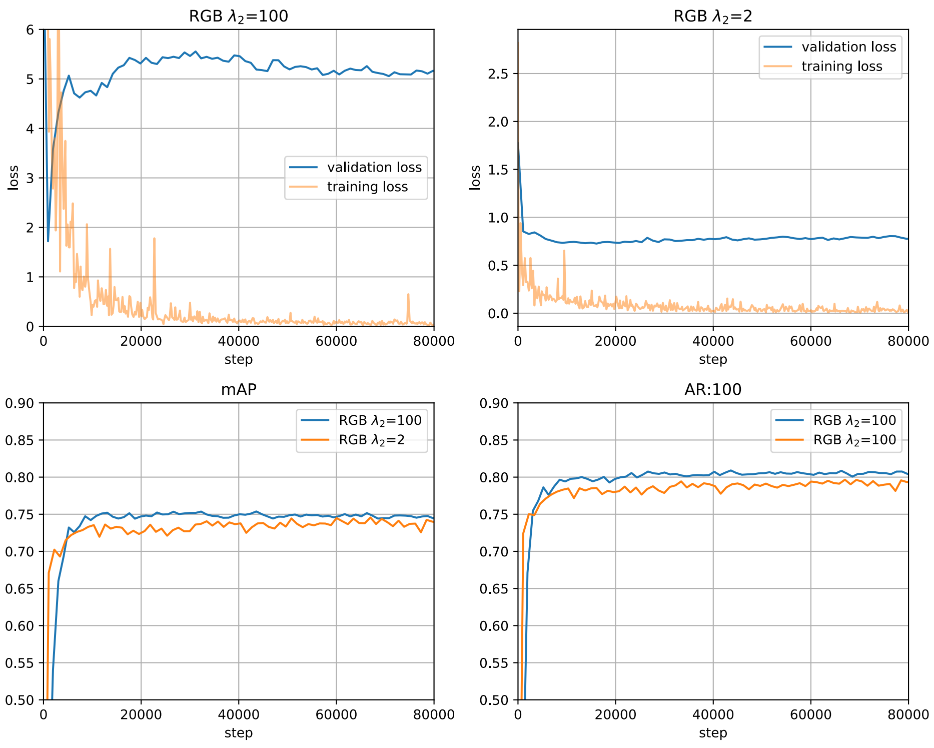


Figure 4.14: Training-validation loss, mAP and AR^{100} score for the $RGB_{\lambda_2 100}$ and $RGB_{\lambda_2 2}$ models.

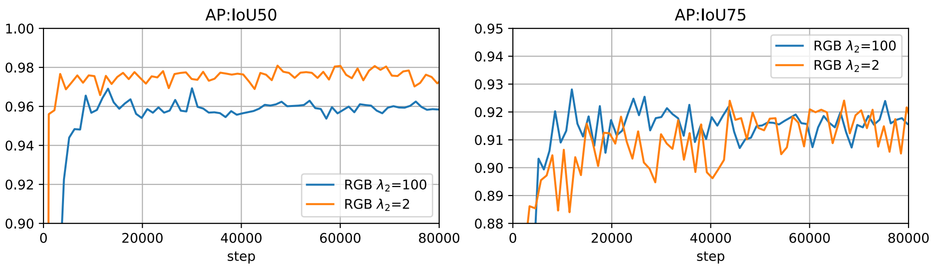


Figure 4.15: The $AP^{IoU=50}$ and AP^{IoU75} for the $RGB_{\lambda_2 100}$ and $RGB_{\lambda_2 2}$ models.

The $AP^{IoU=50}$ shows that the $RGB_{\lambda_2 100}$ model is performing worse with respect to average precision across the three plug classes at IoU50, compared to the $RGB_{\lambda_2 2}$ model with a lower localization loss weight. This indicates that the model with the lower localization loss weight detects fewer false positives at IoU50, which leads to a higher average precision across the plug classes. As the IoU threshold for defining a true positive increases to $AP^{IoU=75}$, both scores decrease and the difference between the two decreases as well. Past step 40k, the $AP^{IoU=75}$ scores can be regarded as equal. The $RGB_{\lambda_2 2}$ model have a greater drop in precision compared to the $RGB_{\lambda_2 100}$ model, when the IoU threshold increases. This suggests that the $RGB_{\lambda_2 2}$ model detects more true positives and fewer false positives than the $RGB_{\lambda_2 100}$ model, but the detected plugs have a lower IoU with the ground truth. Due to the high localization loss weight, the $RGB_{\lambda_2 100}$ model highly favours better localization, thus a larger portion of the detected core plugs have a higher IoU with the ground truth. This can be viewed as the model with the high localization loss weight increases its performance by refining the precision of the already detected core plugs in order to count them as true positives at higher IoU thresholds, while the model with low localization loss increases its performance by increasing the number of core plugs that can be correctly classified and located in the data set, but at a lower IoU with the ground truth.

The performance of the $RGB_{\lambda_2 100}$ model improves as the IoU thresholds increase, while the performance of the $RGB_{\lambda_2 2}$ model deteriorates by increasing the IoU thresholds. Thus, the calculated mAP will be slightly higher for the $RGB_{\lambda_2 100}$ model on the validation set. However, this emphasis for the $RGB_{\lambda_2 100}$ on localization comes at the cost of classification, as seen in the confusion matrix in figure 4.4, where three *hplugs* are misclassified as *scal* plugs, and eight *vplugs* are not identified (false negative). In this case, the number of false positives detected by the initial $RGB_{\lambda_2 100}$ is relatively small and has not affected the mAP to a great extent. This can be attributed to the low variance present within each plug class in this data set. As the variance increases, the number of false positives are expected to increase as well, which is expected to be detrimental to the mAP of the $RGB_{\lambda_2 100}$. Thus, it is preferable with a model that is less prone to detecting false positives when including the high variance data set in the training set and expanding to previously unseen data. As seen in figure 4.16, the confusion matrix for the $RGB_{\lambda_2 2}$ model, no false positives are generated due to misclassification of the core plugs at both IoU thresholds, i.e. every detected core plug is correctly classified. Further, the number of

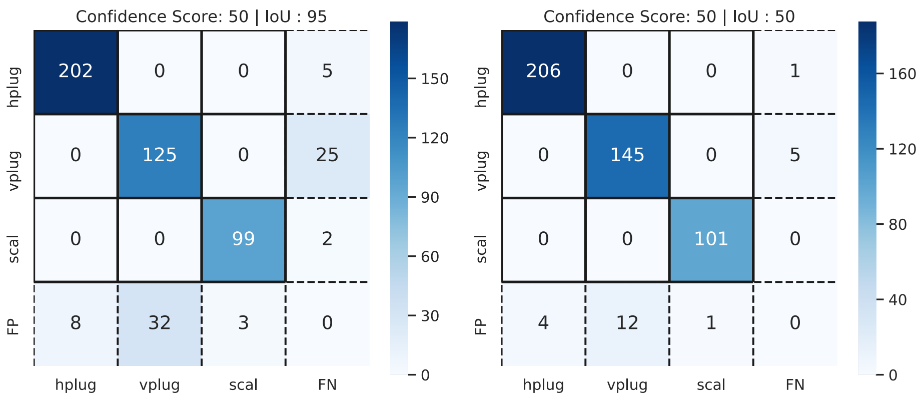


Figure 4.16: The confusion matrices at IoU50 and IoU95 for the $RGB_{\lambda_2 2}$ model

false negatives in the *vplug* class at IoU50 are fewer.

Since the detected core plugs in the $RGB_{\lambda_2 2}$ model have a lower IoU with the ground truth, a greater number of these plugs will be regarded as false positives as the IoU threshold increases. This will affect the AR^{100} as well, since this metric is also averaged across the IoU thresholds IoU=$[0.5 : 0.05 : 0.95]$.

These results show that the choice of the localization loss weight must be based on the object detection task at hand. As outlined in section 2.5, in the specialization project of this thesis, object detection was used to locate and automatically crop the cores from the optical core images. Thus, it was important to locate the cores with a high IoU with the ground truth, so that the entire core would be included in the crop. Although, a similar overfitting behaviour was present in the specialization project the increase to the mAP was significant ($+9\%$), as mentioned in section 4.1, when compared to the performance increase observed in the results in figure 4.14. This is mainly related to the object detection task in the specialization project was a single-class object detection task, i.e. the objects were classified as either a core or nothing. Therefore, there were few opportunities for the model to generate false positives as a result of misclassifications. Since the risk of generating false positives was low, the localization weight could be increased in order to push a larger amount of the detections towards being valid at higher IoU thresholds without compromising the model performance at the lower IoU thresholds.

The results in figure 4.14 show that increasing λ_2 can not be applied with the same success as in the specialization project. Since this is a multi-class task, the risk of generating false positives due to misclassification is much higher. Additionally, due to the similarity in visual appearance between some of the core plug classes, this risk is further increased. Although, locating the objects are important in this task, it is not as essential as it was in the specialization project. This task requires a better balance between localization and classification. Therefore, the localization loss for the final model is set to $\lambda_2 = 2$, since it allows for detecting and correctly classifying a larger portion of the core plugs at an acceptable IoU, rather than fewer at an excellent IoU.

4.2.2 Further Discussion on Localization

The results presented in section 4.2.1 showed that the over-emphasis on locating the core plugs, i.e. higher weighting of the localization loss term, resulted in the detection of fewer plugs, albeit at a higher IoU with the ground truth. From the discussion in the previous section, it is concluded that localization of the objects in this task is important, i.e. accurately locating the core plugs. However, it is not as important as it was in the specialization project. In addition to the previously outlined points, the definition of the boundary of the objects plays an important role when deciding on which evaluation metric best represents the model's ability to locate the core plugs.

When labelling the core plugs in the data set, the boundary between the objects (core plugs) and non-objects (background) needs to be defined, i.e. which pixels constitute the core plugs and which do not. In this task, the definition of this boundary is not clearly defined. This can be attributed to low relief features or damaged core plugs, which requires a decision to be made for where the boundary is drawn. This can be regarded as the bias of the person labelling the data. Figure 4.17 shows predictions with lower IoU scores that are the results of the labelling bias.

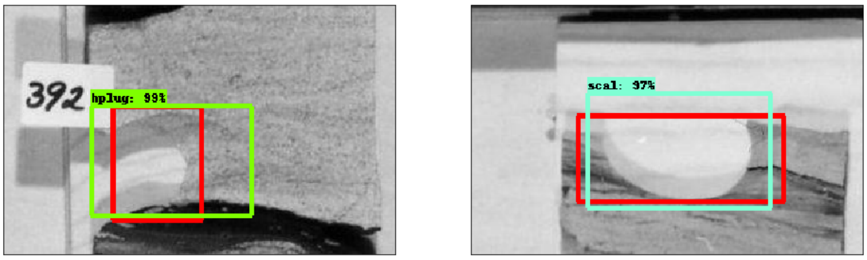


Figure 4.17: Low IoU predictions that are regarded as true positives at lower IoU thresholds. The low IoU scores can be attributed to the labelling bias. Bounding box colors: *hplug* - green, *ground truth* - red, *scal* - turquoise.

The left and right images show the difference between what has been labelled as the core plugs (ground truth - red) and what the model predicts as the core plugs. Although, these predictions yield lower IoU scores, they are still able to locate the core plugs with sufficient accuracy for the task of giving an estimated depth location of the core plug. In the left image, the low IoU score is the result of poor labelling, where the ground truth has been drawn slightly too small and not included the low relief part of the core plug. Similarly, there may be cases where the opposite occurs. Where the ground truth includes the low relief part, while the prediction does not. The right image shows an example of a damaged core plug, where there is a difference in where the upper boundary of the core plug is defined in the ground truth and the model's prediction. Since the definition of the boundary of the core plugs (objects) are not as clear, i.e. where the object starts and ends, examples like these will be present throughout the data set. This, in turn, will result in a larger portion of the core plugs having a low IoU with the ground truth, although they are able to locate and classify the core plugs.

These examples illustrate that having the highest IoU score is not necessarily a prerequisite for solving the task at hand. Thus, the strictest COCO-evaluation metric, mAP, might not fully describe the model's ability to solve the task. The examples in figure 4.17 would be regarded as false positives at higher IoU thresholds, although they can both locate and classify the core plugs. This suggests that $AP^{IoU=50}$ and $AP^{IoU=75}$ might be better metrics to evaluate the model's ability to solve the task at hand, or at least be regarded as equally significant to the mAP.

4.2.3 Learning rate

The initial model used in this section was trained and validated using the data from both the low and high variance data sets from well $6406/3-2$ and $6406/8-1$. The input data was converted to grayscale according to the findings in 4.1.2 and the model was initialized with the hyperparameters in the default configuration file, as outlined in Huang et al. (2017a). A summary of the main hyperparameters and the learning rate used to initialize the model are given in table 4.5. The model initialized with the default hyperparameters in table 4.5 was trained over 20k steps, recording the validation loss.

