Modelling PRACH signals in base station with neural network

Master's Thesis
Linnea Kivioja
2462646
Unit of Mathematical Sciences
University of Oulu
Spring 2020

Abstract

Neural networks are a hot topic in the field of machine learning right now. They are machine learning algorithms that are often used for image classification and big data problems. Neural networks consist of layers which are made of neurons, places where the computation happens. The activity in these neurons is loosely based on the function of human brain. Like most machine learning techniques, the goal of neural networks is to solve an optimisation problem. It is done by trying to minimise the loss function that measures the difference between the target output and the predicted output. Neural computing is based on a so-called back-propagation algorithm. It enables updating the neural network model each training loop, to make the predictions gradually better.

This master's thesis presents neural networks as a method for modelling the PRACH signal in the base station receiver. PRACH is a physical channel where the signal comes first when a user tries to connect to the mobile network. The situation is a supervised learning regression problem. A neural network is trained to learn the correlation between the inputs, i.e. chosen parameters that describe the situation, and the output, i.e. the corresponding signal in I/Q format. In the end, the neural network is used to predict the PRACH data signal when the corresponding input features are provided.

Keywords: machine learning, supervised learning, regression, prediction, neural network, back-propagation algorithm, PRACH, I/Q data

Tiivistelmä

Neuroverkkoalgoritmit ovat juuri nyt pinnalla koneoppimisen piireissä. Neuroverkot ovat koneoppimisen algoritmeja, joita usein käytetään kuvien luokitteluun ja suuren datan ongelmiin. Neuroverkot koostuvat kerroksista, jotka puolestaan koostuvat neuroneista, joissa laskenta tapahtuu. Neuronien toiminta perustuu löyhästi ihmisaivojen toimintaan. Kuten useimmissa koneoppimisen menetelmissä, myös neuroverkkojen tarkoituksena on ratkaista optimointiongelma. Se tehdään yrittämällä minimoida tappiofunktiota, joka laskee oikean vasteen ja ennustetun vasteen välistä eroa. Neuraalilaskenta perustuu niin kutsuttuun vastavirta-algoritmiin, joka mahdollistaa neuroverkkomallin päivittämisen joka opetuskierroksella, jotta ennusteita saisi parannettua vähitellen.

Tämä pro gradu -tutkielma esittelee neuroverkon metodina, jolla voidaan mallintaa PRACH-signaalia tukiaseman vastaanottimessa. PRACH on yksi tukiaseman fyysisistä kanavista, jonne signaali saapuu ensimmäiseksi, kun käyttäjä yrittää ottaa yhteyden mobiiliverkkoon. Tutkimusongelma on regressio-ongelma ohjatun oppimisen tilanteessa. Neuroverkolle opetetaan korrelaatio syötteen (input) ja vasteen (output) välillä, eli valittujen syöteparametrien ja vastaavan signaalin välinen korrelaatio. Lopuksi neuroverkkoa voidaan käyttää tukiaseman vastaanottaman PRACH-datasignaalin ennustamiseen, kun tiedetään vastaavat tilannetta kuvaavat syöteparametrit.

Avainsanat: koneoppiminen, valvottu oppiminen, regressio, ennuste, neuroverkko, vastavirta-algoritmi, PRACH, I/Q data

Foreword

This thesis work has been done for Nokia Solutions and Networks, in a 5G modelling product development team. I want to thank Nokia Solutions and Networks for giving me this great opportunity to gain valuable knowledge and experience.

From Nokia Solutions and Networks, I would like to first thank my supervisor, Pekka Tuuttila, for guiding me through my time at Nokia. I also want to thank Olga Kayo for helping me to get forward with my thesis project.

From the University of Oulu, I would like to thank my principal supervisor, Erkki Laitinen, for introducing me to this opportunity and guiding me through the process of writing my thesis. I also want to thank Leena Ruha for being the second examinor of my thesis. Last but not least, I would like to thank the University of Oulu and the Unit of Mathematical Sciences for offering excellent and flexible education during these five years.

This master thesis work has received funding from the Electronic Component Systems for European Leadership Joint Undertaking (ECSEL JU) under grant agreement No 737494. This Joint Undertaking receives support from the European Union's Horizon 2020 research and innovation programme in Finland, Sweden, France, Spain, Italy and Czech Republic.

Abbreviations

ANN	Artificial Neural Network
AR	Augmented Reality
BS	Base Station
CNN	Convolutional Neural Network
DL	Downlink
gNB	gNodeB, notation for the BS in 5G
GPU	Graphics Processing Unit
IoT	Internet of Things
I/Q	In-phase/Quadrature
LTE	Long Term Evolution
MLP	Multilayer Perceptron
MSE	Mean Squared Error
NaN	Not a Number
NN	Neural Network
NR	New Radio
PRACH	Physical Random Access Channel
PUCCH	Physical Uplink Control Channel
PUSCH	Physical Uplink Shared Channel
ReLU	Rectified Linear Unit
RF	Radio Frequency
RGB	Red, Green, Blue
SGD	Stochastic Gradient Descent
UE	User Equipment
UL	Uplink
VR	Virtual Reality
5G	5th Generation (of mobile network technologies)

Contents

1 Introduction

1.1 Technical background

The Finnish telecommunications company Nokia was known worldwide for the famous Nokia mobile phones. Today, Nokia is a company that focuses on wireless technology and provides mobile network hardware and software. [9] Nokia is among the leaders in 5G development.

1.1.1 Mobile networks

Mobile networks (wireless networks, cellular networks) are complex webs that consist of cellphone tower zones connected to each other. They use various radio frequencies for data transfer. Mobile networks are divided into areas of land called *cells*. Cells have at least one *base station* (BS) cell tower within their area. Cells are connected to each other and hand off packets of signals like data, voice or text messages. They send and receive signals from each other and from the mobile devices (such as phones and laptops with wireless internet connection) in their area. [10] These wireless data networks are used today more than ever. They allow us to make a call to the other side of the world in an instant and to be connected to the internet almost anywhere.

1.1.2 Base station

A base station (BS) is a fixed radio signal transceiver (transmitter/receiver) in a mobile network. The BS connects mobile devices to the network. The BS has a *transmitter* part which sends signals to the mobile device i.e. the UE (User Equipment), and a *receiver* part that receives signals from the UE. The data traffic from the UE to the BS receiver is called *uplink* (UL), and the data traffic from the BS transmitter to the UE is called *downlink* (DL). The notation for the BS in 5G is the gNB (gNodeB).

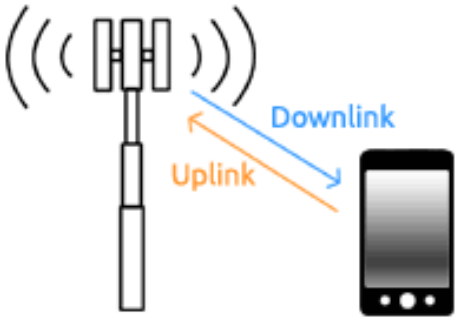

Figure 1: The connections between the BS and the UE in mobile network. [11]

1.1.3 5G Network

5G is called the 5th generation of mobile network technologies. It is a significant development from the 4G LTE (Long Term Evolution) network technologies that are used today. 5G is designed to deliver faster and more connections in today's modern society. An important concept is the Internet of Things (IoT), the idea of billions of smart devices connected to people and each other. It is also designed to have greater capacity to meet the large growth in data transfer today. One of the most important advantages of 5G is the fast response time, *latency*. Latency is the time it takes for data to go from the source to the destination in milliseconds. This means the time between the moment when data is sent by the UE and the moment it is received by the gNB (or similarly, sent by the gNB and received by the UE). [26] In 3G the response time was 100 ms, in 4G it was around 30 ms and in 5G it is only 1 ms. This is basically instantaneous, which opens up a whole new world of connected applications. It enables concepts like real-time tracking of traffic and automated vehicles.

There are three key use case categories in 5G. *Massive machine-to-machine communications* is a use case that means basically the IoT – billions of devices connected without human intervention, in a whole new scale. This might enable new revolutionary improvements in industrial processes,

agriculture, manufacturing and business communications. *Ultra-reliable low-latency communications* means enabling concepts like real-time control of machines, industrial robotics, vehicle-to-vehicle communications, safety systems and autonomous driving. One significant area of use is medical care where remote care, procedures and treatments will be possible. *Enhanced mobile broadband* means that significantly faster data speeds and greater capacity are provided. This will enable easier and faster handling of data, fixed wireless internet access for homes and greater connectivity for people on the move.

All in all, 5G will make significant improvements in all areas of life. The IoT will provide smart homes, smart schools and smart cities connected in a whole new way. We will have smarter and safer vehicles, and enhanced health care and education. For businesses and industry, 5G will provide new efficient ways of taking advantage of the data available, and new smart innovations and robotics. New technologies such as virtual reality (VR) and augmented reality (AR) will be accessible by everyone. Virtual reality provides new virtual experiences like travelling to a new city or watching a live football match, making the experience to feel like one actually is there. 5G will enable many new opportunities that we have not even thought of yet.

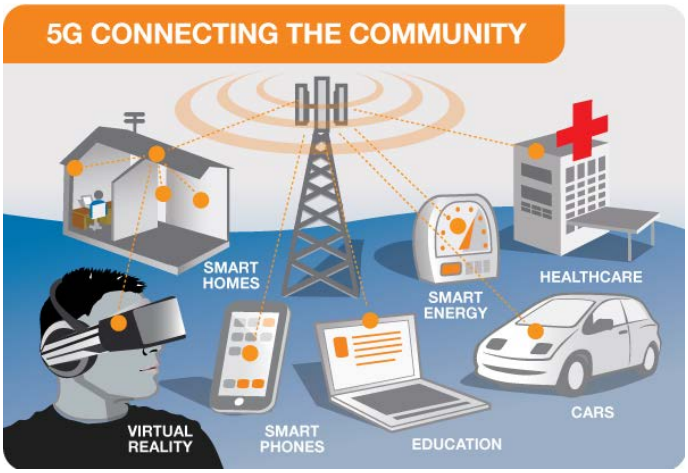

Figure 2: 5G connecting the community. [12]

1.1.4 PRACH

To be able to carry the data across the radio access network, the data is organised into different data channels. The physical channels are the channels which are closest to the actual data transmission over the network. They are used to transport the data over the actual radio interface. The transmission is done over the air.

There are three physical channels in both uplink and downlink direction in 5G. In this thesis work, we are interested in the receiver part of the gNB, which means the uplink direction. The three uplink physical channels in 5G NR (New Radio) are Physical Random Access Channel (PRACH), Physical Uplink Shared Channel (PUSCH) and Physical Uplink Control Channel (PUCCH).

The channel to be studied in this thesis work is PRACH. PRACH is used for accessing the network. The UE transmits an initial random access *preamble* that consists of long or short sequences. PRACH channel carries the preamble signal of the user, and the gNB is trying to detect the channel carrying the signal. [16]

PRACH is the first channel that is used when the signal comes in uplink direction (from the UE to the gNB). The goal of PRACH channel is to carry the user signal that is trying to access the network.

1.1.5 Simulator

The used data is generated with a link level simulator that models the complex 5G network and all its functions. There are multiple scripts for different test scenarios, for each physical channel separately. When a test script is used to run a simulation, it creates multiple folders of measured results – all the data that is obtained from running the simulation with the specific test script settings. In the resulting folders we can find for example the input parameter settings for the scenario, the data signal in I/Q format and the measured output results.

1.2 Goal of thesis

The goal is to model the behaviour of PRACH channel in the gNB – or more accurately, to model the PRACH data signal that the gNB receives in a user-defined situation. For this, a model is needed. The parameter settings of a hypothetical situation are fed in the model. Based on them, the model should be able to predict the outcome – the resulting PRACH data signal in the gNB. The goal is to create an estimator that predicts the PRACH data signal as accurately as possible.

The training data for the model is obtained from the simulations. For generating a data sample with the simulator, the input parameter settings are defined. After running the simulation, the PRACH data signal in I/Q format is acquired. Multiple different simulations are run for different cases, and wanted data from them is saved and collected.

With the data ready, the behaviour can be taught to the machine learning model. The main goal is to use the trained model to see comparable output results when using some self-defined input parameter set. The advantage of the model is not needing to run through time-consuming simulations with the simulator.

A neural network is used as a model. The model learns the behaviour of the data generated by the simulator.

1.3 Scope of thesis

After introducing the basic technical concepts and the goal of this thesis, we continue with describing the data that was used, the machine learning technique that was applied to the data and the results of the project.

Chapter 2 describes the nature of the data used in the simulations, taking a special look into the format of the output data, the I/Q signal. It is explained why neural networks were chosen as the used machine learning algorithm. The data pre-processing steps are also gone through.

In Chapter 3, the concept of neural networks is introduced. The structure

and the calculation methods in neural networks are presented in more detail. The chapter also goes through the mathematical details of the fundamental algorithm for neural networks: the back-propagation algorithm.

Chapter 4 describes the thesis project of designing a neural network for PRACH signal prediction. The used neural network and its hyperparameters are presented along with explanations on why they were chosen.

The results of the project are discussed in Chapter 5, along with some indicative measurements about how accurate the predictions of the network are. It is also tested with the simulator whether the preamble can be detected from the predicted signal. Example plots of predicted I/Q signals compared to their targets are shown and explained at the end of thesis, in Appendix.

Chapter 6 collects together some ideas that came across during the process of writing the thesis. They are saved for further studying and future works. Finally, the thesis work and results are summarized in Chapter 5.

2 Data

The data that is used to train the model has an input part and an output part. The input is the set of parameters that were used to define the simulation case, and the output is the resulting PRACH data signal in I/Q format. These values were collected from multiple simulations to acquire 4800 data samples. Each data sample is a row of data that has one set of input parameters (input features) and one PRACH data signal.

First, the case for study was chosen. It was not possible to study all PRACH behaviour: the situation needed to be limited. Therefore, some input parameters were "frozen" in order to create the model. For example, the number of UEs was chosen to be constant. If it changes between data samples, it causes problems as there are more measured values for more users. For this project, the number of UEs was chosen to be 1, so only single-user data was collected.

One data sample has 9 input features which are measurements that were used to define the simulation case. All the inputs are numerical. The output of one data sample is a PRACH data signal in the form of an I/Q vector, which is a complex integer vector, of length 15408. The input features were chosen to be parameters that have varying values in simulations. Therefore, they might have an effect on the resulting output data signal. Two input features that have an effect on the output signal are introduced as an example: let them be called Input1 and Input2. Input1 is a parameter that defines at which point in the signal the preamble might be seen. Input2 is another parameter which tells how clearly the preamble part might be seen: when Input2 value increases, the values in the preamble part of the signal increase.

As said before, the data was generated by running simulations with the simulator. The simulation scripts randomised the values of some input parameters, in their respective ranges, so that the data would be as diverse as possible. There are no duplicate rows in the data, which means all the simulation cases had different input parameter values.

2.1 I/Q data

The output part of data is the PRACH I/Q data signal received by the gNB. The I/Q data is in complex vector format, where the I part (In-phase) of a data point is the real part of that complex point, and the Q part (Quadrature) is the imaginary part of the complex point.

Basically, I/Q data represents the message signal. It can be presented with two different forms. One form is two separate variables (I, Q), in which case the data is a $(n \times 2)$ matrix. The other form is a complex number $I + Qi$, in which case the data is a complex vector of length n. We can plot the data (either the two variables or the complex number components) to x and y axis to see how it behaves.

I/Q data shows changes in the sine wave signal, specifically the changes in amplitude A and phase ϕ. This changing in amplitude and phase that has a specific order is called the *modulation* of the signal. Signal modulation, i.e. changing of the sine wave, is a way to encode information into the signal. I/Q data is a practical way to represent the signal.

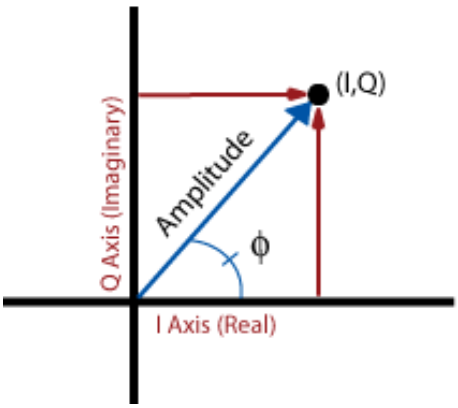

Figure 3: I/Q data point representation. [4]

I/Q data has two parts: I part and Q part. I part is the *in-phase* data, "the real signal". Here, the modulating carrier wave (the wave that carries the signal) is in phase with the real signal. Q part is the *quadrature* data which means that the carrier is offset by 90 degrees. It is 90 degrees out of phase,

so it is orthogonal to the real signal. This is called quadrature upconversion. The I/Q modulator mixes the I-waveform with the RF (Radio Frequency) carrier sine wave, and the Q-waveform with the same RF carrier sine wave, but with 90 degrees phase offset.

I/Q data represents the signal on time domain. This means that I/Q data has basically three dimensions: time, amplitude and phase. Every point of the signal, i.e. point in time, is a row in the $(n \times 2)$ matrix, or a point in the complex vector of length n, depending on which form we use. Every point also provides a new value for I and Q, meaning a new amplitude and phase for every point. This is exactly the modulation of the signal. [3] [4]

Figure 4 shows an example plot of what I/Q data signal might look like, with I part plotted on x axis and Q part plotted on y axis. This is the output part of one training data sample: PRACH I/Q data signal. It is the target I/Q data created by the simulator that the model is trying to predict. Ideal PRACH I/Q signal would resemble a circle like the outer circle seen in Figure 4. However, there is always noise in real life situations which is why the circle is uneven.

The I and Q parts of a PRACH I/Q data signal are plotted separately in Figure 5. Both I and Q parts are vectors of length 15408. Here, the preamble part can be seen at the beginning of the signal, for both I and Q. The I/Q values in the preamble part are large compared to the rest of the signal, so the preamble part is distinct.

2.2 How to choose the algorithm?

While looking for the ideal machine learning algorithm for a specific case, one needs to have a clear picture of the data that is going to be used. The problem in hand and the potential constraints also need to be understood.

