Digital Twin Based on Deep Learning Neural Networks for Latency Prediction in Wireless Communication

Thesis

Adham Taha, Linette Susan Anil, Magnus Melgaard

Aalborg University
Department of Electronic Systems
Fredrik Bajers Vej 7B
DK-9220 Aalborg Ø

Department of Electronic Systems
Fredrik Bajers Vej 7B
9220 Aalborg Ø
www.es.aau.dk

Title

Digital Twin Based on
Deep Learning Neural Networks
for Latency Prediction
in Wireless Communication

Project type

Thesis

Project period

Spring 2022

Participants

Adham Taha
Linette Susan Anil
Magnus Melgaard

Supervisors

Abstract

Abstract

Number of pages: 81
Date of completion: May 9, 2023

Nomenclature

Abbreviations

AI	Artificial Intelligence
AMC	Adaptive Modulation and Coding
AWS	Amazon Web Services
DT	Digital Twin
E2E	End-to-end
IQR	Interquartile Range
LSTMs	Long Short-Term Memory Networks
MAE	Mean Average Error
MCS	Modulation and Coding Scheme
MSE	Mean Squared Error
MSS	Maximum Segment Size
MTU	Maximum Transmission Unit
NN	Neural Network
OFDM	Orthogonal Frequency Division Multiplexing
PT	Physical Twin
QAM	Aquadrature Amplitude Modulation
QoS	Quality of Service
QPSK	Quandrature Phase Shift Keying
ReLU	Recified Linear Unit
RSS	Received Signal Strength
RSSI	Received Signal Strength Indicator
VIF	Variance Inflation Factor

Preface

References follow the Harvard standard, where author name and year of creation are used. Further reference information can be found in the bibliography provided in the end. The bibliography is sorted alphabetically by author. However, if the author is unknown, the publisher is used instead. If the reference is a website, the latest visiting date is also included. An reference example is: [author's last name, year of creation]. Figures and tables are sorted according to the chapters. This means that the first figure in chapter 2 is numbered 2.1, and the next figure in the chapter is numbered 2.2. Every figure found externally will have a source reference. Figures without a reference are created by the group itself.

Numbers are written using periods as decimal separators and spaces as thousand separators.

The authors of this thesis is:

Adham Taha	Linette Susan Anil	Magnus Melgaard
<ataha18@student.aau.dk>	<lanil21@student.aau.dk>	<mmelga16@student.aau.dk>

Contents

Nomenclature iii

Preface v

Chapter 1 Introduction 1
1.1 Initial Problem Formulation . . . 3

Chapter 2 Analysis 5
2.1 State of the Art . . . 5
2.1.1 Latency and sources of latency . . . 5
2.1.2 Latency prediction . . . 6
2.1.3 Addressing the predicted latency . . . 7
2.2 Digital Twins and Physical Twins . . . 8
2.3 Machine Learning . . . 10
2.3.1 Neural Networks . . . 11
2.3.2 Summary . . . 22
2.4 Final Problem Statement . . . 22

Chapter 3 Use Case 23
3.1 Use Case . . . 23
3.1.1 Initial system proposal . . . 23
3.1.2 The Physical Twin . . . 24
3.1.3 The Digital Twin . . . 26
3.1.4 Summary . . . 29

Chapter 4 Design 31
4.1 Design . . . 31
4.1.1 Creating the Physical Twin . . . 31
4.1.2 The Digital Twin . . . 32
4.2 System Inputs . . . 34
4.2.1 Summary . . . 47
4.3 Processing the model inputs . . . 48
4.3.1 Numerical preprocessing . . . 48
4.3.2 Categorical preprocessing . . . 49
4.3.3 Training data splitting . . . 51
4.3.4 Outlier Analysis . . . 52

Chapter 5 Performance Evaluation and Optimisation 57
5.1 Model Tuning and Performance . . . 57
5.1.1 Hyperparameter tuning . . . 58
5.1.2 Validation . . . 61
5.1.3 Amount of inputs used . . . 62

5.2 Analysis of Input Data Behavior 65
5.2.1 Multicollinearity 65
5.2.2 Data sparsity 66
5.3 Evaluation of the preprocessing techniques for different models 67
5.4 Summary 69

Chapter 6 Validation and Performance Testing **71**
6.1 Test Overview 71
6.2 Model Loss Performance 72
6.3 Model Time Performance 73

Chapter 7 Discussion and Conclusion **75**
7.1 Discussion 75
7.2 Conclusion 75
7.3 Reflections 75

Appendix **77**

Bibliography **79**

1 Introduction

When interacting with applications or using digital services, it can often be observed that there is a small period of wait between the cause and the effect. This phenomenon is called *latency*, and is ever-present even when something seemingly appears to be instantaneous.

Latency manifests through the many different small delays that occur between a request, e.g., pressing a button or waiting for a video stream, and the end result. These sources of delays include the physical components, the transmission medium, routing, queuing, processing time, geographic distance, and more [Cisco, 2023]. Even using optic fiber cables induces a delay as low as 5 µs per kilometer [Coffey, 2023].

How each type of service or scenario is affected by latency is not the same either. For something like remotely opening a garage door, the delay would not be problematic, but a video call where the voice and video don't sync up would be noticeable, and in the worst case make it difficult to keep a conversation going. These different requirements for latency, often categorised as soft-, firm-, or hard realtime, can help define to what extend latency is accepted or not. A system with a soft realtime requirement may continue to work and prove useful even if some deadlines are not met, whereas hard and firm requirements will have the result of a process discarded if the latency is too large. In addition, hard realtime will result in a failure of the system as well. A summary of how the results of a system, and the system itself, are vulnerable to latency depending on the different realtime constraints, is included on figure 1.1.

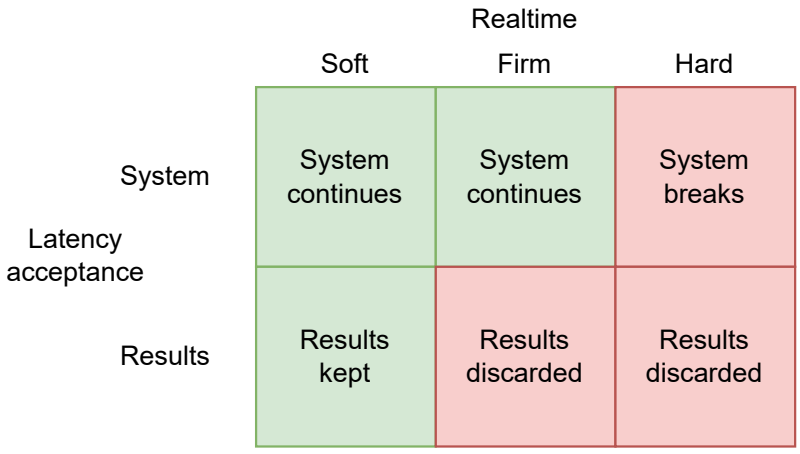

Figure 1.1. Confusion matrix showing three types of realtime constraints and how latency may affect the system and its results. The green cells show when the system or results will not be discarded by latency, and red cells when they will.

However, even if a soft realtime task is not discarded by delays, the value of the results may increase if they are timely. For example, a common soft realtime problem is delay in online games or even controller response times affecting how well the user can play the game, especially if cloud gaming is used. If for example the game servers are hosted in another part of the world, the latency or "ping" would result in noticeable competitive disadvantage, and degradation of the experience. According to data by the American internet service provider Verizon [2023], the latency for the service they provide can easily exceed 100 ms for communication across the world.

Latency may even have unpredictable patterns of rising and falling, referred to as *jitter*. Jitter can be particularly disruptive for applications like video calls and streaming, where having a stable, live experience is important.

Reducing a latency may be as simple as upgrading a CPU, lowering media quality, increasing bandwidth, and reducing the geographical distance. However, as there are many factors to what may cause a high or unpredictable latency, it may be difficult to identify the ones causing the most delay. This may be problematic, as if the source itself cannot be identified, the latency itself may not be able to be addressed, which may as a consequence be problematic for systems with firm or hard realtime requirements.

A possible method to try and isolate the delay is the use of Digital Twins, an emerging and increasingly popular technology in the machine learning fields. By running alongside a system process, the Digital Twin will simulate the same process, and thus give an insight into what the expected result should be. This can be used to figure out at which step of the process it goes wrong and unexpected latency is added. Additionally, by implementing a Neural Network (NN), a specific architecture of machine learning, the Digital Twin can be achieved, and used to also predict the expected values. By predicting the expected latency in advance, it may even be possible to address the cause of latency in time.

1.1 Initial Problem Formulation

In summary, latency is an unavoidable problem when working with either software or physical hardware, where many small contributing factors will add up. These factors can be either deterministic or random, with varying levels of impact and their combined effect can be significant on the Quality of Service (QoS) of the end-users. The idea of using NNs to attempt to predict the latency based on prior data is proposed, which requires that the latency can be isolated in the first place. This raises an initial problem formulation for this project:

How can the individual contributors of latency be identified, and how can the information be used for the purpose of predicting the end-to-end latency?

Analysis 2

The Introduction introduced the concept of latency and Neural Networks. To further understand these concepts, and later address how they may be worked upon, an Analysis chapter is included.

This chapter goes into detail about the State of the Art in latency, latency prediction, and how this latency can be addressed. In addition, the concept of Digital Twins and how a Neural Network may be used as as Digital Twin is included. This section covers how a Neural Network operates, and how a prediction is achieved.

2.1 State of the Art

With the initial problem statement in mind, the current State of the Art approaches to both predicting and handling latency will be investigated.

2.1.1 Latency and sources of latency

[Briscoe et al., 2014] describes latency as the measure of responsiveness in a system, i.e. the how instantaneous an application feels. It can be defined as the time it takes starting from a single critical task being required until the last bit of critical information is received at the destination. Latency is considered to be one of the biggest hindrance in achieving a seamless experience for many applications, and [Briscoe et al., 2014] classifies the sources for these delays experienced during a communication session into multiple categories. The three main categories are: Structural delay, Endpoint interaction delay and Transmission path delay. The nature of delays is such that it is additive over the communication session, i.e. even slight delays experienced in any of these categories per transfer adds up in the overall delay, where numerous small tasks can quickly ramp up to a lot of latency. Below, different sources of delay are explained further.

Structural delay

These delays occur due to suboptimal paths or routes in the network structure. In addition, the distance between client and server also plays an important role in the amount of latency experienced. The physical placement of components such as servers, databases, and caches with respect to client's endpoint also have an impact on the latency at the client's end.

Endpoint delay

These delays occur due to end-to-end (E2E) protocol setup. These protocols can include transport protocols that is used for various control interactions before the data is actually sent. In addition, these E2E protocols are also used during data communication in order to

recover lost packets for reliable transfer. It can also be used for optimisation, for example, to assess new paths to avoid congestion or merge packets to reduce the number of packets on the link. These protocol interactions can introduce additional latency to the system.

In addition to these delays, there are also some general delays that are encountered when transmitting a packet over a network. These delays are elaborated further below:

Transmission delay

The transmission delay is the time taken to push all the packet bits from the host onto the transmission medium or link. It depends mainly on the size of the packet and the bandwidth of the channel. This delay is also influenced by the number of devices using the link. More specifically, when a large number of users are competing for gaining the channel access and even though the transmission error is low, the latency can become unpredictable, therefore impacting the overall latency. Furthermore, the number of collisions may also increase as the number of users increase, which results in more retransmissions, thus introducing more overhead and hence additional latency.

Propagation delay

The time taken for the last bit of the packet to reach the destination after the packet is transmitted on to the medium is known as the propagation delay. Distance between the sender and receiver and the transmission speed are the main factors affecting propagation delay.

Queueing delay

Once the packet is received at the destination, the packet has to wait in a queue, also known as a buffer, for an amount of time, before the packet is processed. This type of delay depends on the size of the queue, i.e., if there are not many packets in the queue, the queueing delay will be small and vice versa. It also depends on the hardware of the queue such as which server is used, relating to how fast the queue may be emptied.

Processing delay

Processing delay is defined as the time it takes a router to process a packet header. A major factor that can affect this type of delay is fragmentation. This is when a packet needs to be split into smaller packets due to its original size being greater than the Maximum Transmission Unit (MTU) supported by the intermediate node (e.g. router). Apart from that, reassembling these fragments also consumes more processing time which is aggregated to the overall latency.

2.1.2 Latency prediction

Predicting the E2E latency can have a significant importance for many real time applications, especially for enhancing the end user's Quality of Experience. However, despite its implications, methods to predict latency in a system are still in the nascent stages and there are only a few approaches that have previously been considered to predict the latency in wireless communication.

[Khatouni et al., 2019] studies and predicts latency in an operational 4G network using a machine learning approach. This is done by exploiting a large data set with more than 200 million latency measurements taken from three different mobile operators. Next, the paper uses the dataset to characterise different features, showing the latency distributions. Then, the Random Forest algorithm is applied to select only the most important features in latency prediction among all features. Finally, it uses three different classifiers to compare which suits the dataset best and produces better results in order to predict latency from a device to a server [Khatouni et al., 2019].

Another survey [Yang et al., 2004] discusses the different kinds of latency prediction mechanisms that can be used theoretically such as queueing theory and time series analysis. However, these approaches are only model-based which provides theoretical upper-bounds, and many applications do not have enough knowledge upon which scientific models can be built and therefore, these mechanisms have their limitations. It also goes on to say that in such cases, machine learning methods such as Artificial NNs can be used for prediction and can be substituted for the other methods overcoming their limitations. However, the survey does not look further into it [Yang et al., 2004].

2.1.3 Addressing the predicted latency

Addressing the predicted latency depends on how critical it is to the considered use case. For instance, the applications that have strict requirement on latency and when the predicted latency exceeds the maximum tolerated value, a proactive action should be taken. This can be achieved by identifying the source that contributes to high latency and then reduce its impact.

Another approach of addressing the predicted latency is using latency compensation techniques, which is widely applied in the gaming industry. The paper [Liu et al., 2022] does a survey on the different latency compensation techniques used for online computer games. Latency compensation techniques refer to software algorithms that are used on either the game server or client side in order to minimize the negative impacts of network latency on the players. It classifies the effect of latency into two main categories: the time for the client to get a response from the server and the difference between game states for one or more clients. Latency concealment is one latency compensation technique surveyed in the paper which masks latency between the server and client by reducing the perception of unresponsiveness. An example of this scenario is when the player does an action which is reflected on their screen, but it has yet to cause any changes on the actual server and for other players [Liu et al., 2022].

The paper however only discusses latency and compensation techniques for video games and does not consider other use cases. Moreover, the goal of this project is to predict latency but how this predicted latency is used or handled is beyond the scope of this project.

As a summary, a few types of latency prediction were investigated. One novel approach includes training and using NNs to predict latency. This approach is further investigated, in order to later be implemented and assessed throughout the rest of the report. This predicted latency can then in theory be compensated with various methods depending on

the type of medium, which is brought up in a discussion after the implementation and testing phases. First, the concept of Digital Twins is introduced, wherein it is investigated how a NN may be used.

2.2 Digital Twins and Physical Twins

Digital Twins (DTs) can be defined as a dynamic virtual copy of a physical object, a system or, even a process within a system [Khan et al., 2022]. This technology aims to mirror and examine, in real time, a physical entity in all of its complexity and with reasonable computation time. Consequently, the given physical entity can be monitored, controlled, and hence optimised.

The idea of DTs was initially applied by NASA in 1970, namely, in the Apollo 13 mission [Liu et al., 2021]. During this mission, NASA replicated and simulated the spacecraft as well as its environment, where a number of potential issues with the mission could be identified. This aided the engineers in resolving a number of problems, including finding the most optimal procedures for getting the Apollo 13 astronauts back to Earth safely [Barricelli et al., 2019].

Since then, DTs have been evolved through various names such as mirrored spaces, virtual spaces, digital copies, and digital mirrors. In 2002, professor Michael Grieves during a presentation about "Conceptual Ideal for Product Life Management" at the University of Michigan, presented DTs for the first time as a concept [Zhang et al., 2021]. Later, [Grieves, 2014] standardised the DT system which encompasses three main parts, as illustrated in figure 2.1; the physical, the virtual, and the link parts. The physical part is denoted as the Physical Twin (PT). This part is the physical system that is under consideration, and typically, it is equipped with sensors that collect data and send it, in real-time or near real-time, to its digital counterpart. The latter is the DT which hosts and processes PT data, and performs a number of tasks e.g data visualisation, system troubleshooting, and simulations. The link part allows for bridging data between the PT and DT components, and additionally lets the output of the DT influence the states and control of the PT, based on the type of implementation.