The learning rate analysis was conducted by training several version of this default

Optimization function	Momentum optimizer
Moment optimization value	0.9
Learning rate (constant), α_{init}	0.0003
Second stage localization loss weight λ_2	2
Second stage classification loss weight	1
Data augmentation	random horizontal flip

Table 4.5: Summary of the hyperparameters used in the learning rate analysis.

model with varying learning rates, with the goal of finding an optimal learning rate. The learning rates were chosen by multiplying the initial learning rate, α_{init}, by a factor of 3 until no further improvement could be recognized in the validation loss. The learning rates in the initial analysis are summarized in table 4.6 The validation losses using these

Learning rate	$C\alpha_{init}$	Value
α_{-1}	$3^{-1}\alpha_{init}$	0.0001
α_0	$3^0\alpha_{init} = \alpha_{init}$	0.0003
α_1	$3^1\alpha_{init}$	0.0009
α_2	$3^2\alpha_{init}$	0.0027
α_3	$3^3\alpha_{init}$	0.0081

Table 4.6: Summary of the learning rates used in the learning rate analysis.

learning rates can be seen in figure 4.18 Since the validation loss is only calculated at a

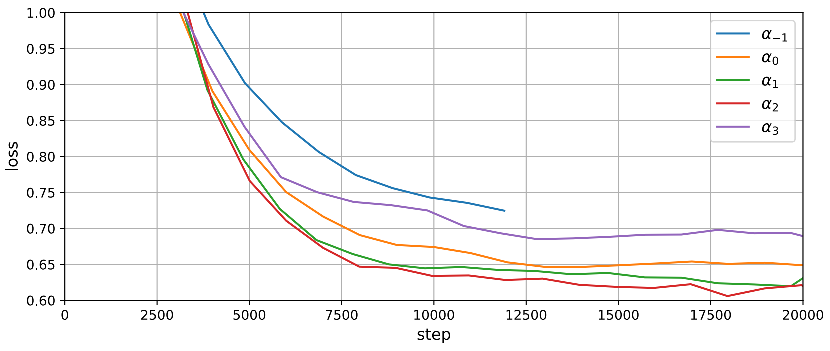


Figure 4.18: The smoothed validation losses for the different learning rates from table 4.6, using exponential smoothing with a smoothing factor of 0.6.

fixed time interval, approximately at every 1000 steps. The number of samples for the validation losses are few, i.e. approximately 20 samples per validation curve in figure 4.18. Since the model is trained with stochastic gradient descent with a batch size of one, both the training and validation loss curves will appear spiky, from which it is difficult to extrapolate any trends on a short training interval. In order to make the plot easier to read without increasing the training time, the validation losses have been smoothed using an exponential moving average. The validation losses show that α_0,α_1 and α_2 are all acceptable choices for the learning rate. They are steadily decreasing at an acceptable rate, and stabilize around a reasonably low loss value. Both α_1 and α_2 achieves lower loss values at faster rates than α_0, which suggests that the initial learning rate might be too low. This could lead to both longer convergence time and getting stuck at a local minima. Further, α_2 reaches the lowest loss value, at the fastest rate, which suggests that it closer to the optimal learning rate.

It is clear that learning rate α_{-1} is too low, which can be seen by the slower decline in loss value. Using this learning rate increases the risk of the model getting stuck at a sub-optimal solution, i.e. a local minima. Even if this should not occur, the time needed to reach the same loss as the larger learning rates is not optimal. Thus, this training-validation run was terminated after 12000 steps. The training-validation run using α_3 shows the opposite case, where the learning rate is too large. Further, the slope of the validation loss in α_3 starts to decrease faster and stabilizes at a higher loss value than the validation loss in α_1 and α_2, which indicates that the learning rate is overshooting the minima, causing the model to stabilize at a higher loss value.

From this initial analysis the optimal learning rate, α_{opt} could be in the range $\alpha_1 \leq \alpha_{opt} < \alpha_3$. However, the significant increase in validation loss from α_2 to α_3 suggests that the optimal learning rate is closer to α_2. Since the difference between α_1 and α_2 is small, and they are intersecting towards the end of the training interval. These were trained for an additional 30k steps until their respective validation losses stabilized and no significant improvement in the validation losses were observed. Figure 4.19 shows the unsmoothed validation losses using α_1 and α_2 and a box plot for the loss values. Both

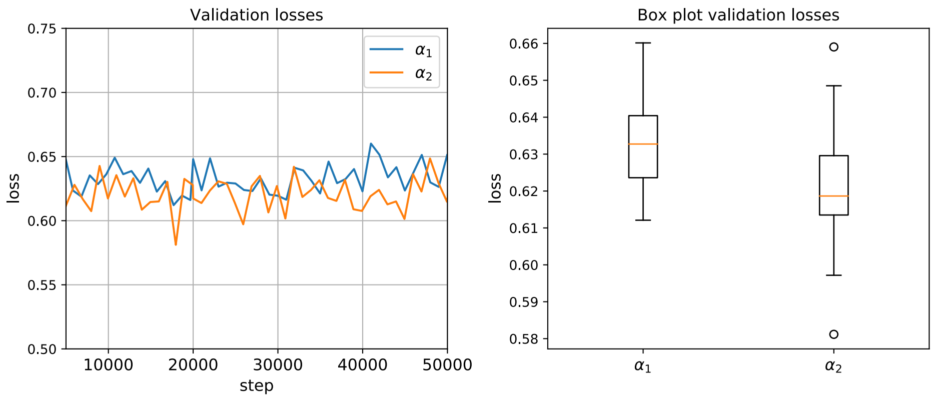


Figure 4.19: The validation loss for the model with α_1 and α_2 on the [5k,50k] interval. The spread of the validation losses achieved on the interval for the two learning rates are shown in the box plot. The circles marks the outlier values for α_2.

the validation loss plot and the box plot illustrate the similarity between the two learning rates. However, there is a slight preference for α_2, since the median of the validation loss is lower. Further, over 75% of the calculated validation losses on the [5k,50k] interval for α_2 is lower than the median of α_1. Expect, for the outlier value at 0.66, the range between the upper quartile and the maximum value is slightly smaller for α_2. These results suggest that α_2 is a slightly better choice of learning rate. Since it converges towards a minimum with a lower median value at a faster rate. From the validation losses, it can be inferred that there is no significant improvement in the validation loss by increasing the training past 10k steps.

4.3 Final Model

The final model is trained on the combined data set (high variance + low variance), which has been converted to grayscale, using a constant learning rate, α_2 from table 4.6 with a second stage localization loss, $\lambda_2 = 2$. The hyperparameters for the training-validation are summarized in table 4.7. The training-validation was initialized with the hyperparameters

Optimization function	Momentum optimizer
Moment optimization value	0.9
Learning rate (constant), α_2	0.0027
Second stage localization loss weight λ_2	2
Second stage classification loss weight	1
Data augmentation	random horizontal flip

Table 4.7: Summary of the hyperparameters used in the training of the final model.

in table 4.7 and trained for 20k steps. The training-validation losses are summarized in figure 4.20. Since the lowest validation loss occurs at step 10k, these weights are saved and will be used for testing. The full evaluation of the performance on the validation set for this initial run can be found in appendix E. The performance with respect to the AP metrics and the AR^{100} are summarized in table 4.8. Additionally, the model is initialized 4

Metric	@ step 10k
mAP	0.707
AP^{75}	0.866
AP^{50}	0.953
AR^{100}	0.758

Table 4.8: Summary of the most important COCO-evaluation metrics for the initial run of the final model.

more times and trained for 10k steps, saving the weights and the end of training, which will be used to evaluate the model's performance on the test set as well. These runs are used to calculate the mean, standard deviation and the standard deviation of the mean (standard error), where the standard error is given as (Walpole et al., 2012):

$$\delta\hat{\mu} = \frac{\hat{\sigma}_{\hat{\mu}}}{\sqrt{n}} \tag{4.1}$$

where n is the number of samples in the population and $\hat{\sigma}_{\hat{\mu}}$ is the standard deviation.

The score for the COCO-evaluation metrics for each run on the validation set can be found in appendix E. The mean, standard deviation and the standard error of the model's performance with respect to the COCO-evaluation metrics for the five runs are summarized in table 4.9. This shows there is some variance between each individual run, which can

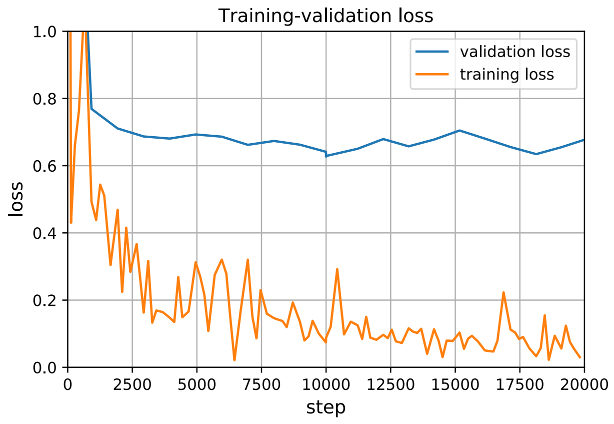


Figure 4.20: The training and validation loss which the early stopping criteria was based on.

Precision metrics	$\hat{\mu}$	$\hat{\sigma}_{\hat{\mu}}$	$\delta\hat{\mu}$	**Recall metrics**	$\hat{\mu}$	$\hat{\sigma}_{\hat{\mu}}$	$\delta\hat{\mu}$
mAP	0.669	0.0097	0.00434	AR^{100}	0.719	0.0086	0.00385
$AP^{IoU=75}$	0.840	0.02043	0.00914	AR^{10}	0.719	0.0086	0.00385
$AP^{IoU=50}$	0.956	0.0038	0.0017	AR^{1}	0.318	0.0055	0.00248

Table 4.9: The mean ($\hat{\mu}$), standard deviation ($\hat{\sigma}_{\hat{\mu}}$) and the standard deviation of the mean ($\delta\hat{\mu}$) (standard error) for the COCO-evaluation metrics for the final model.

be attributed to the stochastic nature of the training neural networks, i.e. the order the model reads the training-validation data and the random initialization of the weights that are fine-tuned (starting point of the momentum optimization GD). However, comparing the average performance in table 4.9 with the performance of the initial run in table 4.8. The differences are marginal, which can be seen in the difference in both the numeric value of the metrics and the standard error. Since the results are not normally distributed the standard deviation and the standard error of the results in the following sections should be regarded as a quantitative description of the uncertainty, and not as a symmetric ($\pm$) range about the mean (Brown, 1982).

As expected, there is a drop in performance when the high variance data set is included. This is reflected in all evaluation metrics in table 4.9. Some metrics are affected to a greater extent. The most evident drop occur in the mAP, AP^{75} and the two average recall metrics, AR^{10} and AR^{100}. These metrics indicate that there are a larger portion of the plugs in this combined data set that are not detected, and those that are detected have a lower IoU score with respect to the ground truth. However, as outlined in section 4.2.1, the more important metric with respect to the task at hand, is the $AP^{IoU=50}$, since it is preferable to detect a larger portion of the core plugs at an acceptable IoU, rather than fewer plugs at an excellent IoU. From table 4.9, the model performance with respect to the $AP^{IoU=50}$ is still high and 95.6 of the core plugs in the data set are identified at this IoU threshold. This shows that the model is still able to perform well for the intended purpose. Due to the small difference between the average performance of the model and the single initial run, the class by class breakdown of the precision-recall scores are conducted using the weights of the initial model. The confusion matrices for the two IoU thresholds can be seen in figure 4.21. As

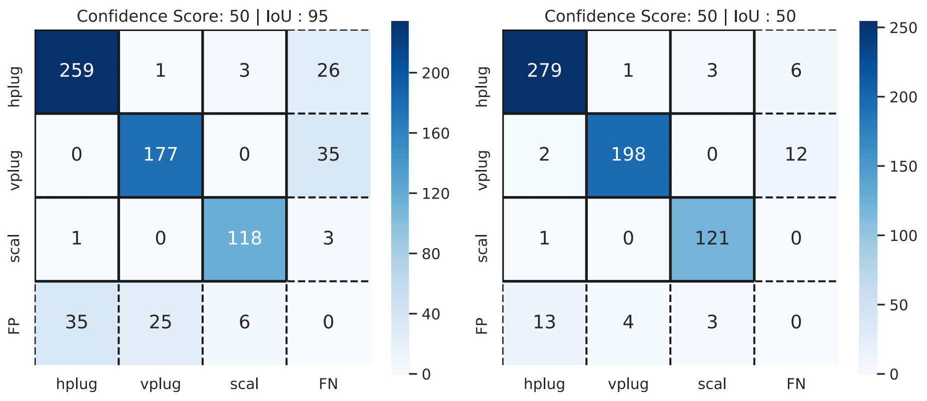


Figure 4.21: The confusion matrices at IoU95 and IoU50 for the initial run of the final model at step 10k.

indicated by the AP metrics, it is clear that the model is unable to identify a large number of the core plugs at higher IoU thresholds, as indicated by the increase in false positives and false negatives in the confusion matrices from IoU50 to IoU95. Further, there is an increase in the number of false positives that are attributed to misclassifications. Similar to the low variance data set, plugs in the *hplug* class are misclassified as the *scal* class and vice versa. Additionally, the introduction of bedding parallel core plugs that are non-circular causes false positives to arise between the *hplug* and *vplug* classes as well. Further, there are 12 false negatives that can not be accounted for as misclassifications (2 *hplugs* + 10 *vplugs*) at IoU50, which shows that there are some core plugs that the model is unable to identify. However, the performance of the model is still acceptable at IoU95 and close to perfect at IoU50, which can be seen in the precision-recall score in table 4.10.