First, it is essential to know and understand the data. One needs to know the amount of data available, and whether it is enough to achieve accurate results. It is important to know for example if the inputs are categorical or if there is missing data that needs filling in. It is useful to look at summary

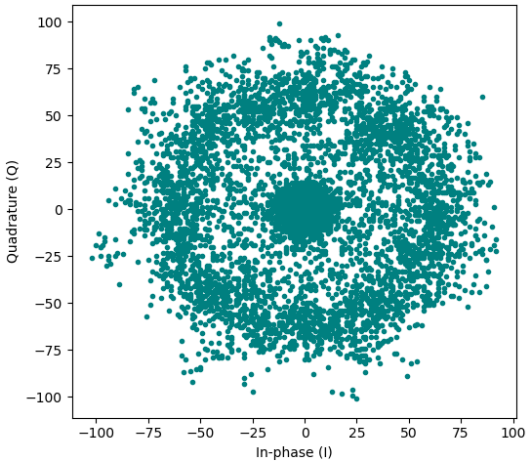

Figure 4: The PRACH I/Q data signal of one data sample.

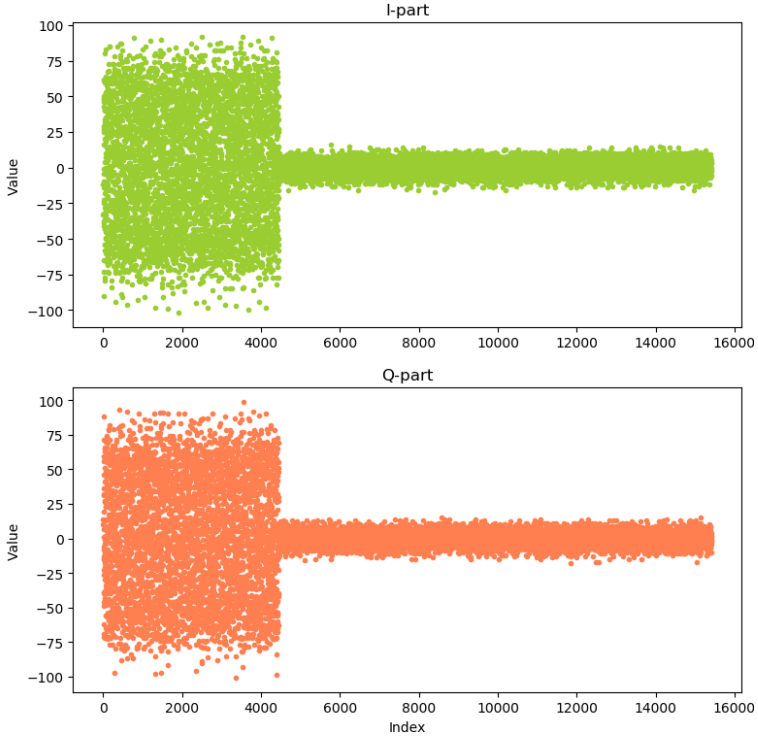

Figure 5: The same PRACH I/Q data signal as above, separated into I and Q parts.

statistics (percentiles, averages, standard deviation, correlations) and visualisations to see the spread of data, relationships between variables and possible outliers.

In most cases, data needs pre-processing in order to be used in a specific case. Data pre-processing is usually one of the most time-taking steps in training a machine learning algorithm. The data needs to be cleaned: only the needed information is taken, data is put in suitable format, and possible missing data cases are handled with a suitable processing technique. If there are radical outliers, they need to be dealt with as well. Data might also need aggregation: summarising or combining. It might be useful to reduce dimensionality in multidimensional cases or remove redundant data. Rescaling or standardising variables is often helpful. The goal is to make data from raw to ready for modelling.

Second, one needs to have a clear picture of the problem in hand. Some problems are very open and need a trial-and-error approach. These are for example supervised learning, classification and regression problems. For them, predictive models that are more general could be built.

It is good to categorise the problem. If the input data is labelled, meaning that it is known which outputs correspond to which inputs, the situation is called *supervised learning*. Otherwise, if the labels corresponding to the inputs are not known, it is called *unsupervised learning*. If the output is numerical, the problem is a *regression* problem. If the output is a class, it is a *classification* problem. If the output is a set of input groups, and the goal is to separate inputs into different groups, it is a *clustering* problem. The goal can also be detecting anomalies, in which case it is an *anomaly detection* problem.

Finally, the constraints and requirements of the problem need to be understood. What is the data storage capacity? Does the prediction have to be fast? Does the learning have to be fast? These qualities also have an effect on which algorithm to choose.

Now, the applicable and practical-to-implement algorithms for the case

can be identified, and the model that suits the problem and meets the goals can be chosen. It is beneficial to make sure that the model is fast enough, is explainable and scalable, and is complex enough to give accurate results. [7]

2.3 Data pre-processing

There needs to be enough data so that the model can be trained to be accurate. The data also needs to be diverse, so that it covers as many cases as possible and the model learns a wide area of cases. There is a large number of data accessible, but it is still finite, so it needs to be decided how much of it is used for training and how much for validation. There are altogether 4800 samples of data. The data is divided into three sets: training set, validation set and test set. In this project, 60 % (2880 samples) of the data is used for training, 25 % (1200 samples) for validation and 15 % (720 samples) for testing.

Before fitting a model to the data, the data needs to be pre-processed. There are no categorical features or missing parts in the data, so there is no need for encoding or filling in. The data is collected from result folders. The wanted 9 features are picked as input and the resulting PRACH I/Q data as output. The data is gathered into one array of 4800 rows. There are no duplicate rows in the data, so there is no redundancy.

The research problem is a situation of supervised learning because there is a corresponding output for each input – the resulting PRACH data signal for each set of input features. The output, the I/Q data vector or matrix, has numerical values, so this is a regression problem. There is a good amount of data and plenty of data storage. The memory capacity and computing power are quite good as well. The learning of the model does not have to be very fast, but the prediction that the trained model provides should be more or less instant.

Why use a neural network as an algorithm to fit to this problem? Neural networks are a hot topic at the moment. They are often very efficient in classification and regression problems in machine learning. However, using

them requires a large amount of data and usually plenty of memory and computing power. On the other hand, they can be used as a "black box" method, which means the data does not need to be understood completely beforehand – the network learns the non-linear correlations straight from the data anyway. Neural networks seemed to be a good choice for this research problem, so the model is created by building and training a neural network.

2.3.1 Standardisation

When working with neural networks, it is very common to standardise the data. Standardising the data generally speeds up learning and leads to faster convergence. It ensures that the magnitude of the values in one feature are of similar extent. If the inputs have different scales, the weights connected to some inputs will be updated much faster than others. This generally hurts the learning process, so when features have different scales, standardisation is usually needed. Standardisation also has a significant effect on the loss function values because they depend on the scale of data. [18]

Standardisation transforms the data to have a mean of 0 and a standard deviation of 1. The data is standardised by standardising each feature in input and output. The original data point x in a feature is scaled and represented as a standardised point $\hat{x}$ by subtracting the mean μ of that feature and dividing by the standard deviation σ of that feature

$$\hat{x} = \frac{x - \mu}{\sigma} \ .$$

Here, the mean μ and standard deviation σ are defined as

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i \quad \text{and} \quad \sigma = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \mu)^2} \ ,$$

where $\{x_1, \ldots, x_n\}$ are the observed values in the feature and n is the number of observations in the feature.

After standardising all data points in all features like this, the standardised features have a new mean of 0 and a new standard deviation of 1. This

means that most of the data is in the range of $[-1, 1]$, with only the "tails" of the original data distribution exceeding this range. However, the original data distribution is large and uneven, especially for the output part, the I/Q data. With certain input parameter values, the preamble part of the output signal gets very large values. In other cases, the values in preamble part are low, so the preamble cannot be easily detected from the rest of the signal. Basically, this means that even when the data is standardised to have a mean of 0 and a standard deviation of 1, there are several values in the data, especially in I/Q data, that easily exceed the range of $[-1, 1]$.

After standardisation, the values in every feature are centered around 0. The average deviation from 0 is 1. The distribution of the data is as before; the data itself does not change, only the scale of the data. When the standardised data is "unstandardised" back to original values, inverse standardisation is performed: first, the standardised data point is multiplied with the standard deviation, and then it is summed with the mean.

When the I/Q data is standardised feature by feature, some features in standardised output data get NaN values (Not a Number). This results from the fact that some features in output data have only one fixed value. Therefore, the standard deviation for that feature is 0. When the feature is standardised, each data point is divided with the standard deviation, which in this case is 0. This results in an error, and the standardised point will be a NaN value.

Before feeding the standardised data into the network, all NaN values in output data are replaced with 0. NaN is not a numerical value, so replacing them with zeros makes all the data numerical again. This way, the model is able to handle it. Replacing NaN with 0 is allowed because the data points get the correct original values again when they are unstandardised.

In the final stage, when the trained network is tested, a prediction of the PRACH signal is produced based on given inputs. The prediction can be compared to the real output signal, the target. At this point, both the prediction and the target have the standardised scale. The real signal and its

prediction can be seen in the original scale if the signals are unstandardised. This means converting the standardised values back to original scale with the same mean and standard deviation that were used for standardisation.

However, the predicted values are only estimates that seldom reach the exact target value. When the predictions are unstandardised, the resulting prediction signal is not necessarily integer-valued. The real PRACH signal (the target) consists of integer values, so the predicted signal is rounded to be in integer format, to be able to do comparison. In some cases, rounding the values might make the difference between the target value and the predicted value bigger. For example, after unstandardisation, if some target I/Q point is $(1, 1)$ and the predicted point is $(0.4, 1.6)$, the rounded prediction will be $(0, 2)$ which is further from the real point than the unrounded one.

3 Neural networks

3.1 Artificial neural networks

An artificial neural network (ANN), or neural network (NN), is a set of algorithms that is loosely based on the functioning of human brain. Basically, neural networks combine the working flow of human brain and mathematical logic. They were originally designed for pattern recognition, a part of machine learning. Nowadays, the neural network technique development focuses also on the fields of statistics and signal processing.

Neural networks interpret data; they help us label, cluster and classify data. They group unlabelled data by finding similarities among the input samples. They can also classify data if they have a *labelled training data* – a data set with the input data and their corresponding labels. Neural networks can be trained based on the training data, meaning that the network learns which kind of inputs produce which kind of outputs.

Basically, neural networks map the inputs to the outputs. Besides classification problems, they are also widely used for predictive analysis, regression. When a neural network is trained with some training data, the network gives a prediction of outputs if the inputs are fed in. If the inputs are changed, the network produces new predictions based on them. This provides information on when there might be errors, what are they caused by and how to possibly avoid them. Knowing this is useful for example in predicting hardware behaviour and hardware breakdowns. "The better we can predict, the better we can prevent." Neural networks are introducing us to a world with fewer surprises – not zero surprises, just fewer of them. [5]

Neural networks are composed of two or more *layers*. These layers are made of *nodes*, which are the places where the computation happens. Nodes are loosely based on neurons in human brain. A node in neural network combines the input data with *weights*, coefficients that assign a significance to each input, depending on the task that the algorithm is trying to learn. Weights give more significance to those inputs that are more helpful for pre-

dicting without error. All the inputs and their weights are summed together, and the sum is passed through an *activation function*. The activation function determines whether and to what extent the signal should progress further through the network, to affect the outcome.

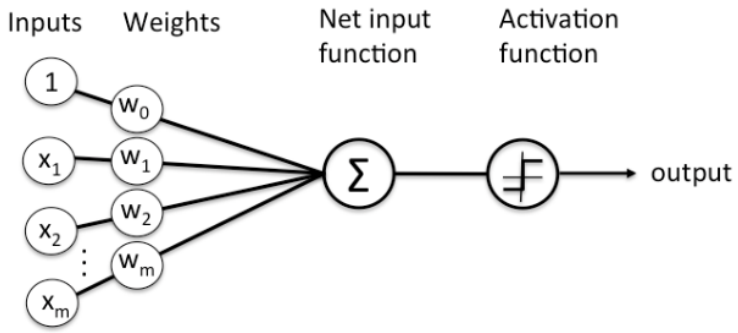

Figure 6: A diagram of what one node might look like. [5]

A layer is a row of nodes through which the input data is fed. The output of each layer is the input for the next layer. The network starts from an initial *input layer* that receives the input data. The number of nodes in input layer is equal to the number of features in the input data. There is no computation happening in the input layer; it just passes the inputs through to the next layer. The network ends in an *output layer* which makes a prediction of the outcome. The number of output nodes (nodes in output layer) depends on whether the output to be predicted consists of a single value or multiple values.

There can be multiple *hidden layers* between the input and output layers, to make the network more complex and the predictions better. If the neural network is composed of several hidden layers, the situation is called *deep learning*. Each layer of nodes trains on a distinct set of features, from the outputs of the previous layer. The further in the network, the more complex the features that the nodes recognise are. This is known as *feature hierarchy*, the increase of complexity and abstraction. This is why deep learning networks can handle large, high-dimensional data sets with billions of parameters.

They can also discover latent structures (hidden structures) in unlabelled, unstructured data – which is most of the data in the world. They perform automatic feature extraction without human intervention, unlike most traditional machine learning algorithms.

Deep neural networks are distinguished from the single-hidden-layer networks by their *depth*, the number of layers through which the data must pass. First versions of neural networks, such as the first perceptrons, were composed of only one input and one output layer, and at most one hidden layer between them. If a network has more than one hidden layer, it qualifies as deep learning.

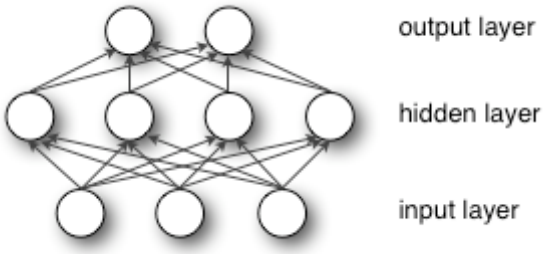

Figure 7: An example of what a simple network might look like. [5]

A neural network that has been trained on labelled data can then be applied to unlabelled data. However, it is very important to have enough data to train on. The more data a network can train on, the more accurate it is likely to be. Bad algorithms trained on lots of data can actually perform better than good algorithms trained on very little data.

Neural network is "born in ignorance". This means that at first, it does not know which weights will provide the best translation of the input to make a correct guess. It has to start with just a guess and then try to make better guesses after each time, learning from its mistakes.

What exactly happens during learning in the neural network? First, the input enters the network, and the weights map the input to a set of guesses. The network compares the guess to the correct result. It calculates the *loss*,

the error due to the difference between the guess and the correct result. Finally, the network updates the model based on the error, adjusting the weights to the extent they contributed to the error. These are the three key functions of neural networks: mapping the input, calculating the loss and updating the model. The network repeats this three-step process over and over again. Neural network is a corrective feedback loop, rewarding weights that support its correct guesses and punishing weights that lead to errors.

What happens at every node of a neural network is essentially a form of *multiple linear regression*. In multiple linear regression, we have many input variables producing an output variable. It can be expressed for example as an equation

$$\tilde{Y} = b_1 \cdot x_1 + b_2 \cdot x_2 + b_3 \cdot x_3 + a \ ,$$

where the estimated output variable $\tilde{Y}$ is produced by summing the input variables x_i, each multiplied with a corresponding coefficient b_i $(i = 1, 2, 3)$, and adding a constant a.

The inputs that enter a node are all combined with each other: they are mixed in different proportions according to their weights. The weights are different for each node of a layer. The network tests which combinations of inputs are significant as it tries to reduce the error. [5]

Like the brain, neural network "learns" by changing the strengths of the connections between nodes, or by adding or removing such connections. These connections are the weights. Learning itself is accomplished sequentially from increasing amounts of experience. [1]

Deep neural networks can be used if the wanted outcomes are labelled, i.e. if the link between inputs and outputs is known – in other words, if the problem is a supervised learning problem. This correlation between inputs and outcomes can be taught to the neural network algorithm.

Deep learning is an extremely popular and widely used method nowadays. Unlike many other machine learning algorithms, neural networks get better with more data, and the amount of data available today is massive. Traditional machine learning algorithms will eventually reach a level where

adding more data does not improve their performance any more, but that does not happen with deep learning. The computational power available today is also increasing, and it enables us to process more data, which is needed when using neural networks. The development of neural network algorithms has made them run much faster than before which makes them even more effective.

The main advantage of neural networks is their ability to outperform nearly every other machine learning algorithm, but there are also some disadvantages. One of them is the "black box" nature of neural networks, which means it is unknown how and why the network came up with a certain output. Due to that, neural networks are not easily interpretable or explainable, and those traits are important in some cases where it is essential to justify the decisions in a human-understandable way. There is not always a large amount of data available, which neural networks usually require in order to work well. In these cases, other machine learning algorithms that deal much better with little data would be a better choice. Neural networks also tend to be more computationally expensive than traditional algorithms. It depends on the size of the data and the depth and complexity of the network, but usually neural networks are very resource and memory intensive. The research of neural networks is relatively new because the hardware to support neural computation did not become available until the 1980s.

Figure 8: One disadvantage of neural networks is their "black box" nature. [6]

Of course, neural networks are not the best solution for every problem.

Sometimes other algorithms provide better results. The resources used for the training also need to be considered. Some algorithms might give better results with less resources, like less time spent for training or less memory or computing power needed. Choosing the method or the algorithm depends strongly on the problem and the situation. There is no "perfect" algorithm that will perform well with any problem. [6]

3.2 Single-Layer Perceptron

One of the first neural network developments was done in 1958 by Frank Rosenblatt who constructed a minimally constrained system called a *perceptron*. [1] A perceptron is essentially a single node. It consists of multiple inputs X_m $(m = 1, \ldots, r)$ and a single output Y. Each input can be either binary, meaning it has a value of either 0 or 1 ("off" or "on"), or real-valued. For each input, there is a real-valued connection weight β_m $(m = 1, \ldots, r)$. Weights can be positive or negative. The magnitude of the weight shows the importance of the connection.

