Master's thesis

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

Caroline Bakkene

Optimization of a Convolutional Neural Network for Classification of Radar Signals

Master's thesis in Electronics Systems Design and Innovation
Supervisor: Torbjørn Karl Svendsen
Co-supervisor: Jon Alm Eriksen
June 2021

NTNU
Norwegian University of
Science and Technology

Caroline Bakkene

Optimization of a Convolutional Neural Network for Classification of Radar Signals

Master's thesis in Electronics Systems Design and Innovation
Supervisor: Torbjørn Karl Svendsen
Co-supervisor: Jon Alm Eriksen
June 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Preface

This thesis is the final project in the Master of Science (MSc) degree in Electronics Systems Design and Innovation at the Norwegian University of Science and Technology. The master thesis was carried out during the spring of 2021 in collaboration with Novelda.

I would like to thank my supervisors Torbjørn Karl Svendsen at NTNU, and Jon Alm Eriksen from Novelda for their supervision and help structuring this work process. I would also like to thank Novelda for giving me the opportunity to collaborate with them for my master thesis, and for providing the data sets used to train and evaluate the neural networks.

Trondheim, 11.06.2021

Caroline Bakkene

Abstract

Novelda's Ultra Wide Band (UWB) radar is able to detect human presence by observing small movements, like breathing and heartbeats. This radar technology has many possible areas of application, and is among other things used in computers to implement touch-free log-in. The radar will reduce the energy used by the computer and increase the security by automatically log off when the user is leaving. To make sure the radar is behaving optimally, it should be able to classify whether or not there is human presence in the detection range by using information from the radar. Applying convolutional neural networks to solve classification problems can be beneficial. When using neural networks, the choice of hyperparameters can make a significant impact on the final performance.

This thesis studies the possibility to apply hyperparameter optimization to develop neural networks with improved accuracy and performance. Two different approaches were tested for the hyperparameter optimization, both using the same search space. A genetic algorithm was implemented, and several model structures were compared using this algorithm. This algorithm was used to search for the optimal hyperparameters for a set of defined models. The Python library *Keras Tuner* that uses the Hyperband algorithm for optimization was also implemented and compared to the genetic algorithm.

The results indicate that the best performance was achieved using two dropout layers with a rate of 0.5, in addition to a Max-pooling layer. When using the genetic algorithm to search for the optimal hyperparameters, the model achieved a validation loss of 0.168 and a true positive rate (Sensitivity) of 0.82 for a given false positive rate (Specificity) of 0.01. When using the Hyperband algorithm, those values were respectively 0.156 and 0.83.

Sammendrag

Noveldas ultrabredbåndsradar kan oppdage menneskelig tilstedeværelse ved å detektere svært små bevegelser, slik som pusting og hjerteslag. Denne radarteknologien har mange mulige bruksområder, og blir blant annet benyttet i datamaskiner, for å implementere berøringsfri innlogging. Radaren vil redusere datamaskinens energiforbruk og øke sikkerheten ved å logge av når brukeren beveger seg vekk fra maskinen. For å være sikker på at radaren fungerer optimalt, bør den være i stand til å bruke innhentet informasjon til å klassifisere om det er menneskelig tilstedeværelse eller ikke i radarens dekningsområde. Det kan være nyttig å bruke konvolusjonelle nevrale nettverk til å løse slike klassifiseringsproblemer.

Denne avhandlingen undersøker mulighetene ved å benytte hyperparameteroptimalisering til å utvikle nevrale nettverk med høy nøyaktighet og ytelse. To forskjellige tilnærminger for hyperparameteroptimalisering ble testet med det samme søkeområdet. En genetisk algoritme ble implementert, og flere ulike modellstrukturer ble sammenlignet ved hjelp av denne algoritmen. Denne algoritmen ble også brukt til å søke etter optimale hyperparametere for et sett med modeller. Biblioteket *Keras Tuner* i Python, som bruker *Hyperband* som algoritme for optimalisering, ble også implementert og sammenlignet med den genetiske algoritmen.

Resultatene viser at den beste ytelsen ble oppnådd ved bruk av to dropoutlag med rate 0.5, i tillegg til ett Max-pooling-lag. Ved å benytte den genetiske algoritmen til å søke etter optimale hyperparametere, oppnådde den et valideringstap på 0.168, og en sann positiv rate (sensitivitet) på 0.82 for en gitt falsk positiv rate (spesifisitet) på 0.01. Ved å benytte Hyperband som algoritme, ble disse verdiene henholdsvis 0.156 og 0.83.

Contents

List of Figures

List of Tables

Abbreviations

ANNs	Artificial Neural Networks.
CNNs	Convolutional Neural Networks.
FPR	False Positive Rate.
GA	Genetic Algorithm.
MSE	Mean Squared Error.
ReLU	Rectified Linear Unit.
RNNs	Recurrent Neural Networks.
ROC	Receiver Operating Characteristics.
SGD	Stochastic Gradient Descent.
TNR	True Negative Rate.
TPR	True Positive Rate.
UWB	Ultra Wide Band.

Chapter 1

Introduction

This chapter will introduce the motivation for this project and why it is a necessary field to study. A thorough description of the problem this thesis aims to solve will be presented. Previous work in the field of hyperparameter optimization will be discussed, as well as the development of classifiers used in Novelda. Finally, the outline of the report is presented.

1.1 Motivation

The use of machine learning and artificial neural networks are increasing, and the need for optimization methods is getting more important. A manual selection of hyperparameters for a network is challenging due to the high number of possible combinations. This will result in a random search, where experience from previously developed models is useful. For some systems the requirements for security and accuracy are high, and it is important that security breaches are avoided. To achieve the highest possible accuracy, an extensive numbers of hyperparameter combinations should be tested. To achieve this, algorithms that helps choose a set of optimal parameters that minimizes a given loss function for a neural network is beneficial.

1.2 Problem Description

Novelda's ultra wide band (UWB) radar is designed for detection of human presence in a room. It is able to detect breathing from humans and animals. This application is promising reliable precision and improved security. For

Novelda it is therefore important to be able to deliver a product that the costumers can rely on every time.
The objective of this master thesis was to explore hyperparameter optimization for a classifier. This includes some pre-processing of provided data, and the design and implementation of a convolutional neural network. Using optimization, this project aims to achieve the lowest possible validation loss and the highest positive *True Positive Rate* for a given *False Positive Rate*. The *True Positive Rate* was used as a metric due to the security Novelda promises their costumers. Novelda's radar detects human presence, and this project will focus on improving the *no presence* category. As long as there is no actual human presence in the room, the radar should not detect any presence.

With exception of a couple of earlier master theses, Novelda has not performed any research of classifiers earlier. It will therefore be interesting to explore how well a classifier is performing on their radar data.

1.3 Previous Work

Multiple theses have been performed in collaboration with Novelda. [1] was a master thesis conducted in the fall of 2016. This project aimed to reduce wrongly classified objects with a Doppler spectrum that is similar to the humans respiration, such as oscillating fans and roof-lamps. The results of this thesis is very encouraging, and it suggest that future work should include exploration of deeper convolutional networks. [2] is another master thesis that was carried out in collaboration with Novelda, in the spring of 2018. It focused on the classification of radar signals, where the classifier should be able to separate humans from animals.

In the field of hyperparameter optimization, much research has been performed in recent years. [3] describes several optimization algorithms and search components. Some of the optimization methods that are mentioned are evolutionary algorithms, ant colony optimization and particle swarm optimization. These are all *Population based metaheuristics* that focuses on improving a population. This research also focuses on the components of optimization algorithms and the importance of the search space. [4] uses an enhanced genetic algorithm for optimizing neural networks. This enhancement was based on the use of mitochondrial DNA in combination with a genetic algorithm. This resulted in an improved exploration and exploitation of the solution space. [5] discuss different approaches for hyperparameter optimization and neural architecture search. These approaches includes gradient based methods, population based methods, and Bayesian optimization. The aim of this paper is to discover more efficient methods for model configuration, including both hyperparameter optimization and neural architecture search.

1.4 Outline

Chapter 2 presents background theory that is useful for understanding the development, training and evaluation of neural networks. It also provides some theory of hyperparameter optimization, and the belonging algorithms. The third chapter cover the methods used to solve the project, and some information about the software used. In chapter 4, the results of the project are presented. These results are discussed further in chapter 5, together with suggestions for further work. Finally, in chapter 6, a conclusion of the work and result are presented.

Chapter 2

Theory

This chapter will present some background theories that are useful for this project. It is assumed that the reader has basic understanding of mathematics, signal processing and machine learning. This chapter will mainly cover the methods that are used in the project. It will also give a brief description of some other important methods.

2.1 Ultra Wide Band Radar

Ultra Wide Band (UWB) uses wide bandwidth for short-range communication, and operates using radio waves. The bandwidth of an UWB should exceed 500 MHz and have a maximum power spectral density of 75 nW/MHz. Compared with the traditional narrowband signals, the UWB offers low cost, low energy consumption and high capacity. The sensor technology that is using UWB, is commonly used in areas such as surveillance and medical supervision. [6]

2.1.1 Novelda's UWB Radar

The information about the UWB radar that the data in this project is collected from, is described on Novelda's homepage [7]: "Our groundbreaking innovation is an UWB short-range impulse radar transceiver System on Chip (SoC). Designed for human presence detection in indoor applications, it offers product innovators a unique combination of reliable sensing precision and unobtrusive, intelligent functionality."

Using the micro-Doppler effect, the radar is able to detect very small movements, even breathing and heartbeats. It has a high distance accuracy of approximately $1cm$ and a resolution of approximately $10cm$ within a range of ten meters.

2.2 Neural networks

2.2.1 Introduction to Artificial Neural Networks

Artificial Neural Networks (ANNs) are models that are designed to simulate the human brain with a large number of neurons that transmits signals to each other. These models can be used to solve several problems in machine learning, including regression and classification. ANNs are built by using multiple connected layers of neurons. A neural network consists of an input layer, an output layer, and one or more hidden layers. The networks where the data is passed from one layer to the next layer are called a *feedforward network* as illustrated in figure 2.1. Some networks have *feedback loops* where the data also are passed backwards in a loop. These networks are called Recurrent Neural Networks (RNNs).

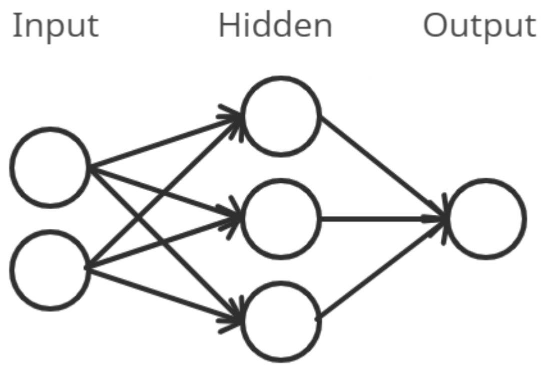


Figure 2.1: Feedforward ANN with one hidden layer. Figure modified from [8]

An ANN is created by designing an architecture for the network, and then training it by using a data set with training data. The training data that is used in machine learning consists of pre-labeled data. A separate validation and test dataset will determine the accuracy of the network.

[8] gives a thorough description of ANNs and deep learning, and the theory presented in this chapter will mainly be based on this book.

2.2.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are well suited for classification problems. In convolutional networks, all the neurons in the first layer are not connected to all the hidden neurons in the next layer. A small field of the input is connected to one neuron in the first hidden layer. This field is called the receptive field for that specific hidden neuron. A hidden layer consists of multiple channels where each channel will be searching for one specific feature in the input data. This means that every neuron in a channel has the same parameters and weights.

CNNs are mainly built of convolutional layers, pooling layers and dense layers, as shown in figure 2.2. These layers will be described in ch. 2.3.5.

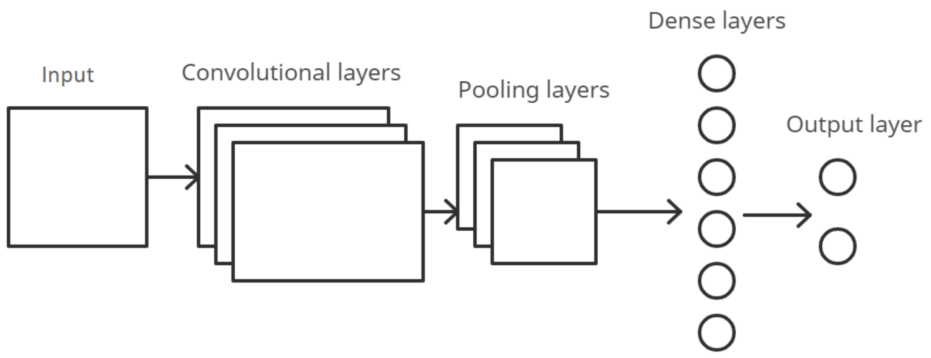


Figure 2.2: A simple CNN architecture. Modified figure from [8]

2.3 Training Neural Networks

2.3.1 Loss Function

ANNs use loss functions, also called cost functions, to evaluate how well the neural network is performing while the model is trained. The loss functions measure the difference between the true and predicted value. The values provided by the output layer are used to calculate the loss. Two of the most common loss functions are the Mean Squared Error (MSE) and the cross-entropy.

The MSE is given by eq. 2.1. The total number of inputs is represented by n, y is the true value, while a represents the value predicted by the network. Ideally should $y = a$, which would indicate that the model returns the correct value. The MSE loss function is mainly used for regression problems.

$$C = \frac{1}{2n} \sum_x ||y(x) - a||^2 \tag{2.1}$$

The cross-entropy loss function is defined by 2.2, and it is well suited for classification problems. It calculates the difference from the predicted probability to the true values. The output value is therefore in the range [0,1], and a value close to 0 is preferred as this indicates that the predicted output is close to the true output.

$$C = -\frac{1}{n} \sum_x [y \ln a + (1 - y) \ln (1 - a)] \tag{2.2}$$

Training loss and validation loss

During training of a network, the loss function is used to calculate the performance of the network. The training loss is calculated during training using the training data set, and indicates how well the model's prediction was on the same data it is trained with. The training loss will decrease over time when the model is learning more. The validation loss is calculated in the same way during training, but is using a different data set. The validation loss will therefore give an indication on how well the model is performing when presented with new data. For a perfectly designed and trained model, the validation loss should be equal to the training loss.

2.3.2 Optimization

Gradient Descent

Gradient descent is an optimization method that is used to minimize the loss of the network. The learning algorithm will find a set of weights and biases that will provide the lowest possible loss. This is done by computing the gradient for the loss function and then use this result to change the parameters.

Stochastic Gradient Descent

The Stochastic Gradient Descent (SGD) computes the gradient descent for small batches of input data, this is called *mini-batch*. Using small batches of random input data will make the learning more effective and faster. The SGD method will use short time to give a good estimate for the gradient. This will result in the network learning fast and providing an accurate result.

Adam

The Adaptive Moment Estimation, also called Adam optimization, is an efficient stochastic optimization method. It combines the best features from the AdaGrad and the RMSProp methods. The advantage of AdaGrad is the ability to deal with sparse data, while the RMSProp is suited for stochastic objectives. The Adam optimization method is based on the individual computation of learning rates for the different parameters. The Adam optimization method requires less memory than other methods, and does only use first-order gradients. [9] shows that Adam is a robust method and is applicable for a wide range of optimization problems.

2.3.3 Activation Functions

The activation function in machine learning is the function that decides how the output of the current layer should be presented. It calculates a weighted sum of the input and adds a bias to it. This can be done to achieve non-linearity between the input and the output, as most of the activation functions are non-linear. The choice of activation functions is important as it decides what kinds of predictions the model can make. [10] describes some of the most common activation functions.

Rectified Linear Unit

The Rectified Linear Unit (ReLU) activation function is a non-saturating activation function. It does also avoid the vanishing gradient problem, which could be an issue for the sigmoid function. When using ReLU, not all neurons are activated at the same time which makes it more effective than other activation functions. The ReLU function is given by equation 2.3:

$$f(x) = max(0, x) \tag{2.3}$$

The output of the function is zero if the weighted input is negative, and for a positive input it will output the input value directly. The ReLU is a common activation function that is used in most kinds of neural networks. The ReLU activation function is illustrated in figure 2.3.

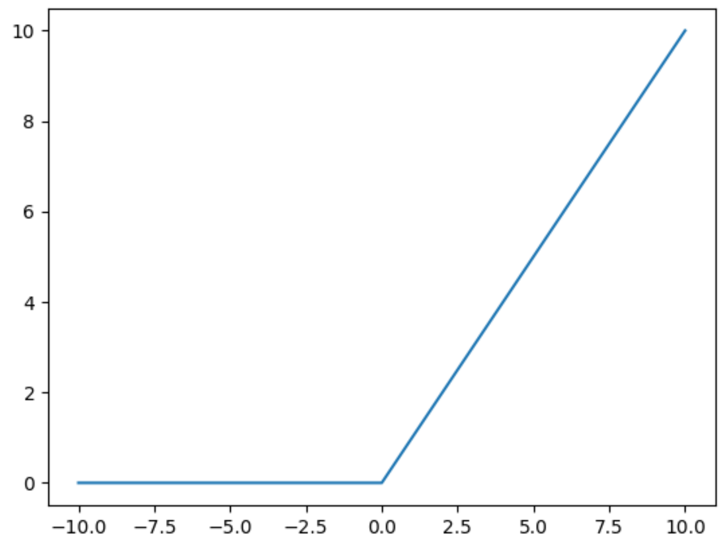


Figure 2.3: ReLU function

Sigmoid and Softmax

For classification problems, the sigmoid or the softmax activation function are mainly used in the output layer. The sigmoid function will give outputs between 0 and 1 which represent the probability for each class. The sigmoid function is showed in figure 2.4, and its function is presented in eq. 2.4

$$\sigma(z) = \frac{1}{1+e^{-z}} \tag{2.4}$$

The softmax function will also give outputs between 0 and 1, but those values will always sum to one. This means that the outputs of the softmax function are related, and will also give the probabilities for each class. The softmax function is mainly used for multiclass classification while the sigmoid is used for binary classification. It is given by the function presented in eq. 2.5. K is the number of classes in the classifier and z_i are the input values to the softmax function. Normalization of the values is performed by using the denominator in this expression.

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{2.5}$$

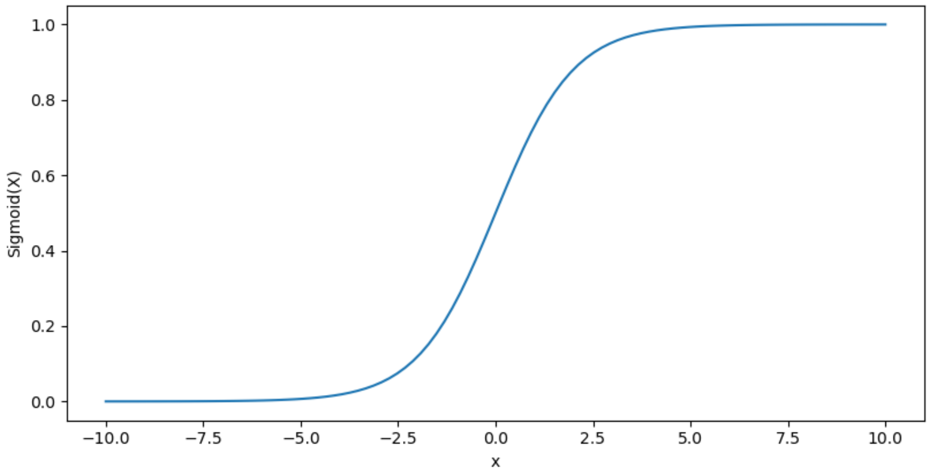


Figure 2.4: Sigmoid function

2.3.4 Overfitting

A network is presented with a given dataset during training, and it will often provide good results using this data. This doesn't necessary indicate that it is a good model. The performance of the model is dependent of how generalized it is, and its ability to adapt to new data. Overfitting may occur when a model is trained too much, while too little training could cause underfitting. Problems regarding overfitting can also appear when the amount of data is limited compared to the parameters in the network.

There are several techniques to reduce overfitting when it occurs. Some options could be to increase the amount of training data, reduce the complexity of the model, or use regularization techniques.
The regularization techniques used for this project was dropout layers and L2 regularization, which will be described in ch. 2.3.5.

2.3.5 Techniques and layers in CNNs

Convolutional layers

In convolutional layers, filters of a given size are used to search for important patterns in the data. The size of this filter, often called kernel size, should be chosen so that meaningful patterns will be detected. When a filter has been applied to the input data, a feature map is created. Each map can detect one specific feature as each map has shared weights and biases. The number of filters should be decided based on the complexity in the data, and how many features that are desired to map. An illustration of a convolutional layer with the creation of one neuron in a feature map is shown in figure 2.5.

Dense Layers

Dense layers, also called fully-connected layers, are layers that connect every input in one layer with every neuron in the next layer, as illustrated in figure 2.6. For CNNs, the last couple of layers are normally fully-connected. Fully-connected networks with several dense layers consist of a high number of trainable parameters, while a CNN with only one or two dense layers will have remarkably less parameters. A reduction in parameters results in faster learning which is an advantage for building deeper networks. In CNNs the fully-connected layers are used for the classification at the end of the network, while the convolutional layers are used for the feature extraction.

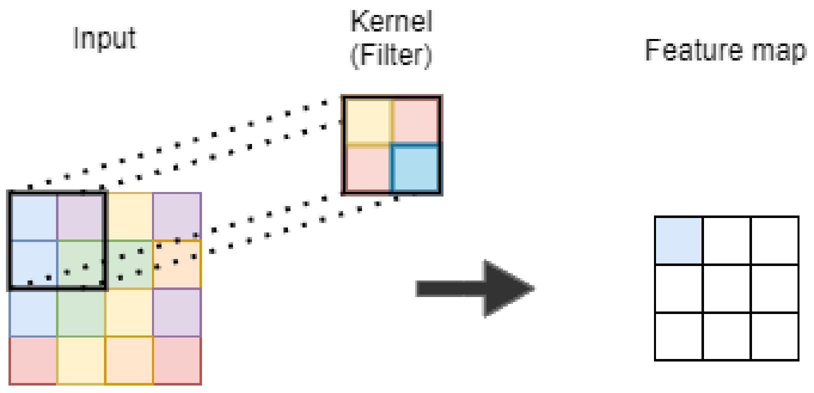


Figure 2.5: Convolutional layer

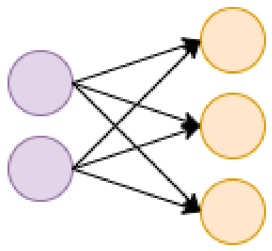


Figure 2.6: Dense layer

Pooling Layers

The pooling layer is downsampling the convolutional layer to decrease complexity, and it is normally used directly after a convolutional layer. Two variations of pooling layers, are Max-Pooling and L2-Pooling. Max-Pooling will output the maximum value in the pooling region, while L2-Pooling will output the squared root of the sum of the squared values in the pooling region. An illustration of an one dimensional Max-Pooling with a 3x1 pool-size is shown in figure 2.7.