The precision-recall scores shows that more than 93% of the core plugs in each class in the data set can be identified, and the precision score is greater than 94% at IoU50. This high performance at IoU50 is also reflected in the $AP^{IoU=50}$ score for the model in table 4.8. Similarly, at IoU95 at least 83.5% of the core plugs in each class can be identified with a precision score greater than 87.2%. The largest drop in performance between the IoU thresholds occur in the *hplug* and *vplug* classes. The drop in performance in the *vplug* class is mainly related to the same causes as outlined earlier in this chapter, e.g. low relief vertical core plugs. Since the size of the data set is increased, the likelihood of

	IoU50		IoU95	
	P	R	P	R
hplug	0.946	0.965	0.878	0.896
vplug	0.975	0.934	0.872	0.835
scal	0.953	0.992	0.929	0.967

Table 4.10: Summary of the precision-recall scores of each plug class at IoU50 and IoU95, for the initial run of the final model at step 10k.

encountering these problematic vertical core plugs increases, which in turn increases the number of false negatives. Therefore, the main cause of the drop in performance in the COCO-evaluation metrics is mainly believed to be related to the *hplug* class, due to having higher variance in visual appearance. There is a small decrease in precision-recall for the *scal* class as well. This is also believed to be attributed to the higher variance for the core plugs that are extracted parallel to the bedding. However, this decrease is less pronounced and can be related to the fact that there are fewer 1.5 inch plugs in the data set, compared to the horizontal CCA plugs. This reduces the influence of the poorly predicted plugs, of the *scal* class, on the average precision scores. Further, the amount of false positives in the *scal* class at both IoU thresholds suggests that the variance within this class is less compared to the *hplug* class.

These obtained results show that the model is able to reasonably detect the core plugs in the validation set with respect to the strictest evaluation metrics, achieving a $mAP = 66.9\%$ at $AR^{100} = 71.9\%$. At the more lenient metrics the model performance is significantly better, and almost perfect at the most lenient metric $AP^{IoU=50}$. Achieving an $AP^{IoU=75} = 84.0\%$ and an $AP^{IoU=50} = 95.6\%$. From the class by class breakdown with respect to the recall metric, it is clear that the model is able to identify almost every core plug in each class at IoU50. As outlined in section 4.2.2, it is argued that that an $IoU = 50$ and $IoU = 75$ with the ground truth might be sufficient to solve the task at hand. Judging the models ability to solve the task by these metrics, will result in a model with close to perfect performance at $IoU = 50$ when evaluated on the validation set.

4.3.1 Testing

Finally, the model is tested on the test sets. This is done to provide an unbiased estimate of the model's performance. Two data sets have been prepared by labelling core images from wells that were not present in either the training or validation data. The first data set consists of 63 labelled images from the wells 6406/3-3 (32) and 6407/1-3 (31). The second data set consists of 67 images that are randomly sampled from the remaining 25 wells in the data set, i.e. from the wells that are not part of the wells used in the training-validation set. These data sets can be regarded as having two different levels of difficulty. The first data set consists of images from only two wells, chosen based on the visual appearances of the core plugs in these well, which are relatively similar to some of the core plugs in the core images used in the training and validation of the model. This was done to measure the model's performance when the variance in visual appearance between training-validation data and test data is kept relatively low. Since it is attempted to keep the visual appearance low, this test set can be regarded as less challenging, and will be referred to as the *easy* test set.

The second test set is randomly sampled from the remaining 25 wells, putting no constraint on the variance in the visual appearance of the core plugs, which will make this test set more challenging. This test set will be referred to as the *hard* test set. In the testing, the weights of each run of the final model at step 10k are used to evaluate the model performance on both the *easy* and *hard* test set. The performance with respect to the most important evaluation metrics for each run with the calculated mean, standard deviation and standard error for both the *easy* and *hard* test set are summarized in table 4.11 and 4.12, respectively. The performance with respect to the full COCO-evaluation metric for each run, with the calculated statistics can be found in appendix F.

From the large drop in the evaluation metrics, it is evident that the mapping of the relationship between input to output learned on the training-validation set is not fully able to capture the same relationship in the test sets. As expected, the decrease in performance is greatest in the *hard* test set, due to the higher variance in visual appearance, where the model achieves a $mAP = 36.4\%$ at an $AR^{100} = 41.5\%$. Although, the performance on the *easy* test set is slightly better, there is a significant decrease in performance on this data set as well. On this data set the model achieves a $mAP = 41.4\%$ at an $AR^{100} = 46.6\%$. A similar trend can be seen in the more lenient evaluation metrics, where the performance on the *easy* set is slightly better than the *hard* set. However, the decrease in performance on both sets at the most lenient metric is significant as well, where only 68.22% and 58.6% of the located core plugs have an $IoU = 50$ with the ground truth. This shows that a significant portion of the core plugs in both data sets are not detected, or detected with an IoU score lower than the most lenient IoU threshold. The difference between the

Metric	run1	run2	run3	run4	run5	$\hat{\mu}$	$\hat{\sigma}_{\hat{\mu}}$	$\delta\hat{\mu}$
mAP	0.414	0.404	0.377	0.419	0.378	0.3984	0.01774	0.00355
$AP^{IoU=50}$	0.696	0.696	0.654	0.699	0.666	0.6822	0.01855	0.00371
$AP^{IoU=75}$	0.465	0.453	0.403	0.462	0.382	0.433	0.03396	0.00679
AR^{100}	0.466	0.46	0.429	0.471	0.438	0.4528	0.01639	0.00328

Table 4.11: Condensed COCO-evaluation metrics for the 5 runs on the *easy* test set, with the mean, standard deviation and the standard deviation of the mean (standard error).

Metric	run1	run2	run3	run4	run5	$\hat{\mu}$	$\hat{\sigma}_{\hat{\mu}}$	$\delta\hat{\mu}$
mAP	0.364	0.387	0.372	0.34	0.352	0.363	0.01617	0.00323
$AP^{IoU=50}$	0.572	0.63	0.597	0.562	0.569	0.586	0.02497	0.00499
$AP^{IoU=75}$	0.428	0.472	0.424	0.378	0.388	0.418	0.03332	0.00666
AR^{100}	0.415	0.437	0.423	0.39	0.403	0.4136	0.01617	0.00323

Table 4.12: Condensed COCO-evaluation metrics for the 5 runs on the *hard* test set, with the mean, standard deviation and the standard deviation of the mean (standard error).

$AP^{IoU=75}$ and $AP^{IoU=50}$ shows that a larger number of the detected core plugs are in the IoU range $75 > IoU \geq 50$ compared to in the validation set. Further, the low mAP

scores indicate that the plugs in this range are closer to $IoU = 50$ than $IoU = 75$.

Although, the model has the ability to learn from the training set and generalize that learning on the validation set, indicated by the low validation loss and the high performance as seen in table 4.9. The results in this section show that this is not the case for the test sets. This indicates that the core plugs in the data set used to train the model are not representative of core plugs in general. This can be attributed to the high variance in the visual appearance of core plugs in optical core images and the variance in the visual appearance of the core plugs between the wells. This is illustrated by the significantly higher performance on the validation set. Since the core images used in training and validation are from the same well, the visual appearance of the core plugs will be similar. Thus, almost every core plug can be identified in the validation set by learning from the training set. This is also reflected in the *easy* test set where the performance is slightly higher than for the *hard* test set, due to some similarities in core plug appearance with the training and validation set.

From the class by class breakdown for *run1*, as seen in both the confusion matrices in figure 4.22 and from the precision-recall scores in table 4.13 it is clear that the main source of the performance loss is related to the bedding parallel core plugs, especially the *scal* class.

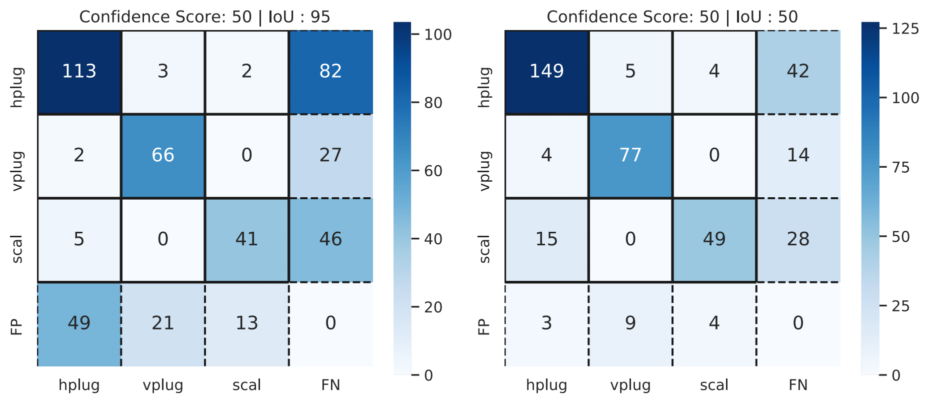


Figure 4.22: The confusion matrices for *run1* for each class in the *easy* test set at IoU50 and IoU95.

	IoU50		IoU95	
	P	R	P	R
hplug	0.871	0.745	0.669	0.565
vplug	0.846	0.811	0.733	0.695
scal	0.860	0.533	0.732	0.446

Table 4.13: The precision-recall scores for *run1* for each class on the *easy* test set at IoU50 and IoU95.

At both IoU thresholds it is clear that the recall score for the *scal* class is significantly lower than for the *hplug* and *vplug* classes. This show that a large portion of the core plugs in the *scal* class are significantly different than the ones in well $6406/3-2$ and $6406/8-1$ since they are not recognized at IoU50. Similarly, the recall score of the *hplug* class sees a performance loss as the IoU threshold increases. This shows that the detected plugs in this class have a lower IoU than the *vplug* class, which can be seen in the increase in false positives between the two IoU thresholds. However, the core plugs that are recognized have a reasonably high precision at both IoU thresholds, which shows that the core plugs that are detected can be classified with reasonably good precision.

In summary, the results on both test sets show that the object detection model trained on core plugs from the wells $6406/3 - 2$ and $6406/8 - 1$ is not able to generalize and predict the core plugs in other wells, without a significant loss in performance with respect to the COCO-evaluation metrics, as seen in table 4.11 and 4.12. This indicates that the training-validation data is not representative of the task data. However, the precision-recall scores for the *vplug* class at IoU50 in table 4.13 show that the model is able to perform reasonably well on this class ($P = 84.6\%$ and $R = 81.1\%$). This furthers the point that there is less variance related to the visual appearance of the vertical core plugs. As outlined in section 3.3, the variance in visual appearance of the bedding parallel core plugs was expected to be high, which was the motivation for including the high variance data from well $6406/8 - 1$. However, the results on the test sets show that this was not sufficient, reflected in the performance loss on the bedding parallel core plug classes (*hplug* and *scal*), with a significant loss in the *scal* class.

4.4 Fine-tuning the Final Model

The final test performed on the model involves fine-tuning of the weights of the final model from section 4.3. This was done in order to verify if the performance of the model could be increased on the core images from specific wells, by fine-tuning the existing model weights with a small number of examples from these wells. This test was conducted by fine-tuning the weights of the final model, using a small subset of the core images from the *easy* test set. The *easy* test set is chosen for this experiment to limit the variance in visual appearance of the core plugs in the data set. This is done to increase the chance of picking a small subset of images that are representative of the entire data set. Since the *easy* test set only contains images from two wells, and the intra-well variance is expected to be small, the chance of a representative subset is increased.

Since the number of images in the *easy* test set are limited (63 images), the model was trained and evaluated using the K-fold cross-validation approach, as outlined in section 2.2.4. Here, the goal is to verify that the result of the fine-tuning of the model is not dependent on the subset of images that are used. From the *easy* test set, 15 images were randomly selected of which 10 were used to train the model and 5 were used for validation. The model was trained and validated on the 15 images and tested on the remaining 48 images. This was done 5 times, reporting the average performance of these runs. The minimum validation loss was achieved after training the model for 1264 steps, which corresponds to approximately 10 minutes of training on a `GeForce RTX 2070 SUPER®` graphics card. The full COCO-evaluation metric scores for the five runs can be found in appendix G, with the most important metrics summarized in table 4.14.

Metric	run1	run2	run3	run4	run5	$\hat{\mu}$	$\hat{\sigma}_{\hat{\mu}}$	$\delta\hat{\mu}$
mAP	0.456	0.446	0.464	0.503	0.486	0.471	0.02073	0.00415
$AP^{IoU=50}$	0.802	0.766	0.803	0.831	0.798	0.8	0.02066	0.00413
$AP^{IoU=75}$	0.456	0.493	0.474	0.605	0.556	0.5168	0.05552	0.01110
AR^{100}	0.508	0.507	0.525	0.572	0.539	0.5302	0.02401	0.00480

Table 4.14: Condensed COCO-evaluation metrics for the cross validation, with the mean,standard deviation and the standard deviation of the mean (standard error).

Comparing these results to the performance on the *easy* test set in table 4.11, show that the performance on a data set with limited variance can be improved by fine-tuning the object detection model developed in this thesis using a few examples from the data set. In this case, the performance on the optical core images in the wells $6406/3 - 3$ and $6407/1 - 3$ was increased across all COCO-evaluation metrics by fine-tuning the model with approximately 20% of the images in the data set. This increase is most evident at the lowest IoU threshold where the achieved average precision, $AP^{IoU=50} = 80.0\%$, is significantly better than the score of $AP^{IoU=50} = 68.22\%$ achieved before fine-tuning. Also mAP, $AP^{IoU=75}$ and AR^{100} scores shows a performance gain after fine-tuning, achieving a score of 47.1%, 51.68% and 53.02%, respectively. However, the standard deviation shows that the improvement in performance, to some degree, is dependent on the images used in training and validation. This is especially notable in the $AP^{IoU=75}$ metric, which has the highest standard deviation. Further, the $AP^{IoU=75}$ score for each run suggests that the distribution of this metric is slightly skewed towards values closer to the $AP^{IoU=75}$ achieved by the model without fine-tuning. Similarly, the standard deviation of the mAP and AR^{100} is relatively large with respect to the increase in performance between this test and the test in the previous section. However, the distributions for these metrics are narrower and seems to be skewed towards higher values, which can be seen in the individual score for each run. This suggests that there is an overall increase in model performance with respect to these metrics.