A weighted sum U of all input values is calculated as

$$U = \sum_{m=1}^{r} \beta_m X_m \ .$$

The output is then

$$Y = \begin{cases} 1 & \text{if} \quad U \geq \theta \\ 0 & \text{otherwise,} \end{cases}$$

where θ is the threshold value (see Figure 9).

Note that the threshold θ can be converted to 0 by introducing an intercept $\beta_0 = -\theta$, so that $U + \beta_0 = U - \theta$. Then 0 can be used as the threshold and compared to

$$U = \sum_{m=0}^{r} \beta_m X_m \ .$$

Note that here $X_0 = 1$. Now the output can be written as

$$Y = \begin{cases} 1 & \text{if} \quad U \geq 0 \\ 0 & \text{otherwise.} \end{cases}$$

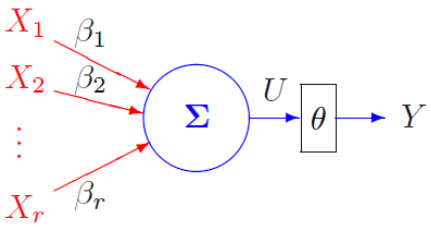

Figure 9: Single-layer perceptron with inputs X_m, weights β_m, weighted sum U, threshold θ and binary output Y. [1]

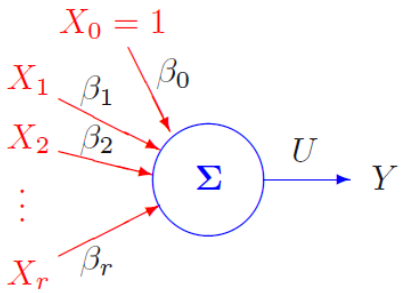

Figure 10: The same perceptron with inputs X_m, weights β_m, weighted sum U and output Y, but now we have the intercept $\beta_0 = -\theta$ and $X_0 = 1$, so the threshold is $\theta = 0$. [1]

3.3 Activation functions

Let $X = (X_1, \ldots X_r)^T$ represent a random r-vector of inputs. If there are multiple, say s output nodes instead of one binary output, each output node computes an activation value using a linear combination of the inputs. For the ℓth output node, the ℓth *linear activation function* can be computed as

$$U_\ell = \beta_{0\ell} + \sum_{m=1}^{r} \beta_{m\ell} X_m = \beta_{0\ell} + X^T \beta_\ell \ ,$$

where $\beta_{0\ell}$ is a constant or an *intercept* related to the threshold for the node, and $\beta_\ell = (\beta_{1\ell}, \ldots, \beta_{r\ell})^T$ is an r-vector of connection weights, $\ell = 1, \ldots, s$.

The collection of s linear activation functions can be rewritten as

$$U = \beta_0 + BX \ ,$$

where $U = (U_1, \ldots, U_s)^T$, $\beta_0 = (\beta_{01}, \ldots, \beta_{0s})^T$ is an s-vector of intercepts, and $B = (\beta_1, \ldots, \beta_s)^T$ is an $(s \times r)$-matrix of connection weights.

The activation values are then filtered through a non-linear *threshold activation function* $f(U_\ell)$, to form the value for the ℓth output node. This can be expressed in matrix notation as

$$F(U) = F(\beta_0 + BX) \ ,$$

where $F = (f, \ldots, f)^T$ is an s-vector function where each element is the function f, and $F(U) = (f(U_1), \ldots, f(U_s))^T$.

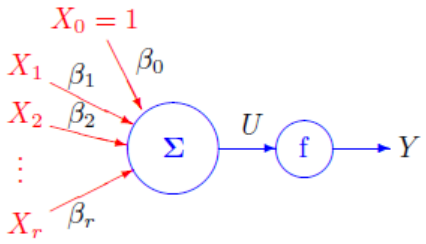

Figure 11: The perceptron with r inputs X_m, intercept β_0, connection weights β_m, weighted sum U, activation function f and one binary output Y. [1]

The most commonly used non-linear activation functions are the *sigmoidal functions*, the "S-shaped" functions, because of their limited range and differentiability. A sigmoidal function is a function $\sigma(\cdot)$ that has the following properties:

$$\begin{cases} \sigma(x) \to 0 & \text{as} \quad x \to -\infty \ , \text{ and} \\ \sigma(x) \to 1 & \text{as} \quad x \to +\infty \ . \end{cases}$$

A sigmoidal function $\sigma(\cdot)$ is *symmetric* if $\sigma(x) + \sigma(-x) = 1$, and *asymmetric* if $\sigma(x) + \sigma(-x) = 0$. Examples of sigmoidal functions are the logistic and the hyperbolic tangent. The logistic function is symmetric, while the hyperbolic

tangent (tanh) is asymmetric. The logistic function is a traditional sigmoidal function which has a range of $[0, 1]$. The hyperbolic tangent has a range of $[-1, 1]$. [13]

ReLU (Rectified Linear Unit) is a widely used non-linear activation function. The formula of ReLU is simple:

$$\mathrm{ReLU}(x) = \max\{0, x\} \ .$$

ReLU is usually the default choice for activation function because it generally works the best and is computationally efficient. Plots of these activation functions are shown in Figure 12.

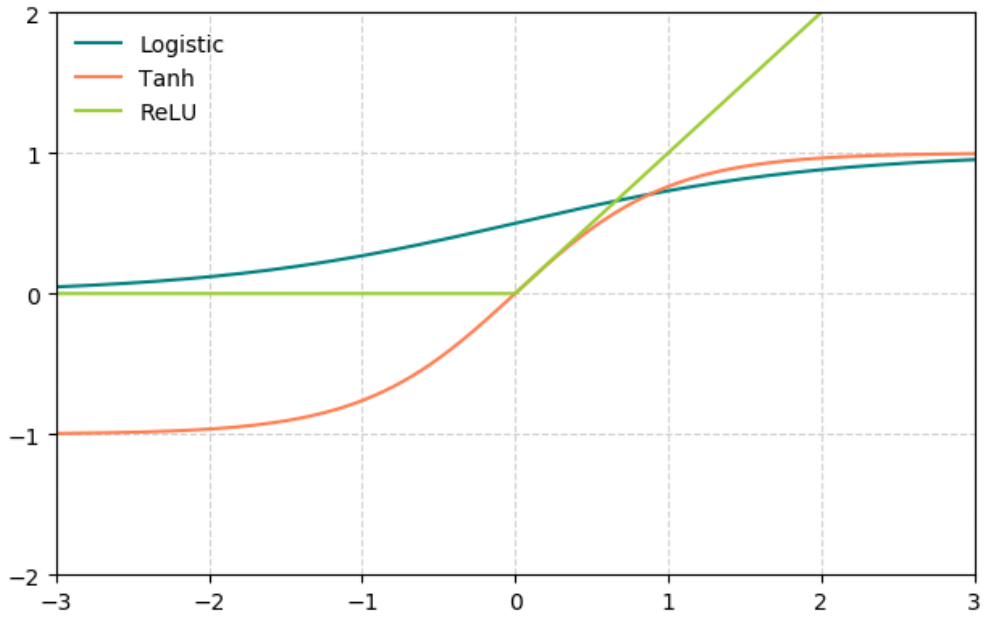

Figure 12: Three examples of activation functions: logistic, hyperbolic tangent (tanh) and ReLU.

A neural network without an activation function is just a linear regression model. If there is no activation function, the weights simply do a linear transformation. A linear equation is easy to solve, but it is not able to solve complex problems and learn complex functional mappings from data. Using non-linear activation functions help the network learn complex data and

provide accurate predictions.

In regression problems, no activation function is used for the output layer. This means that no transformation is done to get the output values (the prediction). Instead, the numerical values without any transformation are needed. If for example a logistic sigmoid function is used, the output values are always between 0 and 1. That is not wanted, unless the data that the predictions are done for is distributed within that range. [28]

3.4 Multilayer perceptrons

The perceptron is still relatively simple and has some limitations. There were suggestions by Minsky and Papert (1969) that the limitations of the perceptron could be overcome with layering the perceptrons and applying non-linear transformations before combining the weighted inputs. At that time these suggestions were not adopted because of computational limits, but when high-speed computers became available and the "back-propagation" algorithm was discovered, Minsky and Papert's suggestions became more meaningful.

Multilayer perceptron (MLP), also called a *feed-forward neural network* or a *deep feed-forward network*, is a multivariate statistical technique that non-linearly maps an input vector $X = (X_1, \ldots, X_r)^T$ to an output vector $Y = (Y_1, \ldots, Y_s)^T$. Between the inputs and outputs, there are "hidden" variables or hidden nodes arranged in layers. A typical neural network has two computational layers: the hidden layer and the output layer. The input layer does not require any computing. It consists of the input nodes which are just the input variables.

Neural networks can be used to model regression or classification problems. In a univariate regression situation, there is only one output variable or output node Y ($s = 1$), while in a multivariate regression situation, such as in this project, there are s output nodes $Y_1, \ldots, Y_s$.

Multilayer perceptrons have r input nodes $X_1, \ldots, X_r$, one or more layers of hidden nodes, and s output nodes $Y_1, \ldots, Y_s$. Each layer of hidden nodes is

called a hidden layer. If there is only one hidden layer, the network is called a two-layer network, where the output layer is the second computational layer. Generally, if there are L hidden layers, the network is called an $(L+1)$-layer network. An illustrative example of a multilayer perceptron can be seen in Figure 13.

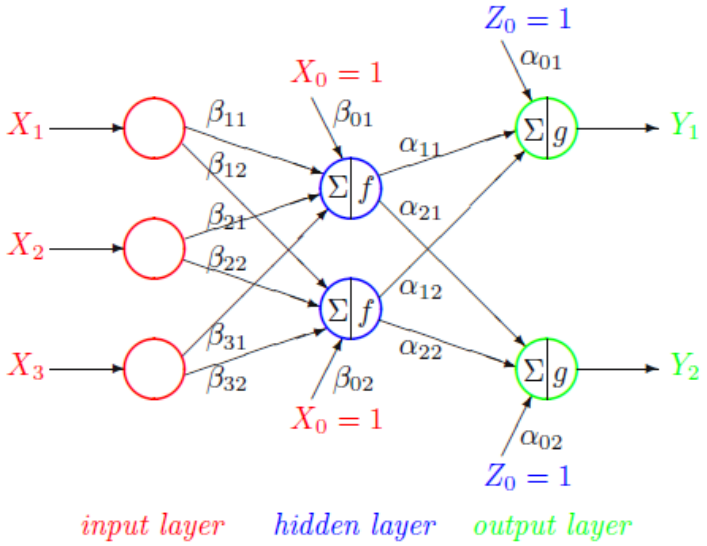

Figure 13: Multilayer perceptron (or neural network) with a single hidden layer. The network has $r = 3$ input nodes, $s = 2$ output nodes, and $t = 2$ nodes in the hidden layer. The α's and β's are weights attached to the connections between nodes, and f and g are activation functions for each computational layer. [1]

A *fully connected network* is a network where for each layer, all nodes of that layer are connected to all nodes in the next layer (if there is one). If some of the connections are missing, the network is a *partially connected network*. However, a partially connected network can always be represented as a fully connected network, by setting the weights of the missing connections to zero.

3.5 Weight optimisation

Neural network tries to make a correct prediction of the output by first taking a guess, then calculating the loss, and then updating the weights in all layers based on the calculated loss, in order to make a better guess next time. It does this repeatedly, trying to make better guesses each time. But how exactly is the updating and optimising of weights done in neural networks?

The most popular numerical method for optimising the network weights is the *back-propagation algorithm* (Werbos, 1974). The back-propagation algorithm computes the first derivatives of a loss function with respect to the weights. These derivatives are used to update the weights by minimising the loss function with an iterative gradient descent method. [1]

3.5.1 Gradient descent

In neural networks, the goal is to minimise the loss function $J(\theta)$ with respect to the parameters θ which are the weights of the network. The purpose is to find a value for θ that minimises the loss function. This value is denoted with

$$\theta^* = \min J(\theta) \ .$$

The derivative $J'(\theta)$ is used for minimising the function $J(\theta)$. The derivative $J'(\theta)$ provides the slope of the function $J(\theta)$ at the point θ. It specifies how to scale a small change γ in the input to obtain the corresponding change in the output:

$$J(\theta + \gamma) \approx J(\theta) + \gamma \cdot J'(\theta) \ .$$

In other words, it explains how to change parameters θ in order to make a small improvement in the loss function $J(\theta)$. For example, it is known that $J(\theta) - \gamma \cdot \text{sign}(J'(\theta)) < J(\theta)$ for small enough γ. Therefore, the loss function $J(\theta)$ can be reduced by moving the parameters θ in small steps of γ, with the opposite sign of the derivative. This technique for minimising a function is called *gradient descent* (see Figure 14). [8]

When $J'(\theta) = 0$, the derivative provides no information on which direction to move. These points where $J'(\theta) = 0$ are called *critical points* or *stationary*

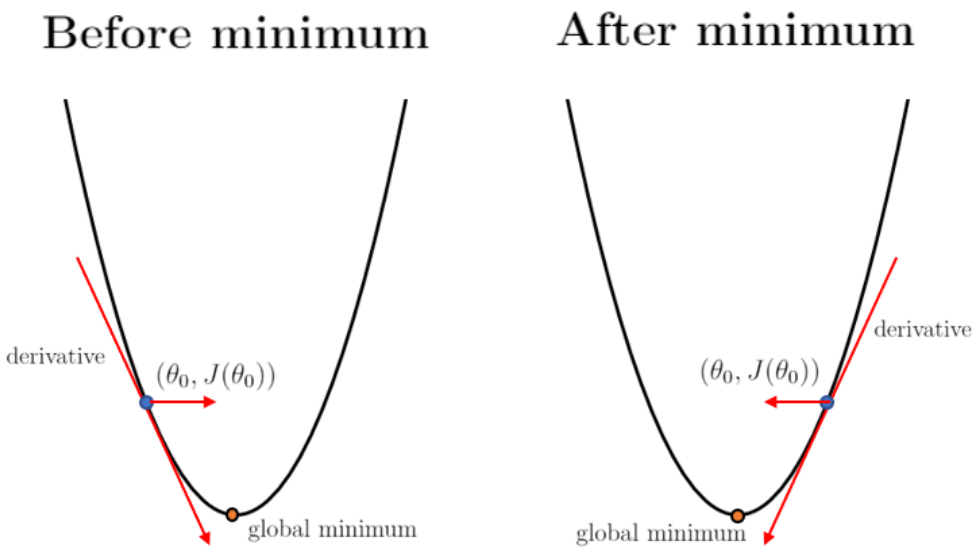

Figure 14: Gradient descent method: finding the direction to minimise function $J(\theta)$. [14]

points. A point where $J(\theta)$ is lower than at all neighbouring points is called a *local minimum*. In this case, it is no longer possible to decrease $J(\theta)$ by making small steps. A *local maximum* is a point where $J(\theta)$ is higher than at all neighbouring points. If a critical point is neither maximum nor minimum, it is called a *saddle point*.

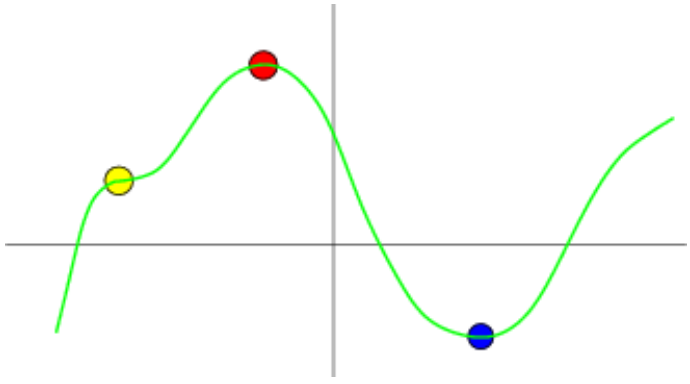

Figure 15: Critical point types: local maximum (red), local minimum (blue) and saddle point (yellow). [15]

A point that obtains the absolute lowest value of $J(\theta)$ is a *global minimum*. A function can have one or multiple global minima. In deep learning, optimisation is often done for functions that may have local minima that are not globally optimal, and for functions that may have many saddle points surrounded by flat regions. This makes optimisation difficult, especially when the function input is multidimensional. Therefore, it is usually sufficient to settle for finding a value of the function that is very low but not necessarily minimal.

For minimising functions with multiple inputs, $J : \mathbb{R}^n \to \mathbb{R}$, *partial derivatives* must be used. The partial derivative $\frac{\partial}{\partial\theta_i}J(\theta)$ measures how J changes as only the variable θ_i increases at point θ. The *gradient* of J is the vector containing all the partial derivatives and is denoted as $\nabla_\theta J(\theta)$. In other words,

$$\nabla_\theta J(\theta) = \left(\frac{\partial}{\partial\theta_1}J(\theta),\ \dots\ , \frac{\partial}{\partial\theta_n}J(\theta)\right) \quad \text{for } \theta = (\theta_1, \dots, \theta_n)\ .$$

In multiple dimensions, critical points are points where every element of the gradient is equal to zero, i.e. $\frac{\partial}{\partial\theta_i}J(\theta) = 0$ for all θ_i. [8]

The *directional derivative* in direction of unit vector u is the slope of the function J in direction u. In other words, the directional derivative is the derivative of function $J(\theta + \alpha u)$ with respect to α, evaluated at $\alpha = 0$. Using the common chain rule of differentiation, it can be seen that

$$\frac{\partial}{\partial\alpha}J(\theta + \alpha u) \ = \ u^\top \nabla_\theta J(\theta) \quad \text{when } \alpha = 0\ .$$

To minimise loss function J, the direction in which J decreases the fastest needs to be found. This can be done using the directional derivative as

$$\min_u \left\{u^\top \nabla_\theta J(\theta)\right\} \ = \ \min_u \left\{\|u\|_2\ \|\nabla_\theta J(\theta)\|_2\ \cos\varphi\right\}\ ,$$

where φ is the angle between u and the gradient. Because $\|u\|_2 = 1$ and $\|\nabla_\theta J(\theta)\|_2$ does not depend on u, this can be simplified to $\min_u \{\cos\varphi\}$. This is minimised when u points in the opposite direction as the gradient. In other words, the gradient points directly uphill, and the negative gradient

points directly downhill. The loss function J can be decreased by moving it in the direction of the negative gradient. This is known as the *method of gradient descent*, or *steepest descent*.