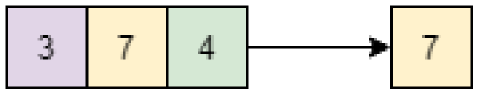


Figure 2.7: 3x1 Max-Pooling

Dropout Layers

Dropout is a regularization method that is used to prevent overfitting when training the data. The dropout modifies the network by temporarily removing hidden neurons that are chosen randomly, as shown in figure 2.8. The amount of neurons that are removed is decided by the dropout rate that has a value: $0 < Rate < 1$. After one epoch, the weights and biases are updated, and a different set of neurons is removed. The effect of using dropout is similar to averaging the result of several different networks, which will help to reduce overfitting.

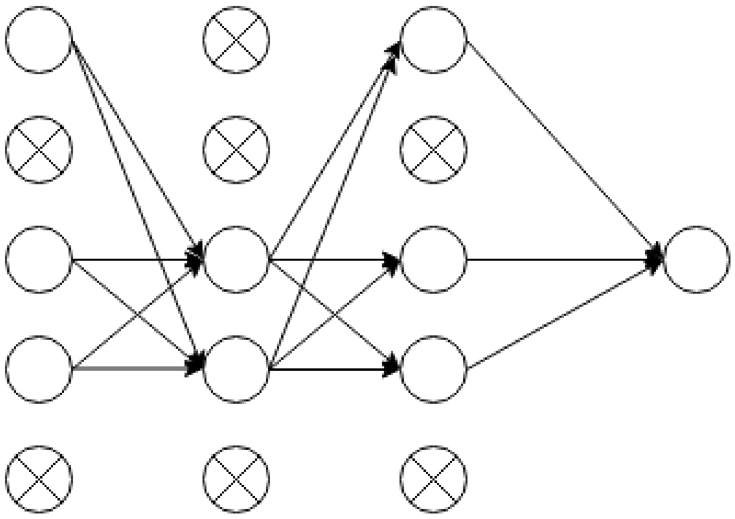


Figure 2.8: Neural network after applying dropout

Figure 2.9 illustrates that overfitting is prevented by using a dropout layer. The two graphs show the validation loss (using a smoothing filter) for two identical models built using two convolutional layers and two dense layers. The only difference is that the model illustrated with a blue graph is using a dropout layer between the two convolutional layers. No other regularization methods are applied. The model with a dropout layer has a loss that is still decreasing at 15 epochs. The yellow graph, the model without dropout, reaches its minimum validation loss at 4 epochs, before the loss increases rapidly. An increasing validation loss indicates that the model fails to generalize to new data.

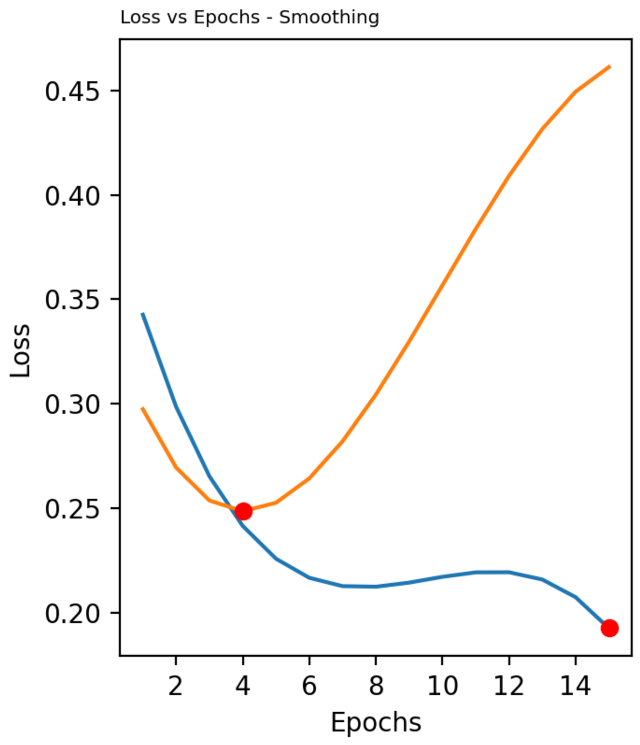


Figure 2.9: The effect of using a dropout layer. Plot from the project.

L2 Regularization

L2 regularization, also called weight decay, is a commonly used regularization technique that is used to prevent overfitting. The idea of this technique is to add an extra term to the cost function that sums the squares of all weights in the model. This is performed to prevent some parameters to grow very large, and will apply penalties for the weight size to the cost function. The L2 regularization term is shown in eq. 2.6.

$$\frac{\lambda}{2n}\Sigma w^2 \tag{2.6}$$

n is the size of the training set, w is the weights, and λ is the regularization parameter. This parameter should be chosen to be a value in the range $0 - 0.1$. A small value will make the network prefer to minimize the original cost function, while a larger value will make the network prefer to learn small weights.

2.4 Classification evaluation

This section will cover some methods for evaluation of the ANNs in this project. The theory in this section is based on [11].

2.4.1 Accuracy

The accuracy of the network is given as the percentage of correctly classified predictions. For a trained network, a test data set is used for evaluation. This evaluation tests the network using unseen data, the test data set, and measures how many predictions are classified as the correct class.

2.4.2 Sensitivity and Specificity

When evaluating the performance of a classifier, the sensitivity and specificity of the model are important statistical measures. The sensitivity, also called True Positive Rate (TPR), gives the probability of the instance classified as positive truly being positive. The specificity, also called True Negative Rate (TNR), indicates how well the classifier has identified the negatives. The metrics are defined mathematically using the number of false and true positives (FP and TP) and false and true negatives (FN and TN):

$$\textit{sensitivity (TPR)} = \frac{TP}{TP + FN} \tag{2.7}$$

$$\textit{specificity (TNR)} = \frac{TN}{TN + FP} \tag{2.8}$$

2.4.3 Receiver Operating Characteristics (ROC)

Receiver Operating Characteristics (ROC) graphs are used for evaluation of classifiers, and the trade-off between the sensitivity and specificity can

be helpful when comparing different models. ROC curves are used in this project when comparing different optimization methods.
The ROC curve is plotting the TPR (sensitivity) against the False Positive Rate (FPR), which is defined by $1 - \textit{specificity}$. These plots can be used to observe whether the system meets the requirements. For some systems, a high FPR can have large consequences. The ROC curve can therefore indicate if the system is achieving a sufficient hit rate compared with the false alarm rate. An example of a ROC curve from the project is shown in fig. 2.10.

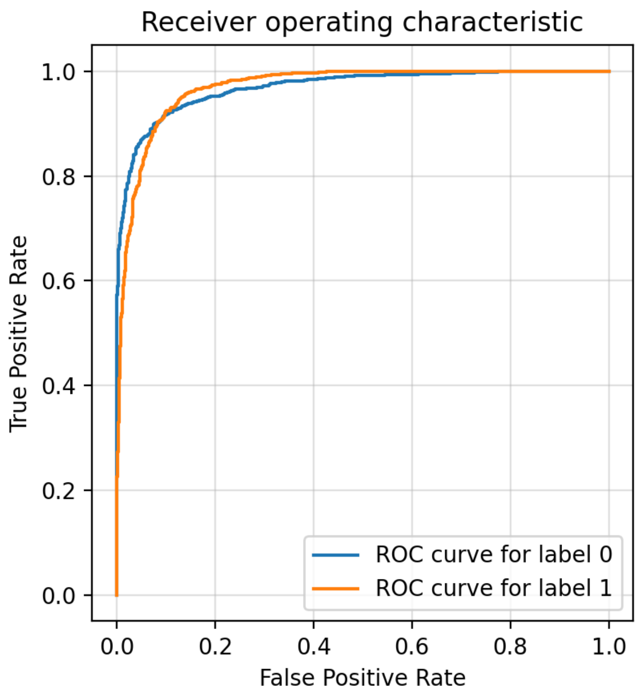


Figure 2.10: ROC curve from project

2.4.4 Confusion Matrix

The confusion matrix is an other technique used for performance evaluation. The two axes represent the true values and the predicted values using a classifier. The matrix will give an insight in how well the classifier is predicting the correct classes. Figure 2.11 show an example of a confusion matrix from this project, with normalized values. Each row will sum to 1, and represents the probability of the classifier predicting the true label or not.

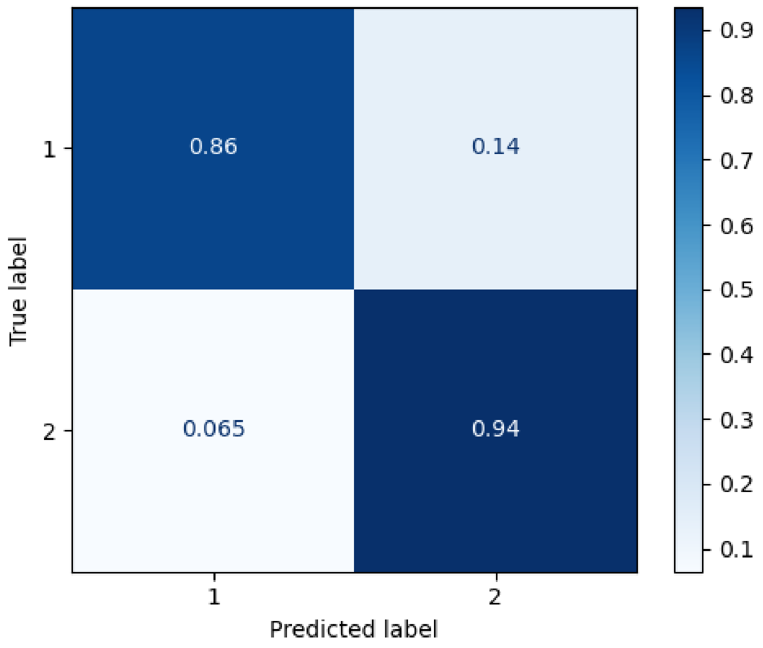


Figure 2.11: Confusion matrix from project

2.5 Hyper Parameter Optimizing

2.5.1 Hyperparameters

Hyperparameters in a neural network are variables that must be chosen before the model can be trained. These are not determined by the learning algorithm, like the weights and biases in the network are. Hyperparameters should be determined for both the network structure and for the training algorithm. The parameters that should be determined for the network structure is e.g. the number of units in the hidden layers, number of filters in the convolutional layers, the dropout rate and the activation functions. The type and number of layers in the architecture is also important to determine. The most common hyperparameters for the training algorithm are the number of epochs to train, the learning rate and the batch-size.

The choice of hyperparameters is important when building and training a network, as the results achieved depend on this. When optimizing a network, different combinations of hyperparameters should be tested. The different combinations of hyperparameters are extensive, and this is why algorithms for hyperparameter optimization are important.

2.5.2 Genetic Algorithm

Genetic Algorithm (Genetic Algorithm (GA)) is an evolutionary based optimization method that is based on the improvement of a population through generations. GA is inspired by the process of natural selection and genetic crossover in the nature. Using genetic crossover and mutations, the genetic algorithm aims to find the optimal hyperparameters efficiently within the solution space. [4] describes optimization of a CNN using genetic algorithm.

The three main functions in a GA are the selection, crossover and mutation. The selection function is choosing parents from the population to mate. The members in the population with the highest fitness value should be chosen for the next generation. These members should be processed by the crossover function, which chooses what genetic information should be passed on to their offspring. Some small amount of mutation can also be applied to ensure genetic diversity between the generations. The mutation rate should be low to avoid large variations between the generations, which could change the genetic algorithm to a random search. The full structure of a basic genetic algorithm is illustrated in figure 2.12.

The number of generations must be chosen, and will define how many iterations the algorithm will run. Another important parameter is the population size, which is defined as the number of solutions in the first generation. The output of the GA will be the set of optimized hyperparameters that resulted in the best performing model.

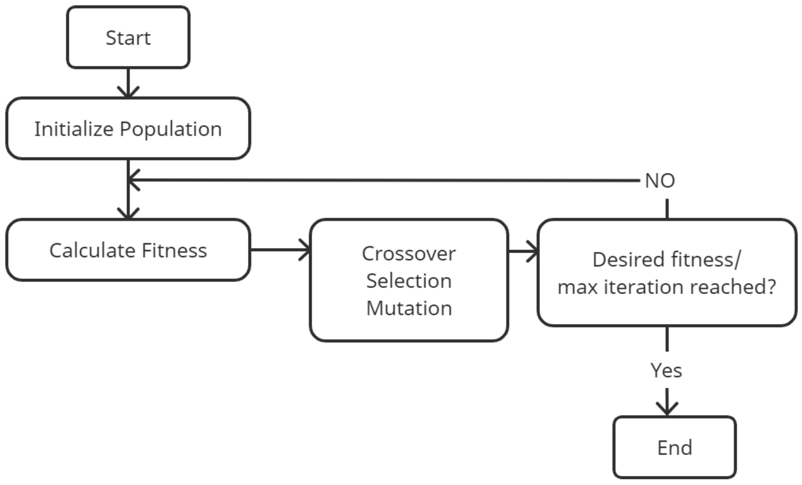


Figure 2.12: Genetic algorithm. Figure modified from [4]

2.5.3 Hyperband Algorithm

Another algorithm that will be used in this project to optimize hyperparameters is the Hyperband algorithm. This algorithm use the principle of the successive halving algorithm to achieve higher efficiency than a standard random search. The successive halving algorithm will randomly choose sets of hyperparameters from a given search space, and evaluate the performance of all configurations. The algorithm performs a number of trials where the half with lowest performance for each trial is discarded. This approach will result in one final model, with the so far best combination of hyperparameters.

The Hyperband algorithm does also use an early-stopping strategy, which makes it very efficient. If a model being trained performs poorly, the training is stopped to avoid using unnecessary time. Hyperband will allocate more time for models performing well. [12] claims that the Hyperband algorithm is 5 to 30 times faster than Bayesian optimization methods.

The algorithm takes two inputs, the resource parameter R and the pruning factor η. The pruning factor decides the number of models eliminated at each halving, while the resource parameter normally defines the maximum number of epochs.

Chapter 3

Methods

This chapter describes the methods used in this project to build and optimize a neural network. The data set and the tools that are used will also be presented.

3.1 Data sets

The data sets used in this thesis was provided by Novelda. The data was created by Novelda using their UWB radar. This radar was recording a person that performed different tasks, and collecting the range bins from the measurements. The person was moving and staying in pre-determined areas in proximity of the radar. All the data provided was collected during one recording, before it was divided into different segments. The different segments consists of the person staying in specific zones relative to the radar.

The data was divided into a training, validation and test data set. The validation data set was used together with the training data set during training to estimate the accuracy and loss after each epoch. The test data set was not a part of the training process, and was introduced to the model after training to calculate the accuracy using new data.

The sizes of the data sets are presented in table 3.1. The data consists of a high number of data samples, that are represented in the outermost dimension. Each data sample is composed of a series of time samples. For every time sample, 32 frames with a resolution of four frames per second were obtained. Each frame contains several range bins that represent the distance of the reflected echo signal from the radar. The innermost dimension contains two values, respectively the real and imaginary part of the echo observation.

Data set	**Size**
Training data	11480 x 32 x 18 x 2
Test data	3828 x 32 x 18 x 2
Validation data	3824 x 32 x 18 x 2

Table 3.1: Sizes of the provided data sets.

In addition to these data sets, associated labels were provided. The labels specify which class the data belongs to. The original data sets had four classes. The radar could either detect presence or no presence, and it could reject or accept the candidate. As this project should focus on improving classification for presence/no presence, both the rejected and accepted class are combined. This results in two classes from the data set, *no presence* and *presence.* One example from each class is plotted and visualized in figure 3.1. The data was applied a Savitzky–Golay filter to smooth the data to make it easier observe the patterns. The data representing *Presence* is more distinct and shows a clear signal with higher amplitudes, while the *No presence* data appears more noisy.

3.2 Pre-processing of data

Pre-processing of the data was done to prepare the data for training. The data provided by Novelda was already divided into different data sets with associated labels, so a minimum of pre-processing was necessary. Using standardization as a scaling technique, the data was re-scaled to avoid large variations. This is performed by subtracting the mean and divide by the standard deviation. The standardization was performed in Python and was applied to every value for all the data sets.

For each time sample in the data sets, the first five range bins from the radar were removed. This data were mostly noise, and did not contain much

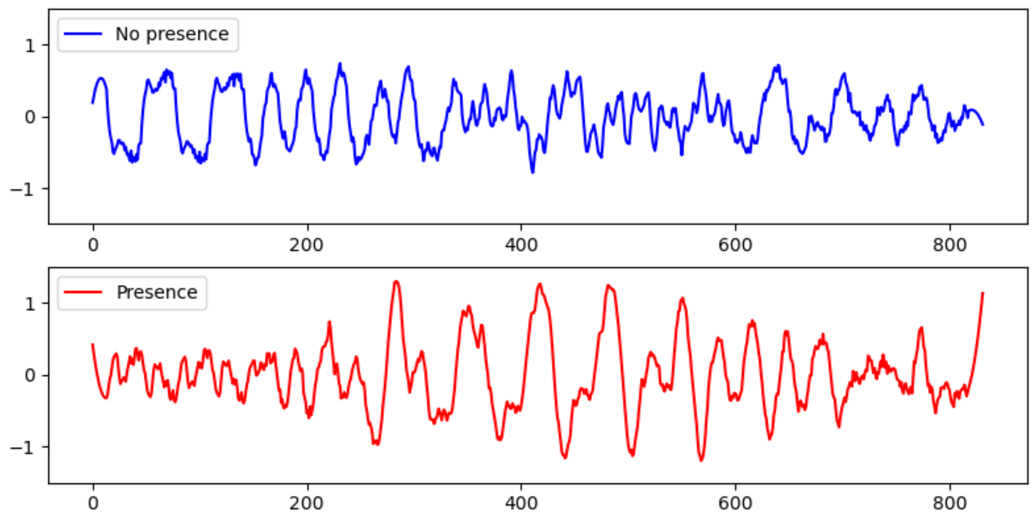


Figure 3.1: Visualization of the data representing the classes *Presence* and *No presence*

valuable information. The remaining data was more accurate, and could make it easier for the network to learn.

3.3 Software tools

3.3.1 Python

For this project, Python was used as the programming language [13]. This was chosen because the author had previous knowledge of this language, and because it provides several libraries for building ANNs.

3.3.2 Tensorflow and Keras

The Tensorflow library was used to create and train neural networks for this project [14]. Tensorflow is an open sourced framework used for machine learning. Keras is an API that simplifies the implementations in Tensorflow [15]. It has multiple built-in functions that make designing and training of ANNs easier.

3.3.3 Scikit-learn

Scikit-learn is a machine learning module in Python, used for predictive data analysis [16]. It is used in this project for model evaluation, like the ROC curve and the confusion matrix.

3.3.4 Keras Tuner

A library that helps to pick the optimal set of hyperparameters for a Tensorflow model [17]. It has four available tuners that can be used for the hypertuning. In this project, the Hyperband tuner was applied.

3.4 Optimization of hyperparameters

The aim of this project was to find a method for optimizing the hyperparameters of a classifier. As described in ch. 2.5.1, the combination of hyperparameter values are extensive and very difficult to optimize manually. In this project, two different methods have been implemented to find the optimal hyperparameters. In ch. 2.5.2, the theory of the genetic algorithm was described. This algorithm has been implemented in Python, and this implementation and the belonging search space of the genetic algorithm will be further discussed. In addition to the genetic algorithm, hyperparameter optimization using a library called *Keras Tuner* has been tested.

The metrics that mainly were used to evaluate the different models, were the validation loss and the ROC curve. For the validation loss, it was desirable to find the models that achieved the lowest loss. When analyzing the ROC curves, a FPR of 0.01 was the maximum accepted rate for Novelda, and the associated TPR was therefore used for comparison.

3.4.1 Search space

When using algorithms for hyperparameter optimizing, the search space should be defined. The search space is a pre-determined domain that contains the possible solutions for the parameters that should be optimized. Table 3.2 shows the search space that was used in this project. The range of the numbers was chosen to be relatively large, to avoid restricting the algorithms too much. Based on knowledge of other CNNs used in similar projects, these limits should be sufficient to allow the optimization algorithms to do a thorough search. A larger range was avoided as this should not be necessary, it would be more time consuming, and could produce very large models.

Parameter	**Minimum**	**Maximum**
Number of filters, Conv layer 1	5	100
Size kernel, Conv layer 1 1	3	5
Number of filters, Conv layer 2	10	120
Size kernel, Conv layer 2	3	5
Units Dense layer	10	150
Epochs	10	20

Table 3.2: Search space

3.4.2 Initial hyperparameters

Some of the hyperparameters that were necessary to build and train a model were chosen before the optimization. This was done to make sure that the models compared in the results had some consistency, and because parameters like the output activation function had to be specified to get the desired output.

Epochs

As shown in table 3.2, the number of epochs during the search for hyperparameters was between 10 and 20. This number was only used for the search, and a pre-determined number of 50 epochs was used for the final training. The models with optimized hyperparameters were trained using these parameters and 50 epochs, to make all the models comparable. If different numbers of epochs were used, it would be more challenging to compare the loss and accuracy of the models.

Activation functions

For all the layers, except the output layer, the *ReLU* activation function was used. This was used due to its efficiency and to avoid the vanishing gradient problem. For the output layer, the *Softmax* activation function was applied. This project aimed to solve a multiclass classification problem, and the softmax function is well suited for this. As mentioned in ch. 2.3.3, the softmax function will output the normalized probabilities for each class.

Optimizer

The optimization technique used when implementing models in this project, was the Adam optimization, that was described in ch. 2.3.2. This was chosen due to its generalized use, the adaptive learning rates and the robustness.

Learning rate

No specific learning rate was set in this project. The Adam optimizer that was used, has an initial learning rate of 0.001. As mentioned earlier, Adam does also use adaptive learning rates for the different parameters.

Loss function

Categorical cross-entropy was used as the loss function in the implementation of the models. The cross-entropy function was described in ch. 2.3.1. Categorical cross-entropy is a version of this that is intended for multiclass classification. One of the reasons to employ the cross-entropy for classification, is that it gives a good indication on how certain the prediction was. Given that the classifier has two classes, it could potentially output the probabilities $[0.51, 0.49]$, and still classify the input to the right class. The cross-entropy loss function is therefore beneficial to understand how well the predictions actually were, even when they classified the input correctly.

Pooling size

The pooling size that was used for the MaxPool-layers was decided to be 2. Pooling layers are used to reduce complexity in the model, and the effect of using these will be explored. The size of 2 was decided because this provided the best results in some tests performed early in this project. A larger pool size reduced the accuracy.

3.5 Architecture search

In this project, the aim was to optimize a classification model. As described earlier, a hyperparameter search within a given search space was performed. When optimizing a model, the architecture of the model should also be studied.