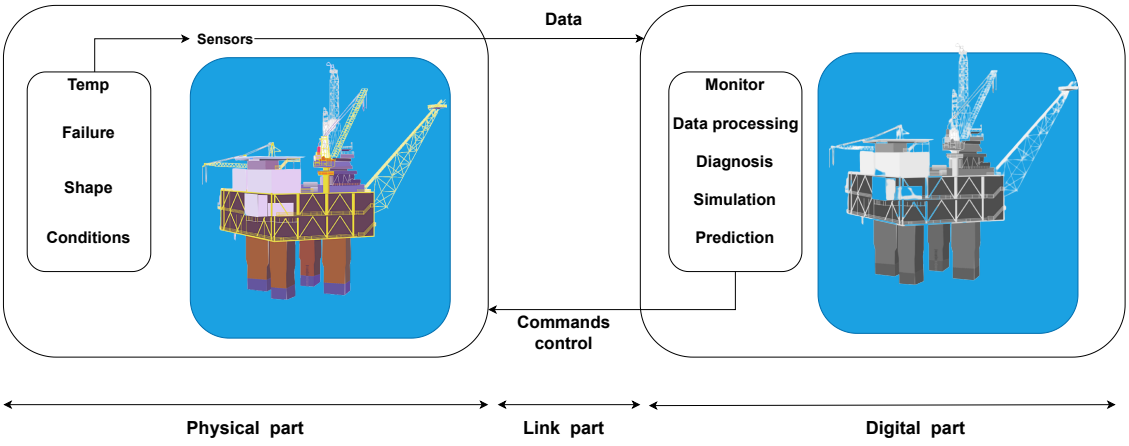

Figure 2.1. DT system parts.

DT systems come in a variety of sophistication levels [Wagg et al., 2020], after having evolved from the most basic level i.e. monitoring DT, up to autonomous DT, as illustrated in figure 2.2.

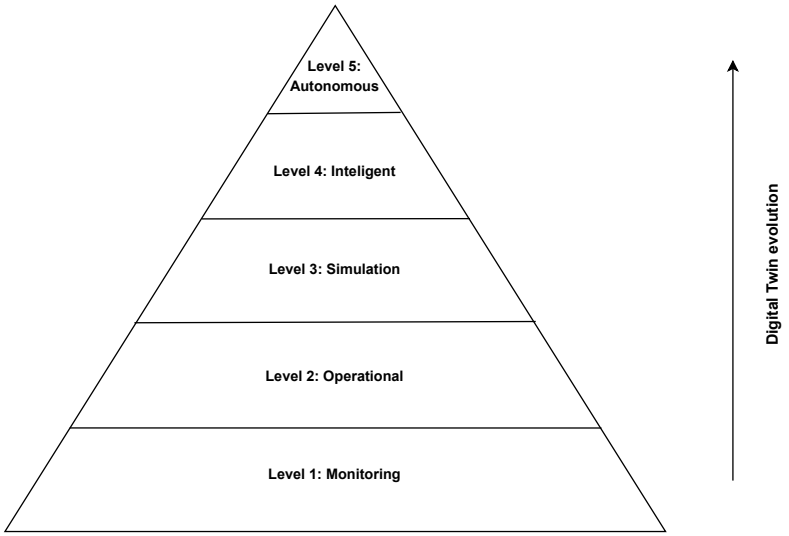

Figure 2.2. Levels of sophistication of Digital Twins.

The most basic level of sophistication of DTs is as a monitoring DT, also named as a supervisory DT. This type allows the user to monitor the condition of a PT. A slightly more advanced type of DT is an operational DT, which incorporates the feature of supporting the operational decisions at the PT based on the collected relevant information. As described by [Tuegel et al., 2011], a simulation DT, in addition to its ability to visualise the current state of a PT, can perform predictions providing the user with quantitative evaluations for the purpose of supporting the operational decisions. An increased level of support and scenario planning is achieved through an intelligent DT, which learns from the collected data through Artificial Intelligence (e.g. machine learning). The most sophisticated DT is the autonomous DT, which controls and manages the PT with low-level human intervention.

A DT can be developed in different architectures, namely as an edge-based DT, cloud-based DT or collaborative DT [Khan et al., 2022], as shown in figure 2.3.

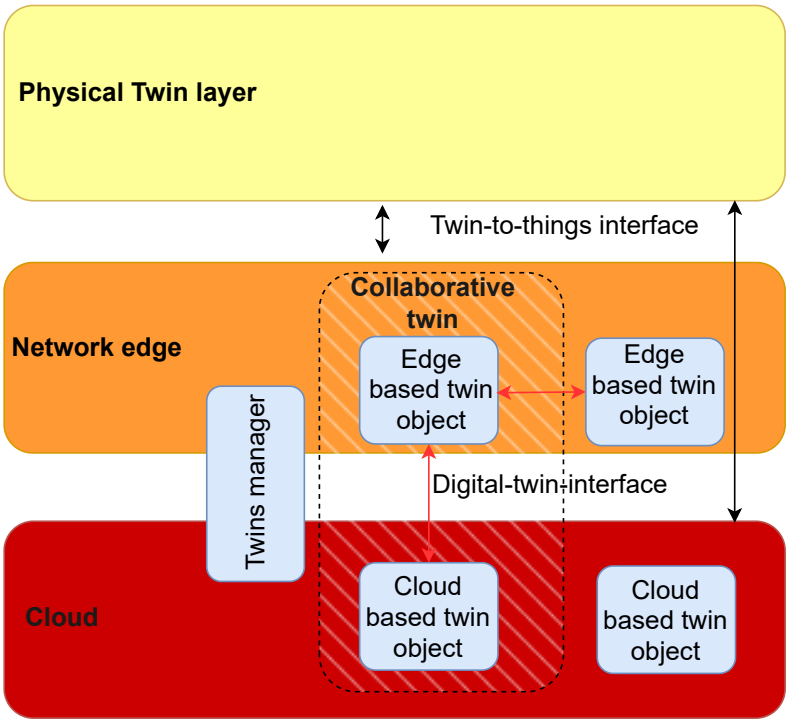

Figure 2.3. Different Digital Twin architectures.

An edge-based DT is characterised by the close proximity of the DT object to the PT object, making it suitable for applications that have strict requirement on latency. On the other hand, a cloud-based DT is more suitable for delay tolerant applications that demand high computational resources, due to the presence of higher power capabilities. In order to address the trade-off between the latency, computational power, and storage capacity, the collaborative DT can be deployed. This architecture is a distributed approach that benefits from both edge-based and cloud-based architectures. For instance, a DT that makes use of a machine learning model can be deployed as a cloud-based object in order to be trained though heavy computational processes. Thereafter, the trained model can be deployed as an edge-based DT to respond more quickly to its PT. In addition, the edge-based DT can continuously retrieve updates from the cloud-based object.

Having looked at DTs and PTs in general, it makes sense to analyse a more specific implementation of a DT, namely an intelligent DT, which first requires an introduction to machine learning.

2.3 Machine Learning

In the last few years, machine learning has been extensively researched and used in various applications including image/speech recognition, product recommendation, and fraud detection [IBM, 2023b]. Machine learning is a sub-field of Artificial Intelligence (AI) that aims to build a model that learns from data through algorithms to be able to make decisions with minimum human assistance. Primarily, machine learning can be categorised into three types: reinforcement learning, unsupervised learning, and supervised learning [IBM, 2021].

Reinforcement learning follows a feedback-based approach in which the model gets positive feedback or reward for every correct action and a penalty or negative feedback for every incorrect action. Consequently, this trains the model to maximise its score obtained from the received rewards through a process of trial and error. Applications of this type of learning can be used in autonomous cars, image processing, robotics, etc.

The unsupervised learning refers to learning from unlabeled data sets (i.e. data which is not tagged with labels or classification), where the output is unknown. It analyses hidden patterns and groupings to discover similarities in the information. An example of a use case for this type of learning is classifying students performances (pass or fail) based on their grades.

Supervised training on the other hand makes use of labeled data, meaning known inputs and their corresponding output(s) in the learning process. It allows for assessing the discrepancy between the actual and the predicted values, either for classification or regression problems. A typical example of classification is image recognition, where labeled data is used to learn the features of different objects. Regression problems on the other hand lets the model learn the relationship between variables, and could for example be used for numerical prediction i.e. revenue forecasting. Because this project revolves around generated data with known inputs and outputs, which can be interpreted as a regression problem, supervised learning is utilised.

2.3.1 Neural Networks

Neural Networks is a subset of machine learning, which consists of three different types of layers. These layers are described in the following paragraphs.

Neural Networks structure

A neural network encompasses number of interconnected nodes which are processing units that hold values. These units are classified into multiple layers, namely the input, hidden and output layers. An example neural network model can be seen in figure 2.4.

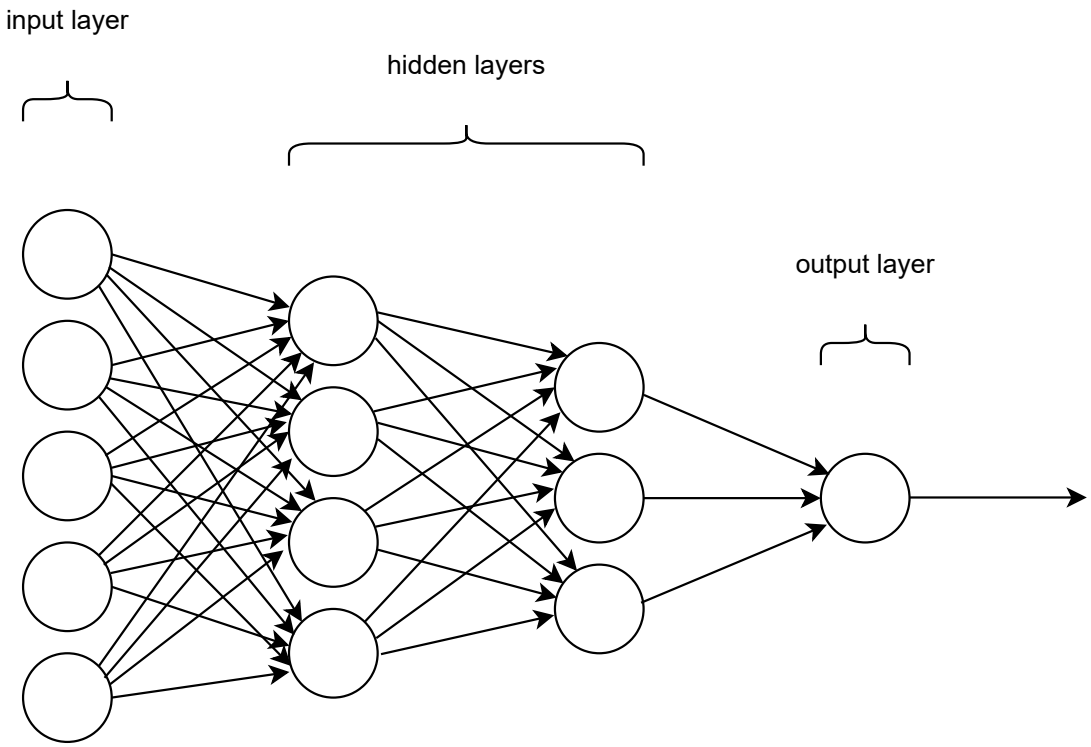

Figure 2.4. Example of a four-layer neural network with two hidden layers.

The input layer consists of nodes that feed the network with information from outside. This information is then passed to the next layer, i.e. the hidden layer(s). This means that the output of one node from the preceding layer is an input for other nodes in the next layer. The neural network may consist of single or multiple hidden layers which are located between the input and the output layers. A Neural Network with a certain level of complexity, i.e. usually a Neural Network with more than one layer is known an deep learning Neural Network. These intermediate layers are responsible for extracting different hidden features and patterns in the data. Finally, the output layer produces some predictions or classifications based on the inputs from the previous layers.

Each of these nodes are connected to the next layer or nodes via edges. These edges are assigned a certain weight w based on the feature's significance. The higher the weight, the more influence it has on the nodes in next layer. The associated weight of the edge is multiplied with the connection's input value and passed through an activation function.

An activation function decides whether a node should be activated or not, referring to whether the output of the node is used. A node that is not activated would simply give a 0. This is used to identify if a node's input to the network is significant or if it adds value to the prediction process. An activation function adds non-linearity to the network by transforming the summed weighted inputs to the node into an output value. There are different types of activation functions. Two main examples are the sigmoid and the Rectified Linear Unit (ReLU) functions.

Sigmoid

A sigmoid is a type of non-linear activation function that takes any real value as input and provides an output which is between the range of 0 to 1. Larger values will be closer to 1, whereas smaller values or negative values would be closer to 0. The sigmoid function is suitable for binary classification where the probability of a binary variable can be calculated using a sigmoid function. The function is represented in equation (2.3.1) along with its Probability Density Function (PDF) in figure 2.5.

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.3.1}$$

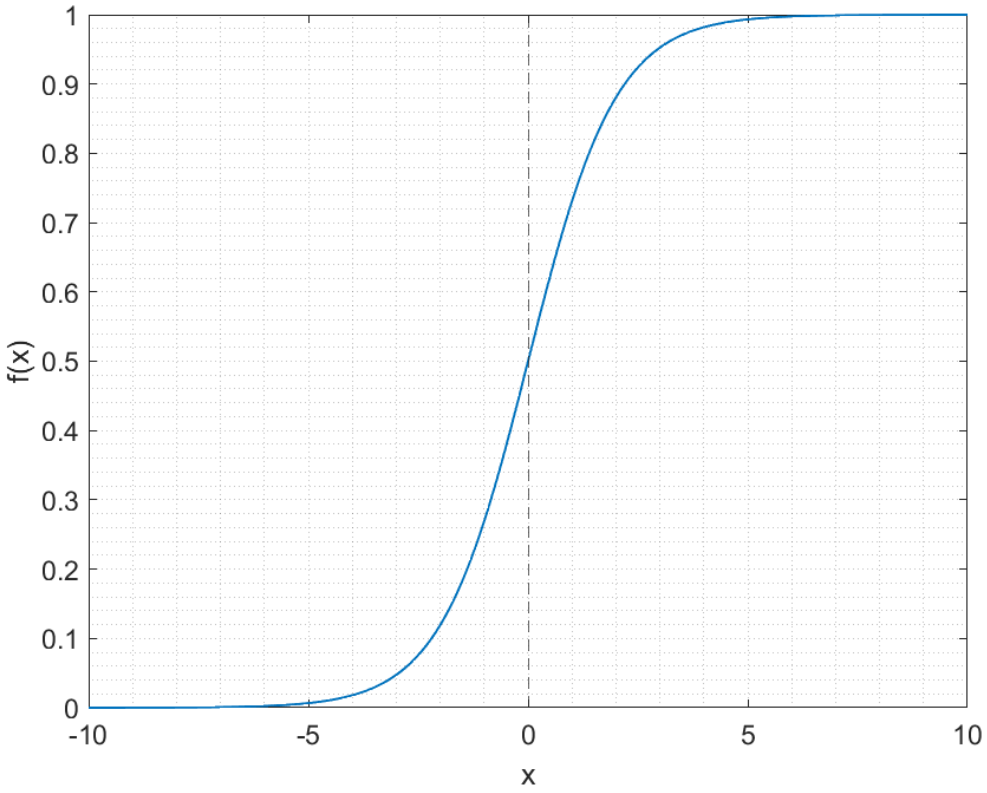

Figure 2.5. The sigmoid function.

ReLU

ReLU is another commonly used function which returns 0 if the input is negative and, for any positive input value, it returns the value itself back i.e. the range of the output is between 0 to infinity. The function is represented in equation (2.3.2) along with its PDF in figure 2.6.

$$f(x) = \max(0, x) \tag{2.3.2}$$

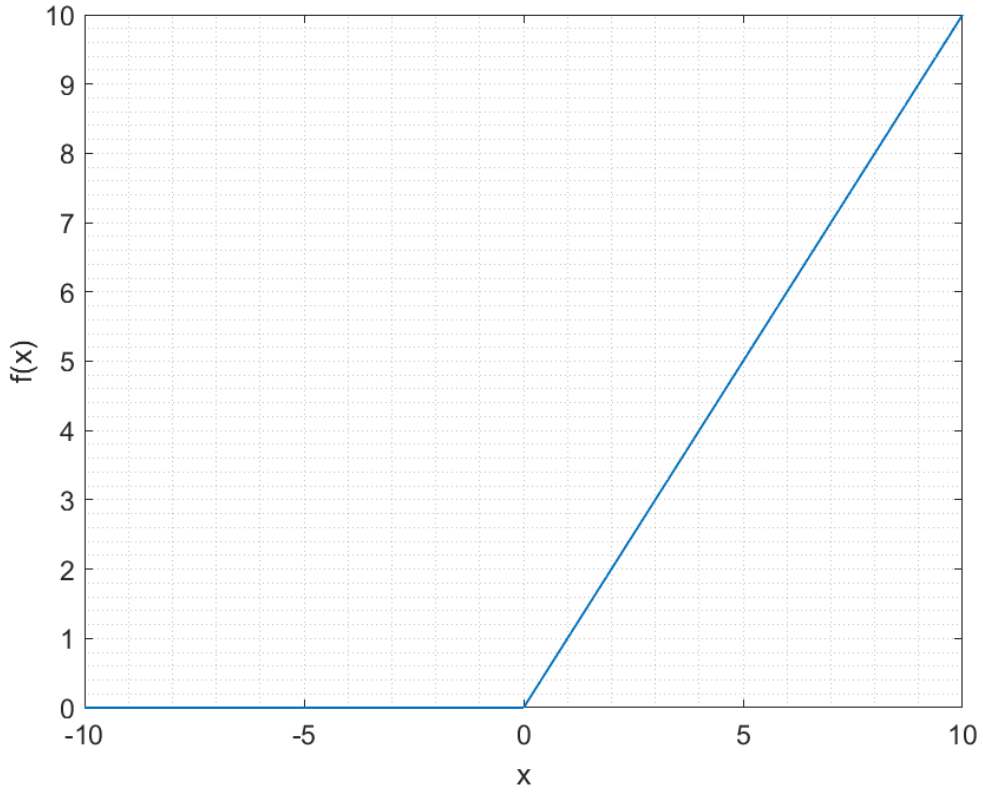

Figure 2.6. The ReLU function

Additionally, a special kind of weight which is called a bias is added to the product of weights and inputs. The bias b is a constant value, usually set by default as 1. The bias is not influenced by the previous layer, however it has an assigned weight due to which it has an impact on the next layer. The bias ensures that even if all inputs are zero, there is still an activation in the node.