Although, there is some increase to the performance with respect to all COCO-evaluation metrics in table 4.14. The most significant increase is related to the $AP^{IoU=50}$ metric. The class by class breakdown of the precision-recall scores is based on the performance of the model in *run1*. Since the $AP^{IoU=50}$ score is relatively similar for each run, they are all expected to give a reasonably similar indication of the cause of the performance increase. From the calculated precision-recall scores for *run1* in table 4.14, which can be seen in table 4.15, it is clear that the improved performance in $AP^{IoU=50}$ is related to better precision scores, but more importantly better recall scores for the bedding parallel core plugs. The increase in precision and recall for the core plugs in the *scal* and *hplug* classes at both IoU thresholds show that the model is able to identify a larger portion of the bedding parallel core plugs with higher precision (fewer false positives). The largest increase in recall occurs in the *scal* class, where the recall score is significantly better at IoU50 after fine-tuning. Similarly, a moderate increase in the recall score for the *hplug* class can be observed as well. This furthers the point that the initial training-validation set was not representative of the variations in visual appearance of the core plugs that exists

	IoU50		IoU95	
	P	R	P	R
hplug	0.931	0.829	0.752	0.664
vplug	0.896	0.833	0.697	0.639
scal	0.944	0.785	0.667	0.554

Table 4.15: The precision-recall scores for *run1* for each class at IoU50 and IoU95.

in the task data. Moreover, the performance of the model in a relatively low variance data set with, with respect to $AP^{IoU=50}$, can be significantly improved by exposing the model to only a few examples from the data set, with relatively short training time.

4.5 Further Discussion and Use Case

The results presented in this thesis indicate that the task of locating and classifying core plugs in optical core images can be achieved with both high precision and recall if the data used to train the model is representative of the task data. However, from the evaluation of the model performance on the test sets, it is clear that the variance in visual appearance of the core plugs in the 27 wells used in this thesis can not be learned from the two wells chosen to train the model. Despite including the high variance data from well $6406/8-1$. As outlined in section 4.2.2, the lowest IoU threshold, IoU=50, is sufficient to solve the object detection task. Although, the model is able to identify some of the core plugs in the test sets at this IoU threshold, the loss in performance is significant. This is below what can be regarded as acceptable for solving the task, dropping from a mean performance in $AP^{IoU=50} = 95.6\%$ on the validation set, to $AP^{IoU=50}$ of 68.22% and 58.6% on the *easy* and *hard* test set, respectively.

Since performance loss is believed to be mainly related to the training data not being representative of the task data. It is expected that the best approach for increasing the performance would be to expose the model to a greater variety of core plugs, by including training data from more wells. Since the task of labelling a new training, validation and test set is both labour intensive and time-consuming, this was not done in this thesis. However, the fine-tuning approach presented in section 4.4 shows that the model's performance can be significantly increased by training the weights of the base model with a small subset of the data set. Achieving a mean performance in the average precision $AP^{IoU=50} = 80\%$, resulting in an acceptable performance for solving the object detection task.

Since the variance in the visual appearance of the core plugs within a well, i.e. the intra-well variance, is expected to be low. It is fair to assume that the likelihood of extracting a small subset of the images from a well that is representative of the remaining images in that well as a whole is relatively high. This is demonstrated by the relatively low standard deviation in table 4.14, indicating that a similar $AP^{IoU=50}$ score can be achieved by training on five different subsets of the *easy* test set. If the same fine-tuning approach should be applied to the *hard* test set, the likelihood of extracting a representative subset of the data is expected to be significantly lower, due to the higher variance in visual appearance of the core plugs between wells, which would result in a higher standard deviation.

Since the prerequisite for this approach to succeed is that the extracted subset can be used to learn the variance in the remaining data, it is desirable to maximize the likelihood of a representative subset. Therefore, it is expected that this approach can be used to both solve the object detection task and be used to generate labelled training data for further training of the model if the data set used in the fine-tuning process is obtained from a single well.

As a proof of concept, a demo script for automatic generation of both labelled data and pixel-depth maps have been written and uploaded to the `GitHub` repository of this thesis (Adelved, 2020), which can be downloaded and used. Both scripts uses the exported inference graph of the fine-tuned weights from $run4$ in table 4.14 and predicts the cores plugs in core images from the *easy* test set.

The automatic labelling script uses the predictions to generate the `.xml` files and saves them in the same directory as the images, which can be inspected and edited using the `labelImage` application. This provides a semi-automatic workflow for labelling core plugs in optical core image data. This approach can significantly reduce the time required for manual labelling of wells with a moderate to large amount of core images. The automatic pixel-depth mapping uses the model predictions to classify the core plugs and locate the centre pixel of the predicted bounding box. Using the top depth of the core (defined in the file name), the pixel values are converted to an estimated depth in meters and outputs a `.csv` file containing the mapping. A sample of the output of the pixel-depth mapping script and a summarized representation of both scripts can be found in appendix J.

As outlined in section 3.3, the data set available for this thesis consisted of 27 wells from the Norwegian continental shelf made available by the courtesy of `GeoProvider`© (GeoProvider, 2019). This data set is already cropped, therefore it did not require a preprocessing step for cropping the core images. Introducing unprocessed data from other wells might need a preprocessing step before training and/or prediction. In such a case, the introduced workflow in this thesis and the specialization project can be regarded as two separate modules, which can be employed to reduce the time needed for generating training data. Using the model from the specialization project as the first module, a large number of core images from a well can be cropped into the desired format with reasonable accuracy. A small subset of these images can be randomly chosen and manually labelled to fine-tune the final model developed in this thesis using the workflow outlined in section 4.4 (second module). Finally, a variation of the scripts mentioned above can be used to predict the core plug labels for the remaining data set to create training data and the pixel-depth mapping.

Chapter 5

Conclusion

This thesis presents the workflow for fine-tuning a pre-trained object detection model, to detect both CCA and non-CCA core plugs in optical core images, using the `Tensorflow` object detection API. The employed pre-trained model is the Faster R-CNN ResNet Inception V2, trained on the Common Objects in Context (COCO) data set and evaluated using the COCO-evaluation metrics. The object detection task was performed by fine-tuning the pre-trained model on images from two wells from the Norwegian continental shelf ($6406/3-2$ and $6406/8-1$). The training and validation data were converted to grayscale, which suggested to increase the generalization capability of the model, indicated by the lower validation loss. This model was tested on two different sets of images, where the first test set, considered the *easy* set, was sampled from the wells $6406/3-3$ (32) and $6407/1-3$ (31) and the second set was randomly sampled from 25 wells from the NCS and considered the *hard* set. After training and testing of the model, the parameters (weights) of the model were further fine-tuned using a small subset of the images from the wells $6406/3-3$ and $6407/1-3$. Later on, this fine-tuned model was tested on the remaining images of the *easy* set.

The resulting object detection model was able to reach high performance with respect to the COCO-evaluation metrics on the validation set. However, evaluating the model on the two test sets resulted in a significant loss in performance. The most significant loss occurs in the randomly sampled set from the 25 wells. These results imply that the chosen training and validation sets are not representative of the existing variance associated with the high inter-well variance in visual appearance of the bedding parallel core plugs. In order to account for variance in the bedding parallel plugs, the data from well $4606/8-1$ (high variance data set), was included in the training of the model. However, this proved to be insufficient, since the amount of variance in visual appearance was greater than initially anticipated. Although, the model was not able to reach an acceptable performance for the intended task on the test sets, it is clear that the model has learned some fundamental features, allowing for a fair amount of the core plugs to be detected in both test sets. The model performance on the images in the *easy* test set was significantly improved at IoU50, from $AP^{IoU=50} = 66.6\%$ to $AP^{IoU=50} = 80.0\%$, by fine-tuning the model weights by incorporating a small subset of the images from *easy* test set, raising the model performance to an acceptable level on the remaining images in the *easy* test set. This illustrates that the model can be fine-tuned with few examples in a short amount of time to perform reasonably well on the data set where the variance in visual appearance within the data set can be represented by the few examples. Since this is expected to be the case for data sets that are obtained from a single well, this fine-tuning approach can be employed for generating both the pixel-depth mapping and labelled training data in individual wells.

5.1 Further work

The model trained in this thesis, in its current state, does not reach a high enough mAP or $AP^{IoU=50}$ score on the unseen core images. Thus, the model can not be directly utilized for an automatic core plug detection algorithm for correlating core analysis and optical core image data. As previously mentioned, this is mainly related to the training data not being representative of the variance that exists in the visual appearance of the core plugs in optical core images. This is especially evident in the bedding parallel core plugs. In order to improve the generalization capability of the model on unseen data, more variance needs to be included. As previously mentioned, this can be achieved by increasing the size of the training set, including core images from a greater number of wells from the available data set. This could be done by randomly selecting the majority of the available wells as training-validation wells, saving the remaining wells for testing. It is highly recommended to train several models using the K-fold cross-validation approach with a new training-validation and testing split approach for each model. In order to reduce the time needed for labelling, the fine-tuning approach from section 4.4 can be used for the wells with a large number of core images to generate label proposals that can be efficiently quality controlled.

As outlined in section 4.5, when new unprocessed data are included the model from the specialization project can be used for cropping the core images to the desired format prior to using the fine-tuning approach to generate label proposals. Currently, these models exist as separate modules which can be used to automate certain aspects of the labelling workflow. Further, the quality control of the labels needs to be performed in a separate software (`labelImg`). In order to make the auto-labelling process more stream-lined, these modules and the quality control step should be combined. Therefore, an alternate suggestion for further work is the development of a pipeline/software that integrates the model from the specialization project (core cropping) and the fine-tuning workflow in this thesis (generate labels/pixel-depth mapping) and allows for quality control and adjustment of the predicted labels in the same environment. An example of such a pipeline can be summarized with the following steps:

- Input optical core images in the standard format from NPD.
- Crop into the desired format using the module from the specialization project.
- Automatically select a random subset to be manually labelled by the user through a graphical user interface (GUI).

- Once labelling is finished, automatically export the data to the correct format and initiate the training-validation pipeline for fine-tuning the weights of the model developed in this thesis.
- Terminate the training with early stopping, export inference graph and predict the labels.
- Allow the user to interact with the predicted core plug labels through a GUI for quality control purposes.
- Save the quality controlled data for further training and as a pixel-depth mapping.
- Select a new well and reiterate.

Such a pipeline would allow the model to be used in its current state and it will reduce the amount of time and effort required to label new data. Once a sufficiently large amount of data have been accumulated, it can be used in the training of a better model or other object detection models that require similar data.

Bibliography

Abdillah, B., Jati, G., Jatmiko, W., 2018. Improvement cnn performance by edge detection preprocessing for vehicle classification problem, pp. 1–7. doi:10.1109/MHS.2018.8887015.

Adelved, D., 2019. Specialization project github repository. https://github.com/Adelved/specialization-project-repo. Accessed: 2020-06-15.

Adelved, D., 2020. Master's thesis github repository. https://github.com/Adelved/core-plug-detector. Accessed: 2020-06-15.

Black, P.E., 2005. "greedy algorithm". https://www.nist.gov/dads/HTML/greedyalgo.html. Accessed: 2020-05-19.

Brown, G.W., 1982. Standard deviation, standard error: Which 'standard' should we use? American Journal of Diseases of Children 136, 937–941. URL: https://doi.org/10.1001/archpedi.1982.03970460067015, doi:10.1001/archpedi.1982.03970460067015.

Canny, J., 1986. A computational approach to edge detection. IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-8, 679–698.

Chollet, F., 2018. Deep Learning with Python. Manning.

Co, L., 2018. Unsplash.com. URL: https://unsplash.com/photos/muj17G6lMjU.

Consortium, C., . Coco data set. URL: http://cocodataset.org/#explore. accessed: 2020-06-14.

Consortium, C., 2019. Coco evaluation metrics. URL: http://cocodataset.org/#detection-eval. accessed: 2019-12-10.

Elgendy, M., 2020. Deep Learning for Vision Systems. Meap ed., Manning.

Erofeev, A., Orlov, D., Ryzhov, A., Koroteev, D., 2019. Prediction of porosity and permeability alteration based on machine learning algorithms. Transport in Porous Media 128, 677–700. URL: http://dx.doi.org/10.1007/s11242-019-01265-3, doi:10.1007/s11242-019-01265-3.

GeoProvider, 2019. Geoprovider. URL: https://geoprovider.no/. accessed: 2020-03-26.

Girshick, R., 2015. Fast r-cnn, in: Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV), IEEE Computer Society, Washington, DC, USA. pp. 1440–1448. URL: http://dx.doi.org/10.1109/ICCV.2015.169, doi:10.1109/ICCV.2015.169.

Girshick, R., Donahue, J., Darrell, T., Malik, J., 2014. Rich feature hierarchies for accurate object detection and semantic segmentation, in: 2014 IEEE Conference on Computer Vision and Pattern Recognition, pp. 580–587. doi:10.1109/CVPR.2014.81.

Gonzalez, A., Kanyan, L., Heidari, Z., Lopez, O., 2019. 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, 15-19 June, The Woodlands, Texas, USA, Society of Petrophysicists and Well-Log Analysts. doi:10.30632/T60ALS-2019_A.

Gonzalez, R.C., Woods, R.E., 2018. Digital Image Processing. 4 ed., Pearson.

Goodfellow, I., Bengio, Y., Courville, A., 2016. Deep Learning. MIT Press. http://www.deeplearningbook.org.