3.5.1 Initial architecture

An initial CNN model was used, and different improvements were performed using this model as a base for multiple tests. This initial model consisted of two convolutional layers and two dense layers, where the last of the dense layers were the output layer of the model. This architecture was chosen based on previous knowledge of how convolutional networks are built. More than one convolutional layer can provide some advantages regarding the classification, but will also increase the time consumption. The dense layers at the end perform the classification task, and for this project 2 dense layers were used, as this is common in basic CNN architectures. This model architecture is presented in figure 3.2

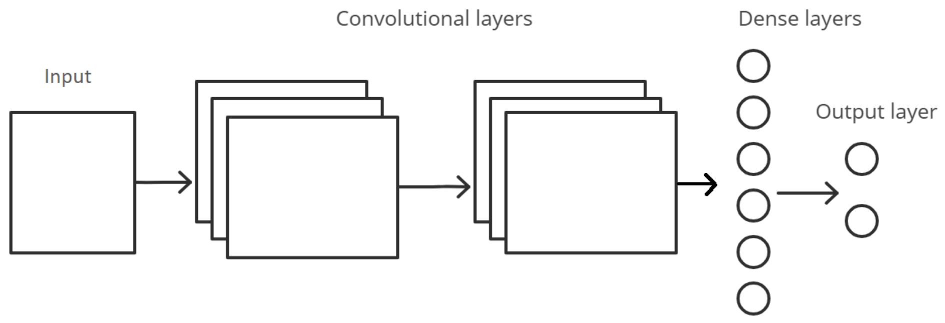


Figure 3.2: Initial CNN model

More convolutional and dense layers could have been applied, but it would also have increased both the number of parameters and the time consumption. This thesis focused on building a high performing network, using as few parameters as possible.

3.5.2 Exploring CNN architectures

As mentioned in the last sub-chapter, ch.3.5.1, some tests were performed to optimize the initial CNN architecture used. The variations that were investigated are listed below:

- The effect of L2 regularization and dropout layers. These methods were tested separately.
- Different combinations of dropout layers and L2 regularization were tested.
- The position and number of Max-pooling layers in combination with dropout layers.

The features from the best models achieved in each test were combined to build an optimal model architecture.

3.6 Genetic Algorithm

3.6.1 System setup

As discussed in ch. 2.5.2, the GA was constructed using three main functions. The selection, crossover and mutation functions are based on natural evolution. The implementation of these functions, and how to build a GA, was inspired by a git repository[1]. A short summary of the hyperparameter search using the genetic algorithm is illustrated in figure 3.3.

The genetic algorithm, with its evolutionary functions, was implemented using Python. Functionality that allowed the genetic algorithm to optimize pre-defined CNNs were also implemented. A class with the necessary parameters and their search spaces, in addition to the initial model was defined. When the model and search space were defined, the fitness function was implemented. This function was used to calculate the fitness, or the accuracy of each combination of hyperparameters. This was used to decide which solutions that performed best. The evolutionary functions mentioned earlier were also implemented. The selection function was returning a list with

[1]https://github.com/ahx-code/CNN-with-GA-Keras

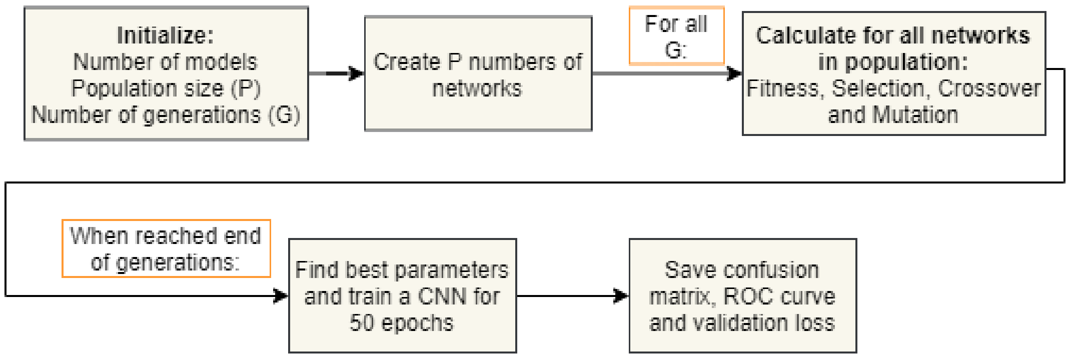

Initialize:
Number of models
Population size (P)
Number of generations (G)
Create P numbers of networks
For all G:
Calculate for all networks in population:
Fitness, Selection, Crossover and Mutation
When reached end of generations:
Find best parameters and train a CNN for 50 epochs
Save confusion matrix, ROC curve and validation loss

Figure 3.3: Implementation of hyperparameter optimization using GA

the solutions in the population that were used for the next generation. The crossover function chose two of these solutions, and created two off-springs based on this. The effect of this function was that the population of possible solutions increased for each generation. These off-springs were passed on to the mutation function. The mutation rate used in this project was 0.1, which increased the epoch and unit number with a random number between 0 and 5, in 1 out of 10 cases. As discussed in 2.5.2, a high mutation rate should be avoided as it may change the genetic algorithm to a random search as the variations between generations become very large.

For hyperparameter search and training of a model using the genetic algorithm, the initial population size and the number of generation should be decided. For the tests used in this project, two combinations of numbers have been used. The first were $Population = 2$ and $Generations = 10$. These were used on smaller tests to see how many layers or which regularization that should be used. These numbers were chosen because they are large enough to do a good search, but not too time consuming. For some larger tests where the optimal hyperparameters should be decided for an optimized architecture, $Population = 4$ and $Generations = 20$ were used. These were used to conduct a more thorough search, evaluating a higher number of combinations.

After the hyperparameter search was finished, the model was trained one final time using these parameters and 50 epochs. A validation training set was used for each epoch, and the training and validation loss were plotted. The loss functions were also applied a Savitzky–Golay filter which has the purpose of smoothing the data. This was used to visually be able to see how the loss varied, as this often has a lot of small and rapid variations. When training a model, the loss and accuracy of the model are not always best at the final epoch. Callbacks were used for each epoch during training to be able to save the best model observed and its weights. This resulted in the program being able to save the model with the highest performance independently of the number of epochs. This was necessary as a high number of epochs were used to ensure that all models being compared would reach its best performance.

The evaluation of the trained model consists of several parts. First the accuracy is calculated using the test data set and the loss function. This accuracy is presented with a percentage of how well the model performed when presented with new and unseen data. Then, the predicted y-values are found using the test data set. The comparison of the true and predicted values is used to create both the ROC-curve and the confusion matrix, as described in ch. 2.4.

3.6.2 Multiple models

For evaluation of the different models that were optimized, it was necessary to compare them to each other. Functionality that made it possible to define several model architectures were implemented with the genetic algorithm. The algorithm performed the given number of generations on each defined model, and repeated this for all of the models. All models used the same search space defined in table 3.2. When all the models were optimized and trained using the GA, the results were represented such that the models could be compared. The ROC curves and loss curves for the models were plotted in the same figure to be able to compare them.

3.7 Keras Tuner

The Keras Tuner library in Python was another method used to search for optimized hyperparameters. Keras Tuner requires that a model structure is given, and that the search space is defined. This search space was the same as for the GA, and is shown in table 3.2.

The Hyperband algorithm, as described in ch. 2.5.3, was used as tuner for the hyperparameter search. The metric to minimize, also called objective, was the validation loss. The Hyperband was finding the best models of each trial to carry forward to the next trial. For each trial the number of models was halved, resulting in one optimal set of hyperparameters for the model. The Hyperband algorithm takes two inputs in addition to the hypermodel, the pruning factor (η) and the resource parameter (R). For this project a $\eta = 3$ was chosen, and the resource parameter was decided to be $\textit{Max epochs} = 20$. A larger η would result in an increased number of models being eliminated at each halving, and 3 was the initial value defined for the function in Python. The maximum number of epochs was chosen to be large enough to evaluate the performance of the models.

When an optimal model architecture was found using the genetic algorithm, the optimized hyperparameters of this model are returned. To verify the hyperparameters found using the GA, a separate search was performed using the Keras Tuner on the same model architecture. A comparison of these search methods was conducted to find the optimal hyperparameters.

Chapter 4

Test and Results

In this chapter, the testing performed on the CNNs and the associated results will be presented. It is mainly the loss and ROC curves that will be presented as results for the networks. Some confusion matrices will also be introduced, and the remaining matrices are included in appendix A.

The first section will briefly describe the procedure used for testing. This was described more detailed in chapter 3. The next sections present the results achieved using the various tests.

4.1 Test Procedure

The code and implementation of the CNNs were created using Python, as described in ch. 3.3. During training of the networks, the hyperparameters of the models were saved locally, as well as the ROC curves, confusion matrices, and the loss curves.

To develop the ROC curves and confusion matrices, prediction in Keras was used. This prediction function uses the test data set to predict the output of the network. These predicted values were compared to the true values, and the relationship between true and predicted values were used to develop ROC curves and confusion matrices. The results obtained using the test data set was quite interesting, as it tests the CNN in new situations.

The accuracy, that will be presented as results for the different tests, was estimated using the test data set in the evaluation function in Keras. The validation loss was obtained by using the validation data set during training of the models.

As described in ch. 3.5.2, different variations of regularization techniques and architectures were tested using the genetic algorithm. Using the results from these tests, an optimized model structure was developed. Hyperparameter optimization was performed on this model to achieve the best possible loss and accuracy. This optimization was achieved using both the genetic algorithm and the Keras tuner, and comparing these.

4.2 Optimization of the Architecture

The following tests of different architectures and regularization techniques were performed using the genetic algorithm with an initial population of 2 and 10 generations. These searches gave an indication of which model architecture would provide the best results. All of the experiments were performed using the same basis, to make sure they were comparable.

The model shown in figure 4.1 is used as a base for the different experiments, as described in ch. 3.5.1.

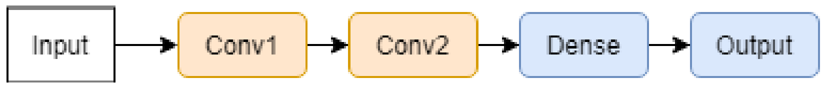


Figure 4.1: Initial model

4.2.1 Test of Initial Model

The initial model used in this project was illustrated in figure 4.1. Before optimizing this model, its performance was tested. The results are presented in table 4.1.

Model	Max Accuracy	Min validation loss	TPR for FPR=0.01
Initial model	90.7 %	0.23	0.71

Table 4.1: Results: Accuracy, Loss and TPR using the initial model

The best validation loss of this model was 0.23, and as seen in figure 4.2 it was constantly increasing. Overfitting of the model was the most likely reason for this, and different regularization techniques was later tested to reduce this.

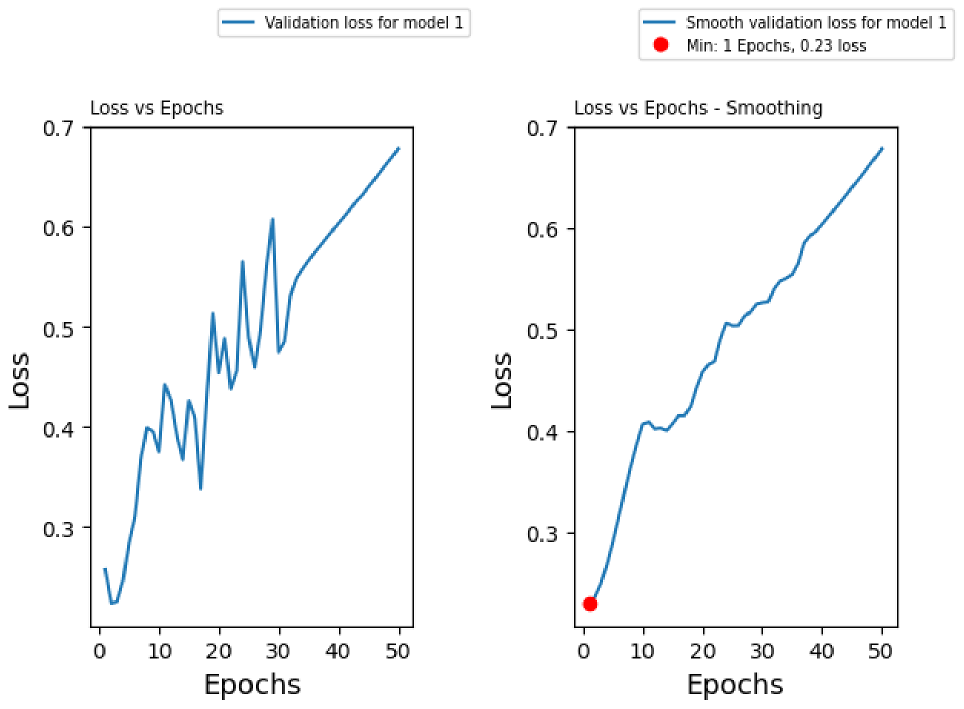


Figure 4.2: Validation loss vs epochs for the initial model. Smoothed graph to the right.

4.2.2 Effect of Dropout layer

When testing the effect of dropout layers in the neural network, different dropout rates were used. The dropout layer was added between the two convolutional layers from fig. 4.1. The dropout rates with corresponding accuracy, validation loss and TPR are presented in table 4.2. The validation loss plotted against the epochs is illustrated in figure 4.3.

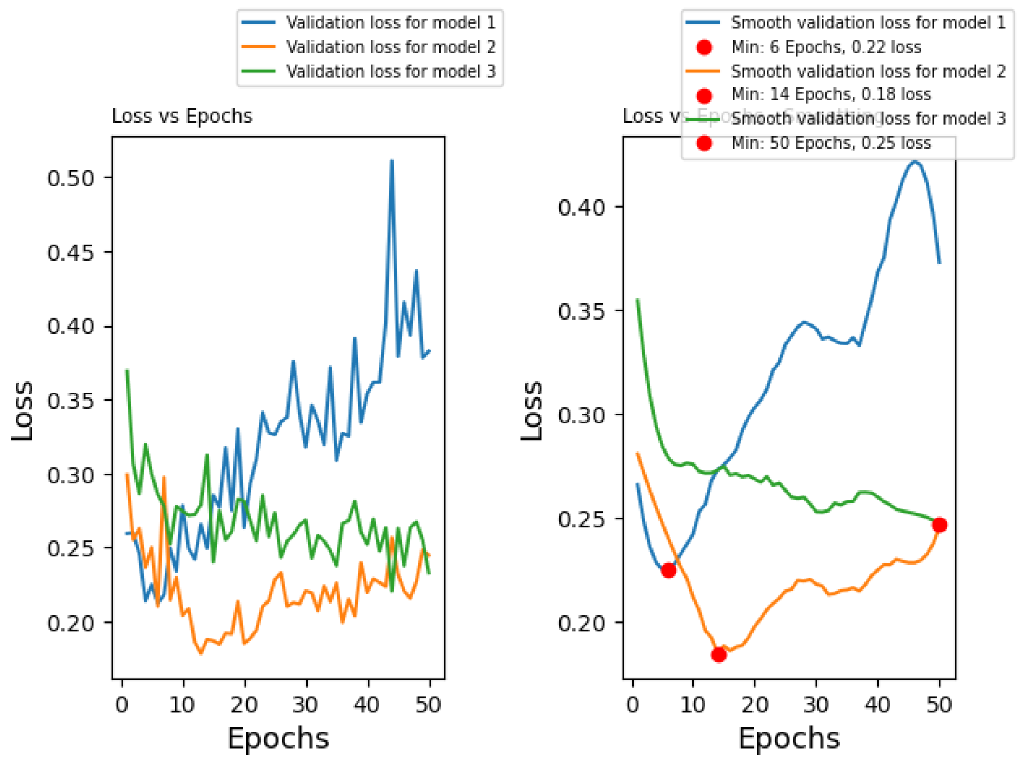


Figure 4.3: Validation loss vs epochs for different dropout rates. Smoothed graph to the right.

Figure 4.3 shows that the validation loss for model 1 with the lowest dropout rate reaches its minimum after 6 epochs, and it is then increasing significantly. Model 2, with a dropout rate of 0.5, is decreasing steadily for the first 14 epochs until it reaches its minimum of 0.18. After this it is increasing, but not as steep as for model 1. Model 3 has a slowly decreasing loss, and has achieved 0.25 in validation loss by the 50th epoch. It is possible that this model eventually could have provided the lowest loss, but this

Model	Dropout Rate	Max Accuracy	Min validation loss	TPR for FPR=0.01
Model 1	0.25	93.3 %	0.22	**0.78**
Model 2	0.50	**94.2** %	**0.18**	0.77
Model 3	0.75	91.1 %	0.25	0.77

Table 4.2: Results: Accuracy, Loss and TPR for different dropout rates

would cause a slower model to train.

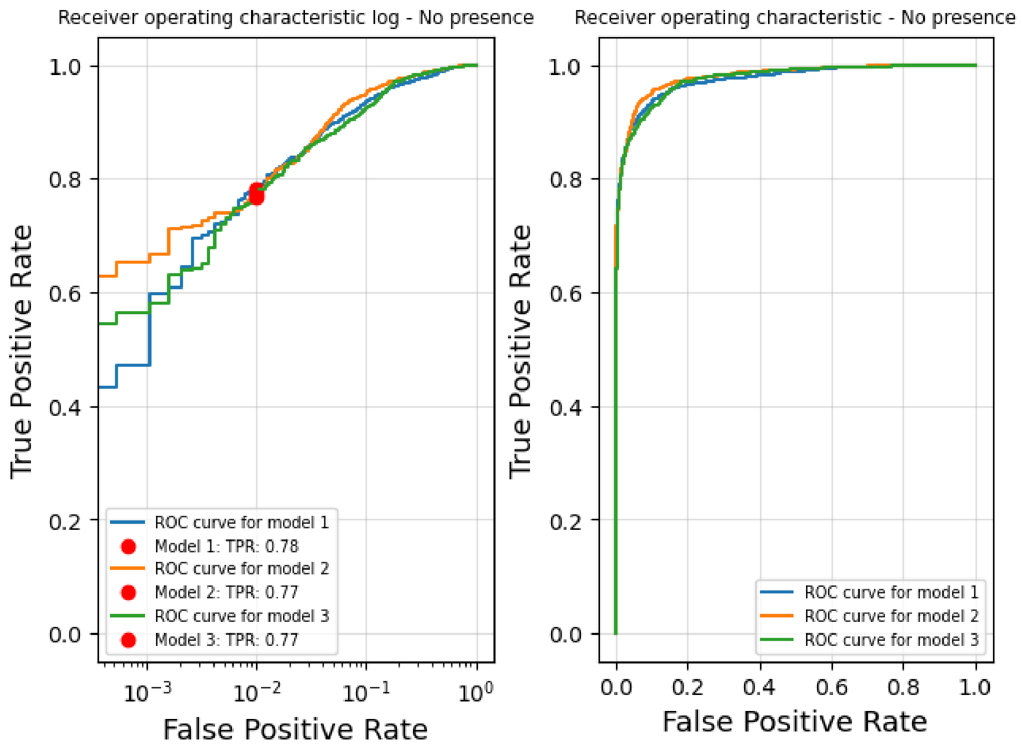
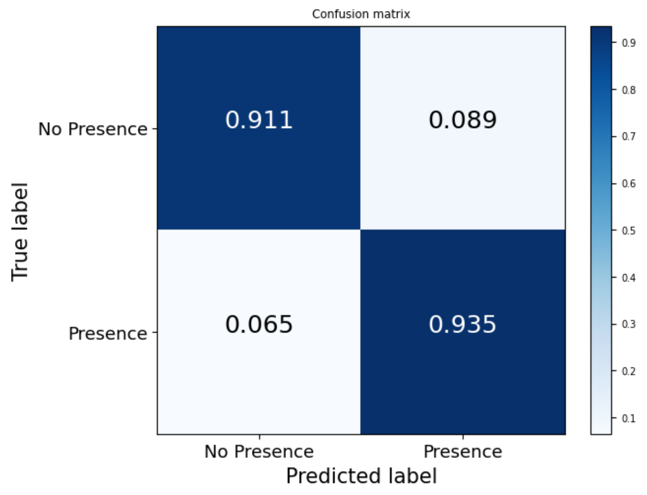
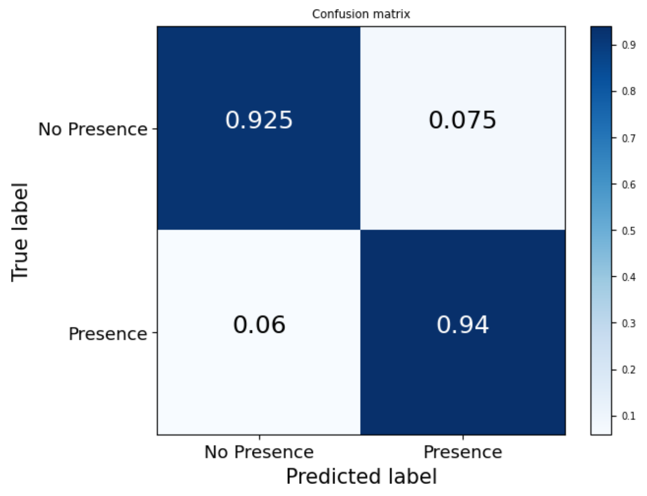


Figure 4.4: ROC curves (log and normal) for different dropout rates

Table 4.2 shows that model 2 performed best in terms of accuracy and validation with a dropout rate of 0.5. Using this rate, the model achieved a maximum accuracy of 94.2%, a validation loss of 0.18 and a TPR of 0.77 for a FPR=0.01. Investigating the logarithmic ROC curve in figure 4.4, model 1 with a TPR of 0.78, performed marginally better than model 2 and 3. Using the confusion matrices for model 1 and 2 to compare, a total evaluation indicates that model 2 with a dropout rate of 0.5 is optimal. These matrices are displayed in fig. 4.5a and 4.5b, and it shows that model 2 predicted *No presence* correctly in 92.5% of the cases, while model 1 detected this label correctly in 91.1% of the cases.



(a) Confusion matrix model 1, Dropout rate = 0.25



(b) Confusion matrix model 2, Dropout rate = 0.5

4.2.3 Effect of L2 Regularization

The effect of using L2 regularization was tested using different regularization strengths (λ). The same strength was used on all layers in the model, except for the output layer where no regularization was applied. Using the result from 4.2.2, a dropout layer with a rate of 0.5 was applied between the two convolutional layers. Table 4.3 presents the accuracy, validation loss and TPR for the different values of λ. The corresponding ROC curves and validation loss curves are shown in fig. 4.6 and 4.7.

Model	λ	Max Accuracy	Min validation loss	TPR for FPR=0.01
Model 1	0.0001	**94.0 %**	**0.24**	**0.80**
Model 2	0.001	93.6 %	0.27	0.79
Model 3	0.01	91.0 %	0.27	0.76
Model 4	0.1	88.0 %	0.35	0.70

Table 4.3: Results: Accuracy, Loss and TPR for different L2 regularization strengths

From the results presented in table 4.3, the best accuracy, loss and TPR were achieved for model 1. It achieved a validation loss of 0.24 and a TPR of 0.80. The conclusion was therefore that model 1 using $\lambda = 0.0001$ was the best option.

When observing the loss in figure 4.6, it is visible that the graph is still decreasing for model 3 and 4. It was known that L2 regularization complicates the computations, and could therefore result in a model that is slower to train. The aim was to find an efficient model, and the results obtained using 50 epochs was therefore used.