Gradient descent proposes a new point

$$\theta' = \theta - \varepsilon \cdot \nabla_\theta J(\theta) \ ,$$

where ε is the *learning rate*, a positive scalar that determines the size of the step in the iterative process. If ε is too large, the iterations move rapidly towards a local minimum, but may possibly overshoot it. In the other hand, if ε is too small, the iterations may take a long time to get anywhere near a local minimum. [1] Choosing and optimising ε can be done in several different ways. A popular approach is to set ε to a small constant. [8]

3.5.2 Back-propagation algorithm

When a feed-forward neural network is used to accept a set of inputs X and predict a set of outputs $\tilde{Y}$, the data flows forward through the network. The set of inputs X have the initial information which "propagates" up to the neurons in hidden layers and finally produces the outputs $\tilde{Y}$. That is called *forward propagation*. During training, the forward propagation continues onward until it produces a scalar cost $J(\theta)$. The *back-propagation algorithm* allows the information from the cost function or the loss function $J(\theta)$ to flow backward through the network in order to compute the gradient. [8]

The back-propagation algorithm is an iterative gradient-descent-based algorithm. Using randomly chosen initial values for the weights, the direction that makes the loss function smaller is searched. The back-propagation algorithm evaluates the gradient using a simple and inexpensive procedure.

It is to be noted that back-propagation refers only to the method of computing the gradient. Other algorithms called *optimisers*, for example Stochastic Gradient Descent (SGD) or Adam (Adaptive Moments), are used to perform the actual learning using this gradient. Learning algorithms usually require the gradient of the loss function J with respect to the parameters θ:

$\nabla_\theta J(\theta)$.

To simplify the description of back-propagation algorithm, we introduce an example of a single-hidden-layer network. All the presented details can be generalised to deeper networks.

Denote the set of r input nodes by $\mathcal{M}$, the set of t hidden nodes by $\mathcal{J}$ and the set of s output nodes by $\mathcal{K}$. Hence, $m \in \mathcal{M}$ indexes an input node, $j \in \mathcal{J}$ indexes a hidden node and $k \in \mathcal{K}$ indexes an output node. There are n input vectors of length r to feed in the network, and the ith input r-vector is denoted by $X_i = (x_{i1}, x_{i2}, \ldots, x_{ir})$, $i = 1, \ldots, n$.

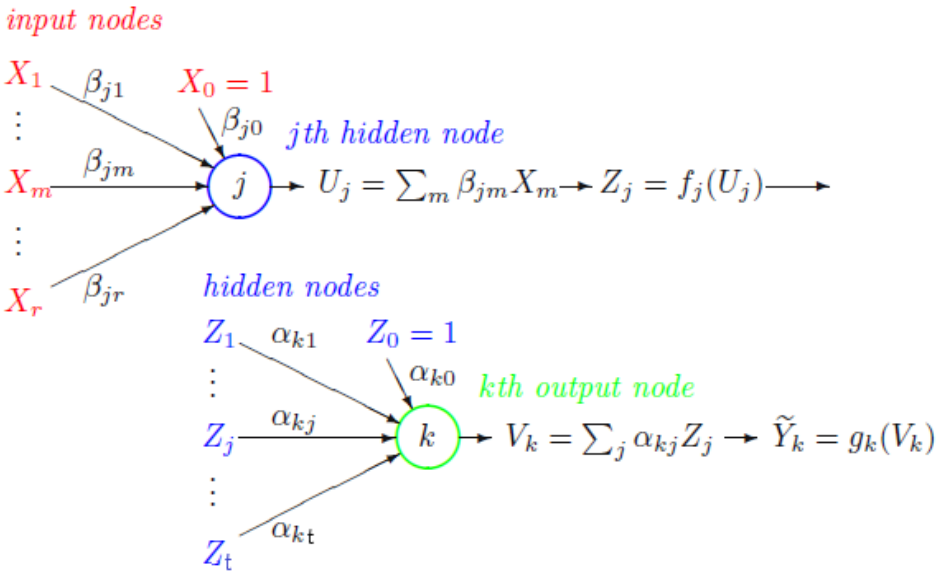

Figure 16: Schematic diagram of single-hidden-layer network. The top diagram relates the input nodes X_m to the jth hidden node, and the bottom diagram relates the hidden nodes Z_j to the kth output node. For simplicity, all reference to the ith input vector has been dropped. [1]

To perform back-propagation, we start at the kth output node. Denote the error at that node by the difference between the predicted output and the target output

$$e_{i,k} = Y_{i,k} - \tilde{Y}_{i,k} \ , \quad k \in \mathcal{K} \ .$$

Denote the loss function for the input vector i by the mean squared error (MSE) of the errors of all s output nodes

$$E_i = \frac{1}{s}\sum_{k=1}^{s} e_{i,k}^2 \ , \quad i = 1, \ldots, n \ .$$

The optimisation criterion is the loss function averaged over all data in the learning set

$$E = \frac{1}{n}\sum_{i=1}^{n} E_i = \frac{1}{n \cdot s}\sum_{i=1}^{n}\sum_{k=1}^{s} e_{i,k}^2 \ .$$

The learning problem is to minimise E with respect to the weights $\beta_{i,jm}$ and $\alpha_{i,kj}$, where $\beta_{i,jm}$ are the weights between the input layer and the hidden layer, and $\alpha_{i,kj}$ are the weights between the hidden layer and the output layer, as seen in Figure 16. Because each derivative of E with respect to these weights is a sum over the learning data set of the derivatives of E_i, $i = 1, \ldots, n$, it is enough to minimise each E_i separately.

Now, for the ith input vector, let

$$V_{i,k} = \sum_{j \in \mathcal{J}} \alpha_{kj} Z_{i,j} = \alpha_{k0} + Z_i^\top \alpha_k \ , \quad k \in \mathcal{K} \ ,$$

be a weighted sum from the set of hidden units to the kth output node, where

$$Z_i = (Z_{i,1}, \ldots, Z_{i,t})^\top \ , \quad \alpha_k = (\alpha_{k1}, \ldots, \alpha_{kt})^\top \ ,$$

and $Z_{i,0} = 1$. Then, the corresponding output is

$$\tilde{Y}_{i,k} = g_k(V_{i,k}) \ , \quad k \in \mathcal{K} \ ,$$

where g_k is the output activation function which is assumed to be differentiable.

Next, we search for the direction that makes the loss function E_i smaller. Consider the weights $\alpha_{i,kj}$ from the jth hidden node to the kth output node. Let $\alpha_i = \left(\alpha_{i,1}^\top, \ldots, \alpha_{i,s}^\top\right)^\top = (\alpha_{i,kj})$, $k = 1, \ldots, s$ and $j = 1, \ldots, t$, to be the ts-vector of all the hidden-layer-to-output-layer weights at the ith iteration. The update rule is then

$$\alpha_{i+1} = \alpha_i + \Delta\alpha_i \ ,$$

where

$$\Delta\alpha_i = -\varepsilon\frac{\partial E_i}{\partial \alpha_i} = \left(-\varepsilon\frac{\partial E_i}{\partial \alpha_{i,kj}}\right) = (\Delta\alpha_{i,kj}) \ .$$

The learning parameter ε specifies how large each step should be in the iterative process.

Using the chain rule for differentiation, we get

$$\begin{aligned}\frac{\partial E_i}{\partial \alpha_{i,kj}} &= \frac{\partial E_i}{\partial e_{i,k}} \cdot \frac{\partial e_{i,k}}{\partial \tilde{Y}_{i,k}} \cdot \frac{\partial \tilde{Y}_{i,k}}{\partial V_{i,k}} \cdot \frac{\partial V_{i,k}}{\partial \alpha_{i,kj}} \\ &= e_{i,k} \cdot (-1) \cdot g_k'(V_{i,k}) \cdot Z_{i,j} \\ &= -e_{i,k} \cdot g_k'(\alpha_{i,k0} + Z_i^\top \alpha_{i,k}) \cdot Z_{i,j} \ .\end{aligned}$$

This can be expressed as

$$\frac{\partial E_i}{\partial \alpha_{i,kj}} = -\delta_{i,k} \cdot Z_{i,j} \ ,$$

where

$$\delta_{i,k} = -\frac{\partial E_i}{\partial \tilde{Y}_{i,k}} \cdot \frac{\partial \tilde{Y}_{i,k}}{\partial V_{i,k}} = e_{i,k} \cdot g_k'(V_{i,k}) \tag{1}$$

is the *sensitivity* (or *local gradient*) of the ith observation at the kth output node. This expression for $\delta_{i,k}$ is the product of two terms associated with the kth node: the error signal $e_{i,k}$ and the derivative of the activation function, $g_k'(V_{i,k})$.

Thus, the gradient descent update to weights $\alpha_{i,kj}$ is given by

$$\alpha_{i+1,kj} = \alpha_{i,kj} - \varepsilon \cdot \frac{\partial E_i}{\partial \alpha_{i,kj}} = \alpha_{i,kj} + \varepsilon \cdot \delta_{i,k} \cdot Z_{i,j} \ , \tag{2}$$

where ε is the learning rate parameter of the back-propagation algorithm.

The next part of the back-propagation algorithm is to derive an update rule for the weights from the mth input node to the jth hidden node. At the ith iteration, let

$$U_{i,j} = \sum_{m \in \mathcal{M}} \beta_{i,jm} X_{i,m} = \beta_{i,j0} + X_i^\top \beta_{i,j} \ , \quad j \in \mathcal{J} \ ,$$

be the weighted sum from the set of input nodes to the jth hidden node, where

$$X_i = (X_{i,1}, \ldots, X_{i,r})^\top \ , \quad \beta_{i,j} = (\beta_{i,j1}, \ldots, \beta_{i,jr})^\top \ ,$$

and $X_{i,0} = 1$. The corresponding output is

$$Z_{i,j} = f_j(U_{i,j}) \ ,$$

where f_j is the activation function at the jth hidden node and is assumed to be differentiable.

Let $\beta_i = (\beta_{i,1}^\top, \ldots, \beta_{i,t}^\top)^\top = (\beta_{i,jm})$ be the ith iteration of the $(r+1)t$-vector of all the input-layer-to-hidden-layer weights. Then, the update rule is

$$\beta_{i+1} = \beta_i + \Delta\beta_i \ ,$$

where

$$\Delta\beta_i = -\varepsilon \frac{\partial E_i}{\partial \beta_i} = \left(-\varepsilon \frac{\partial E_i}{\partial \beta_{i,jm}} \right) = (\Delta\beta_{i,jm}) \ .$$

Using the chain rule again, we have

$$\frac{\partial E_i}{\partial \beta_{i,jm}} = \frac{\partial E_i}{\partial Z_{i,j}} \cdot \frac{\partial Z_{i,j}}{\partial U_{i,j}} \cdot \frac{\partial U_{i,j}}{\partial \beta_{i,jm}} \ ,$$

where the first term is

$$\begin{aligned} \frac{\partial E_i}{\partial Z_{i,j}} &= \sum_{k \in \mathcal{K}} e_{i,k} \cdot \frac{\partial e_{i,k}}{\partial Z_{i,j}} \\ &= \sum_{k \in \mathcal{K}} e_{i,k} \cdot \frac{\partial e_{i,k}}{\partial V_{i,k}} \cdot \frac{\partial V_{i,k}}{\partial Z_{i,j}} \\ &= -\sum_{k \in \mathcal{K}} e_{i,k} \cdot g_k'(V_{ij}) \cdot \alpha_{i,kj} \\ &= -\sum_{k \in \mathcal{K}} \delta_{i,k} \alpha_{i,kj} \ . \end{aligned} \tag{3}$$

Thus, we get

$$\frac{\partial E_i}{\partial \beta_{i,jm}} = -\sum_{k \in \mathcal{K}} e_{i,k} g_k'(\alpha_{i,k0} + Z_i^\top \alpha_{i,k}) \alpha_{i,kj} \cdot f_j'(\beta_{i,j0} + X_i^\top \beta_{i,j}) \cdot X_{i,m} \ .$$

Putting equations (1) and (3) together, we get

$$\delta_{i,j} = f_j'(U_{i,j}) \sum_{k \in \mathcal{K}} \delta_{i,k} \alpha_{i,kj} \ .$$

The expression for $\delta_{i,j}$ is the product of two terms. The first term, $f'_j(U_{i,j})$, is the derivative of the activation function f_j evaluated at the jth hidden node. The second term is a weighted sum of the $\delta_{i,k}$ (which requires knowledge of the error $e_{i,k}$ at the kth output node) over all output nodes, where the kth weight, $\alpha_{i,kj}$, is the connection weight of the jth hidden node to the kth output node. Thus, $\delta_{i,j}$ at the jth hidden node depends on the set $\{\delta_{i,k}\}$ from all the output nodes.

The gradient descent update to $\beta_{i,jm}$ is given by

$$\beta_{i+1,jm} = \beta_{i,jm} - \varepsilon \frac{\partial E_i}{\partial \beta_{i,jm}} = \beta_{i,jm} + \varepsilon \cdot \delta_{i,j} \cdot X_{i,m} \ , \tag{4}$$

where ε is the learning rate parameter of the back-propagation algorithm.

The back-propagation algorithm is defined by the weight-update equations (2) and (4). These update formulas identify two stages of computation in this algorithm: a "feed-forward pass" stage and a "back-propagation pass" stage. After an initialisation step in which the weights are assigned values, these stages in the algorithm are performed.

In *feed-forward pass*, the inputs enter the node from the left and emerge from the right of the node. The output from the node is computed as a weighted sum, and the results are passed from left to right through the layers of the network.

In *back-propagation pass*, the network is run in reverse order, layer by layer, starting at the output layer. First, the error is computed at the kth output node and then multiplied by the derivative of the activation function, in order to get the sensitivity $\delta_{i,k}$ at that output node. The weights $\{\alpha_{i,kj}\}$ that are feeding into the output nodes are updated by using equation (2). Second, the sensitivity $\delta_{i,j}$ at the jth hidden node is computed. Then, the weights $\{\beta_{i,jm}\}$ that are feeding into the hidden nodes are computed by using equation (4).

This iterative process is repeated until some suitable stopping time.

3.6 Loss functions

There are multiple ways to calculate the loss at the output node. Mean Squared Error (MSE) is the most commonly used loss function for regression problems. It can also be called L2 loss. MSE is the sum of squared distances between the target variable Y and the predicted value $\tilde{Y}$

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \tilde{Y}_i)^2 \ .$$

The range of MSE function is from 0 to ∞; the closer to 0 the MSE value is, the smaller the loss.

Whenever a machine learning model is trained, the goal is to find the point that minimises the loss function. However, when the back-propagation algorithm is used to search for the minimum of the loss function, the found point might be a local minimum. In that case, it is not be possible to find a direction to take small steps in, so that the loss function value continues decreasing. In a local minimum, the loss function value always differs from 0. Naturally, the loss function reaches the global minimum, 0, when the prediction is exactly equal to the true value.

Since MSE squares the error e ($e = Y - \tilde{Y}$), the value of MSE increases significantly if $e > 1$. This means that if there is an outlier in the data, the value of e is large, and the value of e^2 is even larger. This makes the model with MSE loss give more weight to outliers. Therefore, MSE is not very robust to outliers. But if the outliers in the data represent anomalies that are important and should be detected, MSE loss should be used.

With neural networks, MSE is a useful loss function because its gradient is not constant anywhere. The gradient of MSE loss is high for larger loss values and decreases as loss approaches 0, making it more precise at the end of training (see Figure 17). MSE behaves nicely even with a fixed learning rate. [17]

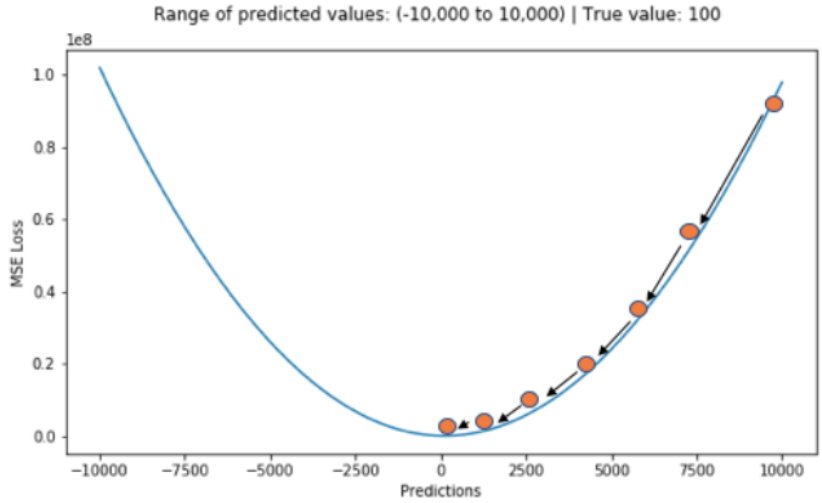

Figure 17: The gradient of MSE loss decreases as loss approaches 0. [17]

3.7 Optimisers

For training neural networks, optimisers are responsible for finding parameters θ of a loss function $J(\theta)$. The most common and established method for optimising neural network loss functions is the gradient descent. However, the need to set the learning rate ε turned out to be particularly crucial and problematic, so many improvements of gradient descent algorithm have been proposed. Setting the learning rate too high can cause the algorithm to diverge, and setting it too low makes it slow to converge. Generally, it is beneficial to decrease the learning rate gradually over time. Several algorithms have been developed to have adaptive learning rates, and they are some of the most popular optimisation algorithms used nowadays.

Adam (Adaptive Moments) is one of the most efficient and widely used optimisers. It adapts the learning rate of every weight, dividing it by the square root of the sum of their recent historical squared values. It uses *momentum*, which means that when calculating the direction to optimise the loss function, in addition to the current gradient, it also takes into account a couple of the previous gradients. For weights with high gradients, the learning rate decreases rapidly. Weights with small updates have a small decrease

of the learning rate. Adam optimiser avoids lowering the learning rates too rapidly by changing the gradient accumulation into an exponentially weighted moving average. [22]

3.8 Convolutional layers

Besides the basic type, the fully-connected linear layers which were presented in Chapter 3.4, there are also different types of layers. Convolutional neural networks (CNNs), which consist of convolutional layers, are a widely used application with many advances. They are often used for image classification, but they can be used with numeric data as well.