Huang, J., Rathod, V., Sun, C., Zhu, M., Korattikara, A., Fathi, A., Fischer, I., Wojna, Z., Song, Y., Guadarrama, S., et al., 2017a. Speed/accuracy trade-offs for modern convolutional object detectors. 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) URL: http://dx.doi.org/10.1109/CVPR.2017.351, doi:10.1109/cvpr.2017.351.

Huang, J., Rathod, V., Sun, C., Zhu, M., Korattikara, A., Fathi, A., Fischer, I., Wojna, Z., Song, Y., Guadarrama, S., et al., 2017b. Speed/accuracy trade-offs for modern convolutional object detectors. 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) URL: http://dx.doi.org/10.1109/CVPR.2017.351, doi:10.1109/cvpr.2017.351.

International Telecommunication Union, I., 2001. Studio encoding parameters of digital television for standard 4:3 and wide-screen 16:9 aspect ratios. Available at https://www.itu.int/dms_pubrec/itu-r/rec/bt/R-REC-BT.601-7-201103-I!!PDF-E.pdf. Accessed: 2020-04-21.

Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S.E., Fu, C., Berg, A.C., 2015. SSD: single shot multibox detector. CoRR abs/1512.02325. URL: http://arxiv.org/abs/1512.02325, arXiv:1512.02325.

McPhee, C., Reed, J., Zubizarreta, I., 2015. Core Analysis: A Best Practice Guide. Elsevier.

NPD, N.P.D., . Npd factpages. URL: https://factpages.npd.no/. accessed: 2020-01-07.

OpenCV, . Canny edge detection. https://docs.opencv.org/trunk/da/d22/tutorial_py_canny.html. Accessed: 2020-06-14.

Prince, C., Shafer, J.L., 2002. Predicting rock properties from digital core images.

Prince, S.J., 2012. Computer Vision: Models, Learning and Inference. Cambridge University Press.

Qian, N., 1999. On the momentum term in gradient descent learning algorithms. Neural Networks 12, 145–151. URL: http://dblp.uni-trier.de/db/journals/nn/nn12.html#Qian99.

Redmon, J., Divvala, S.K., Girshick, R.B., Farhadi, A., 2015. You only look once: Unified, real-time object detection. CoRR abs/1506.02640. URL: http://arxiv.org/abs/1506.02640, arXiv:1506.02640.

Ren, S., He, K., Girshick, R., Sun, J., 2017. Faster r-cnn: Towards real-time object detection with region proposal networks. IEEE Transactions on Pattern Analysis and Machine Intelligence 39, 1137–1149. doi:10.1109/TPAMI.2016.2577031.

Rumelhart, D.E., Hinton, G.E., Williams, R.J., 1986. Learning representations by back-propagating errors. 323, 533–536. doi:10.1038/323533a0.

Russell, S., Norvig, P., 2009. Artificial Intelligence: A Modern Approach. 3rd ed., Prentice Hall Press, USA.

Ryan, H., 1994. Ricker, ormsby, klander, butterworth – a choice of wavelets. CSEG Recorder URL: http://74.3.176.63/publications/recorder/1994/09sep/sep94-choice-of-wavelets.pdf.

Simm, R., Bacon, M., 2014. Seismic Amplitude: An Interpreter's Handbook. Cambride University Press.

Sobel, I., Feldman, G., 1973. A 3×3 isotropic gradient operator for image processing. Pattern Classification and Scene Analysis , 271–272.

Suvrit Sra, Sebastian Nowozin, S.J.W., 2011. Optimization for Machine Learning (Neural Information Processing series). Neural Information Processing series, The MIT Press.

Szegedy, C., Ioffe, S., Vanhoucke, V., Alemi, A., 2016. Inception-v4, inception-resnet and the impact of residual connections on learning arXiv:1602.07261.

Tensorflow, 2020a. Detection model zoo. https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md. Accessed: 2020-06-12.

Tensorflow, 2020b. Tensorflow models repository. https://github.com/tensorflow/models. Accessed: 2020-06-12.

Tensorflow, 2020c. Tensorflow object detection api tutorial. URL: https://tensorflow-object-detection-api-tutorial.readthedocs.io/en/latest/index.html. accessed: 2020-01-06.

Thomas, A., Rider, M., Curtis, A., Macarthur, A., 2011. Automated lithology extraction from core photographs 29. doi:10.3997/2214-4609.201400891.

Tzutalin, 2015. Labelimg. https://github.com/tzutalin/labelImg.

Uijlings, J.R.R., van de Sande, K.E.A., Gevers, T., Smeulders, A.W.M., 2013. Selective search for object recognition. International Journal of Computer Vision 104, 154–171. URL: https://doi.org/10.1007/s11263-013-0620-5, doi:10.1007/s11263-013-0620-5.

Walpole, R.E., Myers, R.H., Myers, S.L., Ye, K., 2012. Probability Statistics for Engineers and Scientists. 9 ed., Pearson.

Zhang, E., Zhang, Y., 2009. Eleven Point Precision-recall Curve. Springer US, Boston, MA. pp. 981–982. URL: https://doi.org/10.1007/978-0-387-39940-9_481, doi:10.1007/978-0-387-39940-9_481.

A Wells

Well name	Number of images
6407/2-3	176
6407/7-2	178
6407/6-1	19
6507/8-1	170
6507/11-3	140
6407/2-1	69
6407/10-1	140
6407/9-2	44
6506/12-4	60
6507/11-1	32
6407/1-3	108 (31 used in *easy*)
6406/3-2	293 (289 used training-validation)
6406/8-1	142 (136 used training-validation)
6507/11-2	48
6507/7-4	514
6407/9-4	43
6506/12-7	78
6406/3-3	32 (32 used in *easy*)
6407/9-1	50
6506/12-1	289
6407/2-2	99
6407/7-4	190
6407/4-1	237
6506/12-3	315
6407/1-2	59
6407/7-1	322
6407/6-3	118

The wells in the data set prepared by `GeoProvider`©. Not all available images from the training-validation wells are used due to core segments without any core plugs (GeoProvider, 2019)

B Default Configuration File

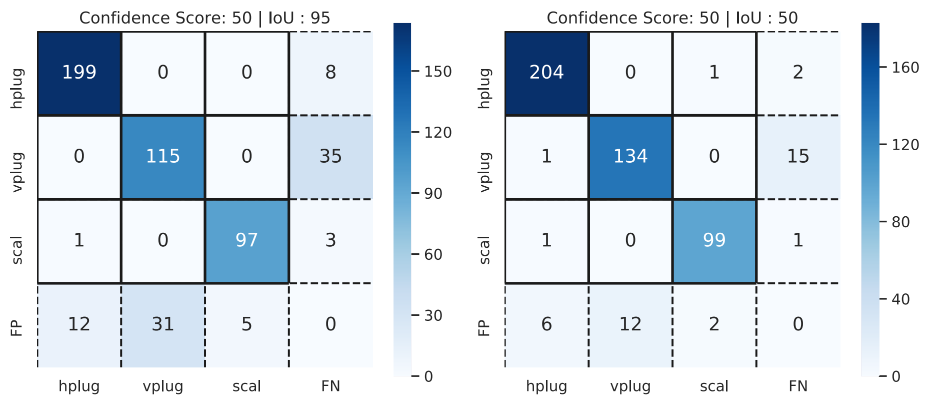
```
model {
  faster_rcnn {
    num_classes: 1
    image_resizer {
      keep_aspect_ratio_resizer {
        min_dimension: 600
        max_dimension: 1200
      }
    }
    feature_extractor {
      type: "faster_rcnn_inception_resnet_v2"
      first_stage_features_stride: 8
    }
    first_stage_anchor_generator {
      grid_anchor_generator {
        height_stride: 8
        width_stride: 8
        scales: 0.25
        scales: 0.5
        scales: 1.0
        scales: 2.0
        aspect_ratios: 0.5
        aspect_ratios: 1.0
        aspect_ratios: 2.0
      }
    }
    first_stage_atrous_rate: 2
    first_stage_box_predictor_conv_hyperparams {
      op: CONV
      regularizer {
        l2_regularizer {
          weight: 0.0
        }
      }
      initializer {
        truncated_normal_initializer {
          stddev: 0.00999999977648
        }
      }
    }
    first_stage_nms_score_threshold: 0.0
    first_stage_nms_iou_threshold: 0.699999988079
    first_stage_max_proposals: 300
    first_stage_localization_loss_weight: 2.0
    first_stage_objectness_loss_weight: 1.0
    initial_crop_size: 17
    maxpool_kernel_size: 1
    maxpool_stride: 1
    second_stage_box_predictor {
      mask_rcnn_box_predictor {
        fc_hyperparams {
          op: FC
          regularizer {
            l2_regularizer {
              weight: 0.0
            }
          }
          initializer {
            variance_scaling_initializer {
              factor: 1.0
              uniform: true
              mode: FAN_AVG
            }
          }
        }
```

```
          use_dropout: false
          dropout_keep_probability: 1.0
        }
      }
      second_stage_post_processing {
        batch_non_max_suppression {
          score_threshold: 0.300000011921
          iou_threshold: 0.600000023842
          max_detections_per_class: 100
          max_total_detections: 100
        }
        score_converter: SOFTMAX
      }
      second_stage_localization_loss_weight: 2.0
      second_stage_classification_loss_weight: 1.0
    }
}
train_config {
  batch_size: 1
  data_augmentation_options {
    random_horizontal_flip {
    }
  }
  optimizer {
    momentum_optimizer {
      learning_rate {
        manual_step_learning_rate {
          initial_learning_rate: 0.000300000014249
          schedule {
            step: 0
            learning_rate: 0.000300000014249
          }
          schedule {
            step: 900000
            learning_rate: 2.99999992421e-05
          }
          schedule {
            step: 1200000
            learning_rate: 3.00000010611e-06
          }
        }
      }
      momentum_optimizer_value: 0.899999976158
    }
    use_moving_average: false
  }
  gradient_clipping_by_norm: 10.0
  fine_tune_checkpoint: "path/to/finetuned/checkpoints/
    faster_rcnn_inception_resnet_v2_atrous_coco_2018_01_28/model.ckpt"
  from_detection_checkpoint: true
  num_steps: 200000
}
train_input_reader {
  label_map_path: ""
  tf_record_input_reader {
    input_path: "/path/to/train.record"
  }
```

```
}
eval_config {
  num_examples: 8000
  max_evals: 10
  use_moving_averages: false
}
eval_input_reader {
  label_map_path: "/path/to/labelmap.pbtxt"
  shuffle: false
  num_readers: 1
  tf_record_input_reader {
    input_path: "path/to/valid.record"
  }
}
```

Listing 1: Default configuration file

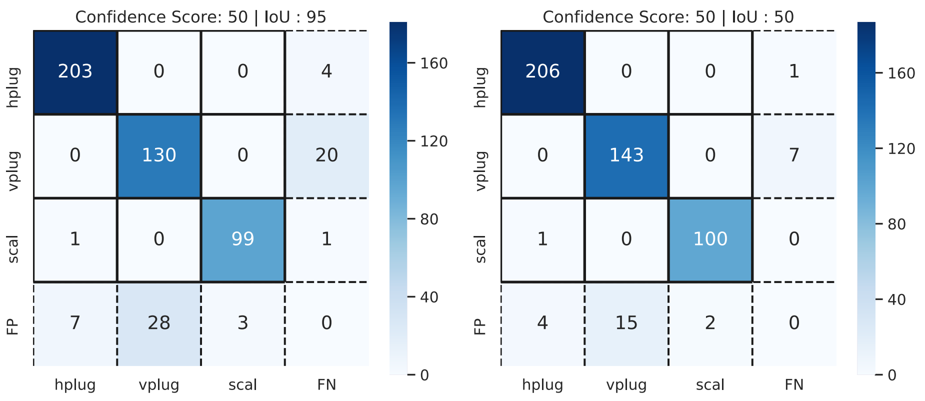
C Precision - Recall Candidate Models



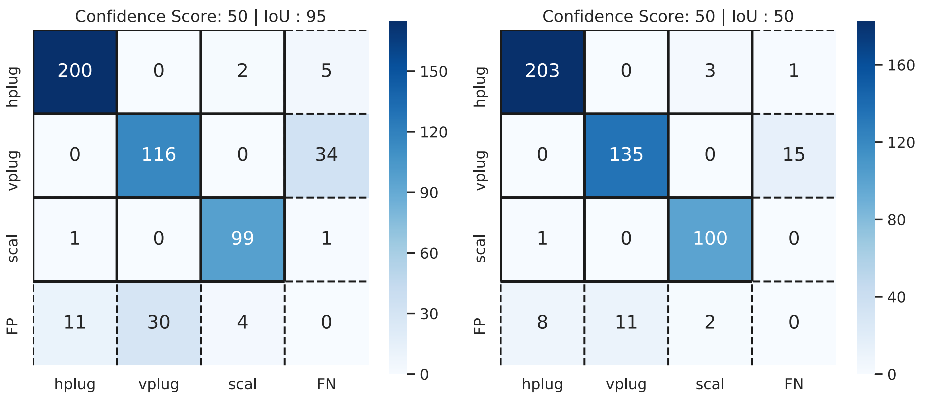
	hplug	vplug	scal	FN
hplug	199	0	0	8
vplug	0	115	0	35
scal	1	0	97	3
FP	12	31	5	0

	hplug	vplug	scal	FN
hplug	204	0	1	2
vplug	1	134	0	15
scal	1	0	99	1
FP	6	12	2	0

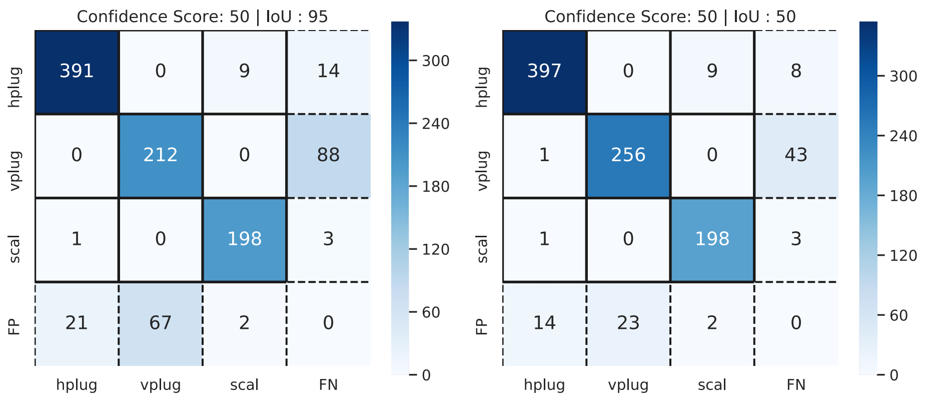
Confusion matrices at IoU=0.95 and IoU=0.5 for the canny model