Model preparation

In order to use machine learning for practical use cases such as prediction or classification, the model trains or learns by leveraging large data sets and this helps the network in processing unknown inputs more accurately [AWS, 2023b]. Training a machine learning algorithm encompasses two phases; the data processing and the training phases.

Data processing phase: In the data processing phase, the data is commonly split into two groups referred to as *training sets* and *testing sets*. The model usually trains itself on these training set. It is used for estimating parameters, comparing model performance and other activities to train the model. The testing set is used only at the conclusion of training, and it is imperative that the test data is not used before this point [Max Kuhn, 2019].

There are multiple ways to split the data into training and testing sets. The way the data is split can have a major influence on its performance. For example, if the training data is too small or if the model trains itself on the same dataset for too long, it may run into a problem called overfitting. Overfitting is a modelling error that occurs when the model

gives very accurate predictions for training data but not for unseen data i.e. the model fits very closely to the training dataset. In order to avoid overfitting, the model must be tested on enough data that is representative of all of the inputs [AWS, 2023c].

One method that can be used to test for overfitting is the 'K-fold cross validation'. In K-fold cross validation, data is divided into K equal subsets, which are called folds. One of these folds is used to test the model, while remaining subsets are reserved to train the model. This process is repeated until each of the fold has acted as a testing set. The performance for each of these test sets is evaluated and a score is retained. Finally, all the scores are averaged and an overall score for the model is attained [IBM, 2023a]. This operation is illustrated in figure 2.7.

Figure 2.7. Example of how K-cross validation works, where K=5

Having established the general terms and features of a neural network model, different types of neural networks are introduced.

Types of neural network models

Neural Network models can be categorised into different categories. Two main examples are Recurrent and Feedforward networks. Recurrent networks - e.g. Long Short-Term Memory Networks (LSTMs) - allow for feedback connections and loops in the network. This means that the nodes become active for a defined and limited duration of time before stimulating the neighbouring nodes, in both directions, which also become active for a limited duration of time. Consequently, the output will be affected by its input after a period of time and not immediately. In addition, each node stores its current state in an internal memory from which the info is later retrieved to be used in the next computation step.

On the other hand, in feedforward networks, the output from a layer serves as an input

for the next layer where the information is never fed back, but only forward. This makes feedforward networks faster, compared to recurrent networks. Thus, for predicting latency using DT technology, feedforward is more suitable, so Recurrent Neural Networks are not considered in this project.

Feedforward algorithm

Essentially, a feedforward neural network aims to estimate the best predictor function for the approximation G^*. To do so, the network applies an estimator $\hat{y} = G(x;\theta)$ which takes input x, learns from parameter θ, and then provides an estimate $\hat{y}$.

Initially, each connection in the neural network is given a `w` weight value, a random small number. Furthermore, at each node, `b` bias value is set to 1, with a corresponding small weight `w`. At every single layer, the inputs are multiplied by the corresponding weight values then summed up together with `b`. Next, the resulting sum is passed through an activation function f. This process is illustrated in figure 2.8

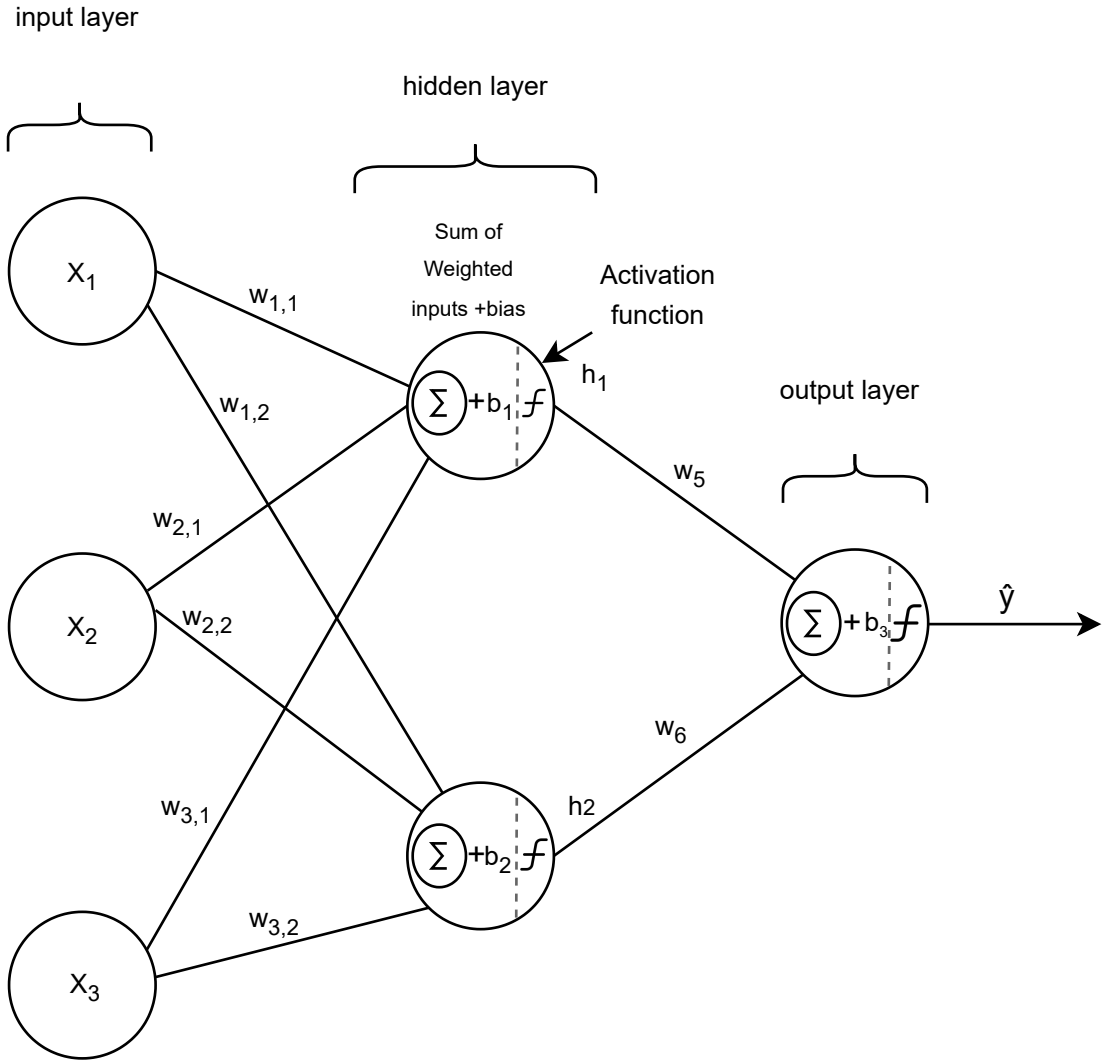

Figure 2.8. Example of a three-layer neural network with one hidden layer.

The output at a particular node on the hidden layer can be represented with equation 2.3.3:

$$h_j = f(x_i \cdot W_{ij} + b_j) \tag{2.3.3}$$

where W_{ij} is a vector holding w weight values of the connections between the input layer and node j at the hidden layer, x_i is the input vector, b_j is the bias in node j and f is the activation function.

Similarly, the output value on the output layer can be represented through the following equation:

$$\hat{y} = f(h_1 \cdot W_5 + h_2 \cdot W_6 + b_3) \tag{2.3.4}$$

where W_5 and W_6 are the weight values of the connections between the input node on the hidden layer and the output node on the output layer, h_1 and h_2 are the output of the nodes 1 and 2 in the hidden layer, b_3 is the bias in the output node, and f is the activation function.

Training phase: This phase starts by quantifying the discrepancy between the predicted value obtained through the feedforward algorithm and the true value. This is done using a loss function. The two main types of loss functions are: Classification Loss Functions and Regression Loss functions.

Classification Loss Function:

In binary classification problems which categorise data into one of two classes, cross entropy is used as a common cost function, which is also known as logarithmic loss. The function is given by:

$$H(P^*|P) = -\sum_i P^*(i) log P(i) \tag{2.3.5}$$

where $P^*(i)$ is the true class distribution and $P(i)$ is the predicted class distribution.

Regression Loss Function:

Regression based neural networks are used to predict a real value quantity. Mean Square Error (MSE) is a common loss function used for evaluating the performance of linear regression. The MSE is calculated using the average of the squared differences between the predicted and actual values. The mathematical equation for MSE is as follows:

$$MSE = \frac{1}{N}\sum_i^N (\hat{y}_i - y_i)^2 \tag{2.3.6}$$

where N is the number of data points, $\hat{y}_i$ is the predicted values and y_i is the observed values.

Backpropagation

Once the loss function is obtained, the goal is to minimise the loss function. This is performed through the backpropagation algorithm. As the name suggests, this algorithm

back-propagates from the output layer towards each node in the input layer, in order to adjust the weights and biases of the nodes based on the individual impact on the overall loss function.

The following example shows how the backpropagation algorithm works. The model shown in figure 2.8 is considered, where the sigmoid function is chosen to be the activation function in all the nodes of the model. Initially, the loss function is computed through the following equation:

$$L = \frac{1}{2}(\hat{y} - y)^2 \tag{2.3.7}$$

where $\hat{y}$ is the predicted value and y is the actual output value. In order to reduce the cost function and find the local minima, different optimisers can be used. An optimiser is an algorithm or method that adapts the weights and biases such that overall loss can be minimised. The gradient descent algorithm is a commonly used optimiser. To adjust the weight and bias values of the system, the gradient descent of the individual weight and bias is computed. This is done by finding the partial derivative of the loss function with respect to each parameter (weights and biases). This indicates in which direction the individual parameter should be updated in order to minimise the cost function. Graphically, this is illustrated in figure 2.9, where the weights are updated until it gets to the lowest value of the cost function where the gradient (slope) approaches zero. The amount that the weights are amended during training is referred to as the step size or the learning rate.

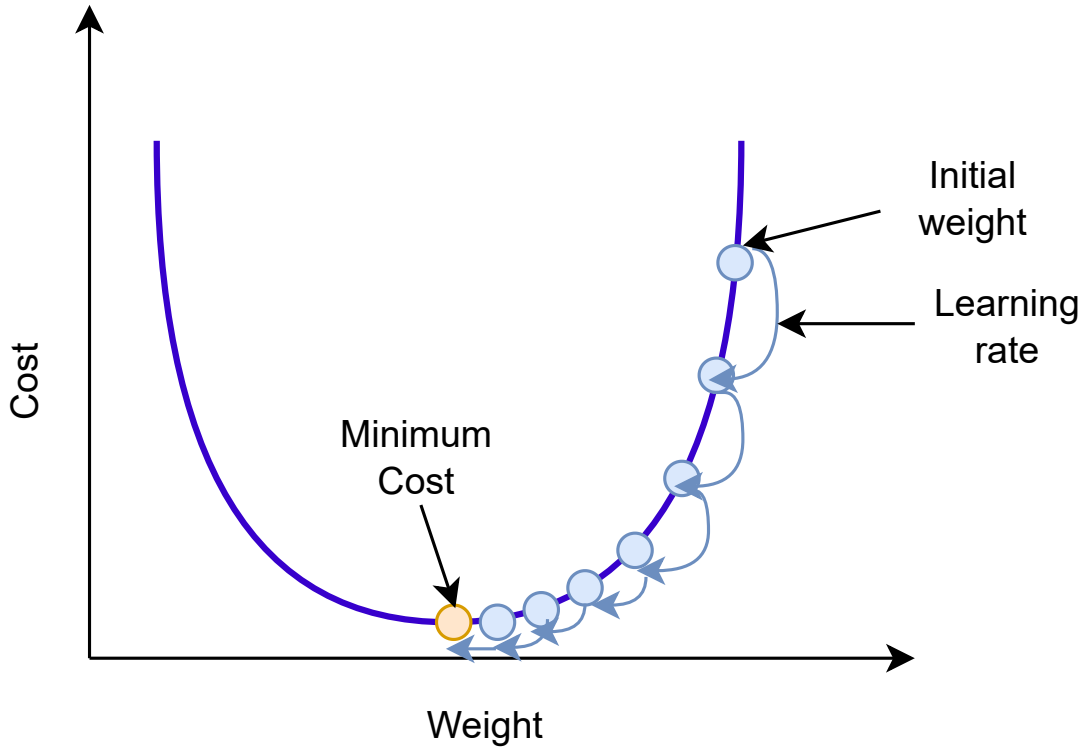

Figure 2.9. Graph showing the cost function with respect to weight.

Returning to the example in figure 2.8, for w_5, the partial derivative of the loss function in terms of w_5 cannot be computed directly, since w_5 is embedded deep inside the output

function, namely in $\hat{y}$. Thus, the gradient with respect to w_5 is calculated by applying the chain rule, as shown in equation 2.3.8:

$$\frac{\partial L}{\partial w_5} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_5} \tag{2.3.8}$$

It is important to mention that the equation of $\hat{y}$, as can be seen in equation 2.3.4, is a result of two operations, namely, a linear and a non-linear operation. The linear operation is represented by the summation of the weighted inputs, h_1 and h_2 with b_3. The non-linear operation is the implementation of the activation function on the result obtained through the linear operation. Consequently, to find the partial derivative of $\hat{y}$ with respect to w_5, the chain rule is applied, so that it results in the following expression:

$$\frac{\partial \hat{y}}{\partial w_5} = \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial w_5} \tag{2.3.9}$$

where

$$z = h_1 \cdot W_5 + h_2 \cdot W_6 + b_3 \tag{2.3.10}$$

and

$$\hat{y} = f(z) = \frac{1}{1 + e^{-z}} \tag{2.3.11}$$

The partial derivative of $\hat{y}$ with respect to z is computed by finding the partial derivative of the sigmoid function.

$$\frac{\partial \hat{y}}{\partial z} = \frac{1}{1 + e^{-z}} \cdot (1 - \frac{1}{1 + e^{-z}}) \tag{2.3.12}$$

Thus, equation 2.3.9 can be written as follows:

$$\frac{\partial \hat{y}}{\partial w_5} = \frac{1}{1 + e^{-z}} \cdot (1 - \frac{1}{1 + e^{-z}}) \cdot \frac{\partial z}{\partial w_5} = \frac{1}{1 + e^{-z}} \cdot (1 - \frac{1}{1 + e^{-z}}) \cdot h_1 \tag{2.3.13}$$

Inserting equation 2.3.13 into equation 2.3.8 gives:

$$\frac{\partial L}{\partial w_5} = (\hat{y} - y) \cdot \frac{1}{1 + e^{-z}} \cdot (1 - \frac{1}{1 + e^{-z}}) \cdot h_1 \tag{2.3.14}$$

By computing equation 2.3.14, a numerical value is obtained and then it is used in equation 2.3.15 in order to find the new value of w_5. Note that η is the learning rate. The value of this rate is usually set to a small number such as 0.001 to avoid the weight being changed drastically from one iteration to another.

$$w_{5(update)} \rightarrow w_5 - \eta \cdot \frac{\partial L}{\partial w_5} \tag{2.3.15}$$

Similarly, the gradient with respect to b_3 is computed, where the result obtained is as follows:

$$\frac{\partial L}{\partial b_3} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial b_3} \tag{2.3.16}$$

$$\frac{\partial L}{\partial b_3} = (\hat{y} - y) \cdot \frac{1}{1 + e^{-z}} \cdot (1 - \frac{1}{1 + e^{-z}}) \cdot 1 \tag{2.3.17}$$

Then, the update value of b_3 can be found through:

$$b_{3(update)} \rightarrow b_3 - \eta \cdot \frac{\partial L}{\partial b_3} \tag{2.3.18}$$

The gradient with respect to w_6 is calculated through the following equation:

$$\frac{\partial L}{\partial w_6} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_6} \tag{2.3.19}$$

$$\frac{\partial L}{\partial w_6} = (\hat{y} - y) \cdot \frac{1}{1 + e^{-z}} \cdot (1 - \frac{1}{1 + e^{-z}}) \cdot h_2 \tag{2.3.20}$$

Next, the update value of w_6 can be found through:

$$w_{6(update)} \rightarrow w_6 - \eta \cdot \frac{\partial L}{\partial w_6} \tag{2.3.21}$$