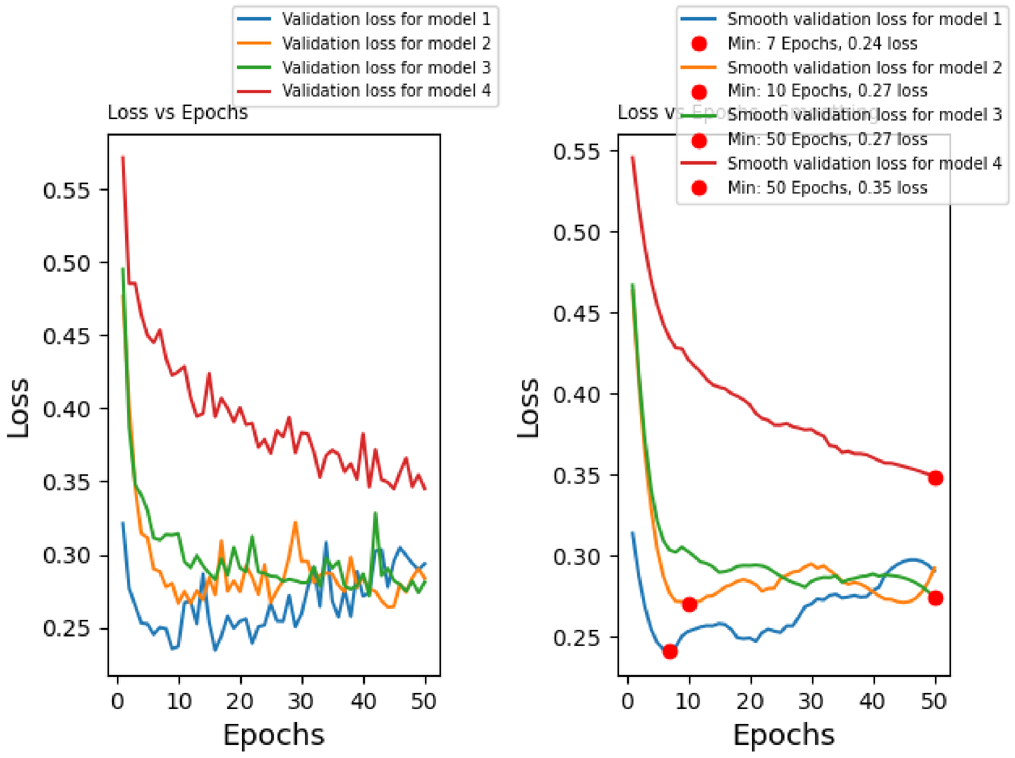


Figure 4.6: Validation loss vs epochs for different L2 regularization strengths. Smoothed graph to the right.

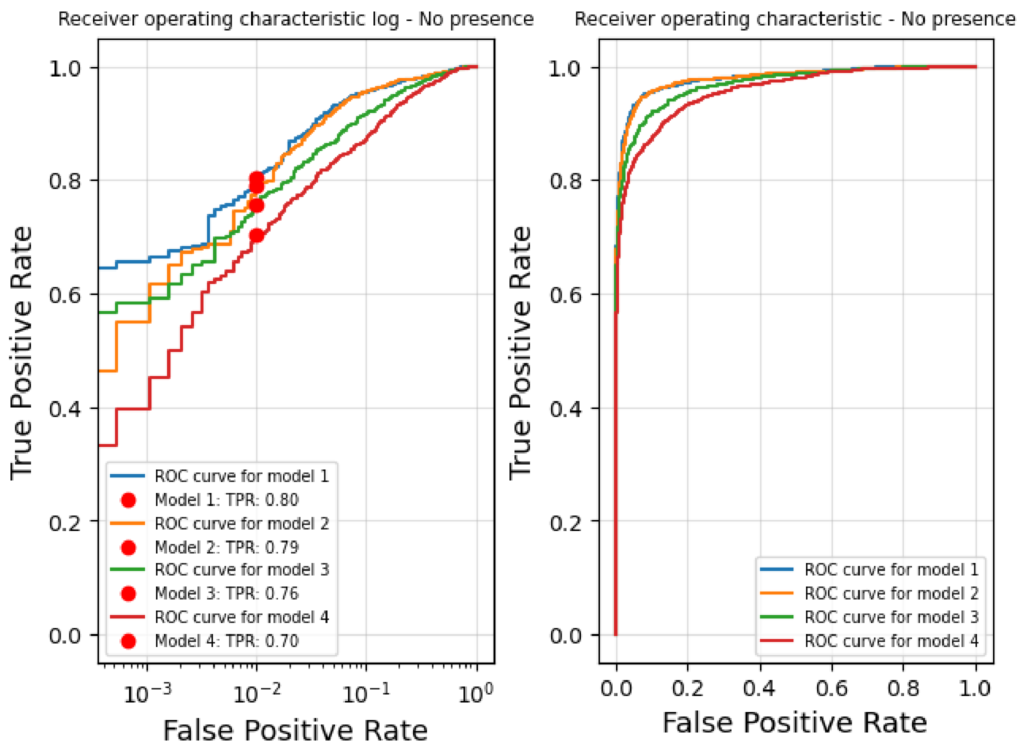


Figure 4.7: ROC curves (log and normal) for different L2 regularization strengths.

4.2.4 Combination of Regularization Techniques

Using the results from 4.2.2 and 4.2.3, a combination of the regularization techniques was tried and compared to the use of only one technique. Four models were tested to decide which combinations that resulted in the best performing model. It was not given that the combination of the best results from 4.2.2 and 4.2.3 would provide the best performance. The model architecture was as shown in 4.1, and the regularization techniques were added to this model. The combinations of techniques and values that were tested are listed in table 4.4. The dropout layer was applied between the convolutional layers while the L2 regularization was applied to all hidden layers. A dropout rate of both 0.5 and 0.25 was tested, due to the even results for these rates. The L2 regularization strength of $\lambda = 0.0001$ provided superiorly better results regarding the validation loss, so only this strength was tested further. For this test, 80 epochs was used instead of the 50 epochs that had been used previously. In ch. 4.2.3 it was observed that the models used longer time to converge when applying L2 regularization. It was therefore interesting to test with a higher number of epochs.

Model	λ	Dropout Rate
Model 1	0.0001	0.5
Model 2	0.0001	0.25
Model 3	0.0001	No dropout
Model 4	No L2	0.5

Table 4.4: Combination of regularization techniques used for each model

Model	Max Accuracy	Min validation loss	TPR for FPR=0.01
Model 1	92.5 %	0.23	0.78
Model 2	91.7 %	0.23	0.78
Model 3	91.2 %	0.28	0.70
Model 4	**93.3 %**	**0.20**	**0.81**

Table 4.5: Results: Accuracy, Loss and TPR for different combinations of L2 and dropout

The results of this test are presented in table 4.5. Model 4 was overall performing better than the other models tested, with no L2 regularization and 0.5 as dropout rate. The validation loss for this model was 0.20, and it achieved a TPR of 0.81. The graphs are presented in figure 4.8 and 4.9.

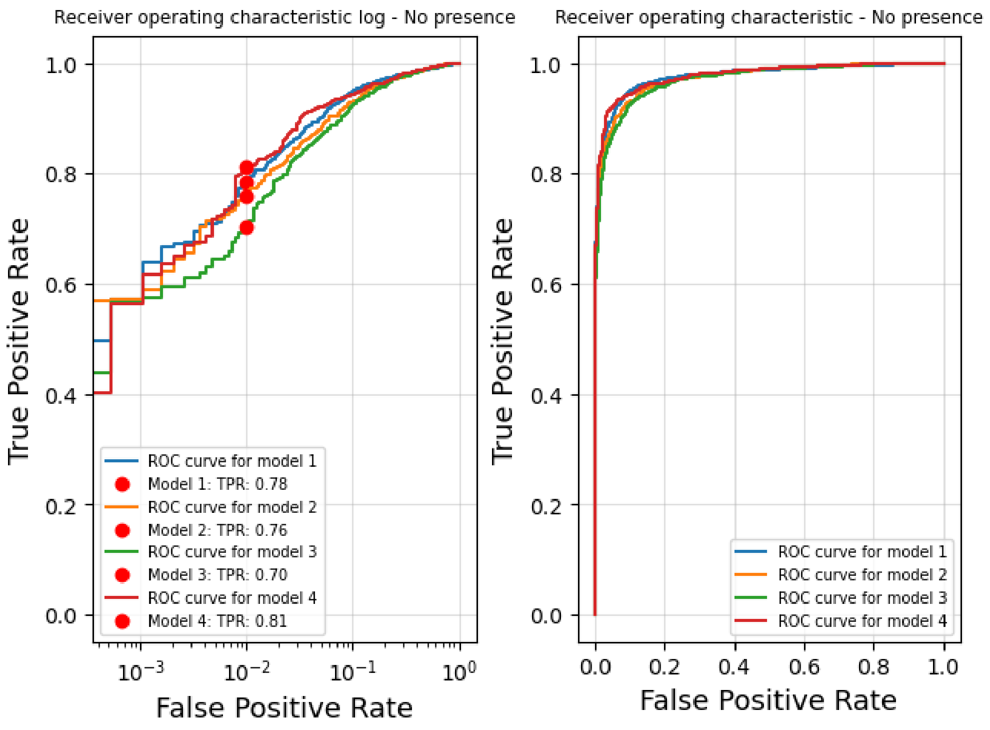


Figure 4.8: ROC curves (log and normal) for different combinations of L2 and dropout

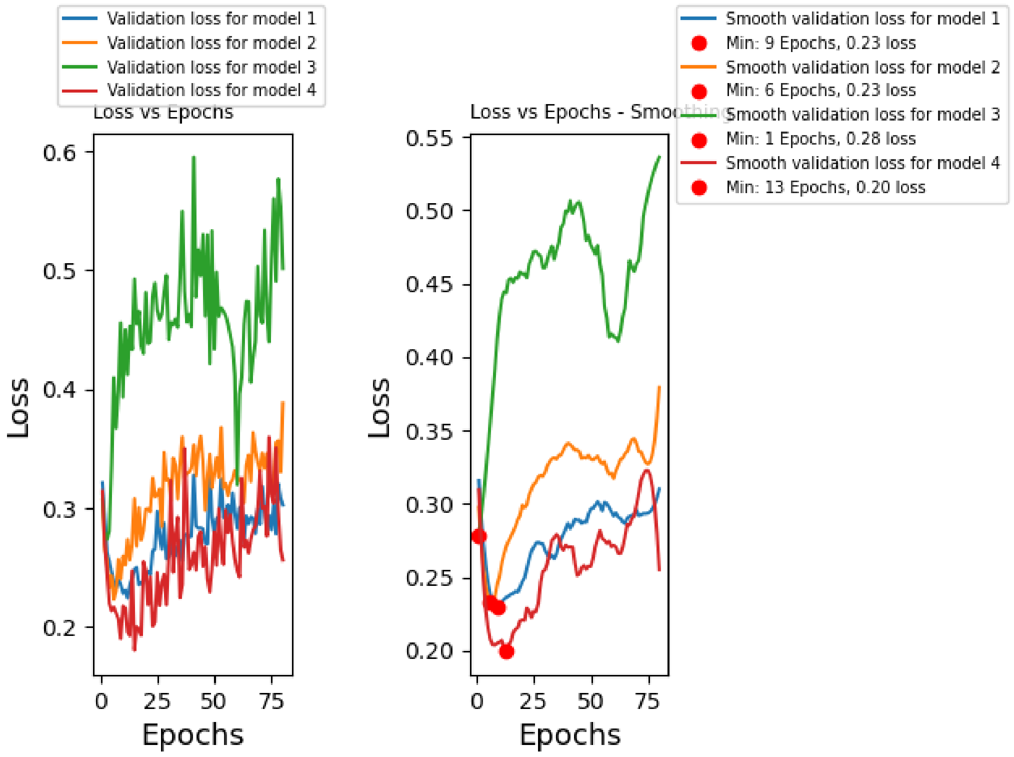


Figure 4.9: Validation loss vs epochs for combinations of L2 and dropout. Smoothed graph to the right.

4.2.5 Max-Pooling layers

In convolutional networks there are common to use Max-Pooling layers to decrease the complexity. The effects of using Max-pooling, and the number and location of these layers was investigated. Based on the results from ch. 4.2.4, the combination of 0.5 as dropout rate and no L2 regularization was carried on for further tests.

When used in combination with a dropout layer, the max-pooling layer was applied first. The order of these layers would probably not make any significant difference. The order was decided such that the max-pooling layer decreased the complexity and number of neurons before removing neurons using dropout.

Table 4.6 shows the combinations and locations of the Max-pooling and dropout layers that were tested. The results are presented in table 4.7, and the corresponding graphs are shown in figure 4.10 and 4.11.

Model	Model structure
Model 1	Max-pooling + Dropout after the first convolutional layer
Model 2	Dropout after first convolutional layer. Max-pooling + Dropout after second convolutional layer.
Model 3	Max-pooling + Dropout after first convolutional layer. Dropout after second convolutional layer.
Model 4	Max-Pool + Dropout after both convolutional layers

Table 4.6: Different locations and number of Max-Pooling layers for each model

Model	Max Accuracy	Min validation loss	TPR for FPR=0.01
Model 1	93.2 %	0.19	0.78
Model 2	**94.1 %**	**0.17**	**0.80**
Model 3	93.5 %	0.19	0.79
Model 4	93.2 %	0.18	0.76

Table 4.7: Results: Accuracy, Loss and TPR for the Max-Pooling

Observing the results from table 4.7, model 2 has the highest performance regarding all metrics. The results for all four models were even, but as model 2 performed slightly better, this was the model that was used for further tests. This model used dropout after the first convolutional layer, and max-pooling + dropout after the second convolutional layer. In figure 4.10, the loss for model 2 is still decreasing at 50 epochs. This model will be tested using a higher number of epochs in the hyperparameter search in ch. 4.3.

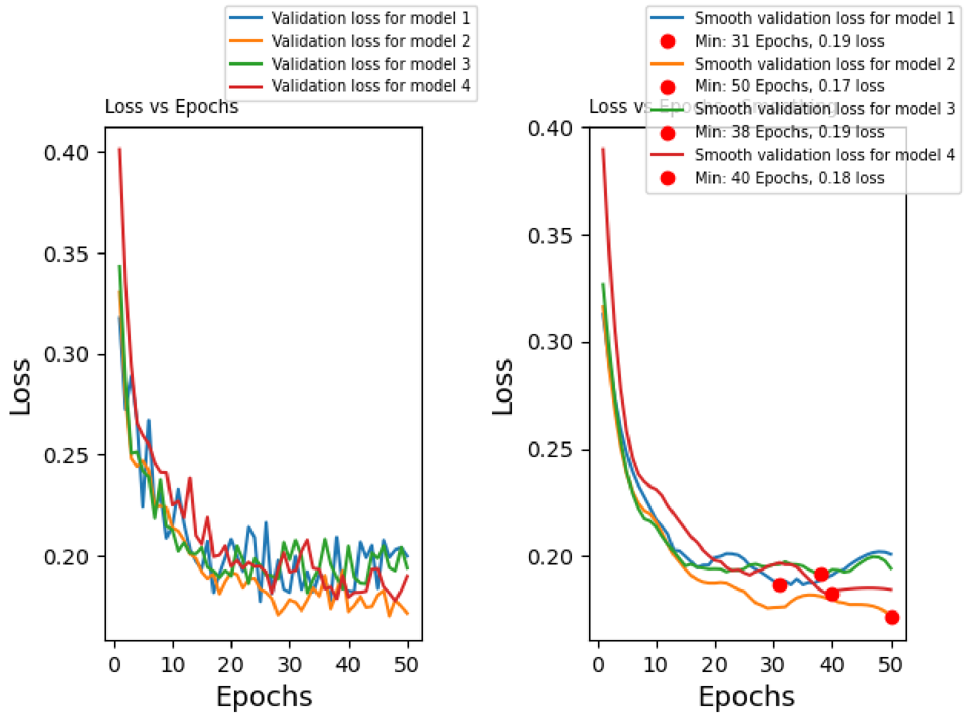


Figure 4.10: Validation loss vs Epochs for different combinations of Max-Pool and Dropout. Smoothed graph to the right.

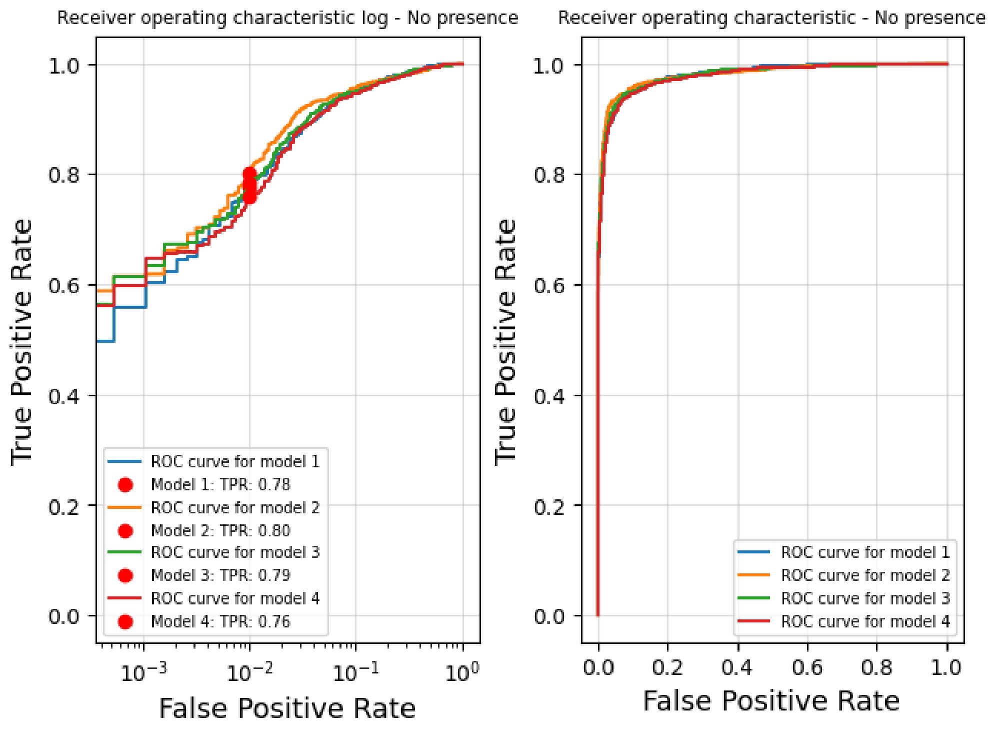


Figure 4.11: ROC curves (log and normal) for different combinations of Max-Pool and Dropout

4.3 Hyperparameter Search

Based on the previous results, an optimal architecture was decided. A hyperparameter search using two different algorithms were used to find the optimal parameters, and the results will be presented in this section.

4.3.1 The Optimized Architecture

From ch. 4.2 the following results were conducted:

- Dropout rate = 0.5 gave the best overall results of the tested rates.
- A L2 regularization strength, $\lambda = 0.0001$ provided the best performance of the tested strengths.
- Only using dropout as regularization gave better results than the combination of dropout and L2, or the use of only L2.
- The combination of a dropout layer after the first convolutional layer and a max-pooling after the second one, provided the highest performance.

Figure 4.12 illustrates the model structure of this optimized architecture. Both dropout layers used 0.5 as rate, and the pool-size for the Max-pooling layer was 2.

4.3.2 Hyperparameters Using Genetic Algorithm

The hyperparameters that were optimized are listed in table 4.8. This consists of the number of filters and the kernel size for both the convolutional layers, and the number of neurons (units) for the dense layer. The epochs are also listed in this table, as the different combinations of hyperparameters were not trained for a given number of epochs during the search. The genetic algorithm was performing a random number between 10 and 20 epochs during the search, as described in ch. 3.4.2. This number could increase throughout the generations, as this is one of the parameters affected by the mutation function. This search was used to give an indication on the performance of the models, and the final test used to collect results in this section was performed using 80 epochs. For this test, the optimized architecture

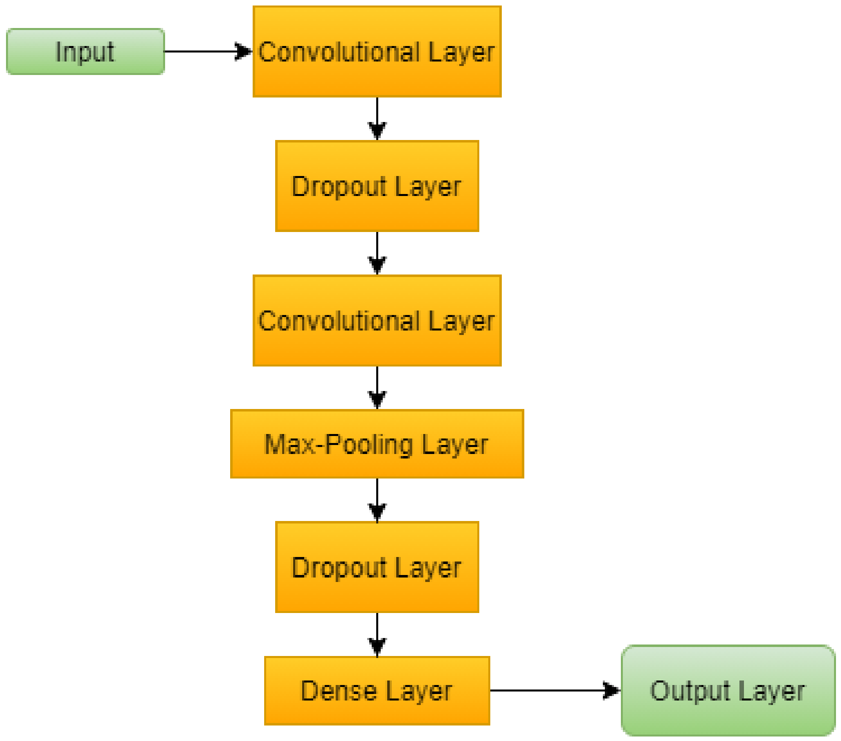


Figure 4.12: Optimized CNN architecture

in figure 4.12 was used in the genetic algorithm with $Population = 4$ and $Generations = 20$.

Four tests were performed when searching for hyperparameters using the genetic algorithm. The resulting parameters are listed in table 4.8, and the corresponding results are presented in table 4.9. Observing the values for the hyperparameters, some of them were consistent throughout the tests, while some varied more. Kernel sizes of 5 resulted overall in the best performance. The number of filters for both convolutional layers was also decided to be relatively large compared with the search space, but with some more variations for layer 2. The parameter that varied the most throughout the tests was the number of units in the dense layer, with values from 36 to 105.

Using the optimized parameters, the network was able to achieve a validation loss down to 0.168 in test 4. The remaining tests had ≈ 0.18, which is only a minor difference from test 4. Test 3 achieves a TPR $= 0.82$, which

Hyperparameter	Test 1	Test 2	Test 3	Test 4
Number of filters, Conv layer 1	95	76	84	72
Kernel size, Conv layer 1	5	5	5	5
Number of filters, Conv layer 2	117	52	83	112
Kernel size, Conv layer 2	5	3	5	5
Units Dense layer	36	56	105	57
Epochs	22	38	25	32

Table 4.8: Hyperparameters found using the genetic algorithm

Test	Max Accuracy	Min validation loss	TPR for FPR=0.01
Test 1	**94.0** %	0.179	0.81
Test 2	93.6 %	0.184	0.76
Test 3	93.9 %	0.176	**0.82**
Test 4	93.8 %	**0.168**	0.79

Table 4.9: Results: Accuracy, Loss and TPR for the hyperparameter search using GA

was the best TPR achieved so far in the project. The results from test 3 are presented in figures 4.13, 4.14 and 4.15. In figure 4.13, the training loss is also included in the non-smoothed plot. The training loss is descending towards zero, while the validation loss is converging around $0.17 - 0.20$. The validation loss can be observed closer in the smoothed graph to the right in the figure. A validation loss that is higher than the training loss indicates that the model is overfitted. The converging of validation loss has been consistent for most of the tests in this project.