Fully-connected neural networks always receive a vector as input. The input vector is multiplied with a matrix of weights to produce an output. This is applicable to any type of input: an image, a sound clip or an unordered collection of numerical features – whatever the dimensionality, their representation can always be flattened into a vector before the transformation. Inputs are usually multi-dimensional arrays with one or more axis, e.g. width and height axis for an image, or time axis for a sound clip. One axis, called the channel axis, is for accessing different views of the data, e.g. the colour channels of an image: red, green and blue (RGB).

A *discrete convolution* is a linear transformation that preserves the ordering of data instead of flattening. It takes advantage of the data structure, which is very handy in some tasks. For example, if the *input feature map* is a two-dimensional data grid, the output values of a discrete convolution can be computed. The *kernel* is a grid that slides across the input feature map. In the following example figures, the kernel is a grid of value

$$\begin{pmatrix} 0 & 1 & 2 \\ 2 & 2 & 0 \\ 0 & 1 & 2 \end{pmatrix} .$$

At each location, the product between each element of the kernel and the input element it overlaps is computed. The results are summed up to obtain the output in the current location. The final outputs of this procedure are called *output feature maps*. Kernel values are often chosen so that the multi-

plication enhances the input values in some way, to highlight the input traits. Kernel values are usually set automatically by the network.

The kernel is moved across the input feature map left to right, top to bottom, one step at a time. This step, which is the amount of movement between the locations of the applied kernel, is called the *stride*. Stride is the size of the step that the kernel takes to move from a location to the next one while sliding across the input feature map. By increasing stride, the size of the resulting output feature map can be decreased. [21]

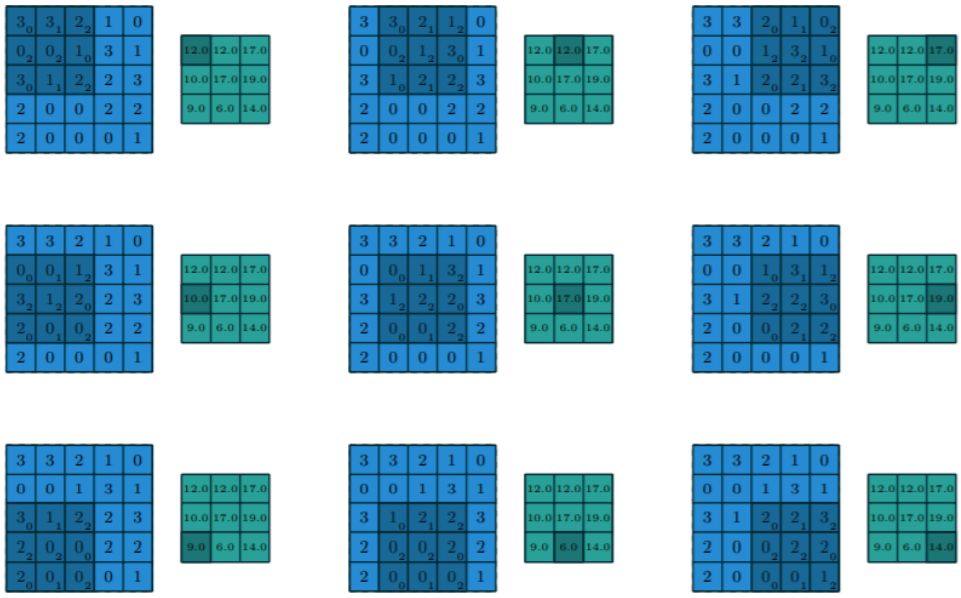

Figure 18: The blue grid is the input feature map of size (5, 5) = 5, where the kernel is the shaded area. In this case, the kernel of size (3, 3) = 3 is fit into the input feature map, and it slides across the map with stride = 1. The output feature map is the green grid of size (3, 3) = 3. The output value of the multiplication in current location is the shaded square. [20]

An alternative approach to applying a kernel to an input feature map, e.g. an image, is to ensure that each pixel in the image is given an opportunity to be at the center of the kernel. Then, a border called *padding* can be added

around the outside of the feature map. [21] The padding is the number of zeros concatenated at the beginning and at the end of the axis.

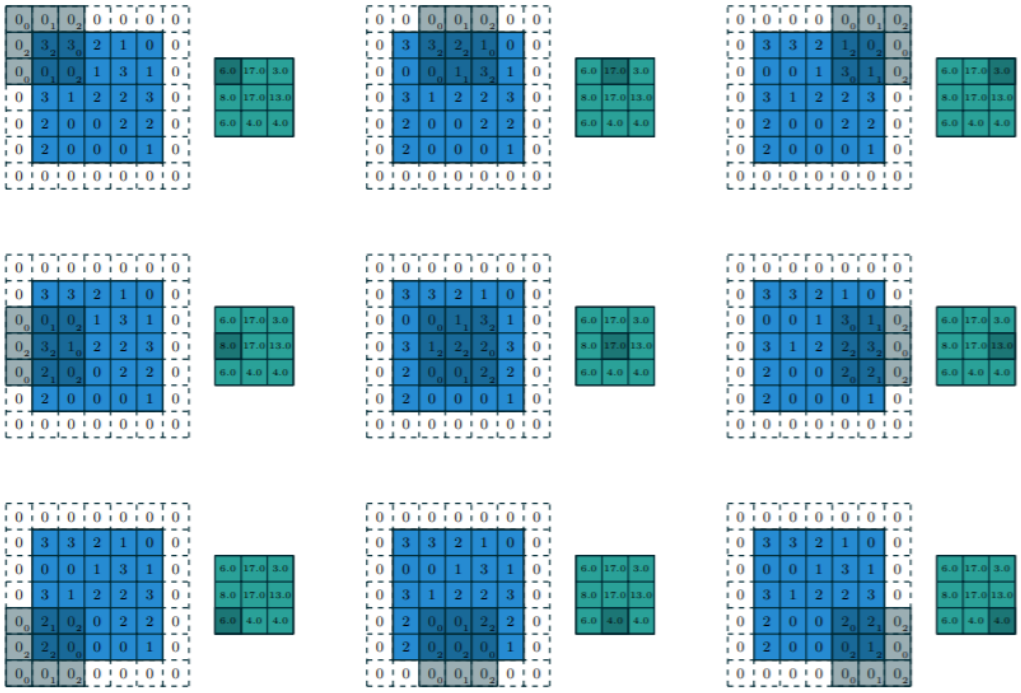

Figure 19: Convolution with input size = 5, kernel size = 3, stride = 2, padding = 1 and output size = 3. [20]

The convolution in Figure 18 and Figure 19 is an example of a two-dimensional convolution, but it can be generalised to n-dimensional convolutions. In this project, only one-dimensional convolutions are used.

This process can be repeated using kernels with different values to form multiple output feature maps, i.e. *output channels*. It is common to have multiple input feature maps stacked onto one another, like the three colour channels for an image. For each output channel, each input channel is convolved with a distinct part of the kernel. The resulting set of feature maps is summed elementwise to produce the corresponding output feature map. The output of the convolution is the set of output feature maps, one for each output channel. [20]

The output shape of a convolutional layer is affected by the input shape,

as well as the choice of kernel shape, padding and stride. The relationship between these properties is not trivial. This is a contrast to fully-connected layers whose output size does not depend on the input size.

A convolution with input size i, kernel size k, stride s and padding p has an output size of

$$o = \left\lfloor \frac{i + 2p - k}{s} \right\rfloor + 1 \ .$$

3.9 Dropout

Dropout is a regularisation technique that randomly zeroes some of the elements of the input feature map with probability p, using samples from a Bernoulli distribution. Each channel will be zeroed out independently on every forward call. This has proven to be an effective technique for regularisation and preventing the co-adaptation of nodes. [23]

Dropout is used to reduce overfitting and improve generalisation. *Overfitting* means that the model learns the training data too well; it learns the detail and noise in the training data to the extent that it negatively impacts the model's performance on new data. Basically, the model over-learns the training data and is not able to generalise to new, unseen data. *Underfitting* refers to a model that can neither model the training data nor generalise to new data. It tries to learn the data way too generally, and thus has poor performance even on the training data. Overfitting and underfitting are the two biggest causes for poor performance in machine learning algorithms. Overfitting is probably the most common problem, and there are multiple techniques to reduce overfitting in machine learning models. However, it is generally a good thing if the model is able to overfit a little bit – it means the model is able to learn the behaviour of the training data. [24]

Dropout can be used with most layer types, but it is commonly used with fully-connected layers because fully-connected layers have the most parameters and tend to cause overfitting. Probabilistically dropping out nodes in the network is a simple method for this. The probability p of dropout can be chosen by testing different values, depending on how much the network is

overfitting. The usual dropout probability for hidden layers is from $p = 0.2$ to $p = 0.5$. [27]

3.10 Hyperparameter optimisation

The network parameters which the network cannot learn itself and which have to be chosen manually are called *hyperparameters*. Hyperparameters are set before training. These are for example the learning rate, number of layers, number of nodes in each layer, batch size, dropout probability and number of epochs.

The learning rate ε defines how quickly the network updates its parameters. It is said to be the most important hyperparameter to tune. Low learning rate slows down the learning process but converges smoothly. Larger learning rate speeds up the learning but may not converge. A good starting point is usually $\varepsilon = 0.01$.

The number of layers in the network is usually chosen by experimenting. A single-layer network can only be used to represent linearly separable functions, meaning very simple problems. Most problems that need solving are not linearly separable. Neural network with one hidden layer can approximate almost any required function. However, the learning can be more efficient with multiple hidden layers. Usually, deep neural networks also perform better. [30]

The number of input nodes equals the number of input features, and the number of output nodes equals the wanted output size. The number of nodes in a hidden layer is usually chosen by experimenting. Large number of nodes in a layer, used with regularisation techniques such as dropout, can increase the accuracy of the network. Small number of nodes may cause underfitting.

The batch size is the number of data samples that are fed into the network at a time, after which the updating of parameters happens. It is commonly found that the larger batch size, the better performance, because the network will generalise better when learning with more data. However, increasing batch size slows down convergence. Due to memory limits, the batch size

cannot be too large. [22] A good default for batch size is usually a power of two, such as 16, 32, 64, 128 and so on.

For the dropout probability, a value from $p = 0.2$ to $p = 0.5$ is generally used. Too low probability has minimal effect, and too high probability results in under-learning. When dropout is used on a larger network, it is likely that the performance gets better.

The number of epochs is the number of training loops: the number of times the whole training data is gone through with the network. The number of epochs needs to be increased until the validation loss stops decreasing (or starts increasing again, in which case the network is overfitting). [29]

The default configurations of the hyperparameters work well on most problems. However, hyperparameter optimisation (hyperparameter tuning) is required in order to get the most out of a given model. Hyperparameter optimisation can be problematic for many reasons: many hyperparameters interact with each other, the computational resources are limited, and the model can easily start overfitting. Hyperparameter selection is both an optimisation and generalisation problem: the solution needs to have good performance but without overfitting. Usually, the most reliable way to choose the hyperparameters is by systematic experimentation, but it always depends on the problem and the data. [31]

4 Thesis project

In this thesis project, the neural network was built by coding it with Pytorch library in Python. The network was trained with PRACH channel data from Nokia. It was used to predict the I/Q data signal in certain user-defined situations. A good source of learning about coding neural networks and using Pytorch is a Udacity online course called "Intro to Deep Learning with Pytorch". [19]

4.1 Designing the network

When designing a neural network, the first step is to get the data ready. In most cases, data needs pre-processing to fit the problem in hand (for data pre-processing, see Chapter 2.3).

Next, the network architecture needs to be defined. Depending on the problem and the size of the data, the structure of the network needs to be chosen: number of layers, layer types, sizes of these layers and the activation functions to use between the layers. The loss function needs to be defined to calculate the losses. The optimiser type that performs the weight optimisation needs to be chosen, and the learning rate parameter needs to be specified. The batch size needs to be chosen, as well as the number of training epochs.

Then, the network model can be created with the chosen settings. After creating the suitable initial model, training can start. The network is trained with the training data set. The training is done with the optimiser which does the weight optimisation with back-propagation. The training data is gone through multiple times, and the training data is shuffled each epoch. After each training epoch, the prediction ability of the network can be checked with the validation data set. It is helpful to keep track of the training and validation losses which measure the difference between the prediction and the target. The losses should be decreasing, meaning that the network is learning to be better in predicting with each epoch.

After suitable number of epochs, the training is stopped. The network

model has now updated weights and should provide a good enough prediction. The prediction ability of the network can be tested with the testing data set. If the network works well enough and the loss is small enough for the requirements, the network can be used for prediction.

4.2 Pre-processing

All the data was generated with the simulator that was presented in Chapter 1.1.5. Some of the input parameter values were randomised in the simulation script, so the input parameter sets were all unique. After pre-processing and standardising the data as described in Chapter 2.3, the data was ready to be fed in the network to start training. One sample of data had an input of size (1×9) as there were 9 input features, and an output of size (2×15408) as this was the I/Q data with two parts I and Q, both of length 15408.

The neural network model "flattens" all matrix format data into vector format if fully-connected layers are used at any point. The output data of size (2×15408) would be "flattened" into a vector of length 30816, twice the length of one part, with first 15408 elements of I part and then 15408 elements of Q part. Using this long vector as the output almost doubles the complexity of the network. Training a network of this size is not possible with the resources available for this project because it takes too much memory. It would also take a huge amount of computing power and time.

Therefore, the I and Q parts of I/Q data signal were separated into two cases, and two separate networks were built: one for I part prediction and one for Q part prediction. The I model is trained with the I part data, and the Q model with the Q part data. The I model learns to predict the I part from the given inputs, and the Q model predicts the Q part from the same inputs. When a prediction of I/Q data is wanted from a set of given inputs, both networks are used: I model to predict I part, and Q model to predict Q part. In the end, the I and Q parts are combined back together to form the complete I/Q data.

Separating I and Q parts means that there might be some lost information

about the relationship between I and Q parts. Separating the I/Q data is normally not recommended. In this project, the output I/Q data is only used for comparison in the loss function at the end of the network. Based on that comparison, the network weights are updated again and again. The network tries to find any correlations between the input features and the output signal.

The case might be different if the output signal was not flattened into one long vector where the I and Q parts follow each other. In the final linear layer, all output nodes get their own respective sets of weights that determine the output value in that node. Therefore, all the values in the output signal get a prediction that does not depend on other output values. This means that even if the I and Q parts were left together and the network could be trained to predict this flattened output vector of length 30816, the network would not look at any connections between I and Q part values. But, as concluded before, predicting an output that long is too much for the memory capacity anyway.

This is the reason why the I and Q parts were chosen to be separated. There is at least one benefit: the models are smaller and less complex, and the memory capacity is enough to handle the models. However, it is not certain that no information is lost when the prediction is done with separate models for I and Q parts. There might also be differences in the predicting abilities of two models: for some cases, the I model might be better at predicting the real I part than the Q model for the Q part, or vice versa. This is why there was no need to calculate any testing loss value for the whole model by somehow combining the I model loss and Q model loss.

4.3 Network architecture

The neural network was chosen to have 6 layers: 4 convolutional layers and 2 fully-connected linear layers. Dropout technique was used before each linear layer. This is a 6-layer network; the input layer is not counted as it is just the input data.

When the input is fed into the network, it goes first through the four convolutional layers (for convolutional layers, see Chapter 3.8). They increase the size of the input by adding more channels. Basically, the input is a vector (or a column) of length 9, and each convolutional layer adds more channels (or more columns) to this input.

After four convolutional layers, the obtained data matrix is flattened into one vector. After the last convolutional layer, the data matrix (output of the convolution) has 2048 channels. The size of this flattened vector is $2048 \cdot 9 = 18432$. With the fully-connected layers, the size is decreased to 15408. This is the size of the wanted output, I or Q part of the I/Q signal. Before each fully-connected layer, dropout of probability $p = 0.2$ is used to reduce overfitting (see Chapter 3.9). The dropout probability was chosen with trial-and-error approach, after experimenting with several values.

The structure of this neural network is demonstrated in Figure 20.

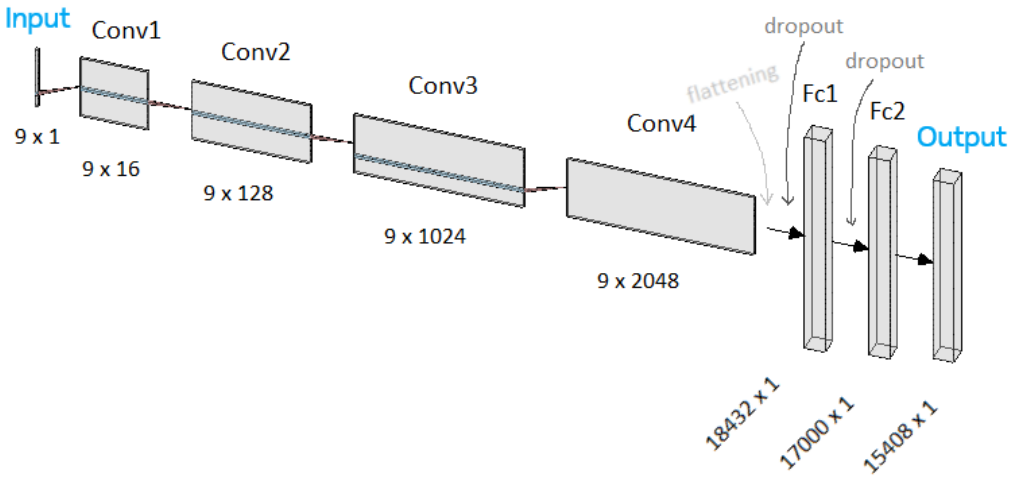

Figure 20: The architecture of the neural network that was built for the project. The diagram is made with the NN-SVG tool by Alex Lenail. [25]

Convolutional layers are used in this network because they produced the best result out of all tested model structures. With convolutional layers, the size of input can be increased with less computing than with only linear layers. Therefore, it is also faster to train a convolutional network. However,

the final layer is required to be a linear layer to achieve better results. This is the part where the network takes most time in training because in linear layers, all nodes get their own respective weights, and the size at the end of the network is so large.