After updating the nearest weights (w_5 and w_6) to the output node, as well as the bias (b_3), all the other parameters should be updated while tracing back towards the input layer, through the hidden layer. For $w_{1,1}$, it can be observed, when propagating back, that to reach this parameter, the following path is involved:

$$w_1 \leftarrow h_1 \leftarrow \hat{y} \leftarrow L \tag{2.3.22}$$

Thus, the descend gradient of the loss function with respect to $w_{1,1}$ can be computed through the following equation:

$$\frac{\partial L}{\partial w_{1,1}} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial h_1} \cdot \frac{\partial h_1}{\partial w_{1,1}} \tag{2.3.23}$$

where

$$\frac{\partial L}{\partial \hat{y}} = \hat{y} - y \tag{2.3.24}$$

$$\frac{\partial \hat{y}}{\partial h_1} = \frac{1}{1 + e^{-z}} \cdot (1 - \frac{1}{1 + e^{-z}}) \cdot w_5 \tag{2.3.25}$$

$$\frac{\partial h_1}{\partial w_{1,1}} = x_1 \tag{2.3.26}$$

Thus equation 2.3.23 can be written as follows:

$$\frac{\partial L}{\partial w_{1,1}} = (\hat{y} - y) \cdot \frac{1}{1 + e^{-z}} \cdot (1 - \frac{1}{1 + e^{-z}}) \cdot w_5 \cdot x_1 \tag{2.3.27}$$

Then, the new value of $w_{1,1}$ is computed through the following equation:

$$w_{1,1(update)} \rightarrow w_{1,1} - \eta \cdot \frac{\partial L}{\partial w_{1,1}} \tag{2.3.28}$$

Afterwards, to update the values of the other parameters i.e. b_1, b_2, $w_{1,2}$, $w_{2,1}$, $w_{2,2}$, $w_{3,1}$ and $w_{3,2}$, the descend gradients of each of these parameters need to be computed. This is done by tracing back the path from the loss function to each of these parameters, then calculating the partial derivative of the loss function with respect to each of these parameters. By doing so, the aforementioned parameters can be updated using the following equations:

$$\begin{aligned}
b_{1(update)} &\rightarrow b_1 - \eta \cdot \frac{\partial L}{\partial b_1} \\
b_{2(update)} &\rightarrow b_2 - \eta \cdot \frac{\partial L}{\partial b_2} \\
w_{1,2(update)} &\rightarrow w_{1,2} - \eta \cdot \frac{\partial L}{\partial w_{1,2}} \\
w_{2,1(update)} &\rightarrow w_{2,1} - \eta \cdot \frac{\partial L}{\partial w_{2,1}} \\
w_{2,2(update)} &\rightarrow w_{2,2} - \eta \cdot \frac{\partial L}{\partial w_{2,2}} \\
w_{3,1(update)} &\rightarrow w_{3,1} - \eta \cdot \frac{\partial L}{\partial w_{3,1}} \\
w_{3,2(update)} &\rightarrow w_{3,2} - \eta \cdot \frac{\partial L}{\partial w_{3,2}}
\end{aligned} \tag{2.3.29}$$

After all parameters of the model have been updated, another values of x_1, x_2, and x_3 are feedforwarded to the network. This results in a new prediction, thereby the loss function is calculated. Next, through the backpropagation algorithm, the weights and biases are updated. This is repeated until the entire data set (all values of x_1, x_2, x_3) has been used once. This is called an *epoch* and in order to fully train the model, many epochs can be taken.

Typically, the data is set to be used in *batches*. For example, if there are 1000 data points for training, a batch size could be set as 50, meaning that 50 of the data points are used at a time. After this has been done 20 times, meaning all 1000 data points have been used, the epoch is completed. The difference between using the complete data set at once is that the weights are updated with every batch, allowing for finer tuning of the weights.

2.3.2 Summary

Some of the State of the Art approaches to latency prediction were investigated, which lead to the idea of using NNs as a type of DT. The math behind training and using a NN was described, in order to understand how the nodes relate the inputs to the output. In the next section, how the NN is trained and used is detailed, starting with the proposed PT structure to generate latency.

2.4 Final Problem Statement

This chapter included a technical analysis of digital twins and one of the possible implementations of such with NNs. Given the problem of predicting E2E latency, the concept of training a NN to predict such a latency is given with an initial proposal. To accomplish this, a Physical Twin is created and run alongside a trained Digital Twin model, feeding the timestamps from the physical twin as inputs. However, if the Digital Twin is to predict the E2E latency, it needs to be faster than the Physical Twin. This raises a new and final problem statement:

How can end-to-end latency be predicted using a digital twin based on deep learning Neural Networks with data from a Physical Twin, prior to the completion of the physical twin process?

Use Case 3

3.1 Use Case

In order to figure out how latency can be predicted using a NN Digital Twin, there must first be a way to obtain the data. This chapter details what kind of Physical Twin could be created for this purpose, and how the Digital Twin models can be structured.

3.1.1 Initial system proposal

As discussed in the previous chapter, latency is the effect of the many individual steps and processes that take place for example during communication. To look at E2E latency, as suggested in the final problem statement, would be to look at the cumulative latency in a process bridging multiple steps.

Given that the purpose of this report following the final problem statement is to consider E2E latency, there must also be a E2E latency that can be observed. This is necessary both to compare the DT with real data, but also to train the prediction. To accomplish this, a PT is created to accompany a DT and provide some manner of input and output.

A diagram of this system is seen on figure 3.1, where the PT obtains an output t_{total}, which is the E2E latency measured by the difference between t_{init} and t_{end}. Meanwhile, the DT is supplied with a number of inputs i, and provides a predicted output $\hat{t}_{total}$. The times each component takes to provide the outputs are given by t_{PT} and t_{DT} accordingly.

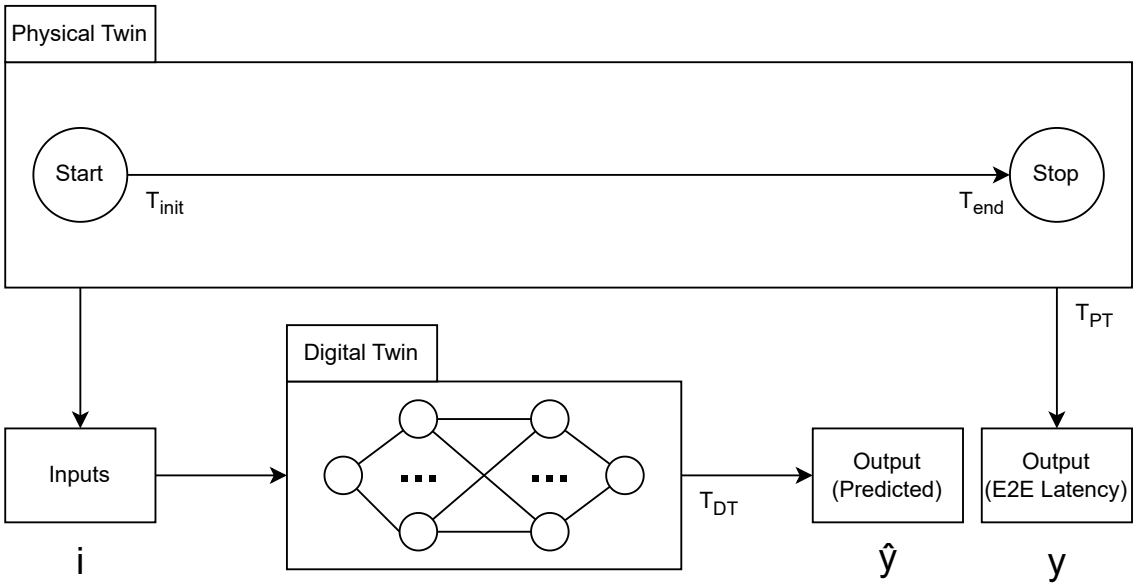

Figure 3.1. Diagram encompassing both the Physical Twin and the Digital Twin.

The DT will naturally have to be trained prior to usage, using both inputs and output from the PT, effectively simulating the process. The training methodology and types of inputs are found during the implementation of the model, as the precision of the model depends not only on the input/output, but also the structure and amount of training iterations.

3.1.2 The Physical Twin

In order to obtain the training data necessary for the DT, and to have a real process to compare results with, a simple server-client architecture is proposed. This PT encompasses the transmission of data between two distributed systems over a wireless connection. This is a multi-step process which is suitable for accumulating different sources of latency, and can be set up with relative ease. A diagram representing this use case can be seen on figure 3.2.

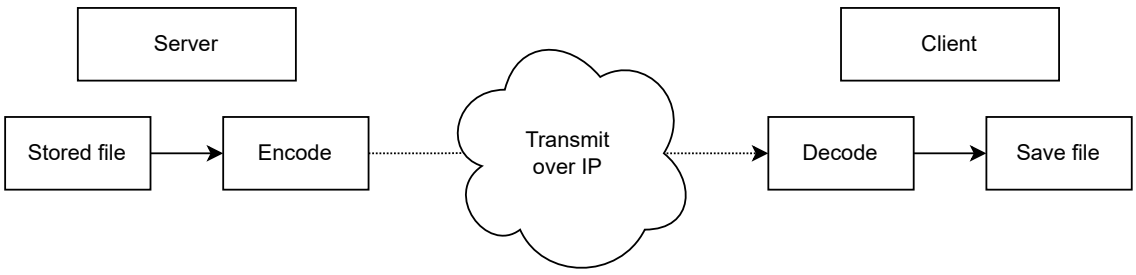

Figure 3.2. A simple server-client architecture including the encoding/decoding of files before and after transmission.

Here, it can be seen that the server part of the PT has to send a file over IP to the client, which requires encoding data prior to transmission, and decoding after receiving. Already here there are a couple of steps that require some processing and reading/writing time,

on top of the latency from transmitting over IP. However, this proposal does not include the act of establishing connection between the client and the server, which adds additional latency.

To show the connection process from start to finish, a sequence chart is included on figure 3.3. Specifically, this shows when the client starts the process of communicating with the server, to when the file is received and connection closed. This figure discounts the elements of decoding the data.

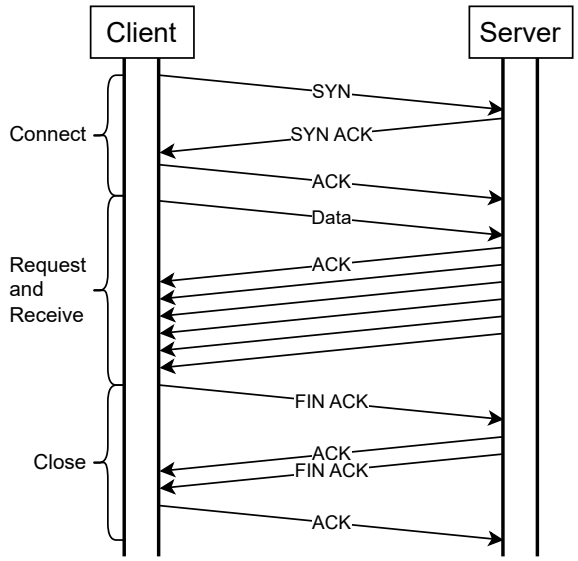

Figure 3.3. A sequence chart showing the process from establishing a connection, to receiving the file, and finally closing the connection.

This figure includes the notion that the client will have to request a file before it can be transmitted. As such, figure 3.2 is updated to include these additional steps, which can be seen on figure 3.4.

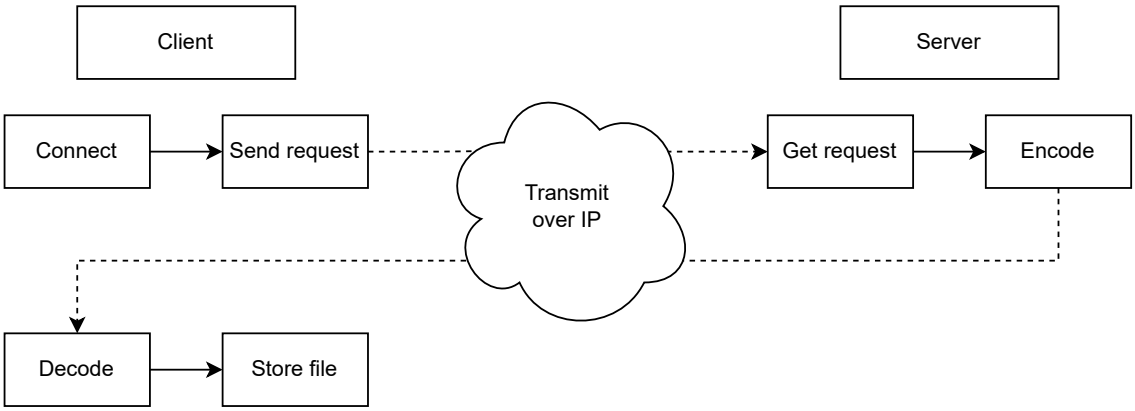

Figure 3.4. An updated server-client architecture, where the client first sends a request.

Definition 3.1.1 (E2E Latency)**.** In this report, the E2E latency, given by t_{total}, would be the total time it takes from establishing the initial connection, to having the file decoded and ready on local storage.

3.1.3 The Digital Twin

Alongside this PT, a DT would be run. As the objective is to simply predict the output from the PT, this DT will only have the limited capability of simulating the PT with the goal of providing an output before the PT is finished. This DT would utilise a model trained on the output from the PT. However, it is not enough just to have the output t_{total} in order to train a model and use it alongside the PT as a DT. To actually predict, the DT should also be given some kind of input data, that would be available *prior* to the output time. Since the output of the proposal consists of the cumulative time that the process takes, it would be ideal to consider using the time each individual step takes as an input for the model. This could be done using timestamps for the different functions seen on figure 3.4, including the time to establish connection, send a request and receive a file, decode the data, and finally store it.

However, this method of predicting based on the entire series of timestamp inputs is not feasible. Since the DT would be waiting until each iteration is done to obtain all timestamps, it would not actually predict the output in a timely manner, and would simply be a simulation. On top of that, the DT itself will also introduce additional latency from the NN model. It is important that the time to predict t_{DT} remains smaller than the PT latency t_{PT}, as this is a live prediction and the DT should be able to finish before the PT. Otherwise it would just count as an offline prediction. As such, it would be necessary to assess how many of the timestamps would be required to include.

For this, the idea of not just using the timestamp data, but also using additional inputs that would be known prior to transmission, is proposed. As an example, the connection time in combination with the size of the transmitted file could be used in order to try and compensate for not having the exact "receiving" time, as the time to transmit a file would depend both on the connection but also the amount of Bytes to transfer. This requires an investigation of which exact inputs and factors affect the PT process, which is included in section 4.2.

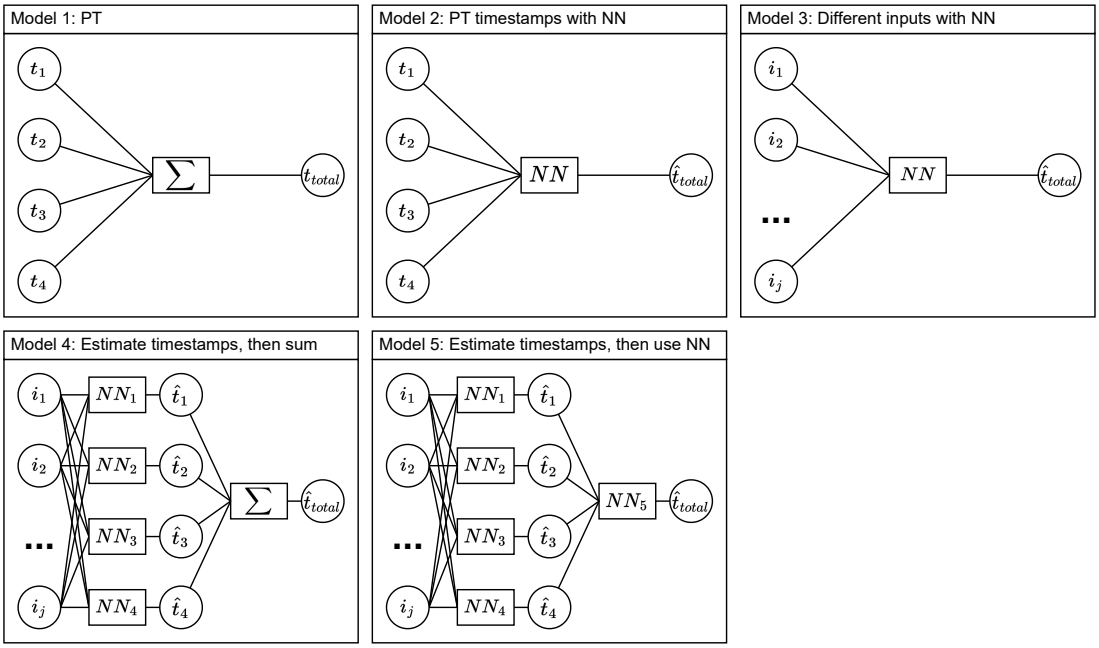

Figure 3.5. Model for obtaining t_{total}.

Another matter is how these inputs are modelled in the DT. A number of different models are proposed:

Model 1: True output

This model consists of just the PT, which collects time data over an iteration, and achieves a final output t_{total} by summation of all the parts.

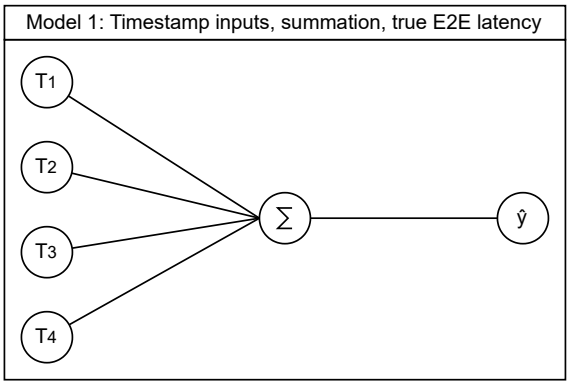

Figure 3.6. Model for obtaining t_{total}.