An additional test using 150 epochs was performed to observe how the loss was changing over time. This test showed that the minimum validation loss was achieved at 38 epochs, before it slowly stared to increase. This loss function is presented in figure 4.16. The plotted results for the remaining tests in this section are presented in appendix B.

4.3.3 Hyperparameters Using Keras Tuner

The Keras Tuner library in Python was also used to search for optimized hyperparameters. The same optimized model as used with the genetic algo-

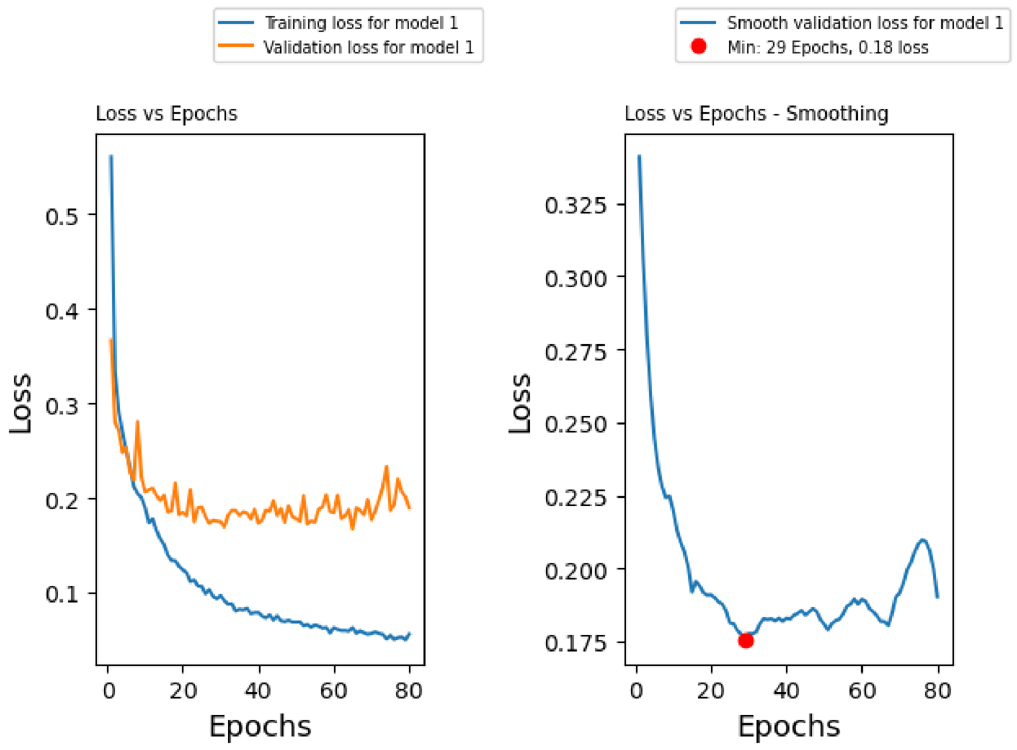


Figure 4.13: Loss vs Epochs for hyperparameter search using Genetic Algorithm. Smoothed graph to the right. (Test 3)

rithm, as illustrated in figure 4.12, was implemented. The hyperparameters that were optimized and the belonging search space were also identical as for the GA.

Hyperparameter	Test 1	Test 2	Test 3	Test 4
Number of filters, Conv layer 1	83	99	91	85
Kernel size, Conv layer 1	5	5	3	5
Number of filters, Conv layer 2	56	90	120	100
Kernel size, Conv layer 2	3	5	3	5
Units Dense layer	94	134	138	136

Table 4.10: Hyperparameters found using Keras Tuner

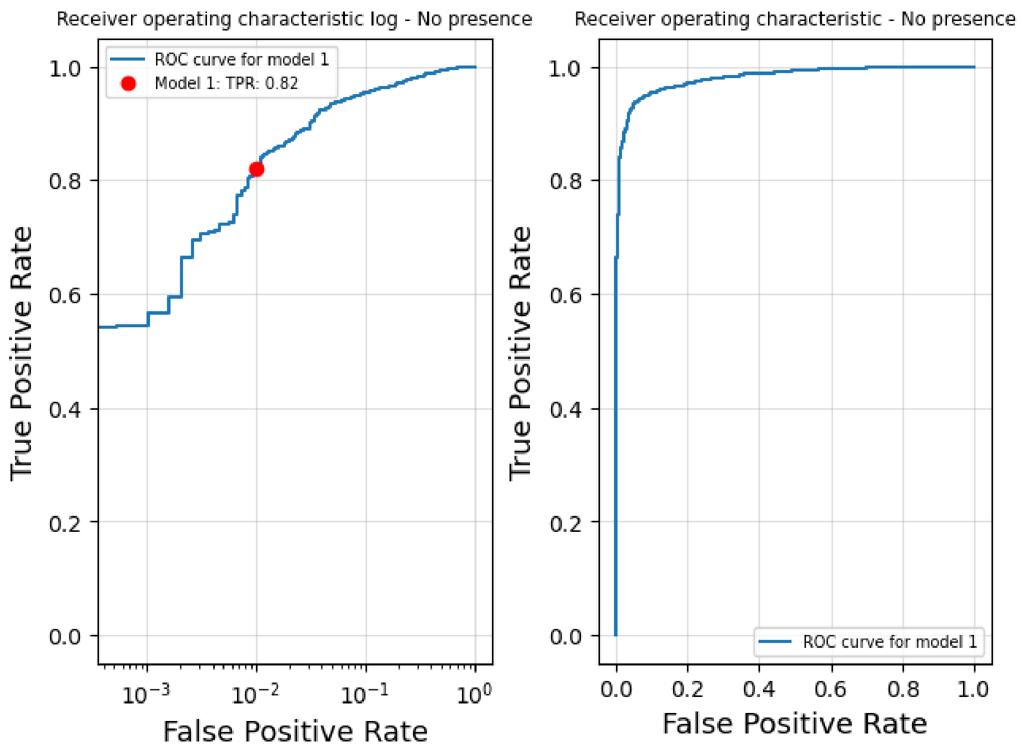


Figure 4.14: ROC curves (log and normal) for hyperparameter search using Genetic Algorithm (Test 3)

Test	Max Accuracy	Min validation loss	TPR for FPR=0.01
Test 1	93.9 %	0.176	0.80
Test 2	**94.5** %	0.174	0.82
Test 3	94.1 %	0.160	0.80
Test 4	94.4 %	**0.156**	**0.83**

Table 4.11: Results: Accuracy, Loss and TPR for the hyperparameter search using Keras Tuner

Observing the resulting parameters in table 4.10, the number of filters for the first convolutional layer was very consistent in the range from 83 – 99. There were larger variations for the second convolutional layer, and for the kernel sizes. The number of units for the dense layer had very small variations for the three last tests, and was in the range from 134 – 138. From the

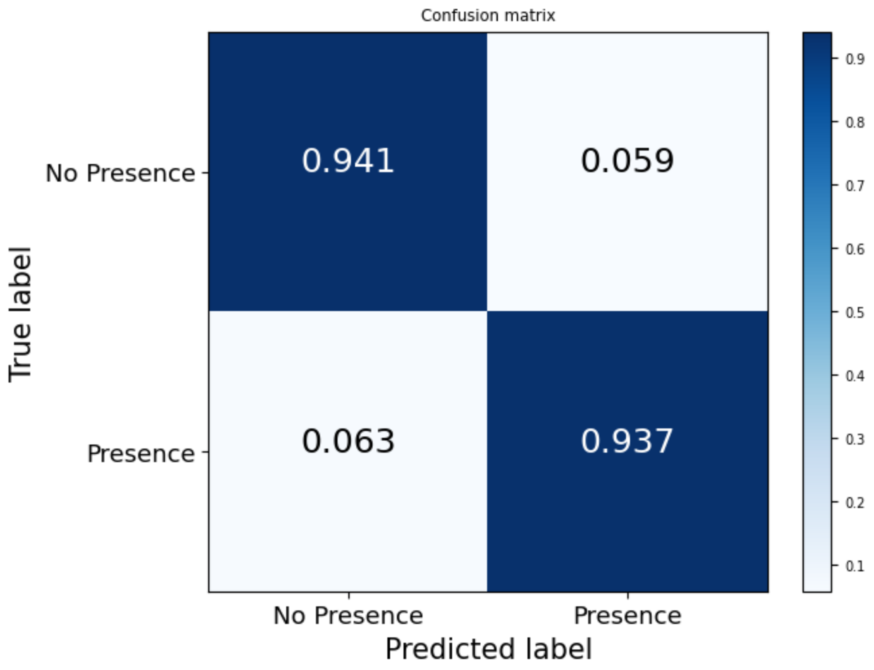


Figure 4.15: Confusion matrix for hyperparameter search using Genetic Algorithm (Test 3)

results presented in 4.11, test 4 performed the absolute best in regards of the validation loss and TPR. It achieved a validation loss of only 0.156 and a TPR of 0.83, which both were the best results obtained during the project. The corresponding graphs are presented in figure 4.17, 4.18 and 4.19. The plotted results for the remaining tests are presented in appendix B.

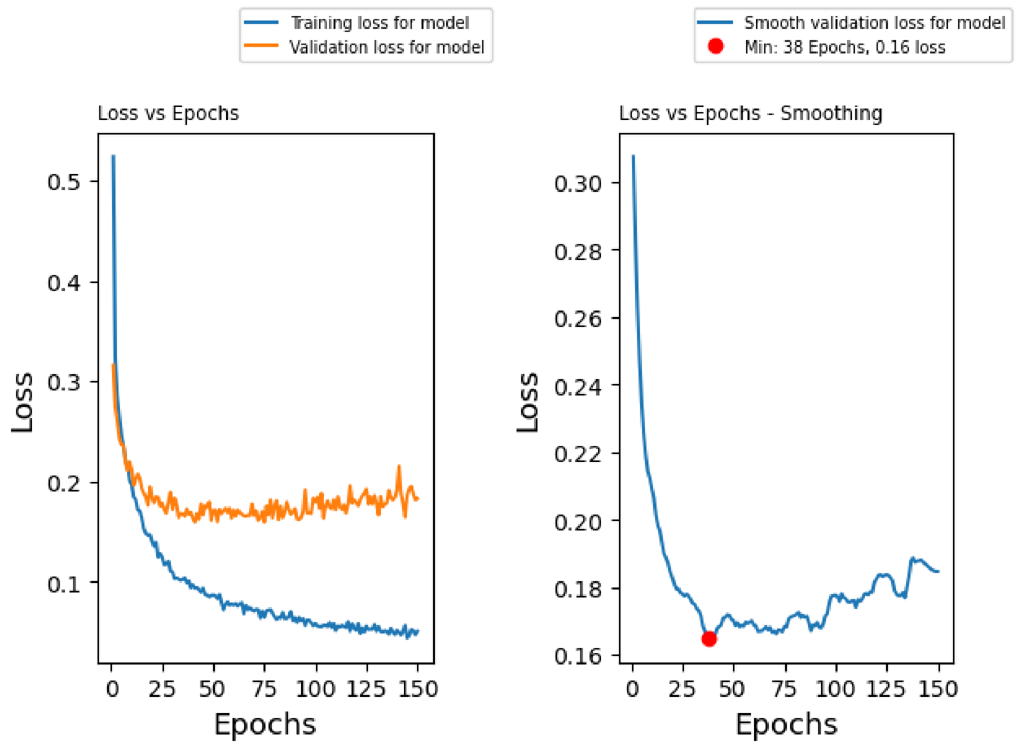


Figure 4.16: Loss vs Epochs for 150 epochs using the genetic algorithm. Smoothed graph to the right.

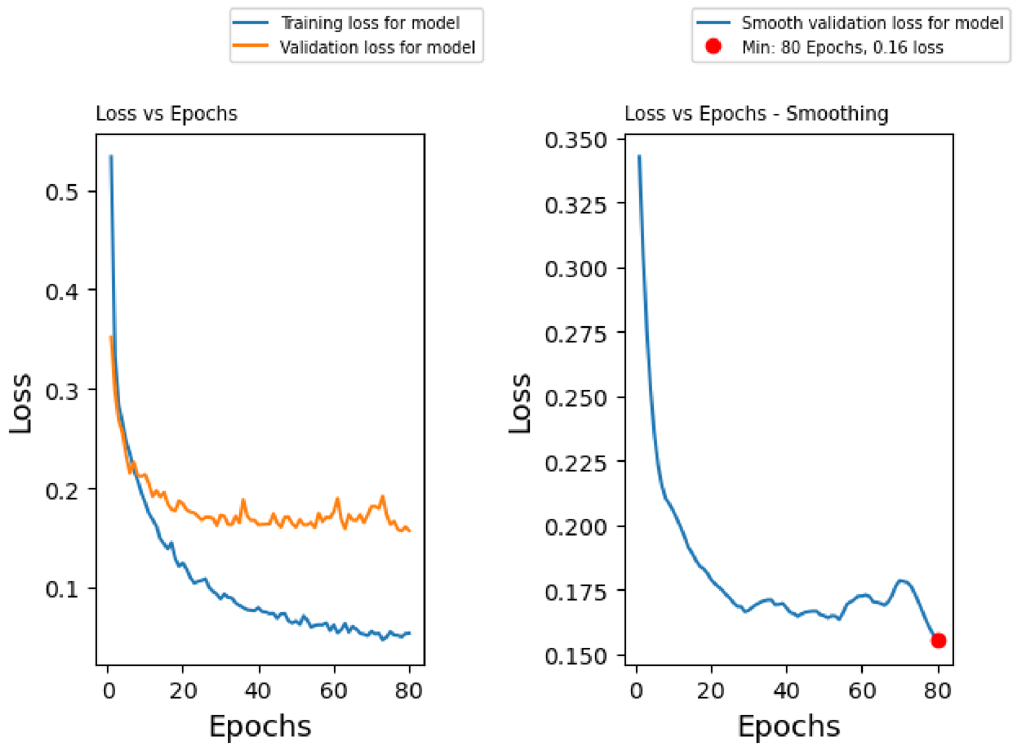


Figure 4.17: Loss vs Epochs for hyperparameter search using Keras Tuner. Smoothed graph to the right. (Test 4)

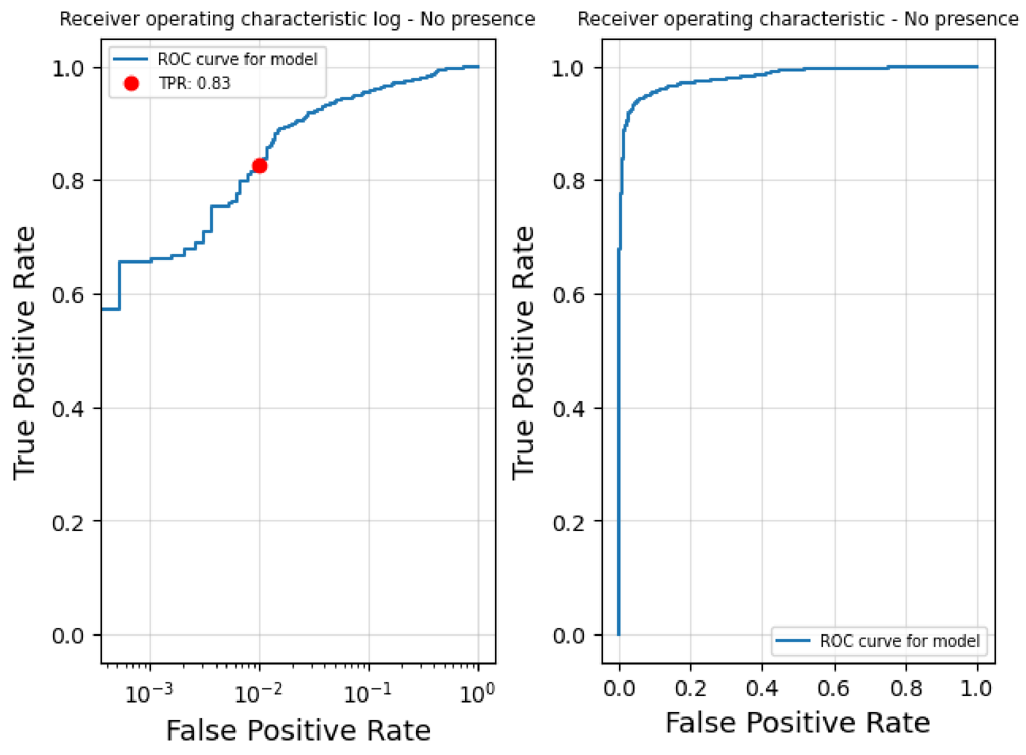


Figure 4.18: ROC curves (log and normal) for hyperparameter search using Keras Tuner (Test 4)

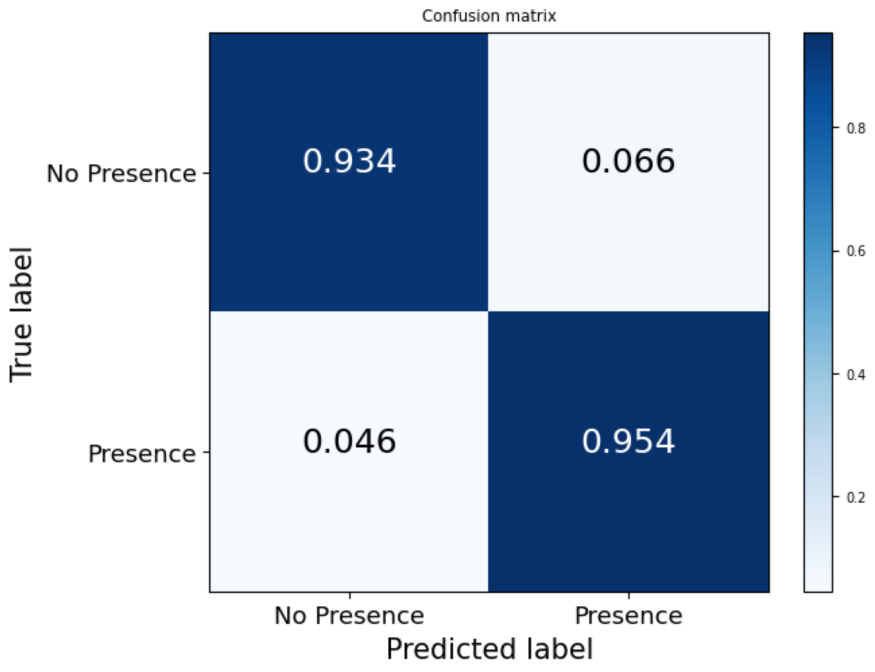


Figure 4.19: Confusion matrix for hyperparameter search using Keras Tuner (Test 4)

Chapter 5

Discussion

This chapter will discuss the different results achieved and the methods used in the project.

5.1 Results Architecture search

The results of the architecture search presented in chapter 4.2 are discussed in this section.

The metrics this thesis aimed to optimize were the validation loss and the TPR for FPR=0.01. When training and testing neural networks the results will differ for each trial. The results presented can therefore only be viewed as an indication of the performance.

5.1.1 Initial model

The initial model achieved a maximum test accuracy of 90.7%, a validation loss of 0.23, and a TPR=0.71. The results for the initial model showed that the validation loss was constantly increasing. This is an indication of overfitting, and that the model is not adapting to new data. This was not an unexpected result as the model did not use any form for regularization. Overfitting occurs when the model is too complex for the data. Reducing the complexity of the model or applying regularization can be beneficial. For this project, different regularization techniques were applied.

5.1.2 Regularization

The two regularization methods that were tested were the L2 regularization and Dropout layers. These were tested both individually and in combination. For a dropout rate of 0.5, a validation loss = 0.18 and a TPR = 0.77 were achieved. This showed significant improvements from the initial model, and indicated that the overfitting problem was improved. When adding L2 regularization to the model, both the loss and TPR were aggravated, for all regularization strengths. When testing different combinations of these techniques, the model without any L2 regularization performed overall best. The same results were achieved for several tests, and concluded that L2 regularization reduced the performance. Only dropout layers were therefore applied for later models.

L2 regularization is commonly used to reduce overfitting in neural networks. There could be several reasons why it did not work as expected for this network. From the results of different weight sizes, the model using the smallest weight, $\lambda = 0.0001$ was performing best. This model achieved a validation loss = 0.23, but it was still higher than for the model without L2. This indicated that L2 regularization might not was necessary for this network, as the regularization strength needed to achieve good performance was very low. As L2 is used to prevent overfitting by penalizing the large weights in the network, it could also make it more difficult to fit the training data. A higher number of epochs could be the solution, but this would also make the network slower.

5.1.3 Max-Pooling

Max-Pooling with a pool-size = 2 was used in combination with the dropout layer with a rate of 0.5. Out of the four models that were tested, the model with a max-pooling layer only after the second convolutional layer performed best. The dropout layer was placed after both the convolutional layers. This model reached a TPR=0.80 and a validation loss=0.17. This was so far the best performing model.
An advantage of using max-pooling layers, is the reduction of complexity in the model. This will reduce the number of trainable parameters which will give a more time efficient network to train. As this layer locates the most important features in a region, it will remove data that may not be necessary

or that is noisy. When applying this layer, the model performed better, just as expected.

5.2 Results Hyperparameter Search

5.2.1 Genetic Algorithm

The results of the hyperparameter search using the genetic algorithm showed that the number of filters for the convolutional layers should be relatively high, and that a kernel size of 5 was selected for almost every test. A high number of filters in a convolutional layer could detect more features in the data, and it makes therefore sense that a high number was chosen. The optimal size of a kernel is dependent on the data, as a larger kernel size will detect larger features in the data, and a small kernel size will detect smaller features better. The search space for these tests did only allow a kernel size of either 3 or 5, as these are most commonly used in CNNs. The largest variations were found for the number of units in the dense layer, and they varied from $36 - 105$. This can be a coincidence, as the algorithm tries a high number of different values. Different combinations can cause the best results for each test, and a hyperparameter search can therefore only be used as an indication.

The results regarding the validation loss and the TPR did also have some variations, like expected. Even when training a model several times using the same hyperparameters, the results will have some variations. This is due to the weights and biases being trained a little different each time. The final results using the GA are still good, and achieved a quite consistent validation loss throughout the tests, from $0.168 - 0.184$. This indicates that the algorithm found a good set of hyperparameters for each test.

5.2.2 Keras Tuner (Hyperband)

The results using Keras Tuner had several similarities with the results from the genetic algorithm. The optimized number of filters for the convolutional layers was relatively large compared with the search space. The number of units for the dense layer was high for all the tests using Keras Tuner, and did not vary as much as for the genetic algorithm. The kernel size is more inconsistent here, and varies between 3 and 5. The size does not increase from the first to the second convolutional layer for any of the tests. If it starts with a kernel size of 5, the next kernel size is 5 or 3. For the one test that starts with 3 as kernel size, the next kernel does also have a size of 3. This was unexpected, as it was anticipated that it would choose the smallest kernel first. It could be easiest for the model to detect the smallest features first, and then use the largest kernel to find high-level features. This would then lead to a gradually decrease of the feature space. One reason for this unexpected behavior could be that both 3 and 5 are considered small kernel sizes, and the performance using either was not that different. If a kernel size of 7 or 9 was used, it could have resulted in another outcome where the smallest kernel sizes were used in the first layer.