ReLU is used as an activation function for all the layers except the output layer. The output layer has no activation function because the research problem is a regression problem (see Chapter 3.3).

For training the network, Adam (see Chapter 3.7) is used as an optimiser to update the weights each loop with a learning rate $\varepsilon = 0.001$. The learning rate was chosen by testing some negative powers of 10, like 10^{-1}, 10^{-2}, 10^{-3} and so on. The loss function that calculates the error between target output and predicted output is MSE loss (see Chapter 3.6).

For the model, a batch size of 128 is used for training and validation. Out of the tested batch sizes, 128 seemed to work best without being too slow to converge. Batch size has more importance on training part, because it is the number of samples fed into the network after which the weights are updated. For validation and testing, no weight updating is done. For them, batch size is just the number of samples for which the trained network does the predictions and calculates the average loss. All the batch losses are averaged into one loss for the epoch. For testing, the used batch size is 16, but other batch sizes were also used during all the testing.

4.4 Training the model

After building the two networks for I and Q parts and separating the preprocessed data into three sets, the training was started. The models were trained with the training data sets for 200 epochs. With each epoch, the training and validation losses were printed out so that it could be monitored that the losses were decreasing as expected. The training loss is the average loss over all training data, and the validation loss is the average loss over all validation data. After 200 epochs, it seemed that the validation loss was not decreasing anymore, so the training was stopped.

Finally, multiple tests were done with the test data sets in order to see the real prediction ability of the models. The same input samples and the corresponding target output (I part vector for I part model, and Q part vector for Q part model) were fed in the models. As a result, the predicted I and Q parts were obtained from the two models. The test loss values between the target and predicted output were also received. The I and Q parts were combined back to form the whole I/Q data. The target and predicted I/Q data sets were plotted in x and y axis to see the differences.

The structure and the used hyperparameters of the network were chosen mostly by experimenting with different options and choosing the combination with the best performance. The most common options for starting points were used for the experimenting. However, it was not possible to test all possible combinations, due to the time limit. Training one network for 200 epochs took around three days, so it was very slow to change just one detail and run the training again, to see if the performance improved.

5 Results

Two networks were trained for this project: one for the I part of the I/Q data, and one for the Q part of the I/Q data. The calculated losses for the models, training loss and validation loss, were of similar extent. Training loss is the average loss function value over the whole training data set, and validation loss is the average loss function value over the whole validation data set.

Training loss is not very significant when deciding whether the model is good or not, because the model has seen and learned the training data. What we want to know is whether the model is able to estimate the output of new, unseen data. This is what validation loss tells us. Therefore, the ideal validation loss value would be as small as possible. It defines how well the model has learned the general behaviour and features of the data.

Both models were trained for 200 epochs, and both training and validation loss were calculated for each epoch. The losses were not decreasing evenly and had some fluctuation at some points, so the model parameters were saved only when the validation loss was smaller than the minimum of all previous validation losses.

For the I model, the last save and the final model was from the 195th epoch. For the Q model, the last save was from the 193th epoch. The losses can be seen in Table 1. The evolution of training and validation losses for the models during these 200 epochs can be seen in Appendix: for I model, in Figure 21, and for Q model, in Figure 22.

After the training was finished, the final model and its ability to predict on unseen data (in this case, the test data) were tested. There were 720 samples in the testing data, and the batch size was chosen to be 16, so the number of batches was $720 \div 16 = 45$. The test loss (average loss over all batches in testing data) for both models are shown in Table 1.

Plotting the predictions and targets in a figure makes it easier to see whether the prediction is completely off or close to the target. Examples of predictions and corresponding targets are illustrated in the figures at the end of the thesis, in Appendix.

Model	Training loss	Validation loss	Test loss
I	0.0658	0.3844	0.4751
Q	0.0606	0.3745	0.4736

Table 1: Losses for I and Q models.

Figure 23 and Figure 24 show quite good predictions and their targets of I part data. For Q part data, these can be seen in Figure 25 and Figure 26. The whole I/Q data prediction and target are plotted in Figures 27 and 28. The model estimates the target signal quite well and captures the correct scale. However, the Q model prediction is slightly worse and does not reach the maximum values of the target. This can be seen in the plot of the combined I/Q data: the shape is not completely a circle but more of an ellipsoid.

Figures 29, 30, 31 and 32 show a prediction of a sample with relatively low I/Q data values. The integer format of the signal can clearly be seen in the whole I/Q signal plots because the scale is so small.

Sometimes the predictions are not good at all. One example of this is presented in Figures 33, 34, 35 and 36. In this example, the model is not able to predict the correct scale of the target signal. The prediction is nowhere close to the target in the plot. In the target signal, the preamble is clearly distinct from the rest of the signal, but it is not visible in the prediction signal.

5.1 Analysis

The large scale of data might be a problem for the network. If the data is not standardised, the model works even worse. However, the standardisation is not working as well as expected because the distribution of the data is wide and uneven, especially for the output.

Some of the samples are hard for the network to learn. Sometimes the network learns to predict the signal to the correct scale, or at least very close.

But for some signals, the scale of the prediction is way too low compared to the scale of the target signal. This makes the model's predictions quite unreliable.

As explained earlier, the model uses the input parameters to predict the output signal. Some of the parameters might have more influence on the output than others. If some input parameters have nothing to do with the output signal, the model might still force some false correlation between them which could cause problems. Therefore, it is important for the input parameters to actually have some correlation with the output.

Finally, the predictions can be checked with the simulator that was used to generate the data. The predicted signal can be fed in the simulator, and the simulator will check whether the preamble can be detected. All the samples were generated with single user, so only one preamble is sent with each signal. The simulator should be able to detect the preamble from the signal.

As noticed in the result figures in Appendix, the preamble is not always visible in the signal if the I/Q values in preamble part are low. Therefore, the preamble cannot always be detected, not even for the target signals in the training data that are taken straight from the simulations. Sometimes there is *misdetection*, which means that the preamble is not detected even though it should be. And sometimes there is *false detection*, which means that a preamble (or multiple preambles) is detected when there should not be anything. Even if the detections are not correct (not even for the target signal), what should matter is whether the detections are same for the target and the prediction. This is not always the case, but it is expected as the predictions were not that close to the target in many cases.

6 Future works

Having a neural network to predict the signal from given inputs is not that useful for PRACH channel use. In this thesis work, the neural network technique was applied to PRACH channel because PRACH is the simplest channel with least parameters. The target was to try if it is even possible to model data in signal format. The goal behind this thesis work was to test the usage of a neural network first with PRACH channel, to see if it might also work for more complex channels with more parameters, such as PUSCH. For PUSCH, predicting the data signal might be more useful. The PUSCH simulations usually take a longer time to run with the simulator, so using a neural network would be easier and quicker, whereas PRACH simulations are quite short and the need for quicker simulations is not as urgent as for PUSCH.

Being able to model simulator data is beneficial, but it would be even more interesting to use real-life data from the gNB, perhaps in addition to simulator data. Real data could be acquired from documented defect notifications that are obtained from real-life use cases. These notifications are usually obtained from encountering so-called *corner cases*, which are extreme cases where the behaviour is not completely reliable. This is natural because defect notifications often turn up in areas where the product is not working perfectly. Focusing more on corner cases might be a more interesting target to study. If some real life data is included in the training data, the gNB behaviour can be modelled more realistically.

When testing a technique for a problem, it is common to first apply it to the simplest case possible. Keeping this in mind, the input features were chosen and some parameter values were frozen for this project. In future, it would be more useful to use data with more than one user. For one user, there might not even be that many defect notifications or corner cases, if any, because the single-user case is very straight-forward. If the number of users is increased, the number of input parameters also increases, and the problem becomes more complex.

One idea that came across during the process of writing this thesis was to use a neural network to model PRACH preamble detection. PRACH is the channel that the UE contacts first. The UE sends a preamble which the gNB tries to detect from the signal that possibly has some noise. The gNB tries to determine if there is a UE somewhere that needs to be connected to the network. The PRACH preamble is included in the PRACH data signal, and the gNB should be able to detect the preamble from the signal. This can also be done with the simulator, so the training data could be generated with the simulator. The input could be the script parameters or the I/Q data signal, and the output would be the detection outcome (detected or not). The network could be trained to determine whether the preamble is detected or not. This would be a classification problem because the output would be either of two classes: true or false (1 or 0). Building such network would probably be simpler than the one in this project, because the output would be one binary value instead of a large integer vector.

For neural networks, hyperparameter optimisation is an area that could be fine-tuned even more. In some cases, it could take even months. There are no clear instructions on how to do hyperparameter optimisation for neural networks: testing the combinations can be done automatically, but usually it needs to be done manually, with experimenting and trial-and-error approach. One limit for hyperparameter optimisation, at least for this project, was time. If there is enough time, it could be possible to concentrate more on fine-tuning all the small parts of the network and trying all possible combinations to find the best model.

Speaking of limitations, along with time, there might be some hardware limitations. Neural networks tend to require a large amount of memory capacity and computing power, especially if the output is as large as in this project. If there is a computer with GPU (Graphics Processing Unit) available, testing different models and training them could be done much faster. However, the GPU computer needs to have enough memory to handle the model if it is very large.

Separating I and Q parts from I/Q data may cause some problems or loss of information. To keep the I/Q data whole, complex-valued neural networks could be applied to the problem. The output could be the I/Q data in complex vector format. That way, the I and Q parts could be kept together, and all information and possible correlations between them could be preserved.

7 Summary

Neural networks are a target of interest for many machine learners, especially in the field of computer vision, because of their success in image recognition tasks. However, they are continuously developed further and widely tested for many other tasks, like regression.

In this thesis work, a neural network was used for predicting the PRACH data signal from known input feature values. The output signal was very large which was one reason for choosing a neural network for prediction.

The mathematical theory shows the way of computing in neural networks. Neural computing is based on the back-propagation algorithm, which enables going "backward" through the network to calculate the gradient of the loss function with respect to the weights. The weights are updated with each training epoch. This is how the learning gradually happens in the neural network, and the predictions become better little by little.

As a result, the trained network creates the predictions of received PRACH signals. Comparing the predictions to the real targets, it is clear that the predictions are not perfect: some predictions are quite close to the target signals, but some are way off. The prediction ability of the network does not seem very reliable.

With the simulator, it can be tested whether the preamble can be detected from the predicted signals. It is not possible for all predictions: some are certainly bad predictions, but some predictions and even their targets just have too low I/Q values in preamble part so that the preamble could be detected. However, the results of this project are satisfactory enough, given the time, memory and computing power resources available.

Some ideas and alternative methods for future works came up during the process of writing this thesis. Neural networks and other machine learning algorithms could possibly be utilised in many parts of modelling 5G networks, such as PRACH preamble detection.

References

[1] A. J. Izenman. *Modern Multivariate Statistical Techniques: Regression, Classification and Manifold Learning*. Springer-Verlag, New York, 2008.

[2] J. Engelstädter. *Lecture 5: A glimpse into stochastic models*. 2019. [Online; accessed October 7, 2019]. URL: https://bookdown.org/janengelstaedter/biol3360modelling3/a-glimpse-into-stochastic-models.html

[3] National Instruments. *What is I/Q Data?* 2019. [Online; accessed October 31, 2019]. URL: http://www.ni.com/tutorial/4805/en/

[4] M. Q. Kuisma. *I/Q Data for Dummies*. 2017. [Online; accessed November 4, 2019]. URL: http://whiteboard.ping.se/SDR/IQ

[5] C. Nicholson. *A Beginner's Guide to Neural Networks and Deep Learning*. 2019. [Online; accessed October 31, 2019]. URL: https://skymind.ai/wiki/neural-network

[6] N. Donges. *Pros and Cons of Neural Networks*. 2018. [Online; accessed November 11, 2019]. URL: https://www.experfy.com/blog/pros-and-cons-of-neural-networks

[7] R. Harlalka. *Choosing the Right Machine Learning Algorithm*. 2018. [Online; accessed November 11, 2019]. URL: https://hackernoon.com/choosing-the-right-machine-learning-algorithm-68126944ce1f

[8] I. Goodfellow, Y. Bengio, A. Courville. *Deep Learning*. The MIT Press, Massachusetts, 2016.

[9] Nokia. *About us: Our history*. [Online; accessed January 30, 2020]. URL: https://www.nokia.com/about-us/what-we-do/our-history/

[10] P. Viswanathan. *How Does a Mobile Network Work?* 2019. [Online; accessed January 30, 2020]. URL: https://www.lifewire.com/how-does-a-mobile-network-work-2373338

[11] Wireless Waffle. *Downlink – Uplink*. 2018. [Online; accessed January 30, 2020]. URL: https://www.wirelesswaffle.com/index.php?d=01&m=04&y=18&category=4

[12] The EMF Explained Series. *5G Explained – What is 5G?* [Online; accessed February 3, 2020]. URL: http://www.emfexplained.info/?ID=25916

[13] N. Kumar. *Deep Learning Best Practices: Activation Functions & Weight Initialization Methods - Part 1*. 2019. [Online; accessed February 3, 2020]. URL: https://medium.com/datadriveninvestor/deep-learning-best-practices-activation-functions-weight-initialization-methods-part-1-c235ff976ed

[14] A. Lai. *Gradient Descent for Linear Regression Explained*. 2018. [Online; accessed March 9, 2020]. URL: https://blog.goodaudience.com/gradient-descent-for-linear-regression-explained-7c60bc414bdd

[15] D. Q. Nykamp. *Introduction to local extrema of functions of two variables*. [Online; accessed March 9, 2020]. URL: https://mathinsight.org/local_extrema_introduction_two_variables

[16] I. Poole. *5G Data Channels: Physical, Transport & Logical*. [Online; accessed March 10, 2020]. URL: https://www.electronics-notes.com/articles/connectivity/5g-mobile-wireless-cellular/data-channels-physical-transport-logical.php

[17] P. Grover. *5 Regression Loss Functions All Machine Learners Should Know*. 2018. [Online; accessed March 10, 2020]. URL: https://heartbeat.fritz.ai/5-regression-loss-functions-all-machine-learners-should-know-4fb140e9d4b0

[18] T. Stöttner. *Why Data Should Be Normalized Before Training a Neural Network*. 2019. [Online; accessed March 11, 2020].

URL: https://towardsdatascience.com/why-data-should-be-normalized-before-training-a-neural-network-c626b7f66c7d

[19] Udacity. *Online course: Intro to Deep Learning with Pytorch.* [Online; accessed March 15, 2020]. URL: https://www.udacity.com/course/deep-learning-pytorch--ud188

[20] V. Dumoulin, F. Visin. *A guide to convolution arithmetic for deep learning.* 2018. [Online; accessed March 16, 2020]. URL: https://arxiv.org/pdf/1603.07285.pdf

[21] J. Brownlee. *A Gentle Introduction to Padding and Stride for Convolutional Neural Networks.* 2019. [Online; accessed March 17, 2020]. URL: https://machinelearningmastery.com/padding-and-stride-for-convolutional-neural-networks/

[22] S. Lathuilière, P. Mesejo, X. Alameda-Pineda, R. Horaud. *A Comprehensive Analysis of Deep Regression.* 2019. [Online; accessed March 17, 2020]. URL: https://arxiv.org/pdf/1803.08450.pdf

[23] Pytorch. *Document: torch.nn.* [Online; accessed March 17, 2020]. URL: https://pytorch.org/docs/stable/nn.html

[24] J. Brownlee. *Overfitting and Underfitting with Machine Learning Algorithms.* 2016. [Online; accessed March 17, 2020]. URL: https://machinelearningmastery.com/overfitting-and-underfitting-with-machine-learning-algorithms/

[25] A. Lenail. *NN-SVG tool: Publication-ready NN-architecture schematics.* [Online; accessed March 27, 2020]. URL: http://alexlenail.me/NN-SVG/AlexNet.html

[26] S. Hill. *4G vs. 5G: How will the next generation improve on the last?* 2019. [Online; accessed April 4, 2020]. URL: https://www.digitaltrends.com/mobile/5g-vs-4g/

[27] J. Brownlee. *A Gentle Introduction to Dropout for Regularizing Deep Neural Networks*. 2018. [Online; accessed April 9, 2020]. URL: https://machinelearningmastery.com/dropout-for-regularizing-deep-neural-networks/

[28] S. Gharat. *What, Why and Which? Activation functions*. 2019. [Online; accessed April 14, 2020]. URL: https://medium.com/@snaily16/what-why-and-which-activation-functions-b2bf748c0441

[29] P. Radhakrishnan. *What are hyperparameters? And how to tune the hyperparameters in a deep neural network?* 2017. [Online; accessed April 14, 2020]. URL: https://towardsdatascience.com/what-are-hyperparameters-and-how-to-tune-the-hyperparameters-in-a-deep-neural-network-d0604917584a

[30] J. Brownlee. *How to Configure the Number of Layers and Nodes in a Neural Network*. 2018. [Online; accessed April 14, 2020]. URL: https://machinelearningmastery.com/how-to-configure-the-number-of-layers-and-nodes-in-a-neural-network/

[31] J. Brownlee. *Recommendations for Deep Learning Neural Network Practitioners*. 2019. [Online; accessed April 14, 2020]. URL: https://machinelearningmastery.com/recommendations-for-deep-learning-neural-network-practitioners/

Appendix

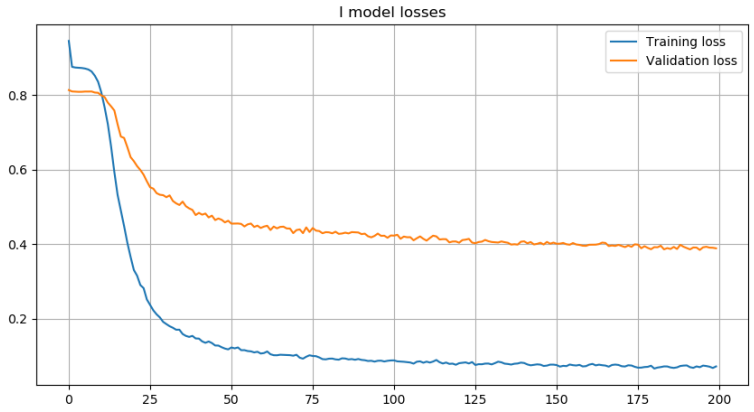

Figure 21: Training and validation losses for the I part model from 200 epochs.