While this model is not a DT and does not utilize NN models, it serves as the ground truth t_{total} to compare the predicted outputs $\hat{t}_{total}$ with.

Model 2: Timestamps, NN

This is the initially proposed model, that would not feasibly be able to finish before Model 1. However, it would still be relevant to see if it could potentially provide a good prediction.

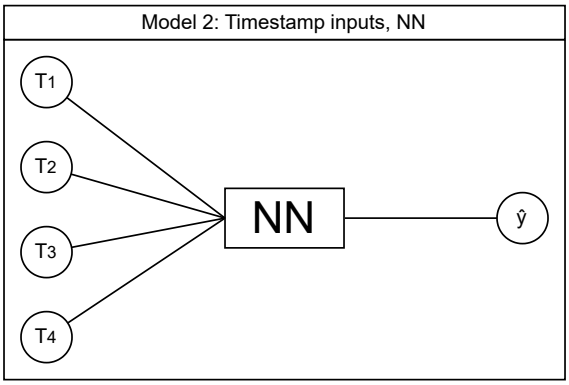

Figure 3.7. Model for obtaining the estimated output $\hat{t}_{total}$ using timestamps.

Model 3: Varied inputs, NN

Similar to the previous model, however except for using just the timestamps, this model would take different types of inputs relevant to the PT process, for example file sizes and connection times. The types and amount of inputs are identified and documented during the implementation in the following chapter.

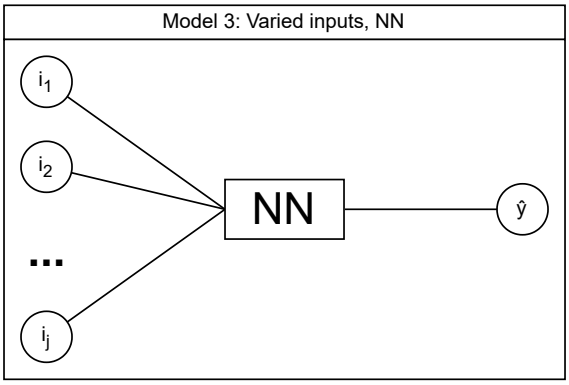

Figure 3.8. Model for obtaining the estimated output $\hat{t}_{total}$ using different inputs.

Model 4: Varied inputs, estimated timestamps, summation

As a way of replicating the first model and method of obtaining the true output, Model 4 predicts the four individual timestamps using relevant inputs, i.e. geographical distance and file size. The sum of these four timestamps is then taken, giving $\hat{t}_{total}$.

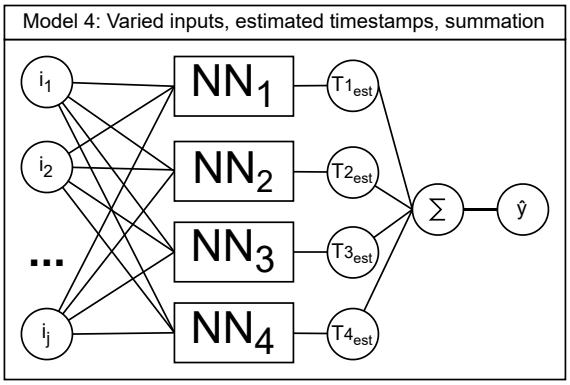

Figure 3.9. Model for obtaining the estimated output $\hat{t}_{total}$ by getting estimated timestamps and taking the sum.

What makes this model particularly interesting, is the generated timestamps that can additionally be analyzed and compared to the real timestamps. However, as this model requires additional NN models, the execution time would be expected to be longer than Models 2 and 3.

Model 5: Varied inputs, estimated timestamps, NN

Similar to Model 4, Model 5 takes all of the predicted timestamps and instead uses them as inputs for another NN model, similar to Model 2.

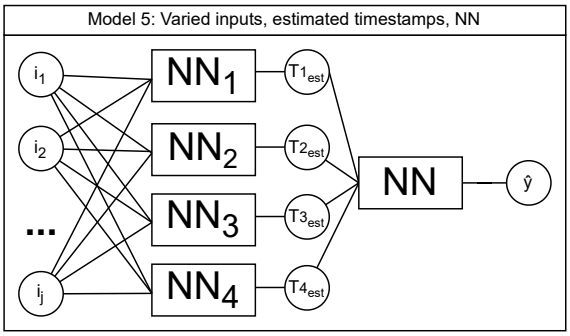

Figure 3.10. Model for obtaining the estimated output $\hat{t}_{total}$ by getting estimated timestamps and passing them to a NN.

This model is included to see if adding the additional NN model could maybe compensate or correct the estimated time inputs, and provide a better prediction than just summing up the estimated timestamps.

3.1.4 Summary

The intention with this project is to create a PT to provide real data, and DT to emulate the PT with the purpose of predicting the E2E latency t_{total}. The PT consists of a client and server architecture, where a connection is established in order to request and receive a decoded file. The proposed DTs consist of different models with different combinations of inputs and NNs in order to later evaluate which would provide better predictions.

Design 4

This chapter contains the implementation and documentation for the PT and DT used in the system. In addition to that, which inputs are used as well as how they affect the latency in each iteration are also discussed.

After that there is a section which thoroughly covers the data capture and data processing processes required for each of the models in the system.

The results shown in this chapter are not representative of the final model, and were obtained using preliminary data and models.

4.1 Design

Chapter 3 addressed the intended design of the latency prediction model, including both a Physical Twin and a Digital Twin. The creation of each of these components is detailed below.

4.1.1 Creating the Physical Twin

The PT consists of both server and client components, which have been created and implemented individually.

Server

The server was created and then configured in the form of a virtual machine, using the Strato cloud environment hosted by Aalborg University [Strato, 2023]. Strato is managed with OpenStack, which is a cloud computing software that provides a range of services for managing cloud-based resources such as storage, networking, and computing. In addition, a public IP was assigned to the server in order to enable external access to it. This means that the server can be accessed remotely, resulting in the latency associated with routing and connecting to the server being generated.

The server is run by a Python script. The script establishes sockets using the `socket` module and can simultaneously listen for incoming TCP and UDP traffic on two different ports using the `threading` module. This required creating Firewall rules on OpenStack to permit the server to receive and handle TCP and UDP traffic. In the event of an incoming TCP request, the server accepts and establishes a TCP connection then starts receiving data from the socket using the function `accept` and `recv` respectively. Next, the server parses the request received from the client to extract the file name. Then the targeted

file is selected from the server database and encoded, then sent back to the client over the TCP socket using the `socket.sendall()` function.

For a UDP request, the server sends back the requested file in chunks to the client after having encoded it, without having to establish a socket connection first.

Client

On a laptop, the client is set up using a Python script that supports both protocols, TCP and UDP. The script starts with a `time.perf_counter()` to initialize a start time, which will be used to obtain the timestamps in the program. Every time a timestamp is taken, all of the previous timestamps on top of the start time are subtracted from a new `time.perf_counter()`.

When TCP is selected, the client sends a connection establishment request using the `socket.connect(server-IP, PORT)` method. After receiving a response from the server indicating that the connection has been accepted, the connection timestamp is recorded. The client then encodes the file name to bytes using UTF-8 and sends it to the server using the `socket.send(File_Name.encode('utf-8'))` method. When the file is received from the socket through the `socket.recv()` method, the request and receive timestamp is recorded. Afterwards, the client decodes the received data and records the decoding timestamp. Finally, the encoded data is saved in MP4 format, while also recording the corresponding timestamp.

In case of UDP, the client creates a UDP socket and sends a request including the file name to the server using the
`socket(socket.AF_INET, socket.SOCK_DGRAM)` and
`socket.sendto(file_name.encode('utf-8'), (IP, port))` methods respectively. Then, the server sends the requested file encoded and in chunks. When the whole file is received, the timestamp of this is recorded. Unlike TCP, UDP does not have a mechanism for indicating that the end of a file has been transmitted. Therefore, it was crucial to implement an approach that handles the situation. This was achieved by using the word 'END' as a delimiter, which indicates to the client that no further data will be received and to stop waiting to receive any more. In addition to that, a timeout feature was implemented to specify the maximum amount of time the client should wait before assuming the packet will not arrive or is lost. Following the arrival of the 'END' delimiter, the client decodes and saves the file in the MP4 format while taking the timestamps for each of these separate operations.

4.1.2 The Digital Twin

Before the DT can make live predictions such as detailed in chapter 3, it first requires to have the existing models. Each model is created using the `Tensorflow Keras` module for Python. By initialising a `keras.Sequential()` object or a so-called "model" with a sequential layer structure, the various parameters such as the amount of input nodes, hidden layers, hidden layer nodes, output nodes, as well as activation functions can be set. `model.compile` is then used to set the loss and optimisation functions for the model, where Stochastic Gradient Descent is used for optimisation and MSE for loss. Given that

this is a regression problem and not a classification problem, accuracy is not a possible metric, and only loss parameters such as MSE are considered, as the result will be how close the predictions are to the ground truth.

model.fit then trains the model, using a set of training input, training output, and parameters such as batch size and amount of epochs. The training data does not include all of the given input and output data, as some of it will be used for validation and requires that it has not been used prior for training. The `train_test_split()` Scikit function accomplishes this by randomly selecting 80% of each input and output data for training and the remainder 20% for validation. In addition to randomly selecting the data, it is also randomly shuffled in advance. As for the epochs and batch size, those should be chosen based on trial and error, while taking in mind not to overfit or underfit the model.

At this point the model is trained, and can be given input values in order to predict an output using `model.predict()`. This is the basis for the DT implementation. However, it is important to consider that these models need to be created using offline data, and will not be trained alongside real-time measurements. This creates a need to obtain enough data before the model can make adequate predictions. A diagram that shows the relationship between offline training and the DT can be seen on figure 4.1.

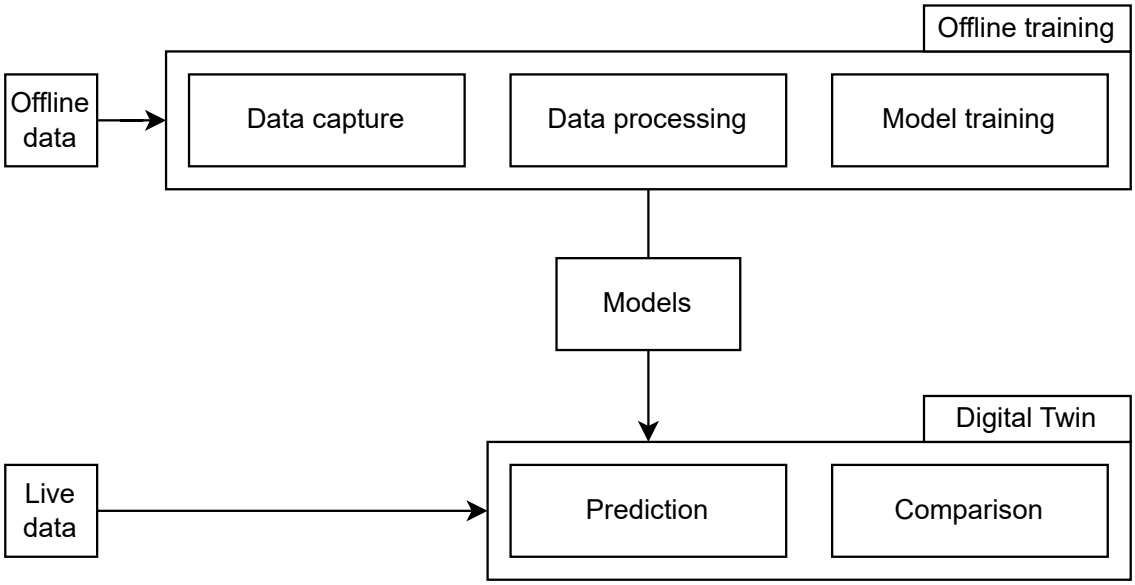

Figure 4.1. A diagram of the offline training procedure and the Digital Twin which requires the complete trained models from the latter to function.

The offline training segment includes three different segments: data capture, data processing, and model training. The resulting models are passed to the DT, which using live data will provide predictions and show comparisons between the real E2E latency and timestamps, and the predicted ones.

Data capture outlines not only how the different inputs are recorded, but also what impact they have on the latency. This comprises an analysis of what information can be extracted from the physical twin outside of just the timestamps. When all of the data sources have been detailed, the different methods of processing this data before training are outlined.

This is to ensure that the data is not only compatible with the models, but also to improve the training results.

The model training segment builds on top of the `Tensorflow` basics described early in this subsection. This includes how the parameters for the model training itself, also known as the hyperparameters, affect the process and can be found in the first place. Similar to the data processing, this has the goal of getting as much use out of the data as possible.

4.2 System Inputs

The operation of downloading a file at the PT encompasses the following main processes; connection, request and receive, store, and decode. The individual time delays of the aforementioned processes are denoted as t_1, t_2, t_3, t_4, respectively, and t_{total} is the sum of these processes corresponding to the E2E delay. Each of these processes may include multiple sub-processes, and examining all of these is not the major focus of this work. As such, the processes are mostly treated as a black box, where as many factors as possible are discovered and analyzed. These factors are listed in table 4.1 which are later employed as inputs to the NN in order to predict $\hat{t}_1$, $\hat{t}_2$, $\hat{t}_3$, $\hat{t}_4$, and $\hat{t}_{total}$, according to the DT structure concepts included in chapter 3. The individual factors are analysed in the subsections below.

Table 4.1. Factor that have direct impact on the latency of the individual processes.

Factor	t_1	t_2	t_3	t_4
Distance	X	X	-	-
Technology	X	X	-	-
Transmissions protocol	X	X	-	-
File size	-	X	X	X
Number of transmissions	-	X	-	-
Encoding algorithm	-	X	X	-
Maximum Transmission Unit (MTU)	-	X	-	-
Received Signal Strength (RSS)	-	X	-	-
Downlink rate	-	X	-	-

Distance

In order to evaluate the impact of the distance on connection, request and receive delays, and eventually on t_{total}, a number of tests have been performed. In these tests, the client performs, sequentially, the following tasks; connect to server, send a TCP request that includes the file name to be downloaded, receive the video file, store then decode it, and finally close the TCP connection. Theses tasks are iterated 10,000 times at different distances between the client and server, while maintaining a constant video and buffer sizes of 10 KB and 900 B, respectively. Practically, the server is physically situated in Aalborg, Denmark, while the client is positioned at a different location for each test. In addition, at each distance, the client communicate with the server via WiFi.

The graph in figure 4.2 shows a clear positive correlation between distance and t_{total}, with increasing of the aggregate of t_1 and t_2 as distance increases. Nevertheless, the latency

remains relatively constant at small distances, namely at 23 km and 42 km. This pattern may be due to a range of factors, including the fact that the physical limitations of data transmission at small distances have a reduced impact. Whereas other factors such as hardware limitations, network congestion and processing time could contribute more to determining latency. Correspondingly, conducting tests involving longer distances, such as intra-continental or even inter-continental data transfer, would be worthwhile to more precisely assess the effect of distance on latency. This may minimise the relative significance of the other factors.

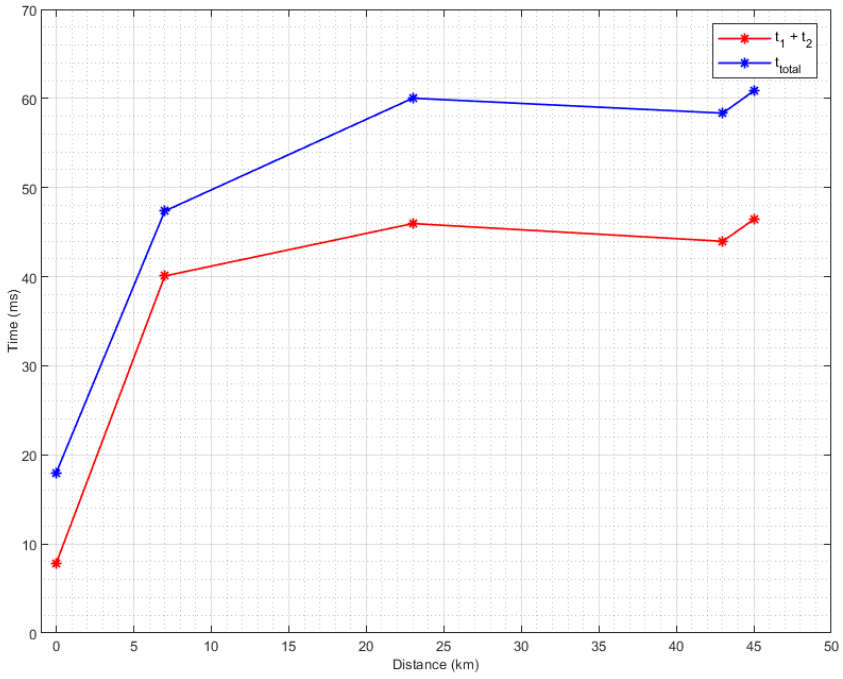

Figure 4.2. Graph depicting the impact of the latency on t_{total} and on $t_1 + t_2$, where the averages of 10,000 iterations are computed for distances; 0, 7, 23, 43 and 45 km.