The results achieved using the Keras Tuner were the best accomplished during this project. It reached a minimum validation loss of 0.156 and a TPR of 0.83, which both are relatively good values compared to other results in this thesis.

5.2.3 Comparison of the two algorithms

Both algorithms achieved good results using the available data. Keras Tuner, using the Hyperband algorithm, obtained slightly better results. There can be several reasons for this. Four tests were performed using each algorithm, and this is not sufficient enough to conclude that it was not a coincidence that one performed better than the other. It is anyhow enough to observe a pattern, as the Keras Tuner performed a little better for almost each test.

The genetic algorithm was implemented and programmed by the author, while the Keras Tuner was implemented using a library in Python. This could be one reason for the difference, and if given more time to further develop the genetic algorithm, it might have provided better performance.

5.2.4 Converging Loss

As observed for many of the previous tests, the validation loss converged when it reached a value of approximately $0.16 - 0.20$. The loss was not decreasing any further than this regardless which techniques that were applied, and there can be different reasons for this. When the validation loss is higher than the training loss, it is often a sign of overfitting. The training loss will in most cases converge towards zero. The validation loss will not always be capable of achieving the same loss values as for the training data set.

The training loss will usually perform well, as this is the exactly same data that the model is trained with. The validation loss will decide how generalized the model is, and how well it adapts to new data. If the data used as input to the model is noisy or has high similarity between the classes, it can be hard to achieve a perfect validation loss.

By observing the the visualization of the data, it could be possible to detect if the different classes are distinct enough. There could also be better ways to pre-process the data, which potentially can make it easier for the model to recognize the patterns.

5.3 Methods

This section will discuss the methods used in the project.

The initial model was build using two convolutional layers and two dense layers. This was based on previous knowledge of similar models, and the requirement to keep it simple. When evaluating this choice later, it is clear that several initial models should have been tested. Models with fewer trainable parameters could have reduced the problem with overfitting. Former developed, and well tested architectures should also have been implemented for comparison.

For each test performed using the genetic algorithm, a different set of parameters was chosen based on which parameters that provided the best results. It is possible that a model could have increased its performance slightly by using more generations. The number of generations was chosen to be high enough for the model to reach its potential, but the optimal number of generations could vary between the different models. If running the same test multiple times, it could present different results. It is important to remember that the results presented in this thesis was obtained during a limited number of tests, and could possibly change if performing a larger number of tests. However, it should give a good indication of the performance of the different models.

The number of filters and the size of the kernels, in addition to the number of units in the dense layers were chosen to be the hyperparameters that were optimized. The remaining hyperparameters, like e.g. the dropout rate and the pool-size for max-pooling were decided by testing their effect, or by using previous knowledge. A network where all the parameters were chosen by the optimization algorithm should have been tested, but it was not conducted in this project.

A higher number of epochs could also have been used for some of the tests. When searching for hyperparameters, the number of epochs was increased from 50 to 80. This seemed to be enough for most of the tests, but it was some cases where it appeared like the loss had not finished converging after 80 epochs. One trial that was running 150 epochs was therefore also tested, but this did not result in any lower validation loss. The loss did slowly start to increase again after 38 epochs, and the conclusion was therefore that 80 epochs should be sufficient enough for most tests.

An uncertainty when comparing models was the rounding of the results. Both the loss and TPR were presented using two decimals for the first tests. This was a problem e.g. for two loss values of 0.18 and 0.19, as these values could be almost equal due to the rounding. In these cases, a total evaluation of all results was used. For the last tests searching for hyperparameters, three decimals were used for the loss.

When observing the results, the optimized values for the number of filters and the unit size were relatively large compared with the range of the search space. The smallest values within the search space were not chosen as optimal values for any of the tests. This could indicate that the search space should have been larger. The initial model did struggle with overfitting, and a reduction in complexity could be a solution to this. It is therefore contradictory to choose a larger search space. As mentioned earlier in ch. 3, a larger search space could also cause larger models that are slower to train.

5.4 Future Work

After the evaluation of the results and methods, some possible improvements were discovered. For future work related to this subject, some suggestions will be mentioned in this section.

5.4.1 Exploration of other Architectures

Deeper networks using a higher number of layers provide many opportunities. This thesis focused on developing and optimizing a simple CNN model. As mentioned, several initial models should have been tested. A well tested model, such as AlexNet [18] or ResNet [19], could have been used as a basis. These architectures can be scaled to fit the amount of data, to avoid creating too complex models. Further work should therefore include a wider test of initial architectures for the hyperparameter search.

It would also be interesting to perform broader tests regarding the dropout layers and max-pooling layers. Other variations of pooling-layers, such as the L2-pooling or Avg-pooling could also be tested. In this project, the use of L2 regularization reduced the performance for the networks. Further testing and research in this field can be interesting, as regularization is a good technique against overfitting in neural networks. For larger data sets or deeper networks, this should probably be implemented. The use of L1 regularization should also be examined, as this was not tested in this project.

5.4.2 Genetic Algorithm

The field of hyperparameter optimization using GA is continuously growing, and it could be interesting to explore this field more. There are several ways to implement genetic algorithms, and different metrics and parameters to optimize. The selection of other parameters to use in the mutation function, and adaptive mutation rates should also be investigated.

5.4.3 Data

The data used for training, validation and testing can be studied more carefully. Other techniques for pre-processing, and data augmentation could be investigated. Also, the data that the classifier fails to classify correctly should be observed. This could make it possible to understand more of the problem, and identify whether it is the data set or the classifier that causes the errors.

Chapter 6

Conclusion

This chapter concludes the results obtained in this project. A short conclusion for the methods used is also presented.

The objective of this thesis was to implement and test methods for hyperparameter optimization for Convolutional Neural Networks (CNNs). The data used was provided by Novelda, and was recorded using their Ultra Wide Band (UWB) radar. The network should classify the radar signals and detect if there is human presence or not. In addition to this, it was desired that the implemented network should not be too complex. To avoid a network with many trainable parameters, a simple model was designed. When optimizing the model, the aim was to reduce the validation cross-entropy loss and maximise the True Positive Rate (TPR) for a given False Positive Rate (FPR). These metrics were used to decide which architectures and hyperparameters performed best.

6.1 Results and Methods

A search for the best architecture was performed using a simple initial model, and investigate the effects of applying regularization and pooling. The final CNN model was implemented using two dropout layers and one Max-pooling layer, in addition to the initial model consisting of two convolutional layers and two dense layers. The effect of L2 regularization was also tested, but it was not implemented as the model achieved reduced performance.

Using the optimized architecture, a hyperparameter search was conducted. To perform the hyperparameter optimization, a genetic algorithm, as well as the Python library *Keras Tuner*, was implemented. *Keras Tuner* uses the hyperband algorithm to search for parameters, and this method was compared to the genetic algorithm. The algorithms searched for optimized values for the convolutional and dense layers, consisting of the number of filters, the kernel sizes, and the number of units.

Using the optimized network in combination with the genetic algorithm, the network achieved a validation loss of 0.168. For a FPR = 0.01, a TPR = 0.82 was reached. The best results were obtained using the hyperband algorithm to search for hyperparameters. It achieved a validation loss of 0.156, and a TPR = 0.83 for FPR = 0.01 was reached.

Regarding the values of the optimized hyperparameters, the two algorithms reached several similarities with their results. A relatively high number of filters for the convolutional layers, and units for the dense layer were selected by the optimizers. In addition to this, a kernel size of 5 for the first convolutional layer, followed by a kernel size of 5 or 3 for the second layer resulted overall in the best results.

Bibliography

[1] H. Blehr, "Ann based respiration detection using uwb radar," M.S. thesis, NTNU Department of Engineering Cybernetics, 2017.

[2] B. wiik, "Ann based classification of humans and animals using uwb radar," M.S. thesis, NTNU Department of Engineering Cybernetics, 2018.

[3] E.-G. Talbi, "Optimization of deep neural networks: a survey and unified taxonomy," working paper or preprint, Jun. 2020, [Online]. Available: https://hal.inria.fr/hal-02570804.

[4] A. Shrestha and A. Mahmood, "Optimizing deep neural network architecture with enhanced genetic algorithm," *18th IEEE International Conference on Machine Learning and Applications*, pp. 1365–1370, 2019. DOI: doi:10.1109/ICMLA.2019.00222.

[5] I. T. Hovden, "Optimizing artificial neural network hyperparameters and architecture," 2019, University of Oslo.

[6] B. A. et al., "Ultra wideband: Applications, technology and future perspectives," 2005.

[7] Accessed 30.04.21. [Online]. Available: https://novelda.com/technology/.

[8] M. A. Nielsen, *Neural Networks and Deep Learning*. Determination press, 2015.

[9] D. P. Kingma and J. L. Ba, "Adam: A method for stochastic optimization," 2015.

[10] S. Sharma, S. Sharma, and A. Athaiya, "Activation functions in neural networks," *International Journal of Engineering Applied Sciences and Technology*, vol. 4, no. 12, pp. 310–316, 2020.

[11] T. Fawcett, "An introduction to roc analysis," *Pattern Recognition Letters*, vol. 27, pp. 861–874, 2006. DOI: doi:10.1016/j.patrec.2005.10.010.

[12] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, *Hyperband: A novel bandit-based approach to hyperparameter optimization*, 2018. arXiv: 1603.06560 [cs.LG].

[13] [Online]. Available: https://www.python.org/.

[14] [Online]. Available: https://www.tensorflow.org/.

[15] [Online]. Available: https://keras.io/.

[16] [Online]. Available: https://scikit-learn.org/stable/.

[17] Accessed 23.05.21. [Online]. Available: https://www.tensorflow.org/tutorials/keras/keras_tuner.

[18] S. Li, L. Wang, J. Li, and Y. Yao, "Image classification algorithm based on improved AlexNet," *Journal of Physics: Conference Series*, vol. 1813, no. 1, p. 012 051, Feb. 2021. DOI: 10.1088/1742-6596/1813/1/012051. [Online]. Available: https://doi.org/10.1088/1742-6596/1813/1/012051.

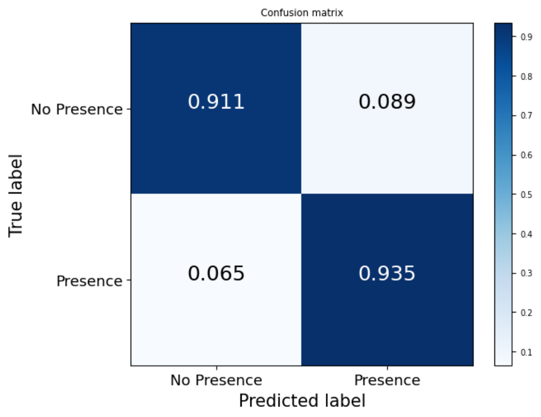
[19] I. Bello, W. Fedus, X. Du, E. D. Cubuk, A. Srinivas, T.-Y. Lin, J. Shlens, and B. Zoph, *Revisiting resnets: Improved training and scaling strategies*, 2021. arXiv: 2103.07579 [cs.CV].

Appendix A

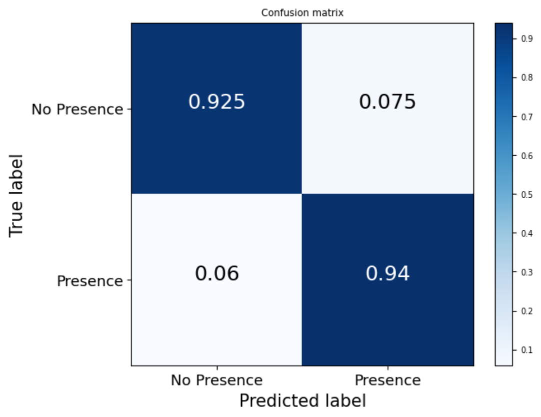
Confusion Matrices Architecture Search

The confusion matrices for all tests were not included in the results. Here, all confusion matrices for the different architecture searches from ch. 4.2 will be presented. These results were not of the same importance as the validation loss and ROC when evaluating models. They are therefore only used as additional information for the different models.

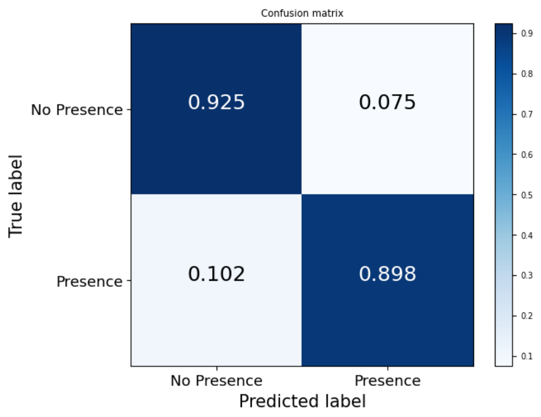
A.1 Effect of Dropout



(a) Model 1. Dropout rate = 0.25



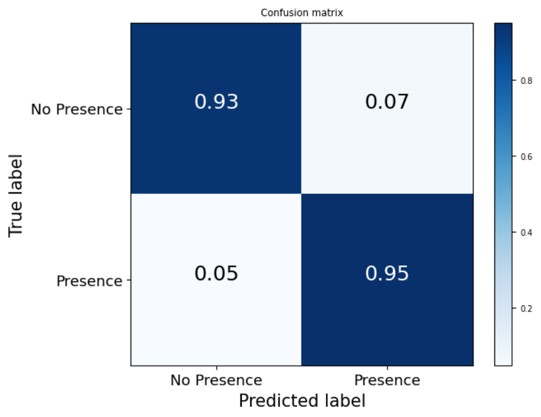
(b) Model 2. Dropout rate = 0.5



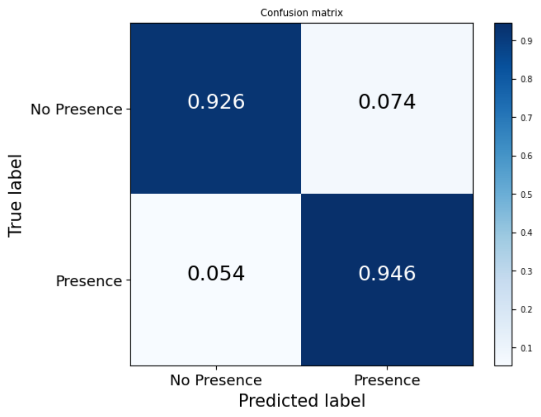
(c) Model 3. Dropout rate = 0.75

Figure A.1: Confusion matrices for variations of Dropout rates

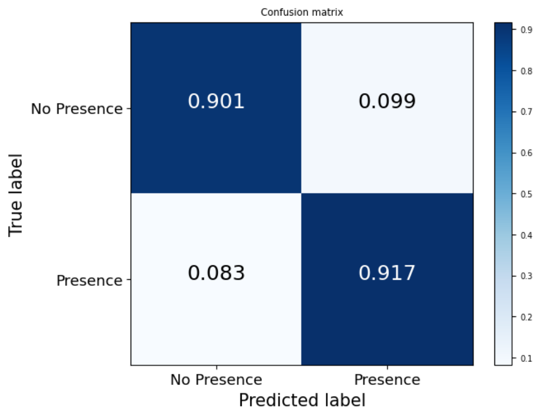
A.2 Effect of L2 Regularization



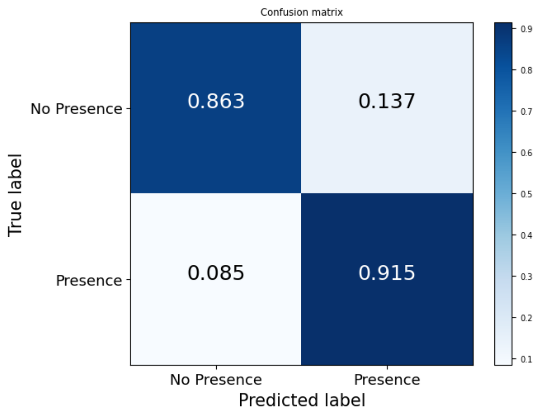
(a) Model 1. $\lambda = 0.0001$



(b) Model 2. $\lambda = 0.001$



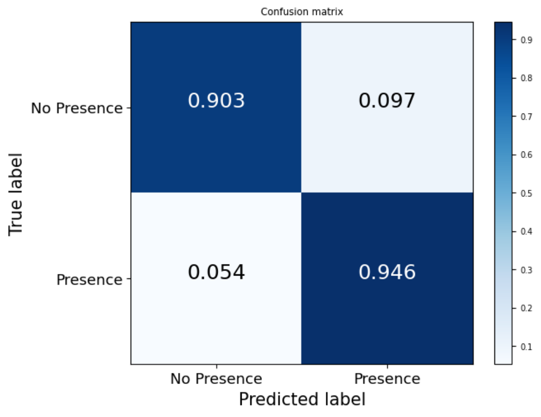
(c) Model 3. $\lambda = 0.01$



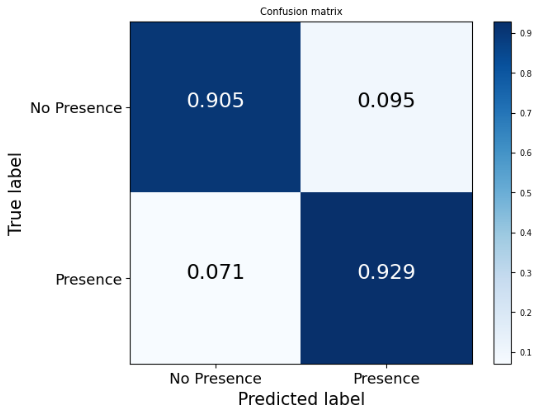
(d) Model 4. $\lambda = 0.1$

Figure A.2: Confusion matrices for variations of L2 regularization strengths

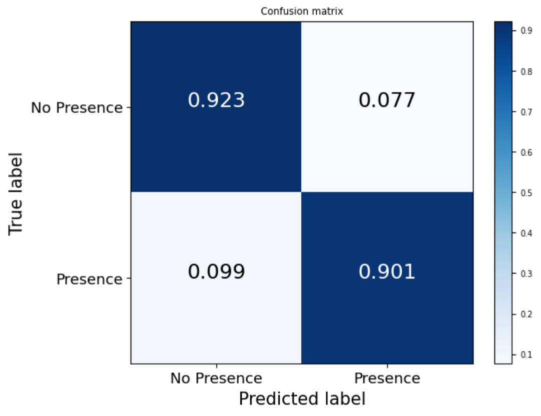
A.3 Combination of Regularization Techniques



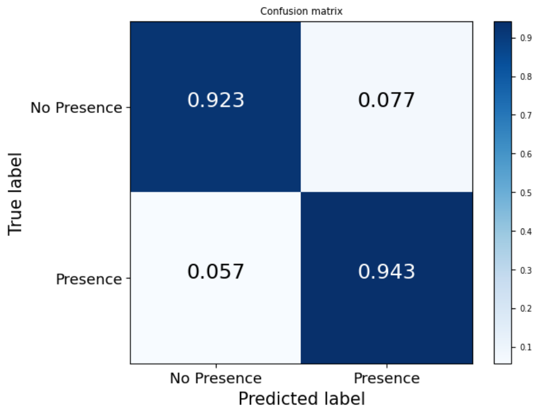
(a) Model 1



(b) Model 2



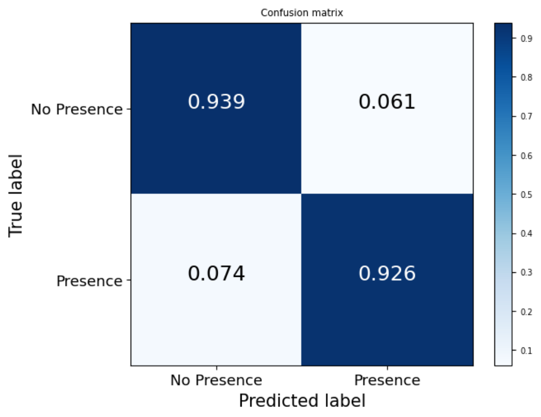
(c) Model 3



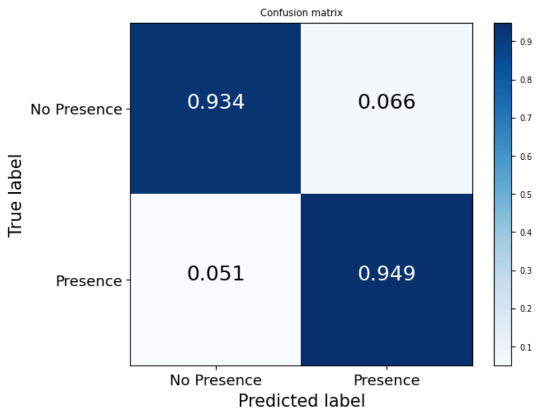
(d) Model 4

Figure A.3: Confusion matrices for combinations of regularization techniques

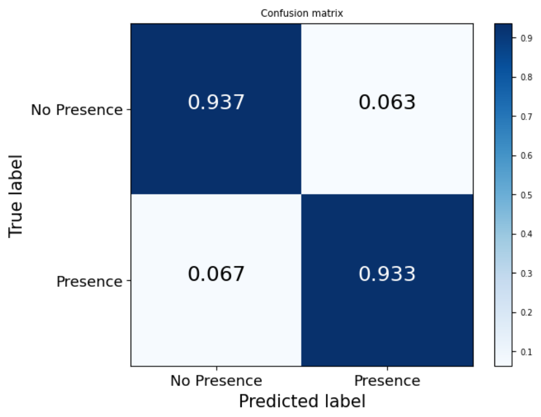
A.4 Max-Pooling Layers



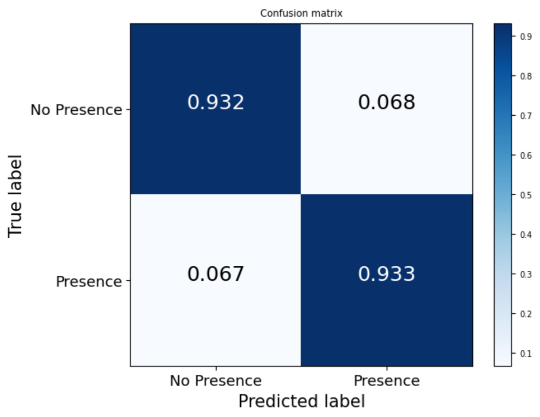
(a) Model 1



(b) Model 2



(c) Model 3



(d) Model 4

Figure A.4: Confusion matrices for combinations of max-pooling and dropout

Appendix B

Hyperparameter Search

In this chapter the different plots from the tests performed in the hyperparameter searches will be presented. This includes the loss, the ROC curve and the confusion matrices.