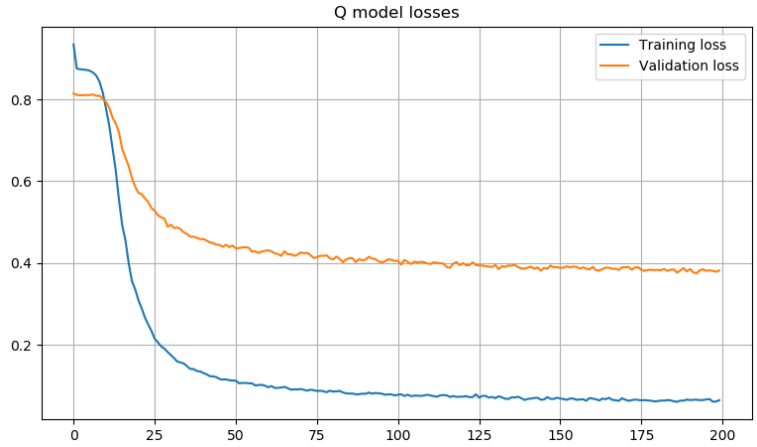

Figure 22: Training and validation losses for the Q part model from 200 epochs.

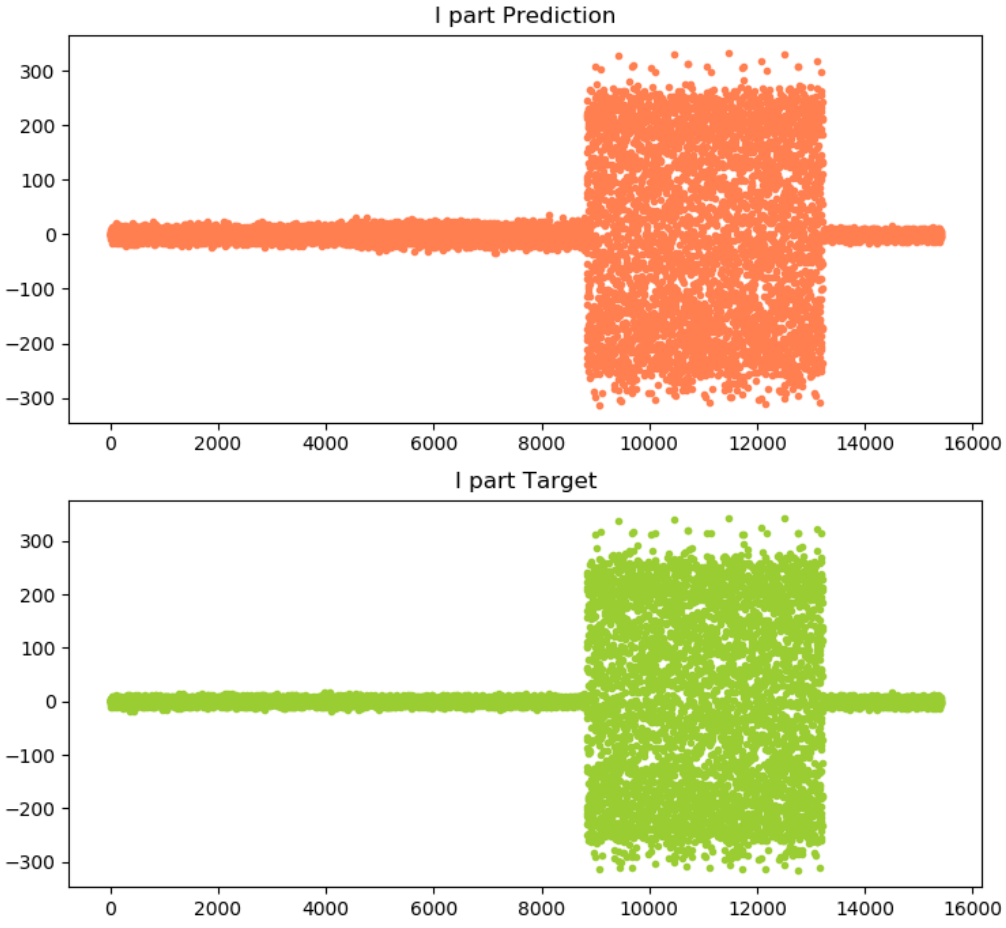

Figure 23: The I part prediction (red) and target (green) of one sample in testing data. The I/Q signal values in PRACH preamble part are relatively large, so the preamble part is clearly distinct from the rest of the signal. The preamble part can be seen near the end of the signal.

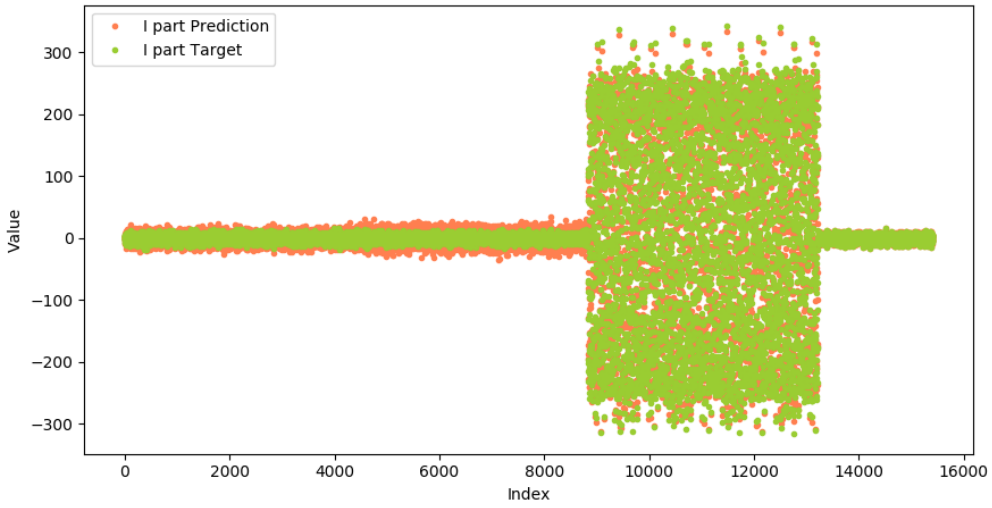

Figure 24: The same I part prediction (red) and target (green) as above, in one figure. Even though the signal values are not exactly same at every point, it is clear that at least the PRACH preamble is detected at the exact same part of the signal, at the same indices.

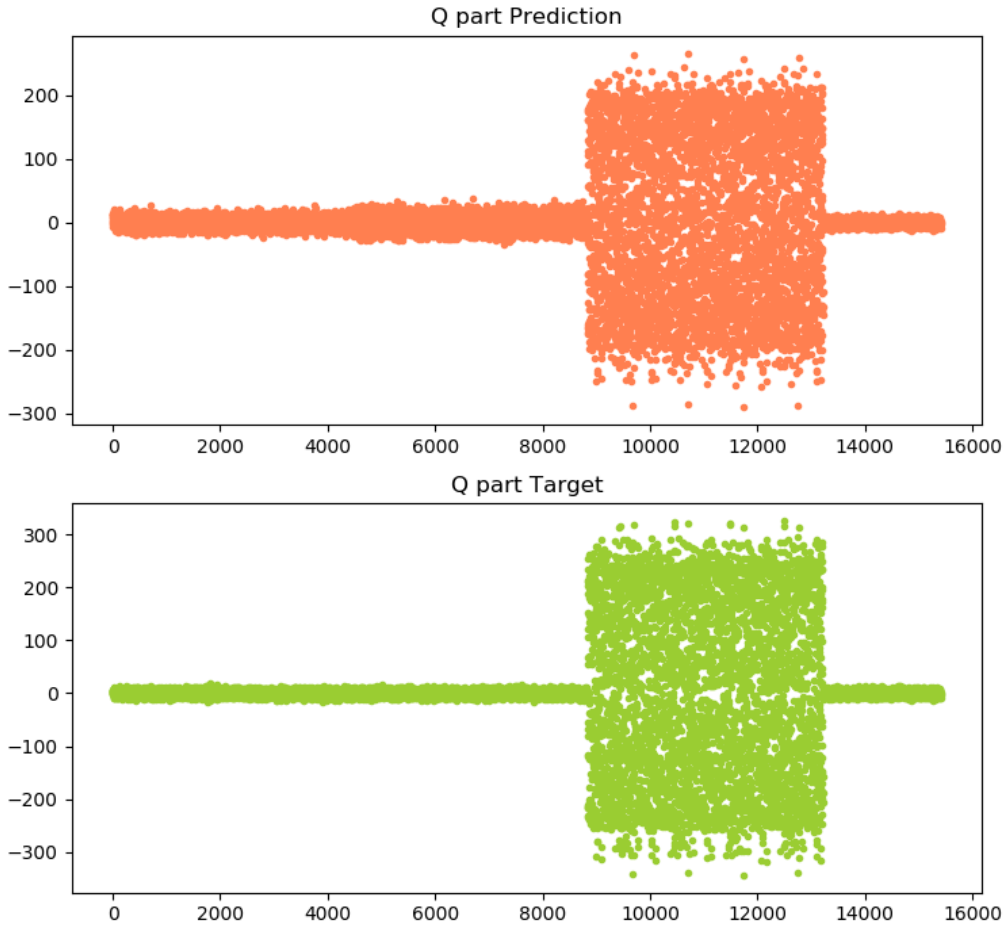

Figure 25: The Q part prediction (red) and target (green) of the same sample in testing data as above.

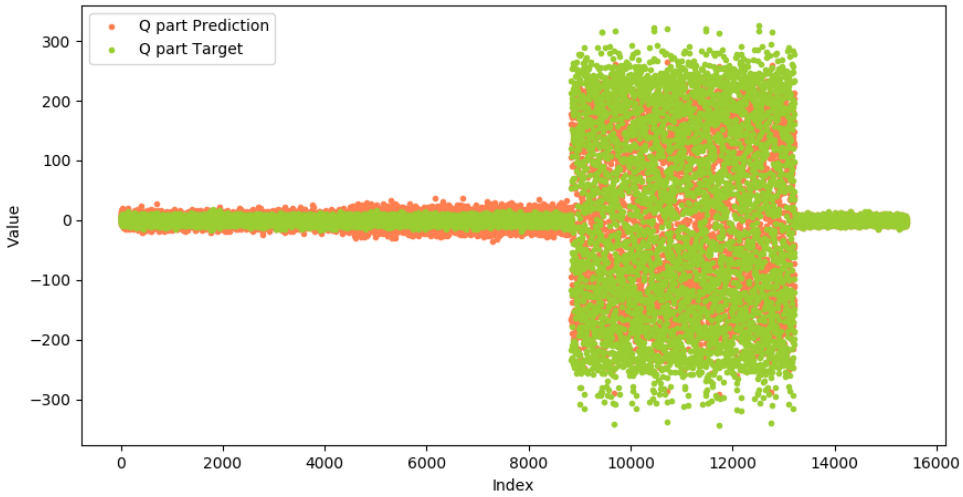

Figure 26: The same Q part prediction (red) and target (green) as above, in one figure. Compared to the I part prediction, the Q part prediction is not as accurate for this specific sample. The predicted values in the preamble part do not reach precisely the level of the target values in preamble part.

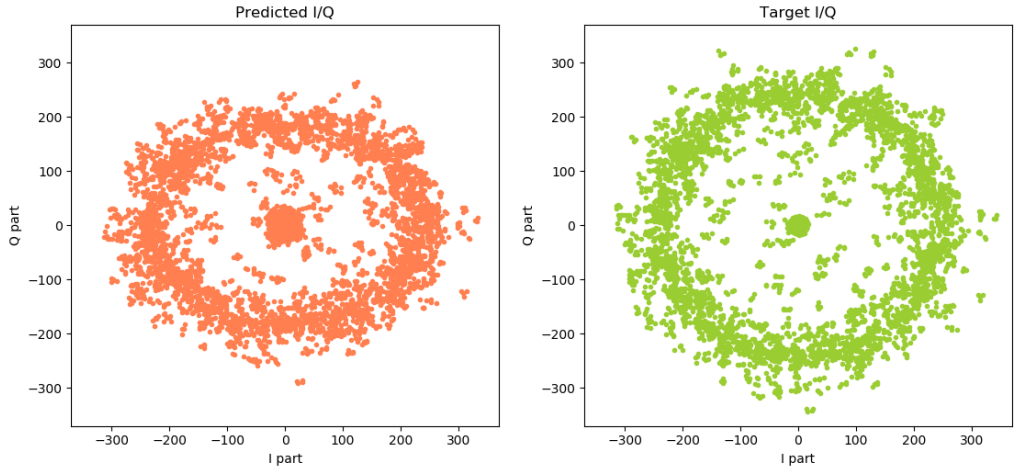

Figure 27: The I and Q predictions of the sample used above are combined to form the whole I/Q data prediction. The I part is plotted in x axis and the Q part in plotted in y axis. The red plot is the predicted I/Q data and the green plot is the target I/Q data. As seen earlier, the Q part prediction did not reach the exact target level. This can be seen in the shape of predicted I/Q data: it is not a complete circle but more oval-shaped.

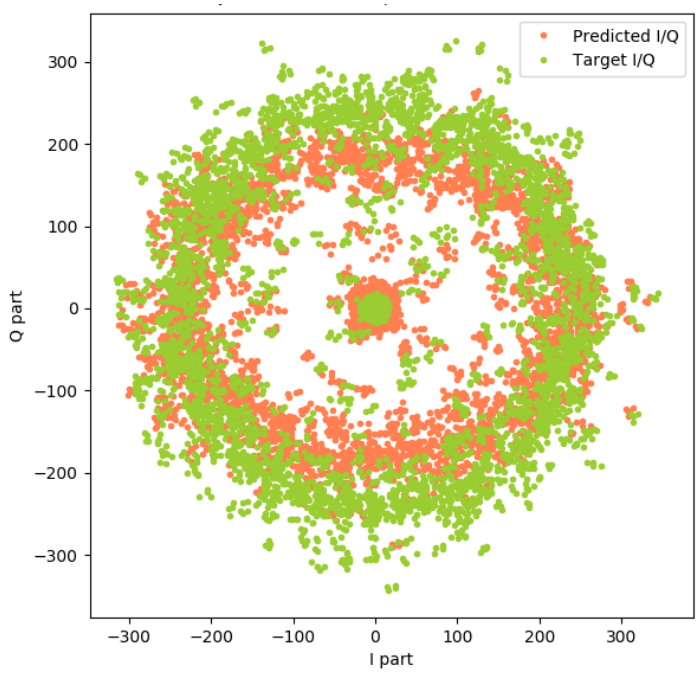

Figure 28: The same I/Q prediction and target as above, in the same figure. The red plot is the prediction and the green plot is the target.

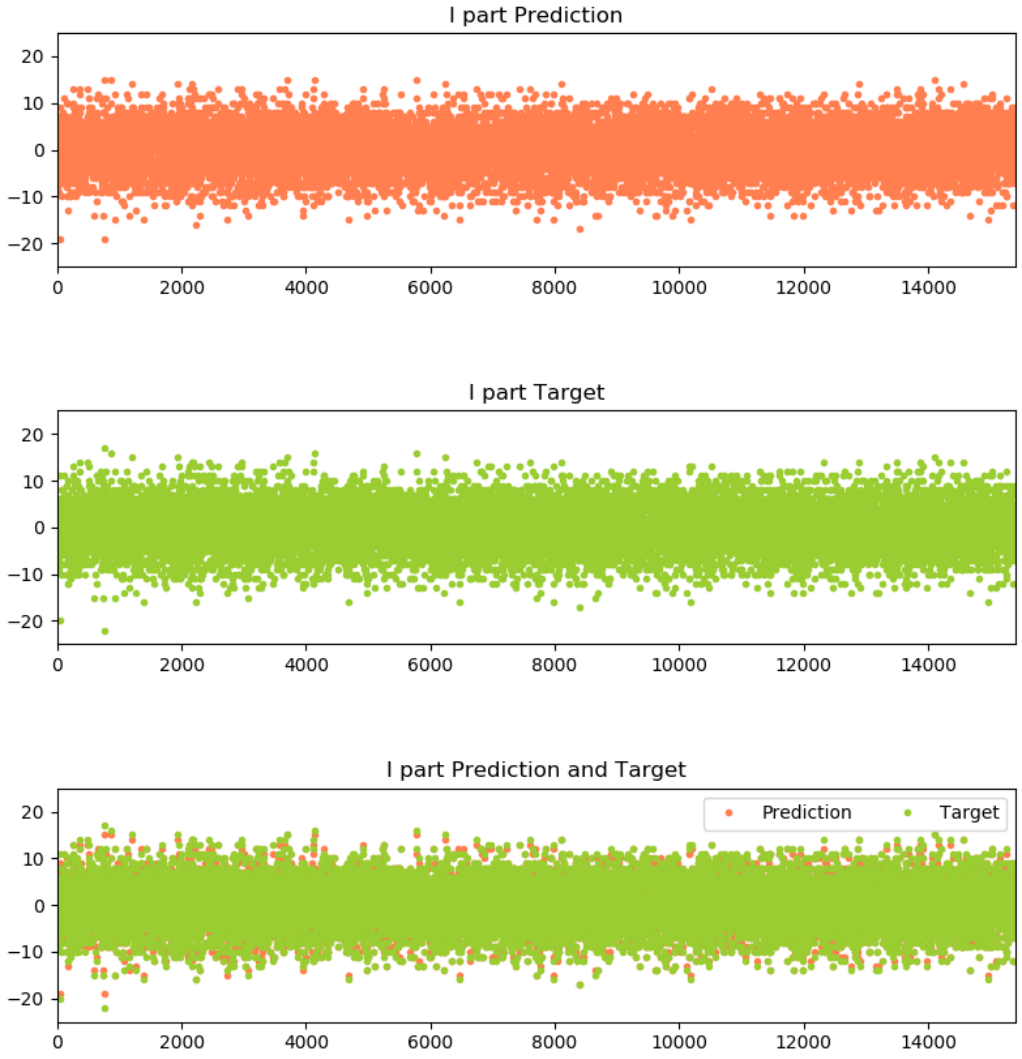

Figure 29: The I part prediction and target for another sample in testing data. The preamble should be seen at the beginning of the signal. However, the I/Q values in preamble part are very low for this sample, so the preamble part does not differ from the rest of the signal.