In order to test the aforementioned hypothesis, five virtual servers have been deployed across multiple geographic locations including Stockholm, Frankfurt, London, Paris, and Virginia using Amazon Web Services (AWS) [AWS, 2023a]. AWS is a cloud computing platform, which provides an array of services such as storage, computing, and networking. In a similar manner as the previous test, this test involve measuring t_1, t_2, and t_{total} of the client to servers located at different distances, as seen in table 4.2. The first two cities listed in this table represent the client locations, with the server located in Aalborg. However, the remaining cities in the table represent the locations of the servers with the client located in Aalborg.

Table 4.2. Input information for cities and the corresponding distance, and data transfer category.

Country	City	Distance to Aalborg (km)	Data transfer category
Denmark	Aalborg	0	Same network
Denmark	Fjerritslev	43	Different network
Sweden	Stockholm	571	Intra-continental
Germany	Frankfurt	790	Intra-continental
England	London	845	Intra-continental
France	Paris	1025	Intra-continental
USA	Virginia	6278	Inter-continental

As seen in the graph in figure 4.3, the total value of t_1 and t_2, and eventually t_{total}, increase alongside the distance between the client and server locations, particularly over longer distances. This finding is consistent with the hypothesis that over greater distances the physical constrains of network communication become more apparent.

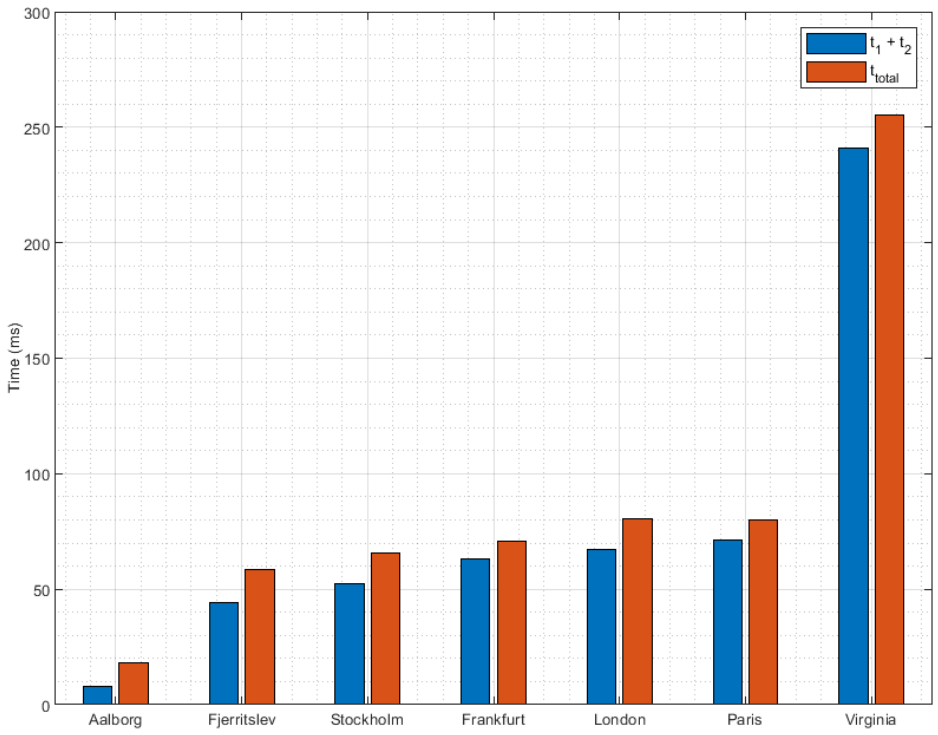

Figure 4.3. Graph depicting the impact of the distance on t_{total} and on $t_1 + t_2$, where the averages of 10,000 iterations are computed for various cities arranged in ascending order of distance between the client and server.

Technology

A number of tests have been conducted where the client repeatedly downloaded files of 10 kB size at different distances and using various technology, specifically, Wifi and 4G. As seen in in figure 4.4, the test results show that WiFi has significantly lower latency compared to 4G. Compared to 4G, WiFi provides a more reliable and stable connection, since it uses cables to transmit data which results in lower latency. On the other hand, a 4G connection is influenced by many factors including the distance to the cell tower, the interference, and network congestion, which may result in slower response times and eventually higher latency.

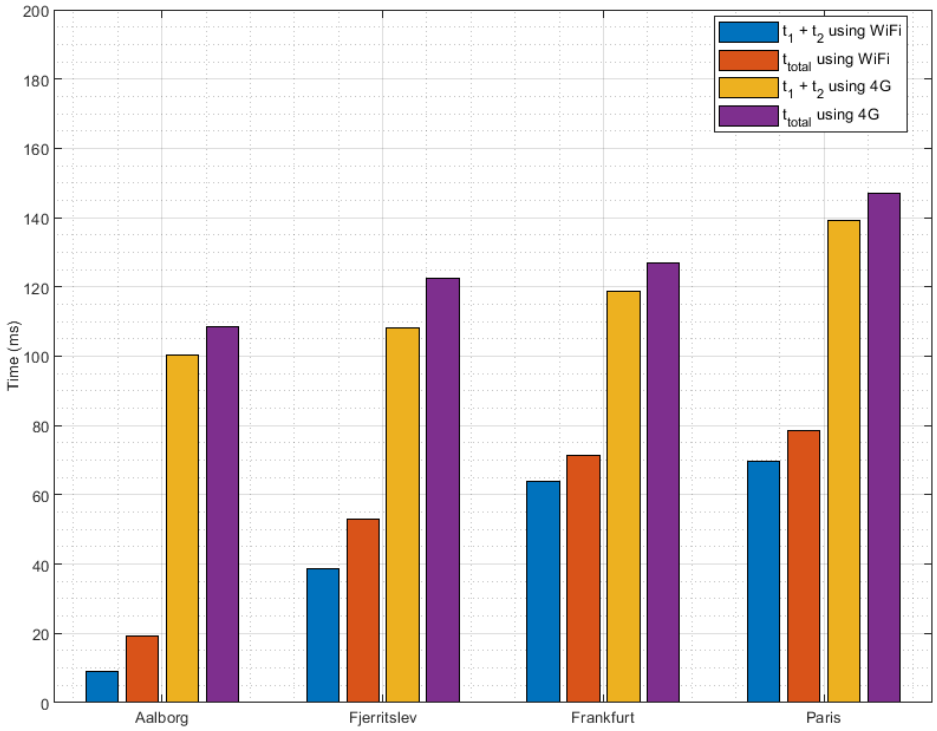

Figure 4.4. Graph depicting the impact of the different technologies (WiFi and 4G) on t_{total} and on $t_1 + t_2$, where the averages of 10,000 iterations are computed for various cities arranged in ascending order of distance between the client and server.

From the results, one can also observe that the the difference in latency between WiFi and 4G is the greatest in Aalborg where the distance between the server and the receiver is 0 km. This is because when it is WiFi, the client and the server are on the same network, which means a fewer number of hops that the packets need to travel through, as can be seen in figure 4.5. Based on some tests, this was not the case for 4G. This is likely due to the fact that the data must travel over a longer path which involves passing through the cell tower, fronthaul, backhaul, core network, and then the target data network, which may also comprise multiple hops.

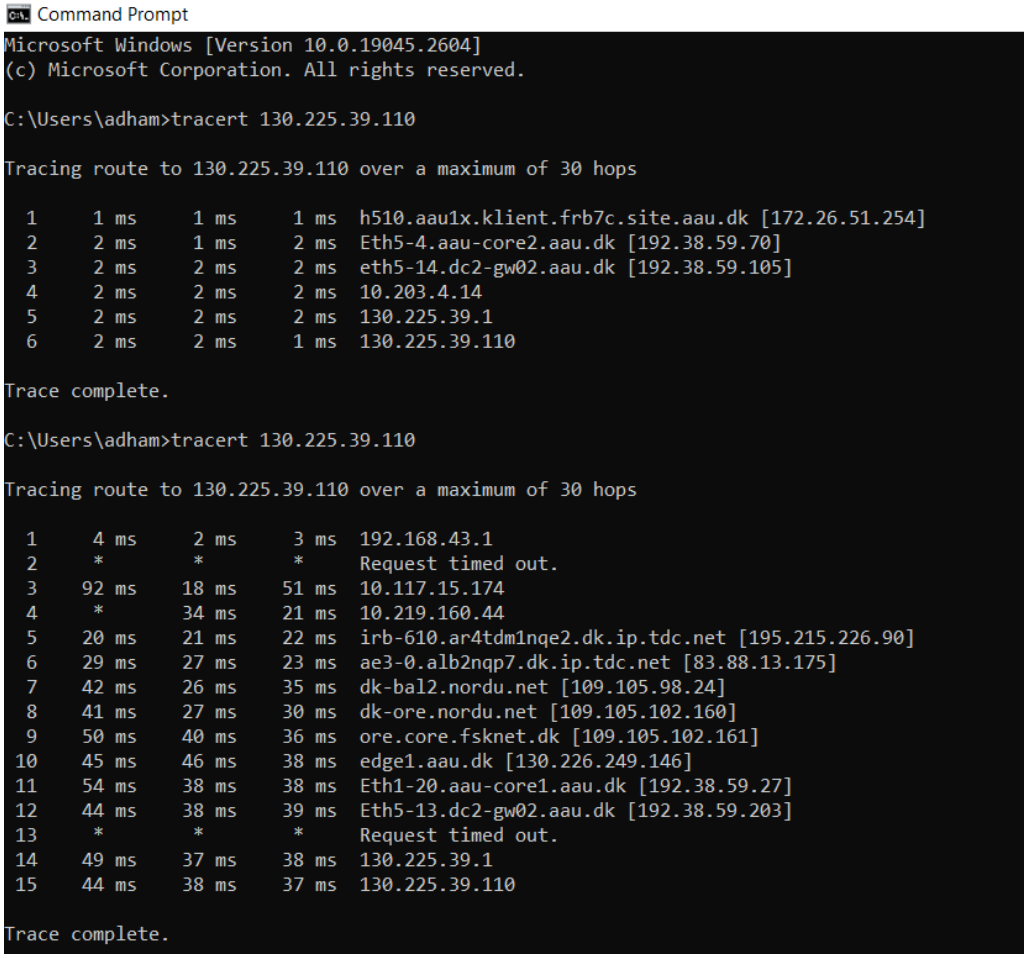

Figure 4.5. Tracert output displaying two different paths and latency of data packets through various network nodes - from the top, one for WiFi and the other for 4G.

File size

The effect of the file size has been evaluated by conducting multiple tests using three distinct file sizes, specifically 1, 10, and 30 kB. From the graph in figure 4.6, it can be seen that as the file size increases, t_2 and t_{total} also increase, since larger files means more data to be transmitted. Similarly, larger file size causes an increase in t_3 and t_4, since more data need to be decoded and stored in the storage media. Additionally, the results show that larger file size have a greater impact on t_2 than t_3 and t_4. This can be explained by the fact that the transmission delay of larger file sizes is mainly affected by bandwidth, which may result in a larger transmission delay. The decoding as well as the storing time however depend on the speed of the CPU, the decoding algorithm, and storage medium, which may be less significant depending on the size of the file.

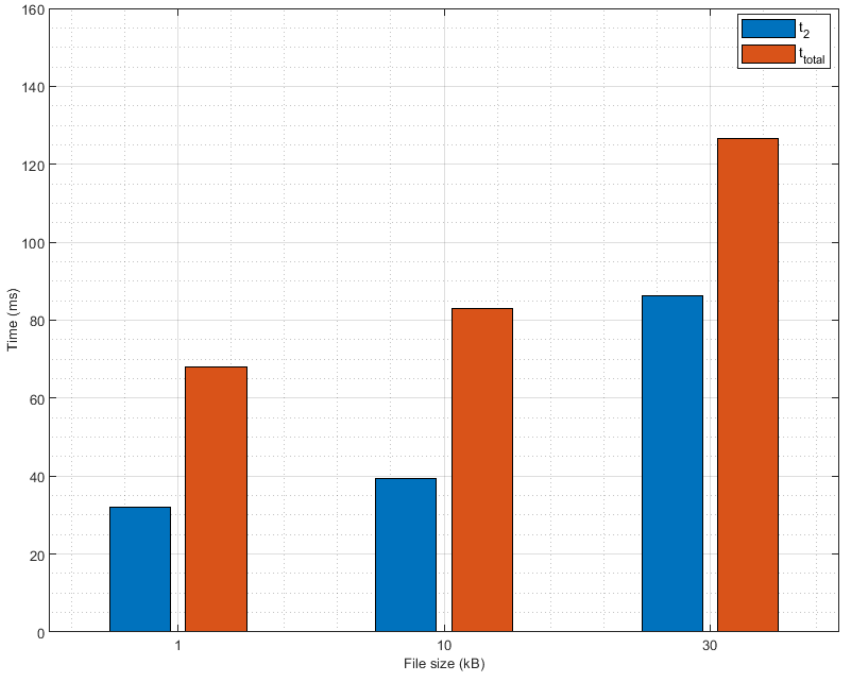

Figure 4.6. Graph depicting the impact of the file size on t_2 and on t_{total}, where the averages of 1,000 iterations are computed for the server located in Frankfurt.

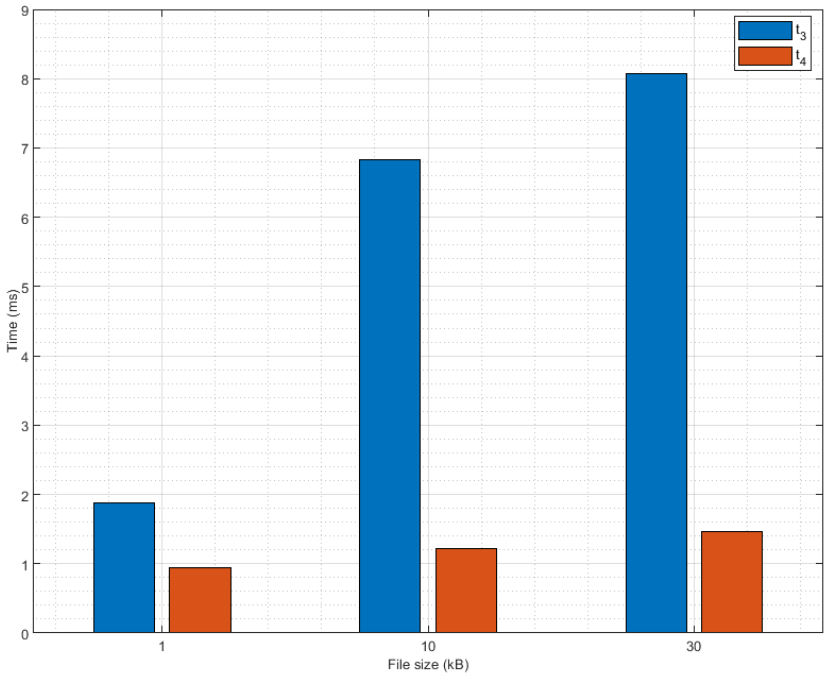

Figure 4.7. Graph depicting the impact of the file size on t_3 and t_4, where the averages of 1,000 iterations are computed for the server located in Frankfurt.

Received Signal Strength

To evaluate the quality as well as the performance of a network in wireless networking, the Received Signal Strength Indicator (RSSI) is commonly used. RSSI is a metric of the signal strength in decibels per milliwatt (dBm), between an access point and a wireless device. In order to study the impact of RSSI on t_2 and on t_{total}, a number of experiments were conducted, where the client was positioned at various distances from the access point using measuring tape. At each position, 200 iterations were performed, where the client requested and downloaded a video file of size 10 kB from the server located in Frankfurt. Furthermore, the RSSI value was obtained at each iteration using 'netsh', a tool that can extract wireless network information from a wireless device. As observed from the graph in figure 4.8, the findings show a clear positive correlation between the distance and the latency (the red line), as well as a negative correlation between the distance and the RSSI value (the blue line).

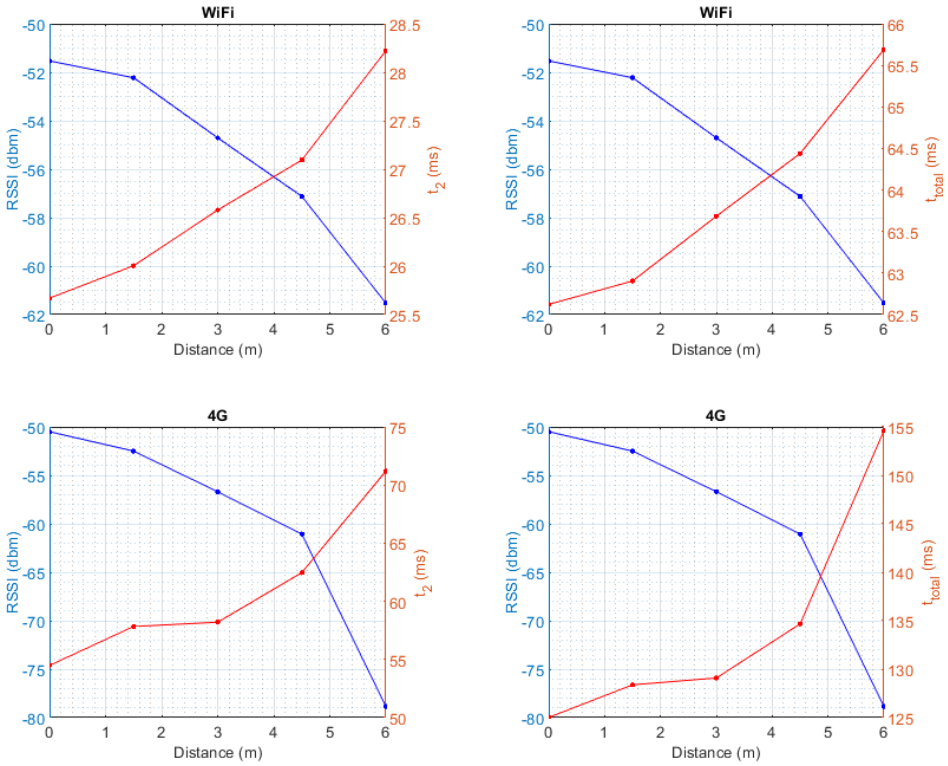

Figure 4.8. Graph depicting the relationship between RSSI, the distance between the client and the access points (WiFi/4G hotspot), t_2 and t_{total} after performing 200 iterations, at each point, to the server located in Frankfurt for 10 kB file size.