B.1 Using Genetic Algorithm

B.1.1 Test 1

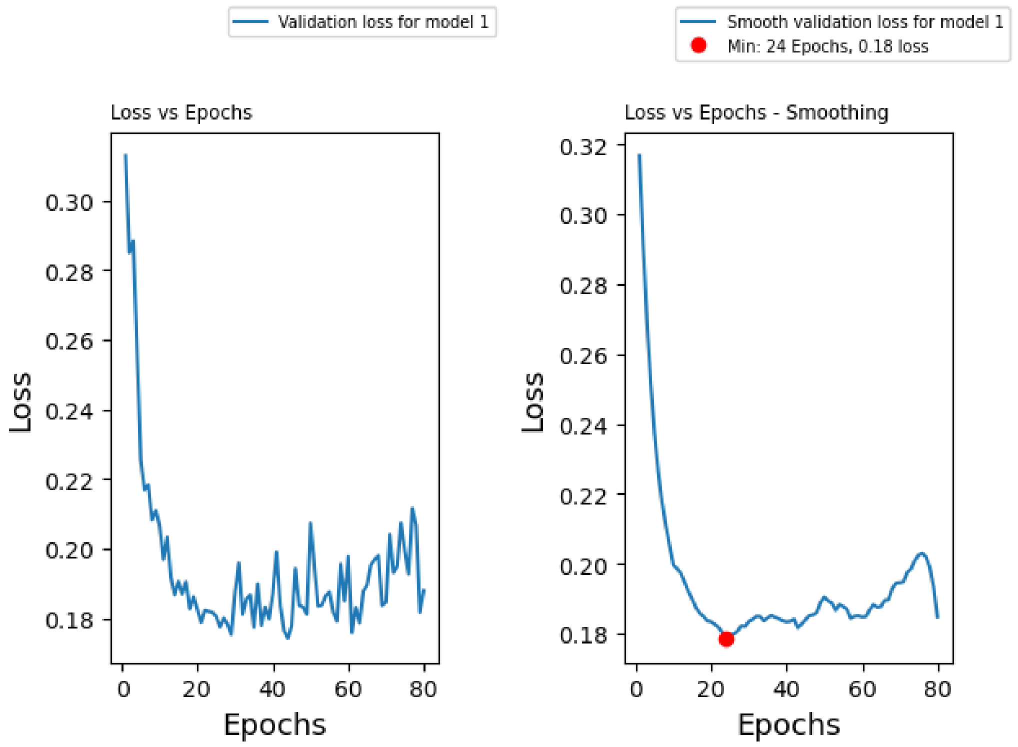


Figure B.1: Loss vs Epochs for hyperparameter search using Genetic Algorithm. Smoothed graph to the right. (Test 1)

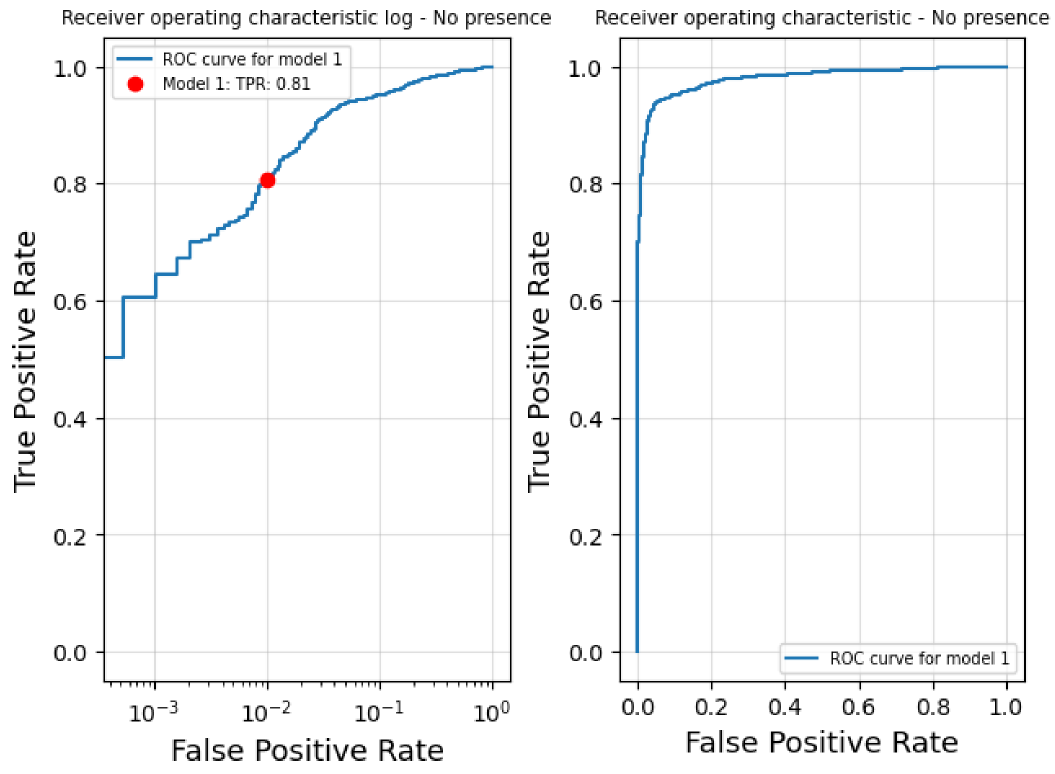


Figure B.2: ROC curves (log and normal) for hyperparameter search using Genetic Algorithm (Test 1)

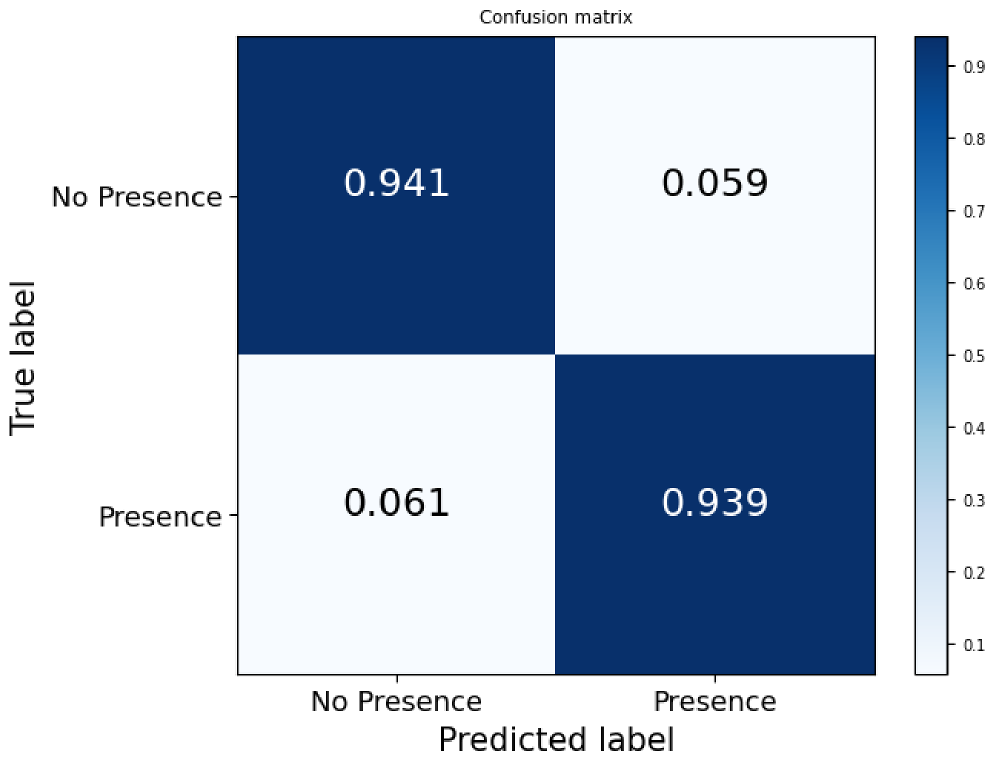


Figure B.3: Confusion Matrix for hyperparameter search using Genetic Algorithm (Test 1)

B.1.2 Test 2

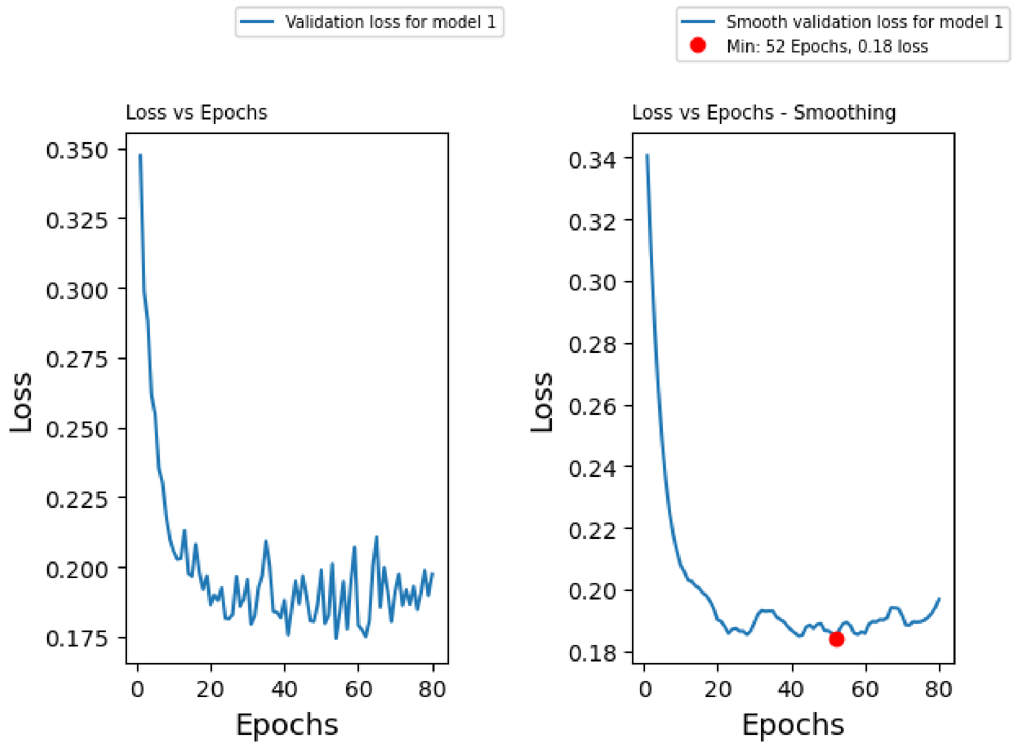


Figure B.4: Loss vs Epochs for hyperparameter search using Genetic Algorithm. Smoothed graph to the right. (Test 2)

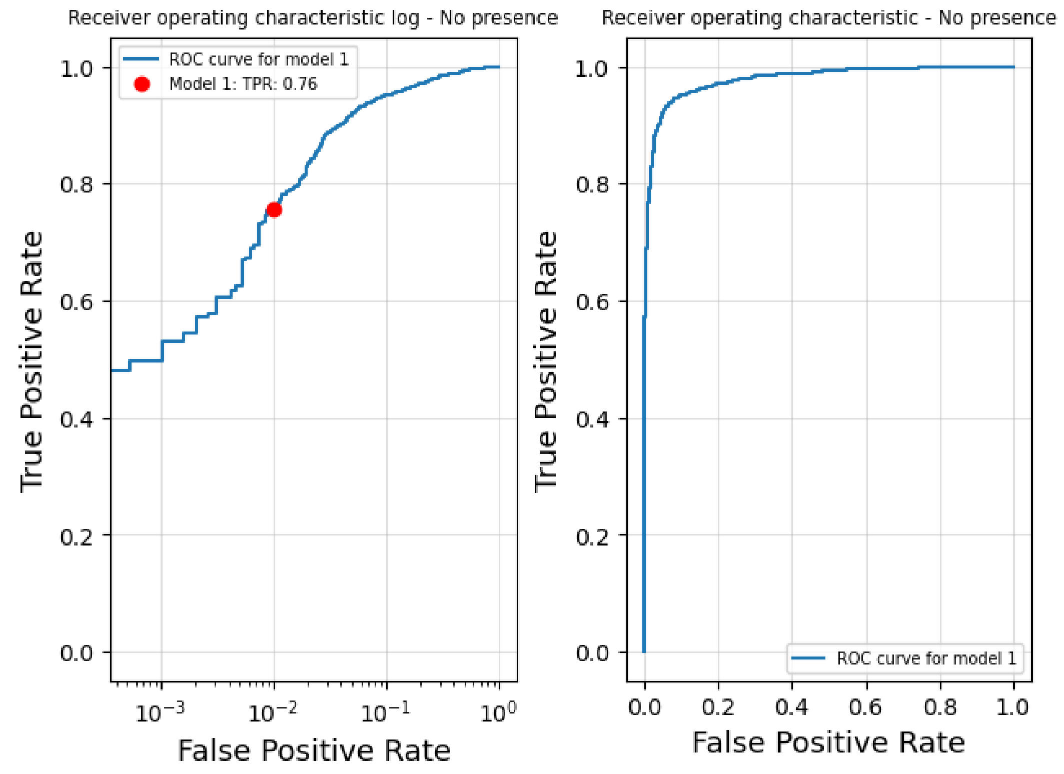


Figure B.5: ROC curves (log and normal) for hyperparameter search using Genetic Algorithm (Test 2)

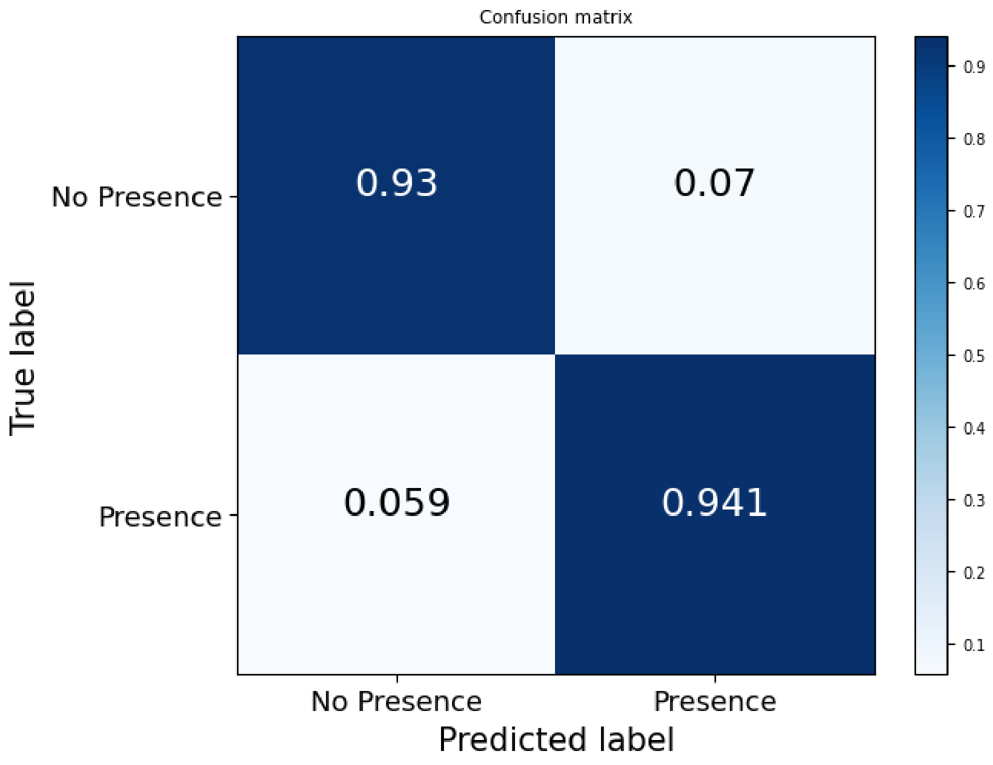


Figure B.6: Confusion Matrix for hyperparameter search using Genetic Algorithm (Test 2)

B.1.3 Test 4

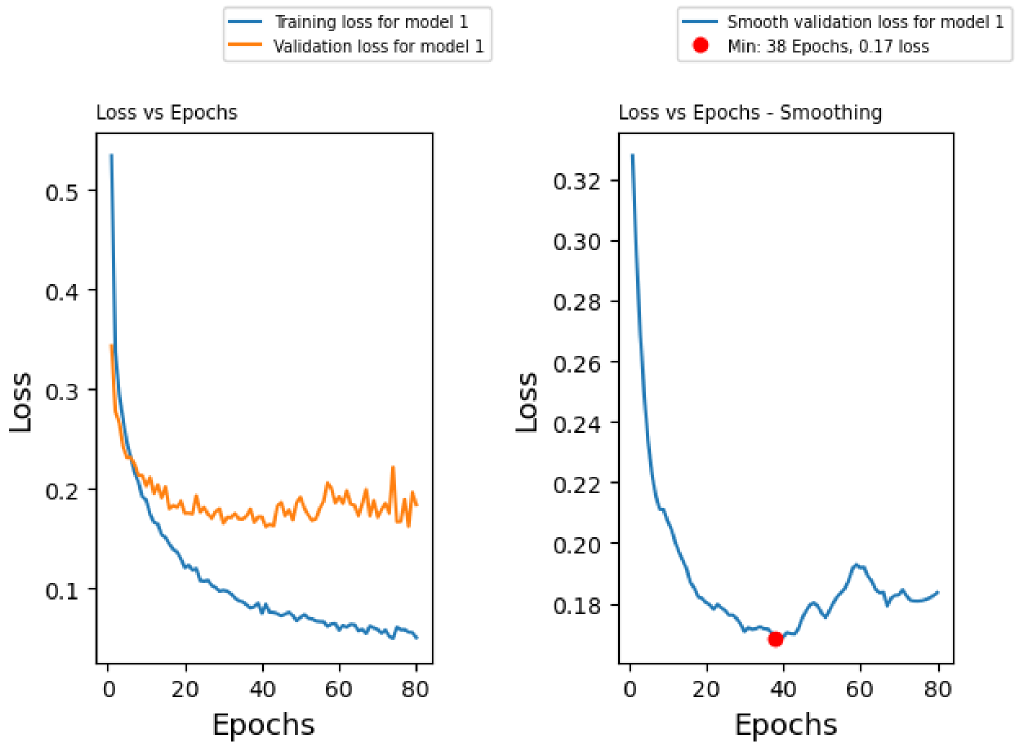


Figure B.7: Loss vs Epochs for hyperparameter search using Genetic Algorithm. Smoothed graph to the right. (Test 4)

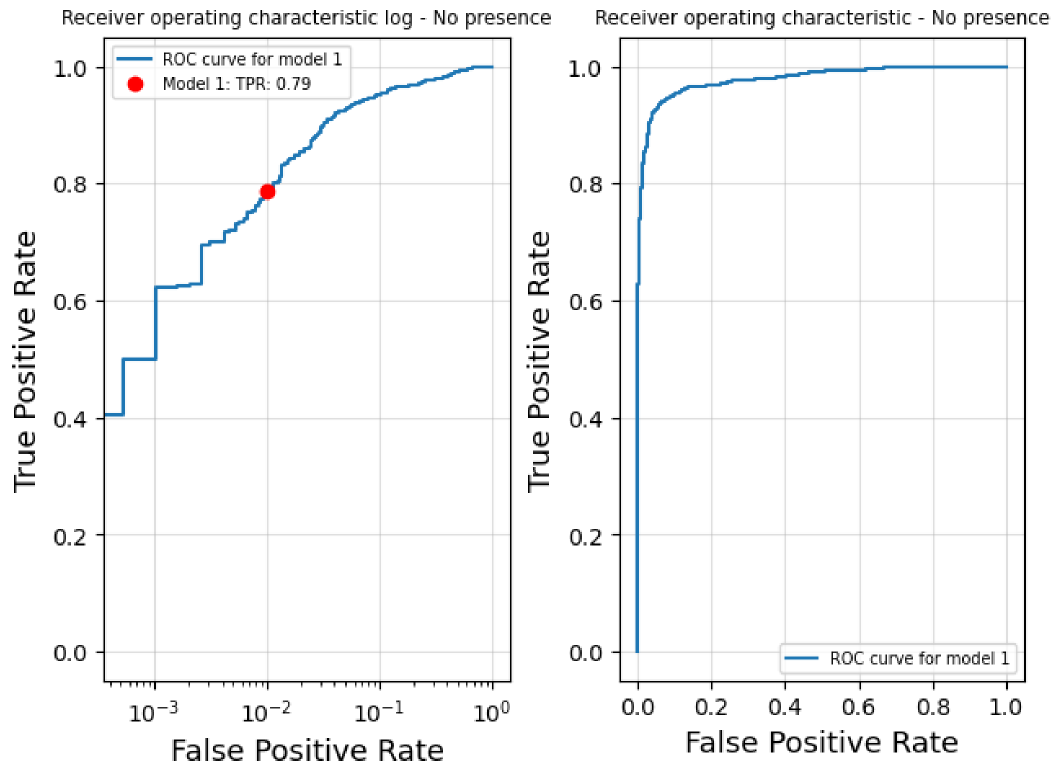


Figure B.8: ROC curves (log and normal) for hyperparameter search using Genetic Algorithm (Test 4)

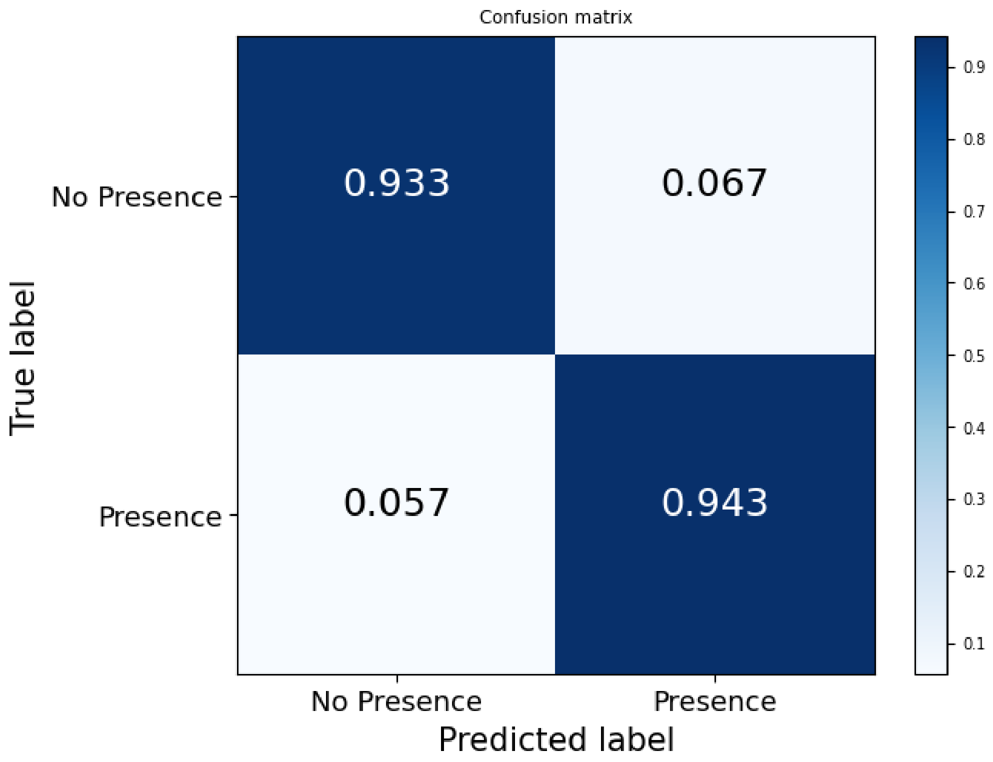


Figure B.9: Confusion Matrix for hyperparameter search using Genetic Algorithm (Test 4)