Confusion matrix at IoU=0.95 and IoU=0.5 for the gray model



Confusion matrices at IoU=0.95 and IoU=0.5 for the sobel model



Confidence Score: 50 | IoU : 95

	hplug	vplug	scal	FN
hplug	391	0	9	14
vplug	0	212	0	88
scal	1	0	198	3
FP	21	67	2	0

Confidence Score: 50 | IoU : 50

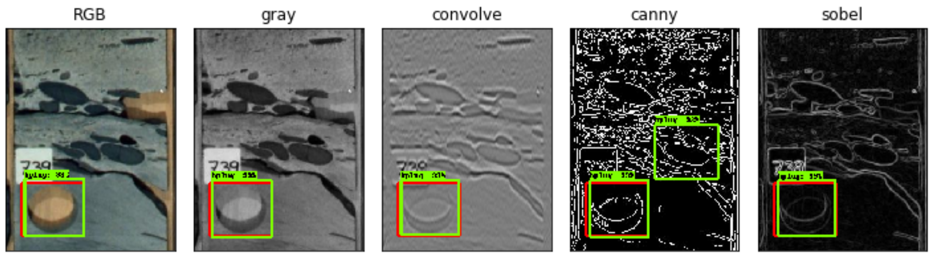
	hplug	vplug	scal	FN
hplug	397	0	9	8
vplug	1	256	0	43
scal	1	0	198	3
FP	14	23	2	0

Confusion matrices at IoU=0.95 and IoU=0.5 for the wavelet model

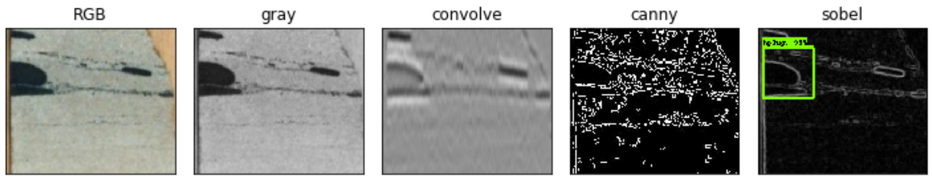
	hplug				vplug				scal			
	P		R		P		R		P		R	
Model	IoU50	IoU95	IoU50	IoU95	IoU50	IoU95	IoU50	IoU95	IoU50	IoU95	IoU50	IoU95
RGB	0.967	0.943	0.986	0.961	0.910	0.833	0.947	0.867	0.943	0.943	0.990	0.990
gray	0.976	0.962	0.995	0.981	0.905	0.823	0.953	0.867	0.980	0.971	0.990	0.980
canny	0.962	0.939	0.986	0.961	0.918	0.788	0.893	0.767	0.971	0.951	0.980	0.960
sobel	0.958	0.943	0.981	0.966	0.925	0.795	0.900	0.773	0.952	0.943	0.990	0.980
wavelet	0.961	0.947	0.959	0.944	0.918	0.760	0.853	0.707	0.947	0.947	0.980	0.980

Precision-recall values for the candidate models at IoU50 and IoU95.

D Detections Candidate Models



False positive in the canny model due to clast.



False positive in the sobel model due to clast.

E COCO Evaluation Final Model

```
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] =
    0.707
```

```
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] =
    0.953
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] =
    0.866
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] =
    -1.000
Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] =
    0.637
Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] =
    0.668
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] =
    0.335
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] =
    0.758
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] =
    0.758
Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] =
    -1.000
Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] =
    0.672
Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] =
    0.713
```

Listing 2: COCO-evaluation metrics for the initial run of the final model.

```
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] =
    0.698
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] =
    0.943
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] =
    0.859
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] =
    -1.000
Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] =
    0.649
Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] =
    0.655
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] =
    0.339
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] =
    0.758
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] =
    0.758
Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] =
    -1.000
Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] =
    0.682
Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] =
    0.711
```

Listing 3: COCO-evaluation metrics, repeated run

```
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] =
    0.681
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] =
    0.946
```

```
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] =
    0.858
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] =
    -1.000
Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] =
    0.644
Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] =
    0.648
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] =
    0.327
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] =
    0.742
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] =
    0.742
Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] =
    -1.000
Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] =
    0.671
Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] =
    0.717
```

Listing 4: COCO-evaluation metrics, repeated run

```
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] =
    0.705
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] =
    0.948
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] =
    0.871
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] =
    -1.000
Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] =
    0.549
Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] =
    0.653
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] =
    0.340
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] =
    0.759
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] =
    0.759
Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] =
    -1.000
Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] =
    0.596
Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] =
    0.700
```

Listing 5: COCO-evaluation metrics, repeated run

```
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] =
    0.701
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] =
    0.950
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] =
    0.852
```

```
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] =
    -1.000
Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] =
    0.655
Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] =
    0.652
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] =
    0.337
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] =
    0.752
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] =
    0.752
Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] =
    -1.000
Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] =
    0.690
Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] =
    0.700
```
Listing 6: COCO-evaluation metrics, repeated run

F The Full COCO-Evaluation Metrics for Test Sets

Metric	run1	run2	run3	run4	run5	$\hat{\mu}$	$\hat{\sigma}_{\hat{\mu}}$	$\delta\hat{\mu}$
mAP	0.414	0.404	0.377	0.419	0.378	0.3984	0.01774	0.00355
$AP^{IoU=50}$	0.696	0.696	0.654	0.699	0.666	0.6822	0.01855	0.00371
$AP^{IoU=75}$	0.465	0.453	0.403	0.462	0.382	0.433	0.03396	0.00679
$AP(small)$	-1	-1	-1	-1	-1	-1	0.00000	0.00000
$AP(medium)$	0.262	0.241	0.235	0.293	0.235	0.2532	0.02224	0.00445
$AP(large)$	0.41	0.402	0.373	0.407	0.376	0.3936	0.01583	0.00317
AR^1	0.278	0.271	0.265	0.278	0.263	0.271	0.00629	0.00126
AR^{10}	0.466	0.46	0.429	0.471	0.438	0.4528	0.01639	0.00328
AR^{100}	0.466	0.46	0.429	0.471	0.438	0.4528	0.01639	0.00328
$AR(small)$	-1	-1	-1	-1	-1	-1	0.00000	0.00000
$AR(medium)$	0.325	0.277	0.295	0.335	0.283	0.303	0.02301	0.00460
$AR(large)$	0.453	0.449	0.415	0.457	0.428	0.4404	0.01617	0.00323

Full COCO-evaluation metric for the 5 runs on the easy test set, with the mean, standard deviation and the standard deviation of the mean.

Metric	run1	run2	run3	run4	run5	$\hat{\mu}$	$\hat{\sigma}_{\hat{\mu}}$	$\delta\hat{\mu}$
mAP	0.364	0.387	0.372	0.34	0.352	0.363	0.01617	0.00323
$AP^{IoU=50}$	0.572	0.63	0.597	0.562	0.569	0.586	0.02497	0.00499
$AP^{IoU=75}$	0.428	0.472	0.424	0.378	0.388	0.418	0.03332	0.00666
$AP(small)$	-1	-1	-1	-1	-1	-1	0.00000	0.00000
$AP(medium)$	0.136	0.164	0.15	0.149	0.152	0.1502	0.00891	0.00178
$AP(large)$	0.391	0.413	0.387	0.363	0.369	0.3846	0.01768	0.00354
AR^1	0.257	0.263	0.253	0.242	0.244	0.2518	0.00788	0.00158
AR^{10}	0.415	0.437	0.423	0.39	0.403	0.4136	0.01617	0.00323
AR^{100}	0.415	0.437	0.423	0.39	0.403	0.4136	0.01617	0.00323
$AR(small)$	-1	-1	-1	-1	-1	-1	0.00000	0.00000
$AR(medium)$	0.168	0.195	0.183	0.192	0.188	0.1852	0.00950	0.00190
$AR(large)$	0.446	0.467	0.438	0.405	0.421	0.4354	0.02121	0.00424

Full COCO-evaluation metric for the 5 runs on the hard test set, with the mean, standard deviation and the standard deviation of the mean.

G The Full COCO-Evaluation Metrics, Cross-Validation

Metric	run1	run2	run3	run4	run5	$\hat{\mu}$	$\hat{\sigma}_{\hat{\mu}}$	$\delta\hat{\mu}$
mAP	0.456	0.446	0.464	0.503	0.486	0.471	0.02073	0.00415
$AP^{IoU=50}$	0.802	0.766	0.803	0.831	0.798	0.8	0.02066	0.00413
$AP^{IoU=75}$	0.456	0.493	0.474	0.605	0.556	0.5168	0.05552	0.01110
$AP(small)$	-1	-1	-1	-1	-1	-1	0.00000	0.00000
$AP(medium)$	0.393	0.269	0.256	0.421	0.26	0.3198	0.07187	0.01437
$AP(large)$	0.452	0.45	0.474	0.506	0.494	0.4752	0.02226	0.00445
AR^1	0.296	0.275	0.281	0.309	0.308	0.2938	0.01382	0.00276
AR^{10}	0.508	0.507	0.525	0.572	0.539	0.5302	0.02401	0.00480
AR^{100}	0.508	0.507	0.525	0.572	0.539	0.5302	0.02401	0.00480
$AR(small)$	-1	-1	-1	-1	-1	-1	0.00000	0.00000
$AR(medium)$	0.475	0.293	0.271	0.466	0.273	0.3556	0.09417	0.01883
$AR(large)$	0.498	0.513	0.541	0.573	0.553	0.5356	0.02704	0.00541

The full COCO-evaluation metrics for the cross-validation.

H Data Processing Functions

```python
# ==============================================================
# Copyright 2020 Dennis Adelved. All Rights Reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
# ==============================================================
#Dependencies
import os
import numpy as np
import shutil
from PIL import Image
import glob
import matplotlib.pyplot as plt
import xml.etree.ElementTree as ET
import bruges
import random
import cv2

#Data augmentation functions

#function to save the relevant attributes(folder,file name and path) of
    the xml's outputted from LabelImg
def change_xml(xml_path,new_folder,new_fname,new_path,channels=3):
    tree = ET.parse(xml_path)
    root = tree.getroot()
    root[0].text = new_folder
    root[1].text = new_fname
    root[2].text = new_path
    if channels==1:
        root[4][2].text = '1'

    return tree

#convolve grayscale image with random frequency in range [min_freq:10:
    max_freq]
def convolve_gray_image_random(im_path, min_freq,max_freq, multiple_axes=
    True):
    image = Image.open(im_path).convert('L')
    image = np.asarray(image)

    f = []

    f.append(random.randint(min_freq,max_freq))
    f.append(random.randint(min_freq,max_freq))

    wavelet = []
    wavelet.append(bruges.filters.ricker(duration=0.100, dt=0.001, f=f[0])
    ) #wavelet
    wavelet.append(bruges.filters.ricker(duration=0.100, dt=0.001, f=f[1])
    )

    if multiple_axes == True:
        vert = np.apply_along_axis(lambda t: np.convolve(t,wavelet[0],mode
    ='same'),axis=0,arr=image)
        hor = np.apply_along_axis(lambda t: np.convolve(t,wavelet[1],mode=
    'same'),axis=1,arr=image)

        return [vert,hor]
    else:
        hor = np.apply_along_axis(lambda t: np.convolve(t,wavelet[0],mode=
    'same'),axis=1,arr=image)
        return [hor]


#augmentation gray
def convert_to_gray(image_path):
    image = Image.open(image_path).convert('L')
    image = np.asarray(image)
    new_im = image.copy()
    return new_im

#augmentation sobel
def augment_sobel(image_path):
    image = Image.open(image_path)
    im = np.asarray(image)
    sobelx = cv2.Sobel(im,cv2.CV_64F,1,0,ksize=3)
    sobely = cv2.Sobel(im,cv2.CV_64F,0,1,ksize=3)
    grad = np.sqrt(sobelx**2 + sobely**2)
    return grad

#augmentation canny
def augment_canny(image_path):
    percentiles = np.array([80, 85, 90, 95])
    image = Image.open(image_path)
    im = np.asarray(image)
    perc = percentiles[random.randrange(len(percentiles))]
    canny = cv2.Canny(im,perc/2,perc)
    return canny


#import xml and jpg and returns a list of xml,jpg pairs
def import_img_xml_paths(data_dir):
    used_cores = glob.glob(os.path.join(data_dir,'*.jpg'))
    used_xml = glob.glob(os.path.join(data_dir,'*.xml'))
    return list(zip(sorted(used_cores),sorted(used_xml)))


#splits training and validation data, simple hold-out split
def simple_holdout_split(image_xml_pairs,split=0.2):

    num_im = len(image_xml_pairs)
    random.shuffle(image_xml_pairs)
    val = image_xml_pairs[0:math.floor(num_im*split)]
    train = image_xml_pairs[math.floor(num_im*split):]

    return train, val
```