Furthermore, it was also noticed that when using a 4G hotspot, the RSSI value decreased at a faster rate compared to when using WiFi. This can be due to many reasons, one of which is the difference in transmission power between a 4G hotspot and a WiFi router, with the latter having a higher power output than the former.

To understand why different access point technologies may behave differently in the context

of latency, the downlink rate corresponding to each distance was obtained using the `netsh` tool. The results show that the downlink rate for WiFi is much higher compared to for 4G hotspot, as can be seen in figure 4.9. It was anticipated that this trend would occur since a 4G hotspot has smaller antennas compared to WiFi, which resulted in lower bandwidth as well as weaker signal, thereby leading to a lower transfer rate. The discrepancy in the downlink rates between WiFi and 4G in addition to the factors mentioned in technology section 4.2 contribute to higher latency when using a 4G hotspot.

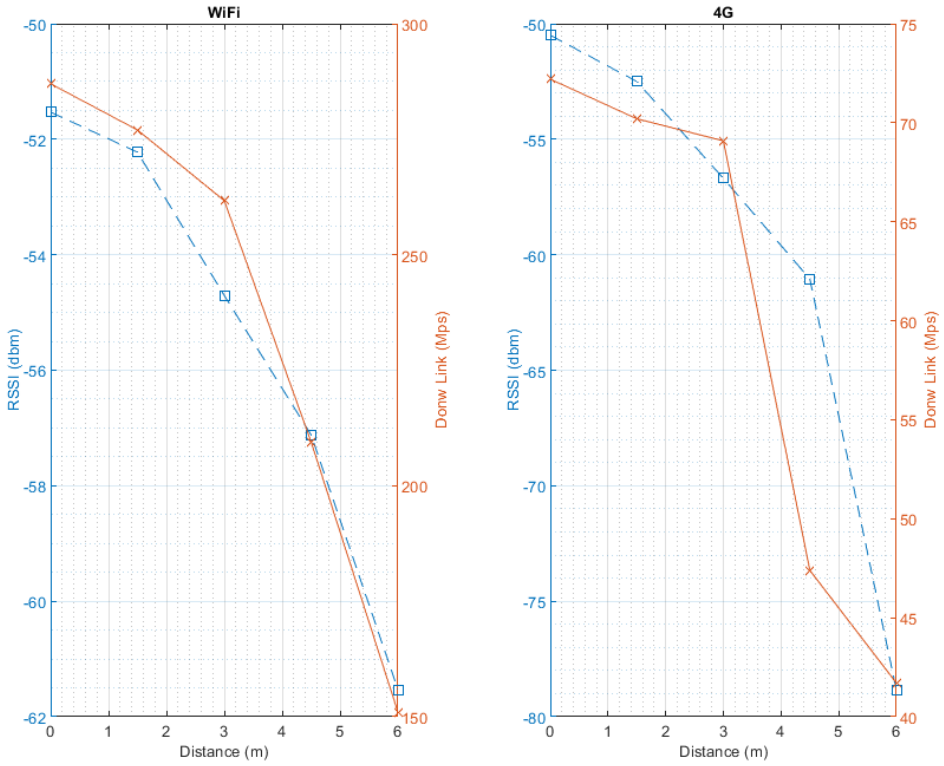

Figure 4.9. Graph depicting the relationship between RSSI (the blue line), the distance between the client and the access points (WiFi/4G hotspot), and the downlink rate (the red line) after performing 200 iterations, at each point, to the server located in Frankfurt for filesizes of 10 kB.

In wireless networking, an access point applies a technique called Adaptive Modulation and Coding (AMC) which allows for bandwidth optimisation to achieve the best balance of reliability and data transfer speed. This technique works by adjusting the data transmission rate and the Modulation and Coding Scheme (MCS) based on channel conditions, i.e., received signal strength and noise level. This can be observed in figure 4.9, where the downlink rate decreases as the distance between the wireless device and the access point increases. The AMC algorithms may vary depending on the model and manufacturer of the access point. For instance, the WiFi standard used in this test is 802.11ax (also known as WiFi 6) uses Orthogonal Frequency Division Multiplexing (OFDM) as a primary modulation scheme, where the available bandwidth is split into multiple sub-carriers which are orthogonal to each other. This feature makes the wireless communication robust against intersymbol interference caused due to multipath propagation [Popovski, 2020,pp 300]. In addition, each sub-carrier is modulated using

same or different modulation schemes depending on the channel condition at each of these sub-carries. For example, at larger distances, where the channel condition is worse, the sub-carriers may be modulated with Quadrature Phase Shift Keying (QPSK), a more robust scheme but with lower transmission rate (i.e. 2 bits per symbol). Whereas, at smaller distances, the channel condition is better, and therefore the sub-carriers can be modulated with higher modulation schemes up to 1024 Quadrature Amplitude Modulation (1024-QAM), resulting in a higher data transmission rate (i.e. 10 bits per symbol). This results in a lower latency at smaller distances. Therefore it was decided to add the downlink rate as an input that affects t_2.

Transmission protocol

Both TCP and UDP have been tested in order to asses their impact on t_1, t_2 and t_{total}. The test results confirm the expected outcome that downloading via UDP is faster compared to TCP, as can be seen from the graph in figure 4.10. The observed discrepancy in latency between UDP and TCP might be explained by the fact that, unlike TCP, UDP is a connectionless protocol meaning that t_1 is equal to 0. Furthermore, neither error correction nor retransmission is required when using UDP.

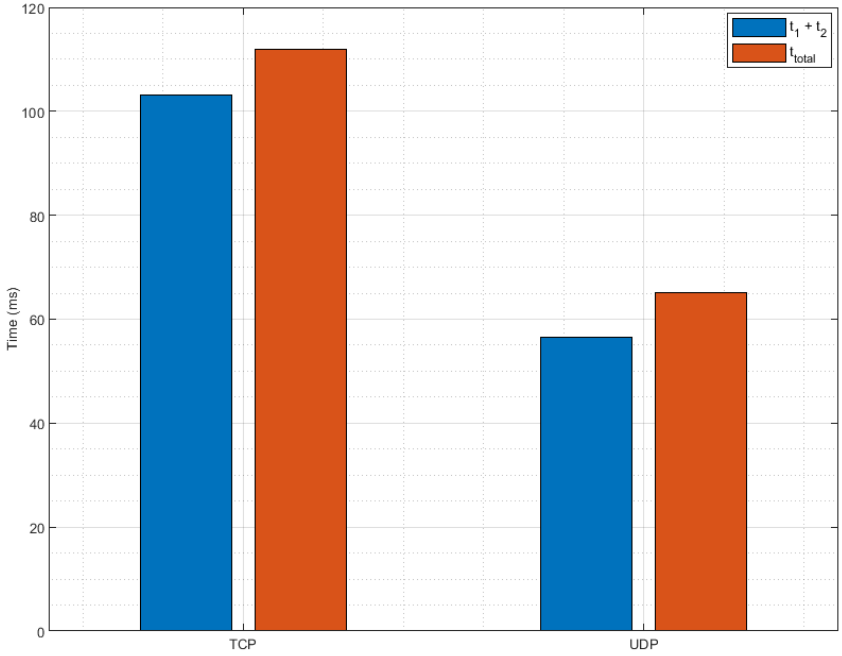

Figure 4.10. Comparison of TCP and UDP latency for downloading a 30 kB file, where the averages of 1,000 iterations are computed for the server located in Frankfurt.

Maximum Transmission Unit and Maximum Segment Size

Maximum Segment Size (MSS) refers to the largest amount of data that can be accommodated within a single segment in a network. The MSS value is negotiated upon the establishment of TCP connection between the communicating entities, namely, during the TCP three-way handshake and prior to transmitting any data. This is to ensure, for

the devices capabilities as well as the network conditions, that the payload of each packet is optimised with the intention of achieving reliable and efficient data transmission. The MSS value is controlled indirectly through the Maximum Transmission Unit (MTU), which is the maximum amount of data that can be transmitted within a single packet over the network. As shown in figure 4.11, the MTU comprises the MSS value in addition to 40 B, which function as the IP and TCP headers. For instance, to attain an MSS of 700 B, the MTU should be adjusted to 740 B.

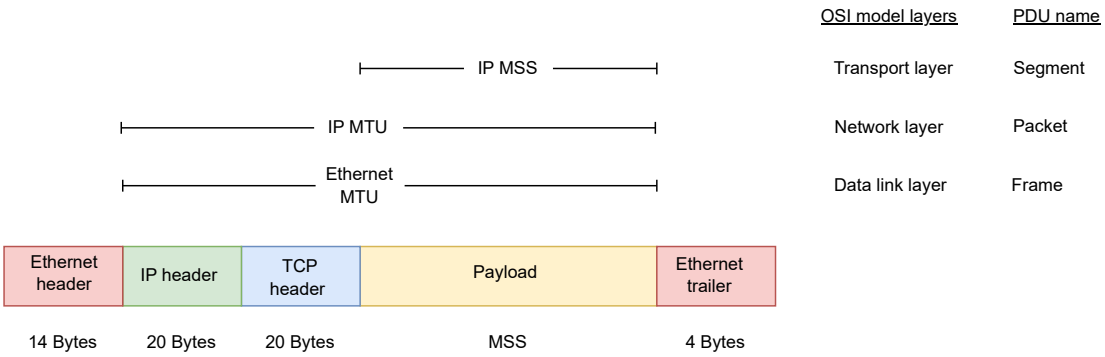

Figure 4.11. Comparison of Maximum Segment Size (MSS) and Maximum Transmission Unit (MTU) for IP and Ethernet protocols in the context of OSI model layers and their corresponding Protocol Data Unit (PDU) names.

In order to assess the impact of the MSS on t_2 and t_{total}, three different MSS values have been tested, namely 300, 700, and 1300 Bytes. Additionally, during these tests, the client consistently requested the same file size, i.e. 10 kB. As observed in the graph in figure 4.12, the findings indicate that decreasing the MSS value leads to an increase in the values of t_2 and t_{total}. This can be attributed to the fact that when the file size exceeds the MSS value, a smaller MSS requires more transmissions to send the entire file. Figure 4.13 depicts a Wireshark screenshot capturing a data transmission from the server to the client, providing clear evidence that the number of packet transmissions increases when the MSS is smaller, which results in a higher t_{total}.

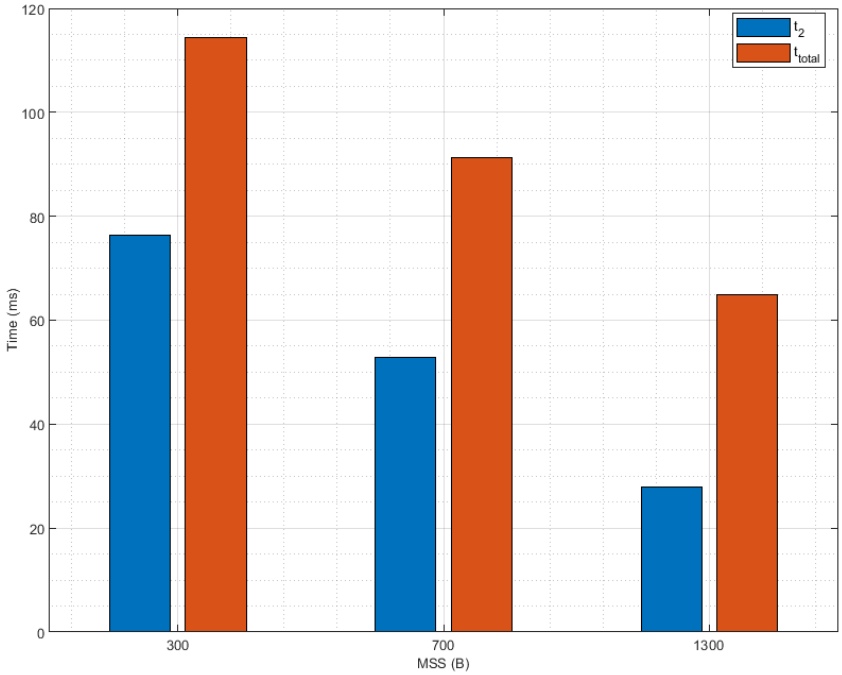

Figure 4.12. Graph depicting the impact of the MSS on t_2 and eventually on t_{total}, where the averages of 1,000 iterations are computed for the server located in Frankfurt.

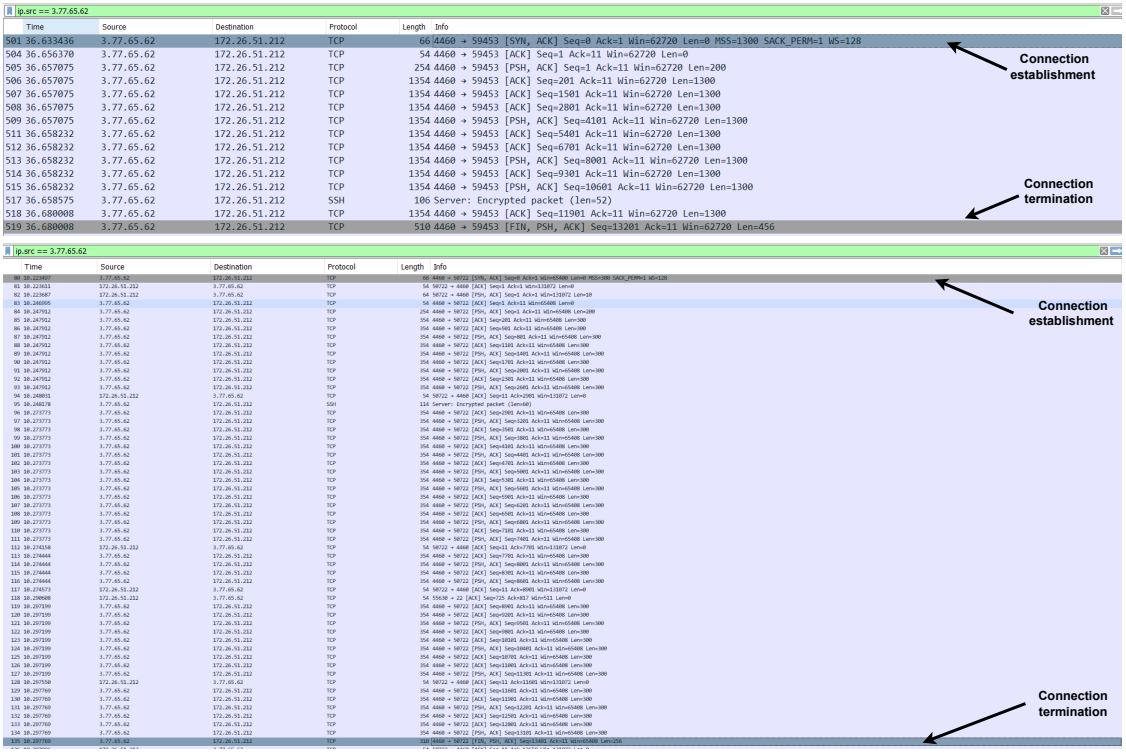

Figure 4.13. Comparing packet transmission count to complete a single 10 kB file download using MSS values of 1,300 (top block) and 300 (bottom block).

When using UDP, the MSS does not apply since no connection negotiation is applied

between the transmitter and the receiver. Thus, to ensure a fair comparison between UDP and TCP when selecting a MSS value during the operations, the server will send the requested file in chunks when UDP is used. These chunks will be equal in size to the selected MSS value.

Encoding algorithm

Before transmitting a video file over any network connection, it is crucial to encode it. This has many purposes, including increasing the transmission efficiency, enhancing security, as well as enabling more accessibility for the receivers in order to view the video on their devices. There are many encoding techniques, each of which have their own advantages and disadvantages. Additionally, choosing the most appropriate technique for a given use case depends on several factors. These include the size and the type of the data, the desired objective of encoding, the data type which the network link support, as well as the available resources. To asses how different encoding schemes may affect t_2, t_3 and eventually t_{total}, two encoding schemes have be tested, namely Base64 and binary. As seen in the graph in figure 4.14, compared to Base64, the binary scheme causes more delay in t_2, t_3 and t_{total}. This is in line with the expectation since Base64 encoding increases the file size by 33%, while binary encoding increases the size by a factor of eight. This is because each byte in the file is represented as a sequence of 8 bits when using binary encoding. Whereas, when using base64, each 3 bytes of the data are represented by 4 bytes. Overall, the binary scheme introduces more delays since more data should be encoded at the transmitter then transmitted and finally decoded at the receiver.