B.2 Using Keras Tuner

B.2.1 Test 1

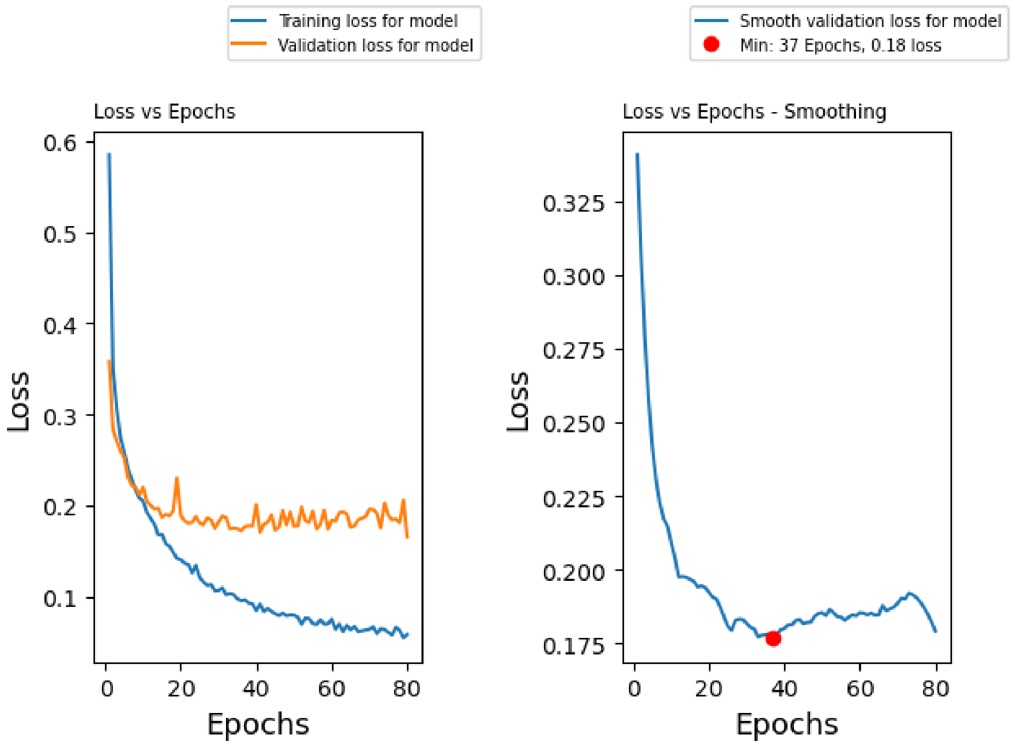


Figure B.10: Loss vs Epochs for hyperparameter search using Keras Tuner. Smoothed graph to the right. (Test 1)

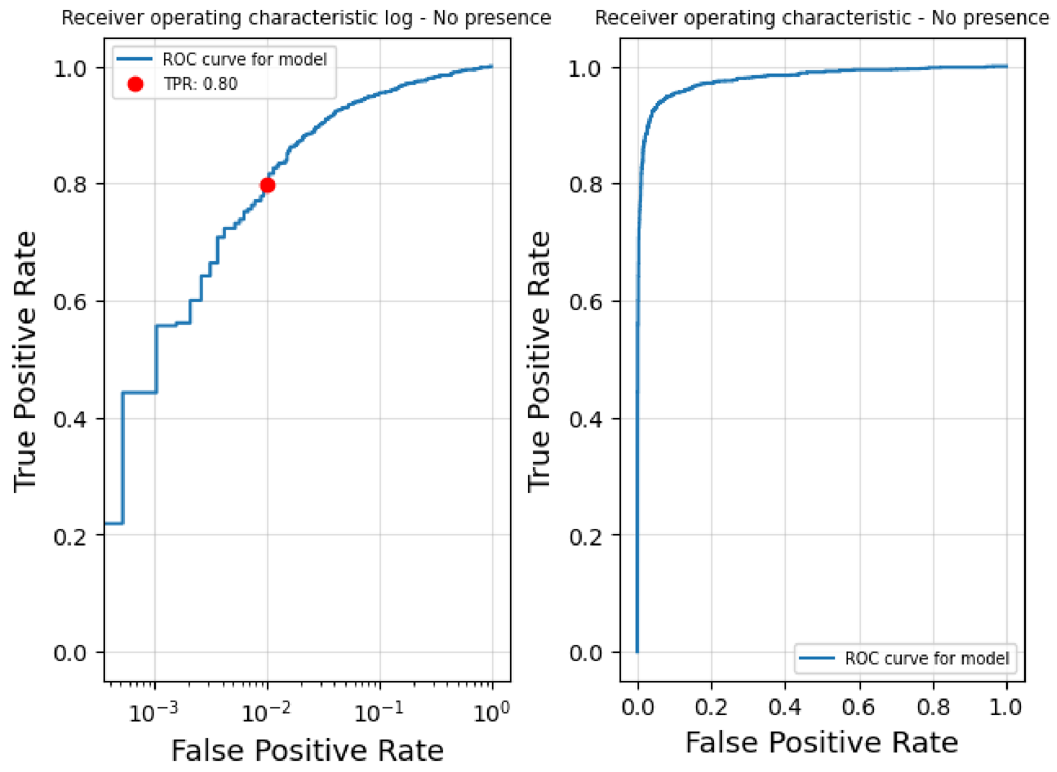


Figure B.11: ROC curves (log and normal) for hyperparameter search using Keras Tuner (Test 1)

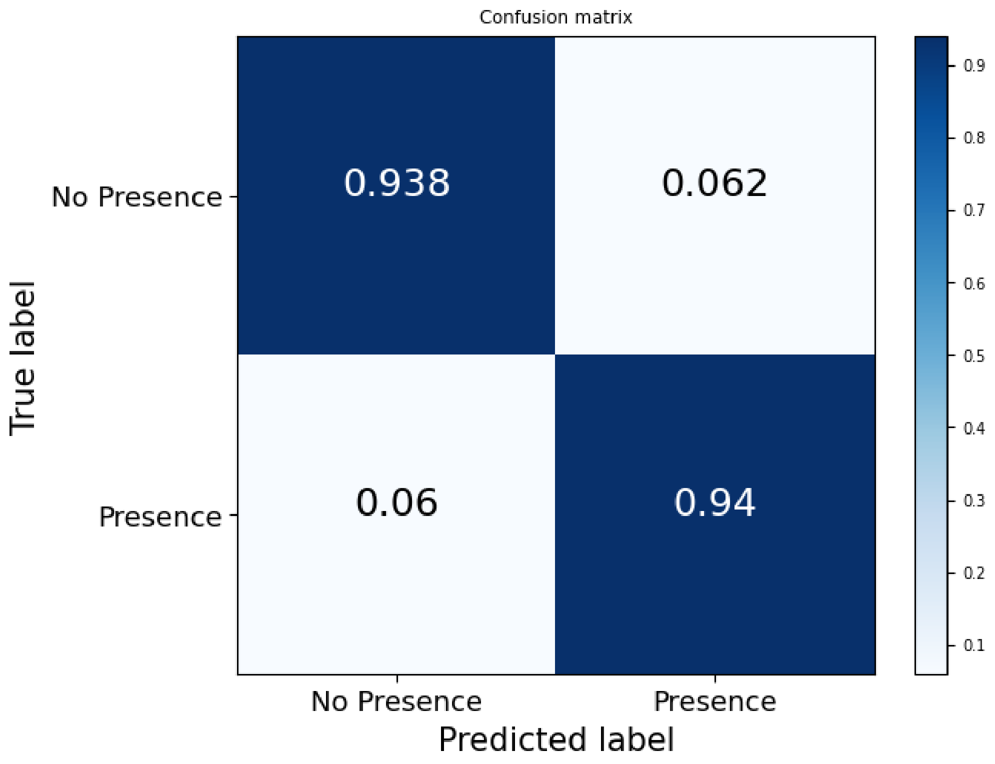


Figure B.12: Confusion Matrix for hyperparameter search using Keras Tuner (Test 1)

B.2.2 Test 2

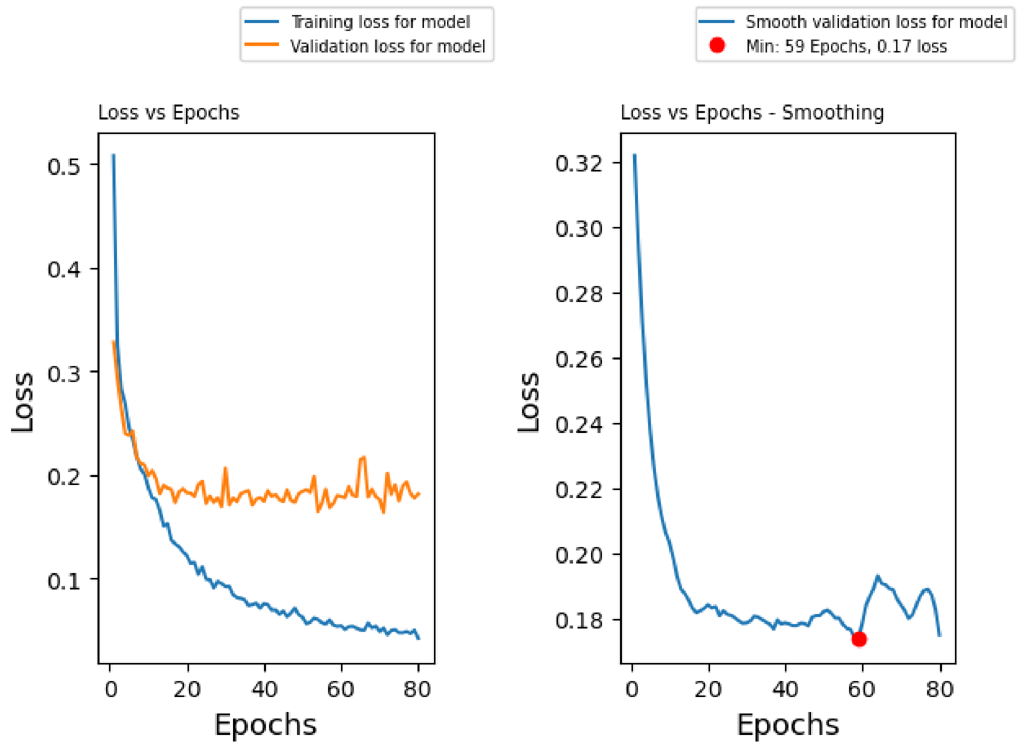


Figure B.13: Loss vs Epochs for hyperparameter search using Keras Tuner. Smoothed graph to the right. (Test 2)

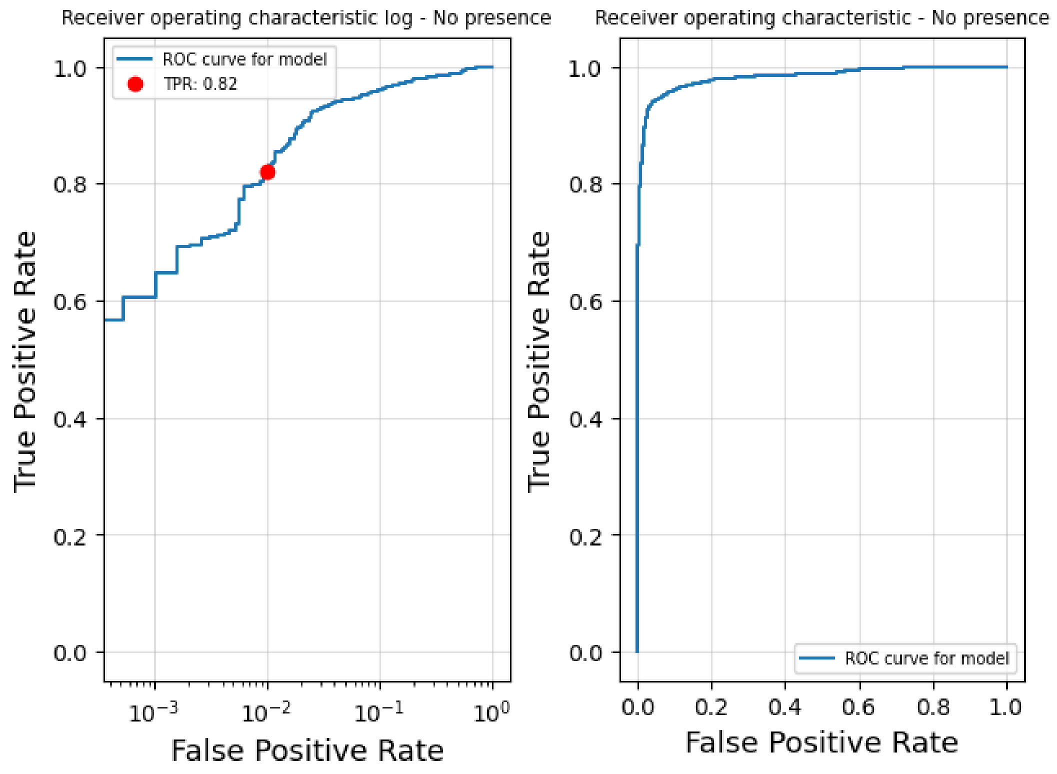


Figure B.14: ROC curves (log and normal) for hyperparameter search using Keras Tuner (Test 2)

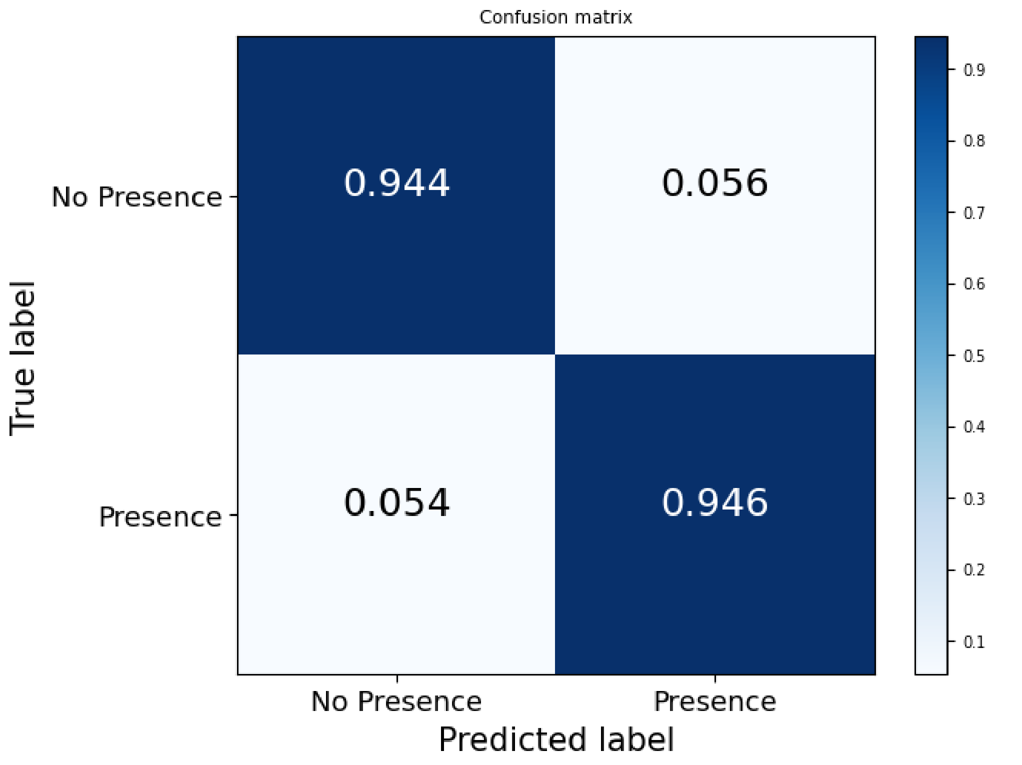


Figure B.15: Confusion Matrix for hyperparameter search using Keras Tuner (Test 2)

B.2.3 Test 3

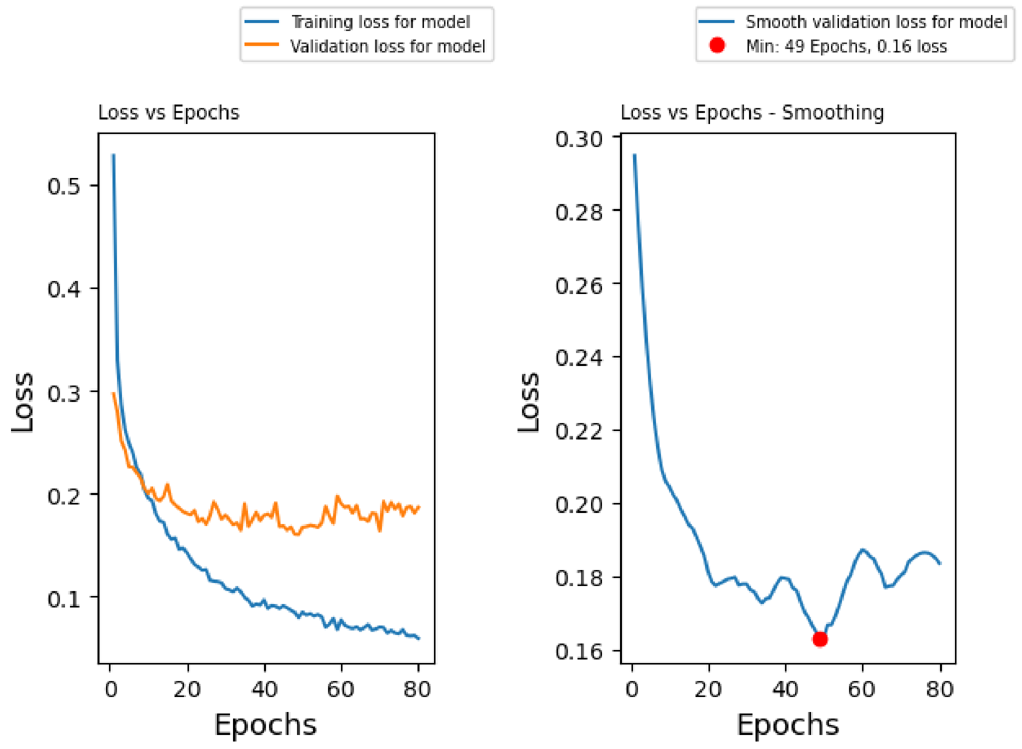


Figure B.16: Loss vs Epochs for hyperparameter search using Keras Tuner. Smoothed graph to the right. (Test 3)

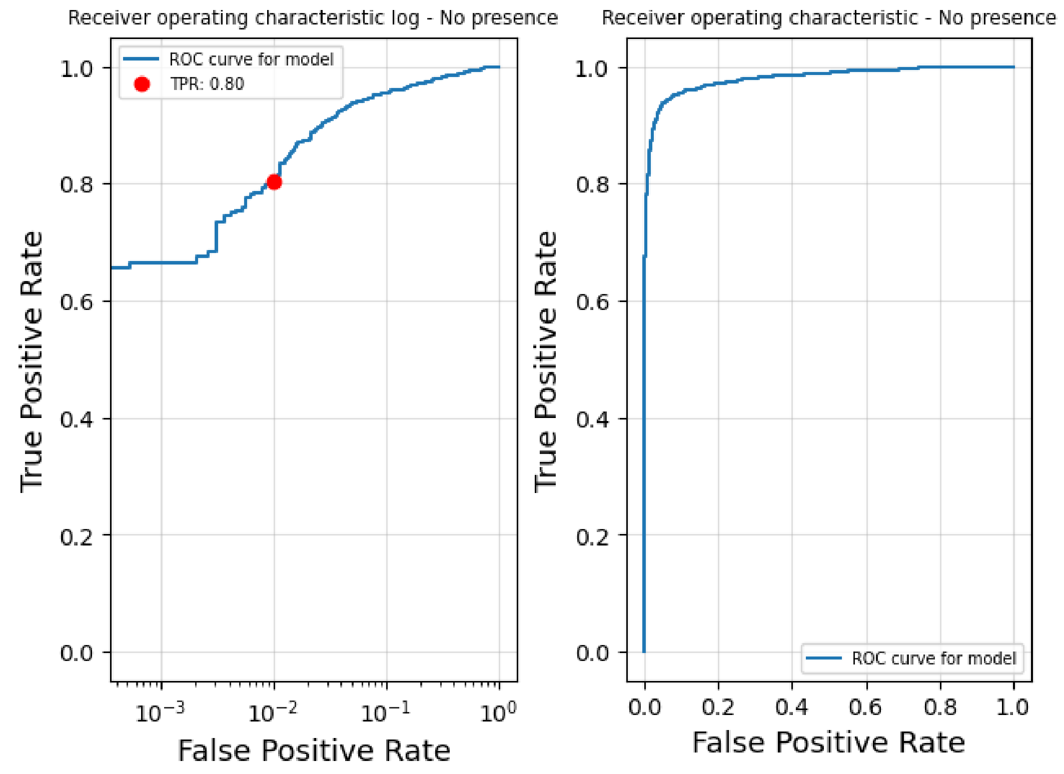


Figure B.17: ROC curves (log and normal) for hyperparameter search using Keras Tuner (Test 3)

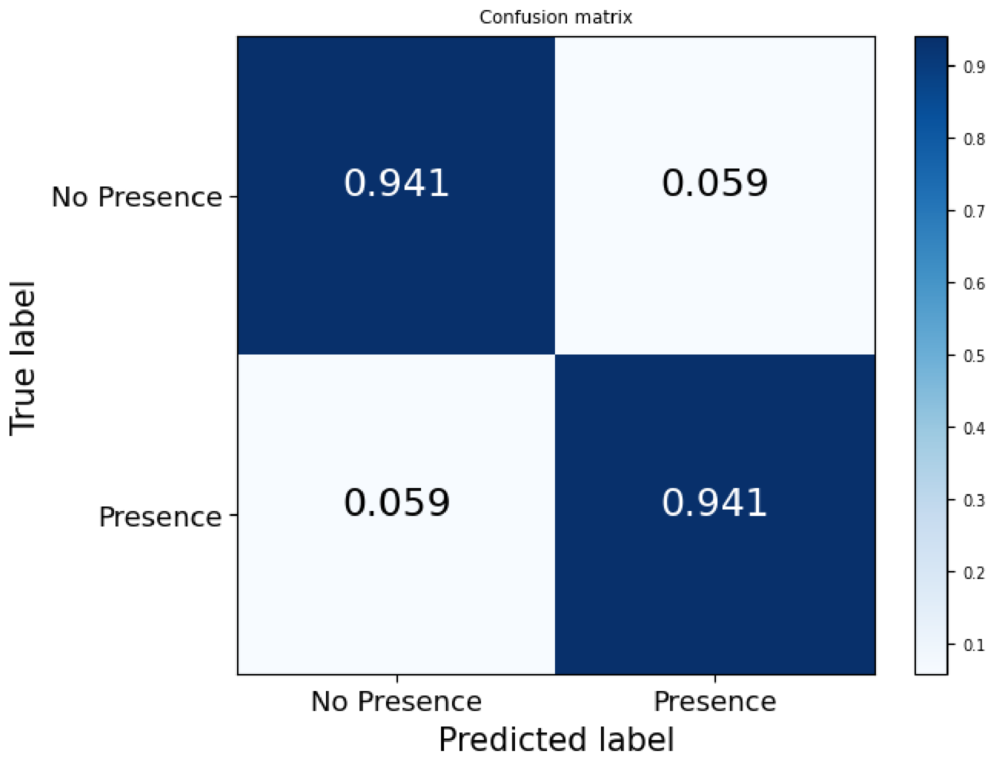


Figure B.18: Confusion Matrix for hyperparameter search using Keras Tuner (Test 3)