Listing 7: Preprocesssing functions used for the data augmentation

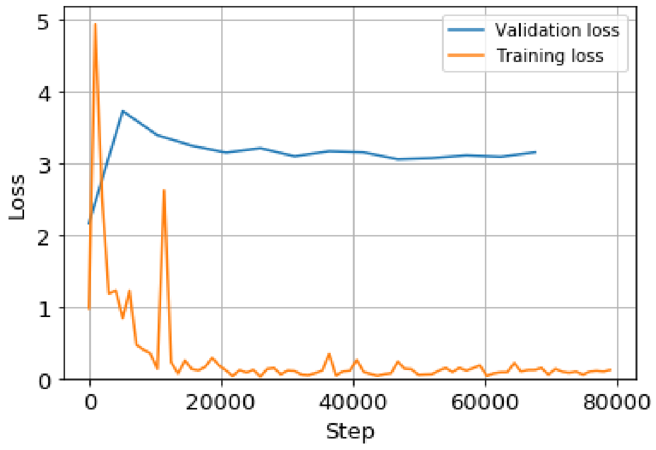
```

# ============================================================
# Copyright 2020 Dennis Adelved. All Rights Reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
# ============================================================

#Dependencies
import os
import glob
import shutil
import random
import xml.etree.ElementTree as ET

#K-fold validation split
#splitting the easy test set into 5 folds (training,validaiton and test
    set)

#data directory containing the xml and jpg files
data_dir = 'test_data'

samples = ['fold1','fold2','fold3','fold4','fold5']
samps = []
for s in samples:
    splits=[]
    splits.append('train_'+s)
    splits.append('valid_'+s)
    splits.append('test_'+s)
    samps.append(splits)

for s in samps:
    xmls = glob.glob(os.path.join(data_dir,'*.xml'))
    ims = glob.glob(os.path.join(data_dir,'*.jpg'))
    xmls=sorted(xmls)
    ims = sorted(ims)
    data = zip(ims,xmls)
    data = list(data)
    random.shuffle(data)
    split = data[0:15]
    test = data[15:]
    random.shuffle(split)
    train=split[0:10]
    val = split[10:]
    for f in s:
        os.mkdir(f)
        if 'train' in f:
            for i in range(len(train)):
                imin = os.path.join(os.getcwd(),train[i][0])
                xmlin = os.path.join(os.getcwd(),train[i][1])
                imout = os.path.join(os.getcwd(),f,train[i][0].split('/')
    [-1])
                xmlout = os.path.join(os.getcwd(),f,train[i][1].split('/')
    [-1])
                shutil.copy(imin,imout)
                shutil.copy(xmlin,xmlout)
        if 'valid' in f:
            for i in range(len(val)):
                imin = os.path.join(os.getcwd(),val[i][0])
                xmlin = os.path.join(os.getcwd(),val[i][1])
                imout = os.path.join(os.getcwd(),f,val[i][0].split('/')
    [-1])
                xmlout = os.path.join(os.getcwd(),f,val[i][1].split('/')
    [-1])
                shutil.copy(imin,imout)
                shutil.copy(xmlin,xmlout)
        if 'test' in f:
            for i in range(len(test)):
                imin = os.path.join(os.getcwd(),test[i][0])
                xmlin = os.path.join(os.getcwd(),test[i][1])
                imout = os.path.join(os.getcwd(),f,test[i][0].split('/')
    [-1])
                xmlout = os.path.join(os.getcwd(),f,test[i][1].split('/')
    [-1])
                shutil.copy(imin,imout)
                shutil.copy(xmlin,xmlout)
```

Listing 8: Splitting the total data in the *easy* test set in to 5 different training-validation-test set splits (K-fold cross-validation split)

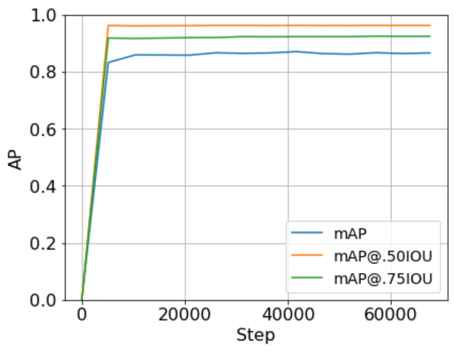
I COCO-Evaluation Metrics: Specialization Project

Metric	AP $\lambda_2 = 100$	AP $\lambda_2 = 2$
$AP^{IoU=0.5}$	0.9623	0.9703
$AP^{IoU=0.75}$	0.9238	0.8819
mAP	0.8655	0.7723

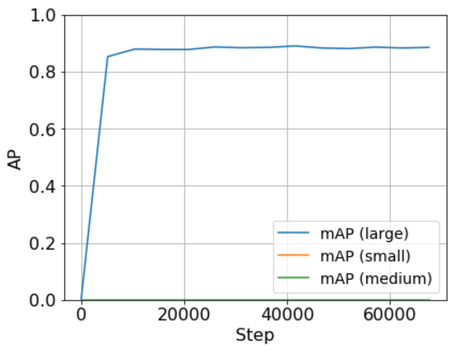
The COCO AP scores for the 2 IoU thresholds and the mAP at end of training for $\lambda_2 = 100$ and $\lambda_2 = 2$ (specialization project).



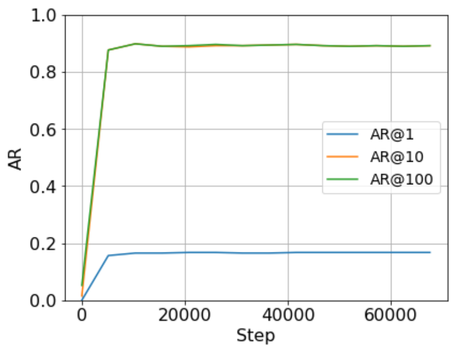
The total loss from training and validation $\lambda_2 = 100$ (specialization project).



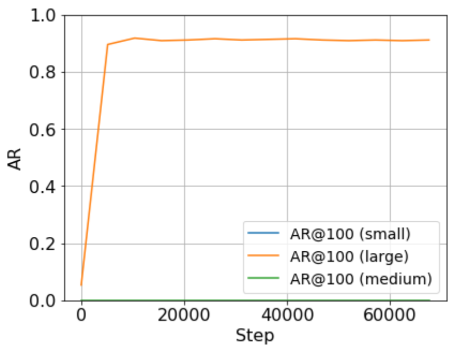
AP for IoU thresholds and mAP.



AP across the different scales.



Max AR given a fixed number of predictions.



AR across the different scales.

The COCO evaluation metrics $\lambda_2 = 100$ (specialization project).

J Pixel-Depth Mapping Script with Output and Auto-Labeling Script

index	Plug Type	Depth	Pixel Location	Source
0	vplug	3632.0269058296	66	images/6407_1_3_3632_3633.jpg
1	hplug	3632.0523848349	128.5	images/6407_1_3_3632_3633.jpg
2	vplug	3632.32409294741	795	images/6407_1_3_3632_3633.jpg
3	hplug	3632.32857725234	806	images/6407_1_3_3632_3633.jpg
4	hplug	3632.75743986955	1858	images/6407_1_3_3632_3633.jpg
5	vplug	3632.79841011007	1958.5	images/6407_1_3_3632_3633.jpg
6	vplug	3632.91765185487	2251	images/6407_1_3_3632_3633.jpg
7	hplug	3632.97472482674	2391	images/6407_1_3_3632_3633.jpg
8	vplug	3634.04197080292	103.5	images/6407_1_3_3634_3635.jpg
9	scal	3634.06184103812	152.5	images/6407_1_3_3634_3635.jpg
10	vplug	3634.2899432279	715	images/6407_1_3_3634_3635.jpg
11	scal	3634.35888077859	885	images/6407_1_3_3634_3635.jpg
12	hplug	3634.66180048662	1632	images/6407_1_3_3634_3635.jpg
13	vplug	3634.71593673966	1765.5	images/6407_1_3_3634_3635.jpg
14	scal	3634.91585563666	2258.5	images/6407_1_3_3634_3635.jpg
15	vplug	3638.02193708609	53	images/6407_1_3_3638_3639.jpg
16	scal	3639.06597938144	160	images/6407_1_3_3639_3640.jpg
17	scal	3639.14969072165	363	images/6407_1_3_3639_3640.jpg
18	scal	3639.3612371134	876	images/6407_1_3_3639_3640.jpg
19	scal	3639.70927835052	1720	images/6407_1_3_3639_3640.jpg
20	vplug	3639.77051546392	1868.5	images/6407_1_3_3639_3640.jpg
21	hplug	3643.03112118714	75.5	images/6407_1_3_3643_3644.jpg
22	scal	3643.10449299258	253.5	images/6407_1_3_3643_3644.jpg
23	vplug	3643.27926628195	677.5	images/6407_1_3_3643_3644.jpg
24	hplug	3643.33429513603	811	images/6407_1_3_3643_3644.jpg
25	hplug	3643.65416323166	1587	images/6407_1_3_3643_3644.jpg
26	vplug	3643.67497938994	1637.5	images/6407_1_3_3643_3644.jpg
27	scal	3643.9451772465	2293	images/6407_1_3_3643_3644.jpg
28	vplug	3643.95692497939	2321.5	images/6407_1_3_3643_3644.jpg
29	hplug	3648.02703243893	67.5	images/6407_1_3_3648_3649.jpg
30	hplug	3648.31137364838	777.5	images/6407_1_3_3648_3649.jpg
31	vplug	3648.3454144974	862.5	images/6407_1_3_3648_3649.jpg