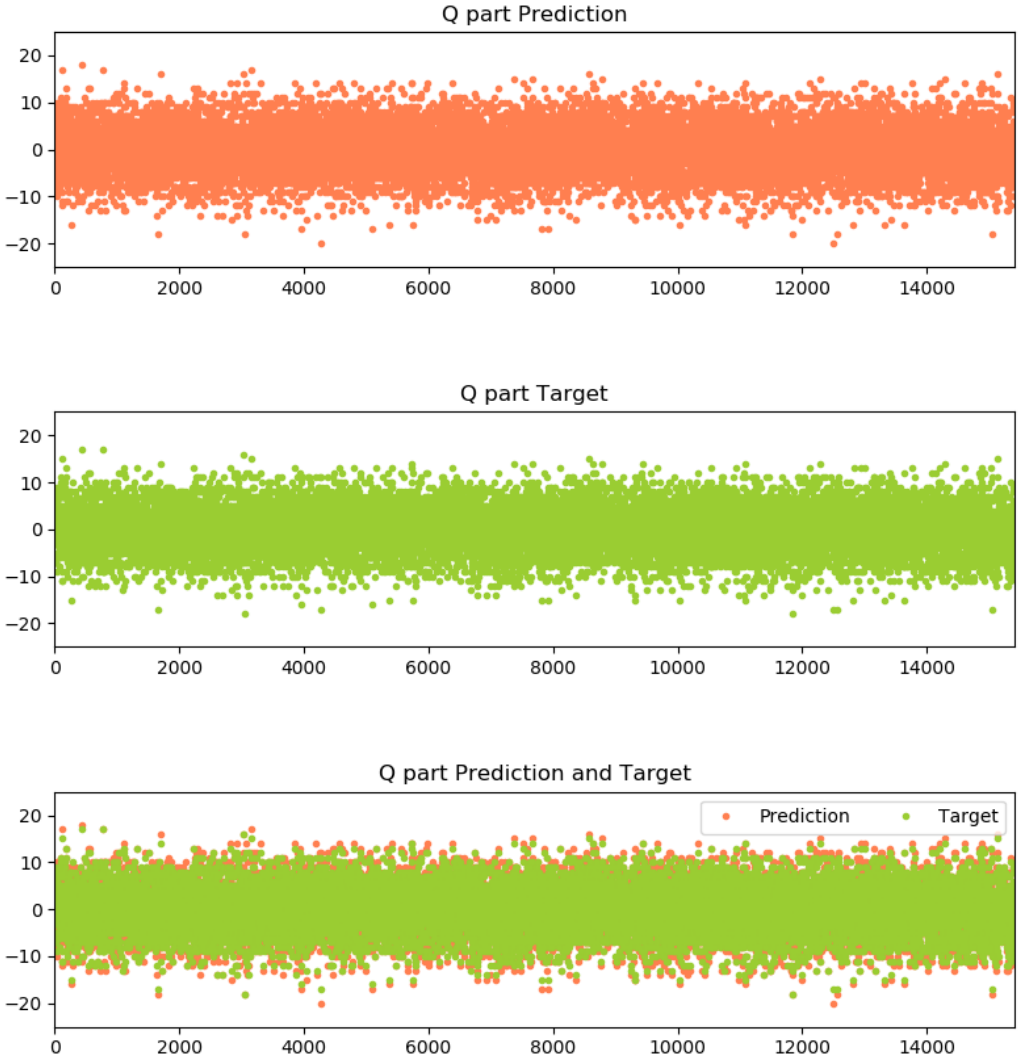

Figure 30: The Q part prediction and target for the same sample as above.

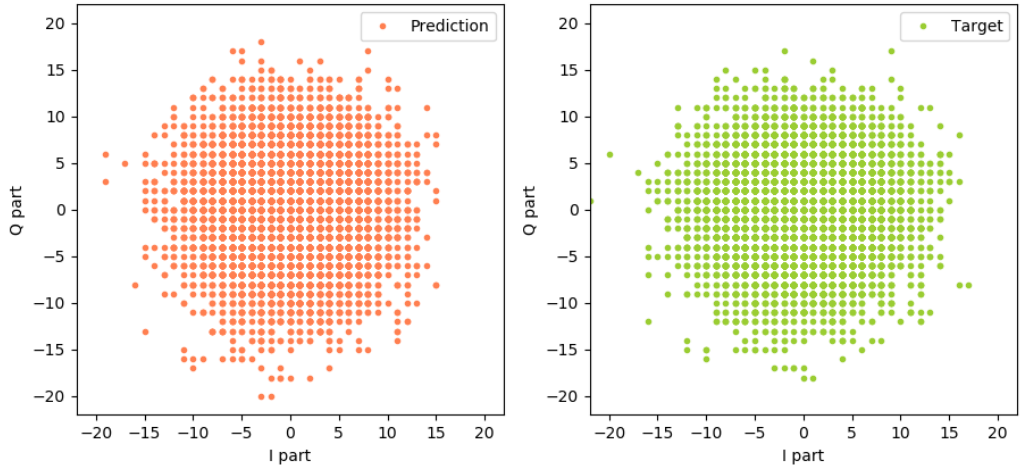

Figure 31: The whole I/Q prediction and target for the same sample as above. Clearly, the signal does not get high values. It is possible to see that all values are integers because the scale is so small.

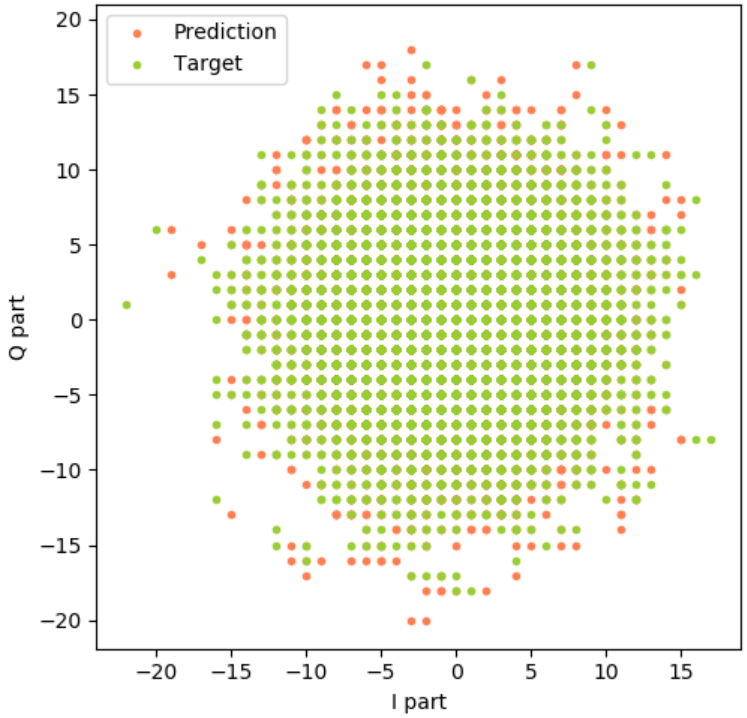

Figure 32: The whole I/Q prediction and target for the same sample as above, where the prediction and target are plotted in the same figure.

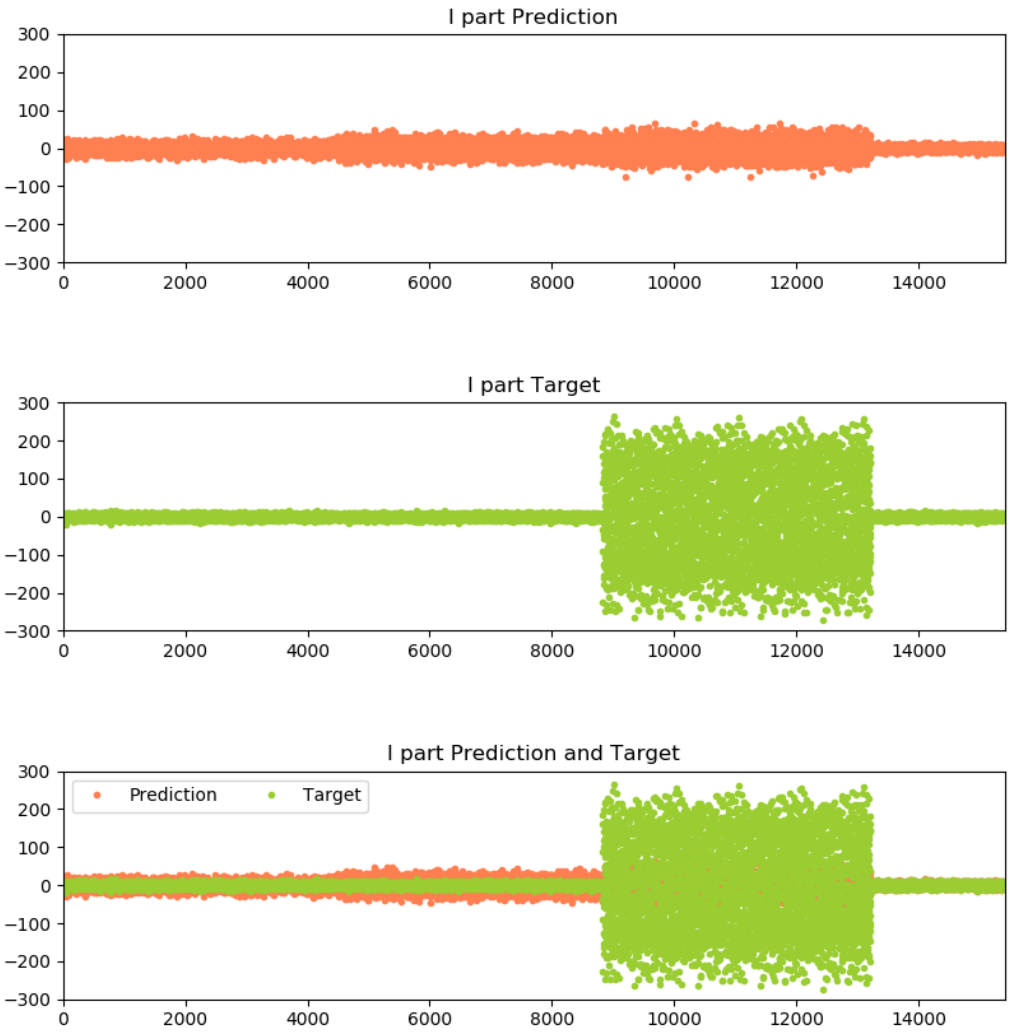

Figure 33: The I part prediction and target for another sample in testing data. In the target figure (green), the preamble part gets relatively large values and can clearly be seen at the end of the signal. It can be easily seen that the prediction (red) is not good, and the model is not able to predict the correct scale. The preamble part cannot clearly be detected from the predicted signal, but in the target signal the preamble part is very distinct.

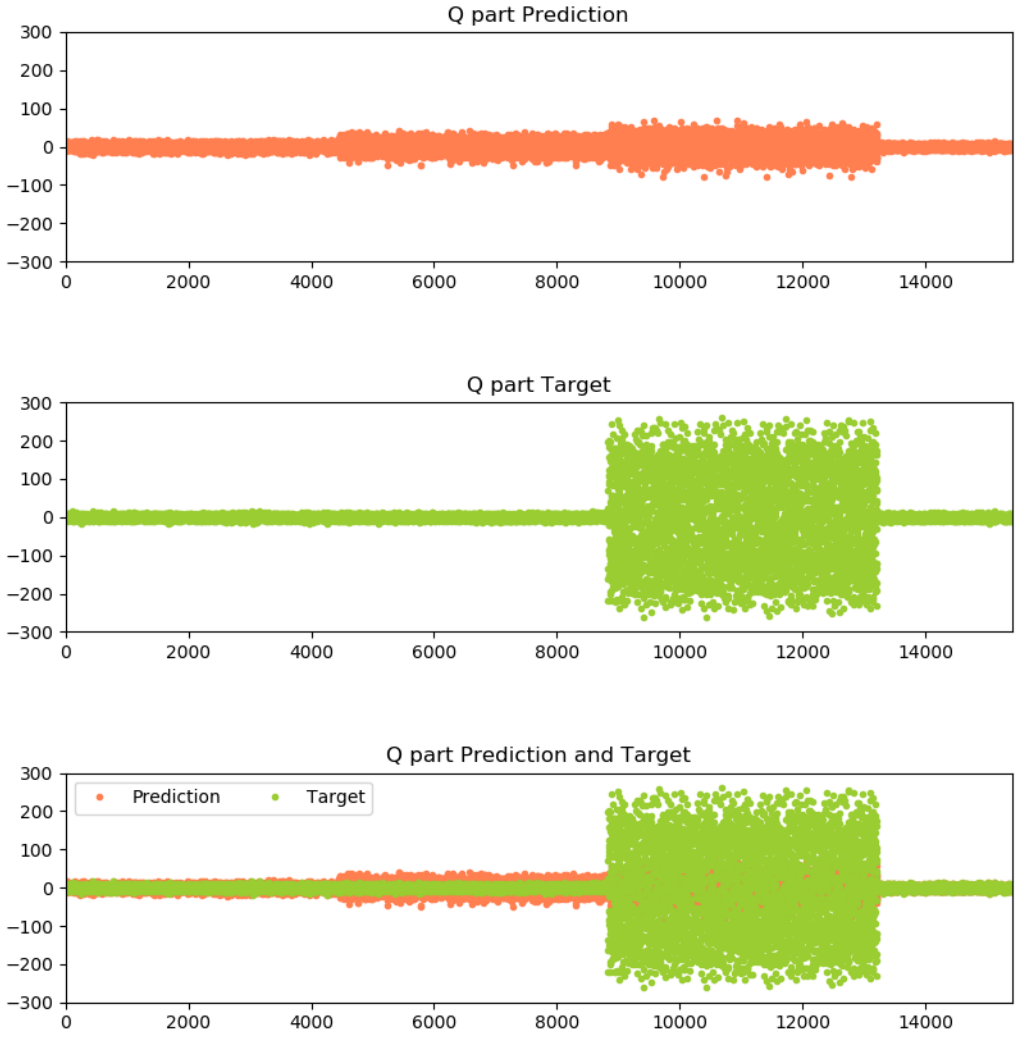

Figure 34: The Q part prediction and target for the same sample as above. The same effect can be seen as in I part prediction: the model is not able to predict the correct scale, and the preamble cannot clearly be detected from the predicted signal.

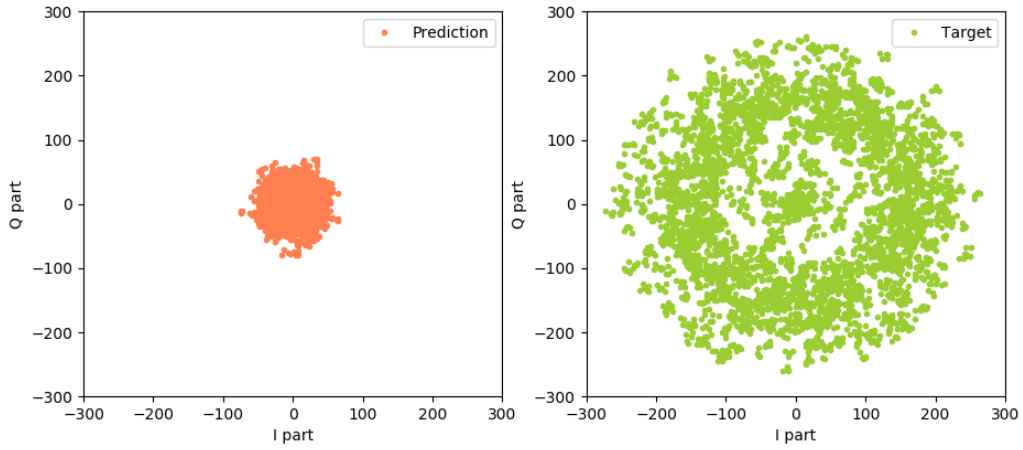

Figure 35: The whole I/Q prediction and target, combined from the I and Q parts above. It can easily be seen that the model is not able to predict the correct scale of the output for this sample.

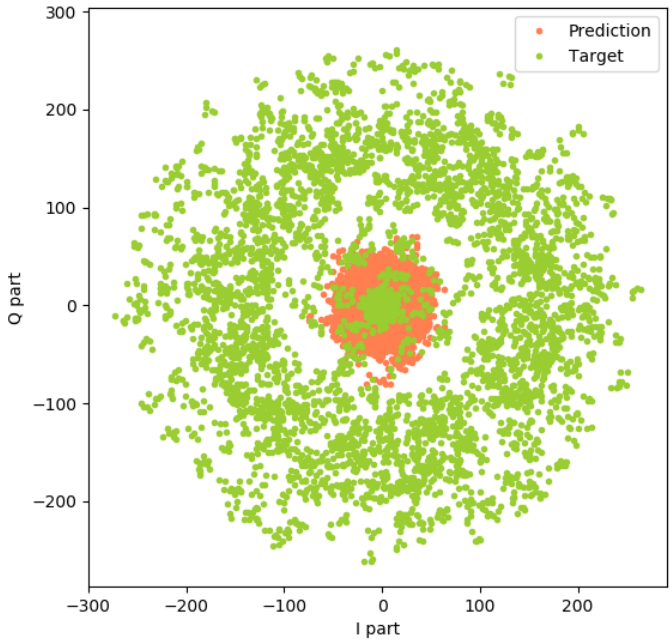

Figure 36: The whole I/Q prediction combined from the I and Q predictions above, in one figure.