A subset of the output from the pixel-depth mapping script on the *easy* tes set

```
# ==============================================================
# Copyright 2017 The TensorFlow Authors. All Rights Reserved.
#
# Modifications copyright 2020 Dennis Adelved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
```

```
#       http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
# ==============================================================

#Dependencies
import glob
import numpy as np
import os
import six.moves.urllib as urllib
import sys
import tarfile
import tensorflow as tf
import zipfile
from distutils.version import StrictVersion
from collections import defaultdict
from io import StringIO
from matplotlib import pyplot as plt
from PIL import Image
import pandas as pd
import shutil

sys.path.append("..")
from object_detection.utils import ops as utils_ops

#Run inference on image using the frozen inference graph
def run_inference_for_single_image(image, graph):
    if 'detection_masks' in tensor_dict:
        # The following processing is only for single image
        detection_boxes = tf.squeeze(tensor_dict['detection_boxes'], [0])
        detection_masks = tf.squeeze(tensor_dict['detection_masks'], [0])
        # Reframe is required to translate mask from box coordinates to
    image coordinates and fit the image size.
        real_num_detection = tf.cast(tensor_dict['num_detections'][0], tf.
    int32)
        detection_boxes = tf.slice(detection_boxes, [0, 0], [
    real_num_detection, -1])
        detection_masks = tf.slice(detection_masks, [0, 0, 0], [
    real_num_detection, -1, -1])
        detection_masks_reframed = utils_ops.
    reframe_box_masks_to_image_masks(
            detection_masks, detection_boxes, image.shape[1], image.shape
    [2])
        detection_masks_reframed = tf.cast(
            tf.greater(detection_masks_reframed, 0.5), tf.uint8)
        # Follow the convention by adding back the batch dimension
        tensor_dict['detection_masks'] = tf.expand_dims(
            detection_masks_reframed, 0)


    image_tensor = tf.get_default_graph().get_tensor_by_name('image_tensor
    :0')
    # Run inference
    output_dict = sess.run(tensor_dict,
                          feed_dict={image_tensor: image})

    # all outputs are float32 numpy arrays, so convert types as
    appropriate
    output_dict['num_detections'] = int(output_dict['num_detections'][0])
    output_dict['detection_classes'] = output_dict[
      'detection_classes'][0].astype(np.int64)
    output_dict['detection_boxes'] = output_dict['detection_boxes'][0]
    output_dict['detection_scores'] = output_dict['detection_scores'][0]
    if 'detection_masks' in output_dict:
        output_dict['detection_masks'] = output_dict['detection_masks'][0]
    return output_dict


def generate_predictions(model_name,path_to_frozen_inference_graph,
    path_to_labels,path_to_images,detection_graph):

  # This is needed since the notebook is stored in the object_detection
    folder.
  sys.path.append(”..”)
  from object_detection.utils import ops as utils_ops

  #if StrictVersion(tf.__version__) < StrictVersion('1.12.0'):
    #raise ImportError('Please upgrade your TensorFlow installation to v1
    .12.*.')

    dicts = [] #to save detections
    image_dims = [] #to save image dimentions

    #Retrieve the detection data from the inference
    with detection_graph.as_default():
        with tf.Session() as sess:
        # Get handles to input and output tensors
            ops = tf.get_default_graph().get_operations()
            for op in ops:
                op._set_device('/device:CPU:*')
            all_tensor_names = {output.name for op in ops for output in op
    .outputs}
            tensor_dict = {}
            for key in [
            'num_detections', 'detection_boxes', 'detection_scores',
            'detection_classes', 'detection_masks'
        ]:
                tensor_name = key + ':0'
                if tensor_name in all_tensor_names:
                    tensor_dict[key] = tf.get_default_graph().
    get_tensor_by_name(
                tensor_name)

            for image_path in path_to_images:

                #Convert RGB image to grayscale
                img = np.array(Image.open(image_path).convert('L'))

                #Expand the number of channels to 3, i.e. expand the
    grayscale channel.
```

```
                new_im = np.ndarray((img.shape[0],img.shape[1],3))
                for d in range(new_im.shape[-1]):
                    new_im[:,:,d] = img[:,:]

                image_dims.append((new_im.shape[0],new_im.shape[1]))

                #Predict bounding boxes on grayscale image.
                output_dict = run_inference_for_single_image(new_im[None
    ,:,:,:], detection_graph)


                #append detection from image to the detection list
                dicts.append(output_dict)

    return dicts,image_dims

def sorted_boxes(denrom,classes,sortby=0):
    concat = np.hstack((denorm,classes)).astype(int)
    sort_val = np.zeros_like(concat)
    sortind = np.argsort(denorm[:,0].argsort())
    for i in range(sortind.shape[0]):
        row = np.argwhere(sortind == i)
        sort_val[i,:] = concat[row,:]
    return sort_val

def pixel_to_depth(top,base,pixel_height):
    return (base-top) / pixel_height

def plug_to_depth(impath,boxes):
    im = np.asarray(Image.open(impath))

    top_str,base_str =impath.split('.')[0].split('_')[-2], (impath.split('
    .')[0].split('_')[-1])

    if ',' in top_str:
        top_str = top_str.split(',')[0] + '.' + top_str.split(',')[-1]

    if ',' in base_str:
        base_str = base_str.split(',')[0] + '.' + base_str.split(',')[-1]


    top,base = float(top_str), float(base_str)
    meter_per_pixel = pixel_to_depth(top,base,im.shape[0])
    mid_diff = ((boxes[:,2] - boxes[:,0]) / 2)
    mid = mid_diff + boxes[:,0]
    names = []
    depth = []
    pix = []
    image = []
    for i in range(len(mid)):

        names.append(class_to_name(boxes[i,-1]))
        depth.append(mid[i] * meter_per_pixel + top)
        pix.append(mid[i])
        image.append(impath)

    return names,depth ,pix ,image


#Denormalize the bounding box coordinates generated by the model
def denormalize(array ,image_dims):
    if array.ndim==1:
        array=array[None,:]
    denorm = np.zeros_like(array)
    h,w = image_dims
    denorm[:,0] = array[:,0] * h
    denorm[:,1] = array[:,1] * w
    denorm[:,2] = array[:,2] * h
    denorm[:,3] = array[:,3] * w
    return denorm

def name_from_path(image_path):
    name_components = image_path.split('/')[-1].split('_')[0:3]
    name = name_components[0] + '_' + name_components[1] + '-' +
    name_components[2]
    top = (image_path.split('_')[-2])
    base = (image_path.split('_')[-1].split('.jpg')[0])
    return name,float(top),float(base)


def class_to_name(c):
    if c == 1:
        name = 'hplug'
        return name
    if c == 2:
        name = 'vplug'
        return name
    if c == 3:
        name = 'scal'
        return name
    else:
        return 'Non-defined class'


def main():
    #Set the path to the folder containing the exported inference graph
    MODEL_NAME = 'inference-graph-demo'

    #adding the frozen inference graph to the path
    PATH_TO_FROZEN_GRAPH = os.path.join(MODEL_NAME,'frozen_inference_graph
    .pb')

    #Path to the label map
    PATH_TO_LABELS = 'labelmap_plugs.pbtxt'

    #Path to the images that are used for inference.
    PATH_TO_TEST_IMAGES_DIR = 'images'

    TEST_IMAGE_PATHS = glob.glob(os.path.join(TEST_IMAGE_PATHS,'*.jpg'))

    detection_graph = tf.Graph()
    with detection_graph.as_default():
        od_graph_def = tf.GraphDef()
        with tf.gfile.GFile(PATH_TO_FROZEN_GRAPH, 'rb') as fid:
        serialized_graph = fid.read()
        od_graph_def.ParseFromString(serialized_graph)
        tf.import_graph_def(od_graph_def, name='')


    predictions,image_dims = generate_predictions(MODEL_NAME,
    PATH_TO_FROZEN_GRAPH,PATH_TO_LABELS,TEST_IMAGE_PATHS,detection_graph)

    lnames=[]
    ldepth=[]
    lpix=[]
    limage=[]
    for ind,d in enumerate(predictions):
        scores=np.argwhere(d['detection_scores'] > 0.5)
        boxes = np.squeeze(d['detection_boxes'][scores])
        classes = d['detection_classes'][scores]
        denorm = denormalize(boxes,image_dims[ind])
        denorm = sorted_boxes(denorm,classes)
        names,depth,pix,image = plug_to_depth(TEST_IMAGE_PATHS[ind],denorm
    )
        lnames += names
        ldepth += depth
        lpix += pix
        limage += image


    d = {'Plug Type': lnames, 'Depth': ldepth, 'Pixel Location': lpix, '
    Source': limage}
    dataframe = pd.DataFrame(d)
    dataframe = dataframe.sort_values('Depth').reset_index(drop=True)
    dataframe.to_csv('pixel_depth_map.csv')

if __name__ == '__main__':
    main()
```

Listing 9: Show of work: summary of the pixel-depth mapping script. Dependent on local file structure and requires setup of the `Tensorflow` environment in order to run. For working script and setup guide please consult the `GitHub` repository of this thesis (Adelved, 2020)

```
# ==============================================================
# Copyright 2017 The TensorFlow Authors. All Rights Reserved.
#
# Modifications copyright 2020 Dennis Adelved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
# ==============================================================

import numpy as np
import os
import six.moves.urllib as urllib
import sys
import tarfile
import tensorflow as tf
import zipfile
from distutils.version import StrictVersion
from collections import defaultdict
from io import StringIO
import matplotlib.pyplot as plt
from PIL import Image
import glob

# This is needed since the notebook is stored in the object_detection
    folder.
sys.path.append("..")
from object_detection.utils import ops as utils_ops


def run_inference_for_single_image(image, graph):

    if 'detection_masks' in tensor_dict:
        # The following processing is only for single image
        detection_boxes = tf.squeeze(tensor_dict['detection_boxes'], [0])
        detection_masks = tf.squeeze(tensor_dict['detection_masks'], [0])
        # Reframe is required to translate mask from box coordinates to
    image coordinates and fit the image size.
        real_num_detection = tf.cast(tensor_dict['num_detections'][0], tf.
    int32)
        detection_boxes = tf.slice(detection_boxes, [0, 0], [
    real_num_detection, -1])
        detection_masks = tf.slice(detection_masks, [0, 0, 0], [
    real_num_detection, -1, -1])
        detection_masks_reframed = utils_ops.
    reframe_box_masks_to_image_masks(
            detection_masks, detection_boxes, image.shape[1], image.shape
    [2])
        detection_masks_reframed = tf.cast(
            tf.greater(detection_masks_reframed, 0.5), tf.uint8)
        # Follow the convention by adding back the batch dimension
        tensor_dict['detection_masks'] = tf.expand_dims(
            detection_masks_reframed, 0)


    image_tensor = tf.get_default_graph().get_tensor_by_name('image_tensor
    :0')
    # Run inference
    output_dict = sess.run(tensor_dict,
                           feed_dict={image_tensor: image})

    # all outputs are float32 numpy arrays, so convert types as
    appropriate
    output_dict['num_detections'] = int(output_dict['num_detections'][0])
    output_dict['detection_classes'] = output_dict[
    'detection_classes'][0].astype(np.int64)
    output_dict['detection_boxes'] = output_dict['detection_boxes'][0]
    output_dict['detection_scores'] = output_dict['detection_scores'][0]
    if 'detection_masks' in output_dict:
        output_dict['detection_masks'] = output_dict['detection_masks'][0]
    return output_dict

def generate_predictions(model_name, path_to_frozen_inference_graph,
    path_to_labels, path_to_images, detection_graph):

    dicts = []

    with detection_graph.as_default():
        with tf.Session() as sess:
        # Get handles to input and output tensors
            ops = tf.get_default_graph().get_operations()
            for op in ops:
                op._set_device('/device:CPU:*')
            all_tensor_names = {output.name for op in ops for output in op
    .outputs}
            tensor_dict = {}
            for key in [
            'num_detections', 'detection_boxes', 'detection_scores',
            'detection_classes', 'detection_masks'
        ]:
                tensor_name = key + ':0'
                if tensor_name in all_tensor_names:
                    tensor_dict[key] = tf.get_default_graph().
    get_tensor_by_name(
                tensor_name)

            for image_path in path_to_images:

                #Convert RGB image to grayscale
                image = Image.open(image_path).convert('L')
                img = np.asarray(image)

                new_im = np.ndarray((img.shape[0],img.shape[1],3))

                for d in range(new_im.shape[-1]):
                    new_im[:,:,d] = img[:,:]

                image_np = np.copy(new_im)



                # Expand dimensions since the model expects images to have
     shape: [1, None, None, 3]
                image_np_expanded = np.expand_dims(image_np, axis=0)

                # Actual detection.
                output_dict = run_inference_for_single_image(
    image_np_expanded, detection_graph)

                #append detection from image to the detection list
                dicts.append(output_dict)
    return dicts
```

```
#Denormalize the bounding box coordinates generated by the model
def denormalize(array,image):
    denorm = array.copy()
    h,w = im.shape[0:2]
    for i in range(array.shape[0]):
        ymin = int(array[i][0][0]*h);
        xmin = int(array[i][0][1]*w)
        ymax = int(array[i][0][2]*h)
        xmax = int(array[i][0][3]*w)
        #print(xmin,':',ymin,':',xmax,':',ymax)
        denorm[i] = xmin,ymin,xmax,ymax
    return denorm



def auto_annotate_xml(im_path,xml_template_path,predicted_bounding_boxes,
    output_folder = OUTPUT_DIR):

    output_path = os.path.join(os.getcwd(),output_folder)
    new_xml_path = im_path.split('/')[-1].split('.')[0] + '.xml'
    xml_copy_path = shutil.copy(xml_template_path, os.path.join(
    output_path, new_xml_path))

    tree = ET.parse(xml_copy_path)
    root = tree.getroot()

    im = Image.open(im_path)
    im = np.asarray(im)
    h,w,d = im.shape

    root.find('folder').text = output_path.split('/')[-1]
    root.find('filename').text = im_path.split('/')[-1]
    root.find('path').text = os.path.join(output_path,im_path.split('/')
    [-1])
    root.find('size').find('width').text = str(w)
    root.find('size').find('height').text = str(h)
    root.find('size').find('depth').text = str(d)

    new_objects = match_objects(root.findall('object'),
    predicted_bounding_boxes)


    for obj in root.findall('object'):
        root.remove(obj)

    for new_obj in new_objects:
        root.append(new_obj)

    for ind,obj in enumerate(root.findall('object')):

        obj.find('name').text = 'core'
        obj.find('bndbox').find('ymin').text = str((
    predicted_bounding_boxes[ind,1]))
        obj.find('bndbox').find('xmin').text = str((
    predicted_bounding_boxes[ind,0]))
```

```
            obj.find('bndbox').find('ymax').text =str( (
        predicted_bounding_boxes[ind,3]))
            obj.find('bndbox').find('xmax').text = str((
        predicted_bounding_boxes[ind,2]))



        shutil.copy(im_path,os.path.join(output_path,im_path.split('/')[-1]))
        tree.write(xml_copy_path)



def main():

    #Set the path to the folder containing the exported inference graph
    MODEL_NAME = 'inference-graph-demo'

    #adding the frozen inference graph to the path
    PATH_TO_FROZEN_GRAPH = os.path.join(MODEL_NAME,'frozen_inference_graph
    .pb')

    #Path to the label map
    PATH_TO_LABELS = 'labelmap_plugs.pbtxt'

    #Path to the images that are used for inference. Here only three
    sample images are given.
    #However, more can be added to this directory
    PATH_TO_TEST_IMAGES_DIR = 'images'

    #The output directory for the cropped core images. If no cropping is
    desired set OUTPUT_DIR = None
    OUTPUT_DIR = os.path.join('autolabel')

    TEST_IMAGE_PATHS = glob.glob(os.path.join(PATH_TO_TEST_IMAGES_DIR,'*.
    jpg'))

    detection_graph = tf.Graph()
    with detection_graph.as_default():
        od_graph_def = tf.GraphDef()
        with tf.gfile.GFile(PATH_TO_FROZEN_GRAPH, 'rb') as fid:
        serialized_graph = fid.read()
        od_graph_def.ParseFromString(serialized_graph)
        tf.import_graph_def(od_graph_def, name='')


    predictions = generate_predictions(MODEL_NAME,PATH_TO_FROZEN_GRAPH,
    PATH_TO_LABELS,TEST_IMAGE_PATHS,detection_graph)


    #Make xml files from predictions and export to output directory
    for ind,d in enumerate(predictions):
        s= np.argwhere(d['detection_scores'] > 0.5)
        boxes = np.squeeze(d['detection_boxes'][s])
        classes = d['detection_classes'][s]
        image_np = Image.open(TEST_IMAGE_PATHS[ind])
        image_np = np.asarray(image_np)
        denorm = denormalize(boxes,image_np)
        denorm = sorted_boxes(denorm,classes)


        auto_annotate_xml(TEST_IMAGE_PATHS[ind],'template.xml',denorm)

if __name__ == '__main__':
    main()
```

Listing 10: Show of work: summary of the auto-labeling script. Dependent on local file structure and requires setup of the `Tensorflow` environment in order to run. For working script and setup guide please consult the `GitHub` repository of this thesis (Adelved, 2020)

NTNU

Norwegian University of
Science and Technology