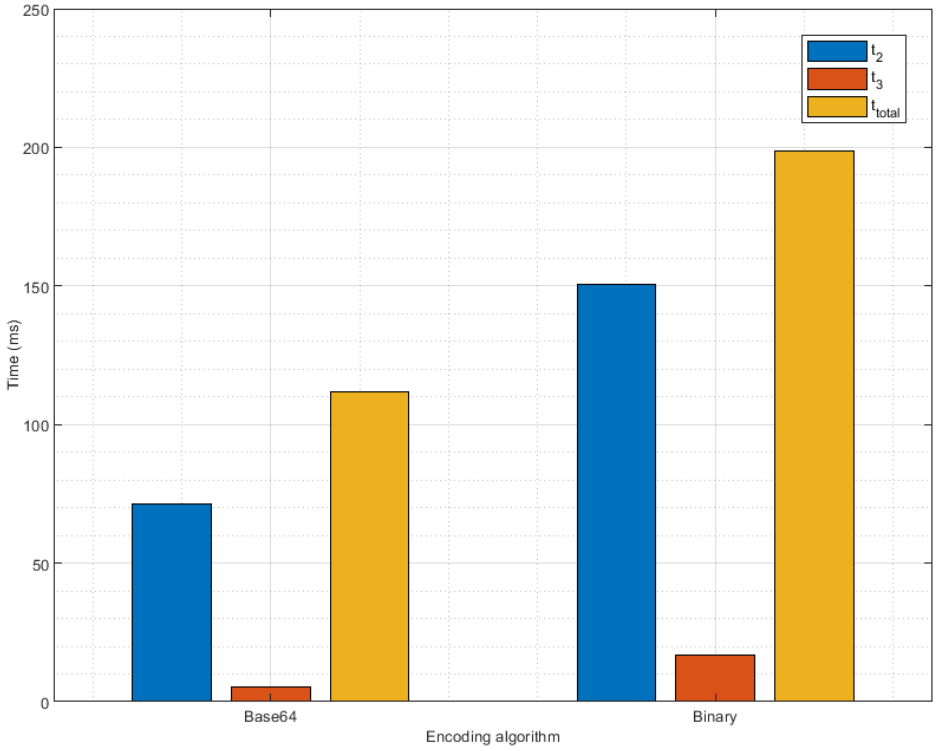

Figure 4.14. Comparison of the time delays (t_2, t_3 and t_{total}) using two encoding algorithms (Base64 and binary) for video transmissions of 30 kB.

Another technique can be used on top of the encoded data is file compression. This technique makes the transfer of a file faster by removing the redundant data and optimising the encoding method which leads to a decrease in the size of the file. However, compressing a small file (i.e. 30 kB which is the largest file used in the PT) may not even lead to a notable decrease in file size. Nevertheless, when applying the binary scheme to a 30 kB video file, the file becomes $\approx$ 246 kB where applying the file compression technique become more feasible. It was however decided to not apply any compression scheme because it can only be applied to one encoding scheme, which would result in an unfair comparison between the Base64 and binary schemes.

The number of transmissions

In general, when downloading a file, the number of transmission can affect the latency in several ways. Initially, each transmission entails certain quantity of overhead which has various functions like error detection and correction, addressing and flow control. This overhead increases with an increase in the number of transmission leading to a reduction in the quantity of the raw data transmitted as well as to an increase in the time required to download the whole file. In addition, the performance of the network may be affected since the extra overhead can contribute to network congestion. Consequently, the packets may be dropped or lost, leading to retransmission of the lost packets and thereby causing additional delays. In this project, it was previously shown (figures 4.12 and 4.13) that t_2 is affected by the MSS value, in which a smaller MSS would cause an increase in the number of segments required for the completion of file transmission. Nevertheless, there are other inputs that can affect the number transmission. These include the file size, the encoding algorithm, and the quality of the connection, as poor connection may cause packet loss. However, quantifying the impact of the connection quality on the number of transmissions may be challenging, since it requires a deep analysis of the network topology as well as the environmental conditions.

The expected number of transmission was derived by testing all the possible combinations of file size, MTU and the encoding algorithm. These tests involves downloading file under all possible distinct combinations of the aforementioned inputs and recording the number of transmissions necessary to successfully finish the download. The results of theses tests can be seen in table 4.3.

Table 4.3. Number of transmissions for all possible combinations of file size, MTU, and encoding.

File size (kB)			MTU (B)			Encoding		Transmission
1	3	30	340	740	1340	base64	binary	#
X	-	-	X	-	-	X	-	5
X	-	-	X	-	-	-	X	28
X	-	-	-	X	-	X	-	2
X	-	-	-	X	-	-	X	12
X	-	-	-	-	X	X	-	2
X	-	-	-	-	X	-	X	7
-	X	-	X	-	-	X	-	46
-	X	-	X	-	-	-	X	274
-	X	-	-	X	-	X	-	20
-	X	-	-	X	-	-	X	118
-	X	-	-	-	X	X	-	11
-	X	-	-	-	X	-	X	64
-	-	X	X	-	-	X	-	137
-	-	X	X	-	-	-	X	820
-	-	X	-	X	-	X	-	59
-	-	X	-	X	-	-	X	352
-	-	X	-	-	X	X	-	32
-	-	X	-	-	X	-	X	190

4.2.1 Summary

As all of the factors included in table 4.1 now have been analyzed, a new table 4.4 has been created to include the type of input as well as what values are possible.

Table 4.4. Inputs that have direct impact on the latency of the individual processes.

Factor	t_1	t_2	t_3	t_4	**Input type**	**Values**
Distance (Location)	X	X	-	-	Categorical	AAU, Fjerritslev, Frankfurt, Paris
Technology	X	X	-	-	Categorical	WiFi, 4G
Transmissions protocol	X	X	-	-	Categorical	TCP, UDP
File size (kB)	-	X	X	X	Numerical	1, 3, 30
Number of transmissions	-	X	-	-	Numerical	2 - 820
Encoding algorithm	-	X	X	-	Categorical	Base64, Binary
MTU (B)	-	X	-	-	Numerical	340, 740, 1340
RSSI	-	X	-	-	Numerical	0 - 1
Downlink rate (Mbps)	-	X	-	-	Numerical	40 - 300

Categorical inputs are inputs that are not numbers, and require further processing to use with a model, which is investigated in section 4.3, alongside how the inputs will be used in general.

4.3 Processing the model inputs

4.3.1 Numerical preprocessing

Numerical preprocessing is the process of transforming raw numerical data into data that is more suitable to build and train a machine learning model. It is often the first step when creating a model and the main objective of preprocessing data is to extract only the relevant information.

Normalisation is one such widely used numerical preprocessing technique. It is a commonly used for input features having a wide range or scale. This is because a large range in the input features can make it more difficult for the model to recognise the underlying patterns or correlation in a dataset, or confuse the model by misjudging the influence of a parameter, thus affecting the accuracy of the model. Normalising the data would ensure that all the input features fall within a similar range and helps reduce any bias in the model and helps reduce any bias in the model, by scaling the data to a fixed 0 to 1 range. The equation for normalising inputs is given by:

$$x_{norm} = \frac{x - \min(x)}{\max(x) - \min(x)} \tag{4.3.1}$$

where x_{norm} is the normalised input data, x is the input data, and $min(x)$ and $max(x)$ are the minimum and maximum values in the dataset respectively.

Table 4.5. Example tables showing how different filesizes appear when normalised.

Default

Filesize (kB)
1
10
30

Normalised

Filesize (kB)
0
0.3103
1

While normalising data is the best suited preprocessing technique for data that is not Gaussian or when the scale is widely different, it also means that the original scale of the data is lost. This can sometimes make it difficult to interpret the results, especially in cases where the results need to be compared to other data sets having different scales. Additionally, normalisation is also sensitive to outliers as it may compress the outlier value to either 0 or 1, making it indistinguishable from other values, thus skewing the distribution.

Another preprocessing technique that is less sensitive to outliers than normalisation is standardisation. Standardisation transforms the data to have zero mean and a standard deviation of one, meaning that all inputs have the same scale, therefore making it easier to compare and comprehend the influence of the different inputs on the final output. It is especially helpful in statistical models such as linear regression, where the model parameters such as weights and biases are evaluated using the mean and variance of the input values. The equation for standardising inputs is given by:

$$x_{stan} = \frac{x - \mu}{\sigma} \tag{4.3.2}$$

where μ is the estimated mean of the input data and σ is the estimated standard deviation of the input data.

Table 4.6. Example tables showing how different filesizes appear when standardised.

Default

Filesize (kB)
1
10
30

Standardised

Filesize (kB)
-0.8533
-0.2470
1.1004

In the event that the output values are the ones chosen to be numerically preprocessed, such as during training to potentially obtain better results, the process would need to be reverted to obtain the actual output. To revert the normalisation, the equation to obtain the input x again is given by:

$$x = x_{norm} \cdot (\max(x) - \min(x)) + \min(x) \tag{4.3.3}$$

Similarly, standardisation can be reverted with the following equation:

$$x = x_{stan} \cdot \sigma + \mu \tag{4.3.4}$$

While standardisation is less sensitive to outliers compared to normalisation as it does not directly use the minimum and maximum values of the input data, it can still be affected by extreme outliers which influence the standard deviation of the data. Therefore, in order to obtain a distribution that is not distorted, it is important to handle the outliers and this is further explained in subsection 4.3.4.

4.3.2 Categorical preprocessing

When dealing with categorical inputs, such as in this case different placements of the server and technology used, it is ideal to transform the data into a numerical value instead. This is because the model will not be able to parse "labels", such as countries.

Label encoding is the most straightforward way to approach this problem. Here, the label itself is replaced with a numerical value, from 1 to n, where n is the amount of total different labels. An example of this encoding applied can be seen on table 4.7.

Table 4.7. Example tables showing how categorical locations are encoded into labels.

Default

Location
Denmark
Frankfurt
Paris

Label encoded

Location
1
2
3

However, label encoding on categorical data means that different categories that may not be correlated will have some range of value instead. This would possibly be interpreted by the model as a numerical range or order, thus skewing the data.

Instead of just giving the categories any number, target encoding can be used to give a category the mean value of a target category, so "Denmark" could become the mean of all of the "Latency" values that include this category. An example of this can be seen on table 4.8.

Table 4.8. Example tables showing how categorical locations are target encoded.

Default

Location	Latency (ms)
Denmark	26.02
Denmark	23.84
Denmark	25.41
Paris	48.53
Paris	56.23

Target encoded

Location	Latency (ms)
25.09	26.02
25.09	23.84
25.09	25.41
52.38	48.53
52.38	56.23

However, if a country is not represented enough in the data, the resulting mean value might not be sufficient. To combat this, the target encoding can be smoothed by using the following equation:

$$\text{Denmark} = \omega \cdot \mu_{categorical} + (1 - \omega) \cdot \mu_{overall} \tag{4.3.5}$$

where $\mu_{overall}$ is the estimated mean across the entire Latency column, $\mu_{categorical}$ is the estimated mean for each country, and the weight ω, which is found by the following equation:

$$\omega = \frac{n}{n + m} \tag{4.3.6}$$

where n is the amount of appearances for said country, and m is the smoothing factor. The smoothing factor m is best chosen based on how noisy the data is, the more noisy the higher the smoothing factor [Kaggle, 2021]. As an example of smoothing, with $m = 2$ and an $\mu_{overall} = 32.524$, the countries from table 4.8 and their subsequent appearances n and categorical means $\mu_{categorical}$ would achieve the following values:

$$\begin{aligned} \text{Denmark} &= \frac{3}{3+2} \cdot 25.09 + (1 - \frac{3}{3+2}) \cdot 32.52 = 28.06 \\ \text{Paris} &= \frac{2}{2+2} \cdot 52.38 + (1 - \frac{2}{2+2}) \cdot 32.52 = 42.45 \end{aligned} \tag{4.3.7}$$

In essence, this weighs appearance frequency of each country in the location category. With the smoothing, every other input in that category also influences the encoding.

Another approach is to instead give the categorical data a "1" or "0" if it is present or active. This is called one-hot encoding, due to each category only having one active input

at a time. For example, on table 4.9, the location is turned from a country into a vector of either 1 or 0, depending on which is active. Only one country is active at a time, hence only one label is "hot".

Table 4.9. Example tables showing how categorical locations are one-hot encoded.

Default

Location
Denmark
Frankfurt
Paris

One-hot encoded

Denmark	Frankfurt	Paris
1	0	0
0	1	0
0	0	1

An apparent effect of one-hot encoding is that a column such as "country" would expand into one column for each country. This could then result in a slower training set, as more labels have to be processed. However, it does so without introducing new information about the labels.

A disadvantage of one-hot is the risk of multicollinearity, meaning two or more of the categorical inputs could be highly correlated. This can result in the model misinterpreting the data, for example if two of the countries chosen for the "location" category are much closer than any other possible inputs. Thus is it usually ideal to analyze the data for correlation followed by manual removal of one of highly correlating columns.

The removal of one of the columns means the absence of any categories, such as all of them being "0", means that the missing country is active. For example, a *0* in both Denmark and Frankfurt would equate the *1* in Paris, if the "Paris" column was to be removed in table 4.10. This removal of a column is also known as dummy encoding, although it is also common to just drop the first of the columns instead of checking for correlation.

Table 4.10. Example tables showing how categorical locations are dummy encoded.

Default

Location
Denmark
Frankfurt
Paris

Dummy encoded

Denmark	Frankfurt
1	0
0	1
0	0

Out of the three types of categorical encoding mentioned, it makes sense to include one-hot encoding as it does not add new information to the model, and target encoding as it adds information based on the t_{total} for each country. A thorough comparison of the encoding methods is conducted during chapter 6.

4.3.3 Training data splitting

In order to prepare the data for training and validation of the models, it must also be split into two categories: training and validation. This is done to ensure that the data used for validation has not been used to train the model before.

To accomplish this, the `train_test_split(input, output, train_size = 0.8)` scikit method is called, which splits the data into 80% data for training, and 20% for validation. The data is also randomly shuffled, so the order of the data when trained will not incur bias.

4.3.4 Outlier Analysis

Outliers are data points with values that are considerably different from the rest of the data points. This could occur due to a number of reasons, for example, measurement errors, the distribution being heavy-tailed, or just truly extreme values. However, outliers can significantly impact the accuracy of a model. This is because a model trains itself to recognise patterns given a certain distribution, but the presence of outliers can cause uncertainty or a larger degree of variance in the pattern, causing it to make biased or incorrect predictions.

Figure 4.15 shows the occurrence of outliers for the t_{total} at Frankfurt for 200 measurements. It can be assumed that the number of outliers will only increase with more measurements. The same is observed in other locations as well. Therefore, in order for the model to make dependable and precise predictions, it is imperative to identify and remove outliers from the dataset.

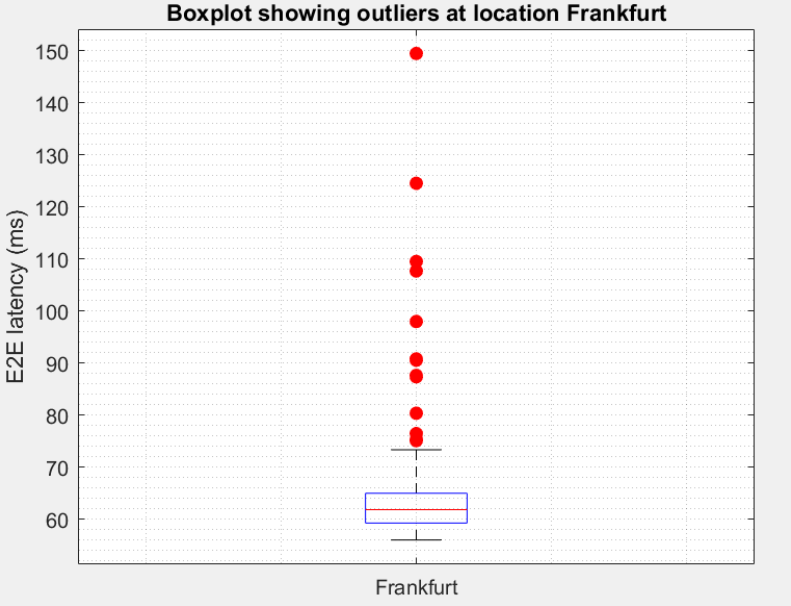

Figure 4.15. Boxplot showing outliers (red dots) in a distribution at Frankfurt.

Quantile intervals are one way of identifying potential outliers in a dataset. Quantile intervals are a range of values in a distribution that are attributed with a certain probability of occurring. In order to use quantile interval to find outliers, we can use the Interquartile Range (IQR). It is usually defined by two quantiles: Q1 and Q3, which represent the 25th percentile and 75th percentile of the dataset, respectively. The range of the IQR which gives the spread of the distribution is calculated as the difference between the 75th and 25th percentile i.e. Q3 - Q1. The IQR represents the range of middle 50% of the entire distribution.

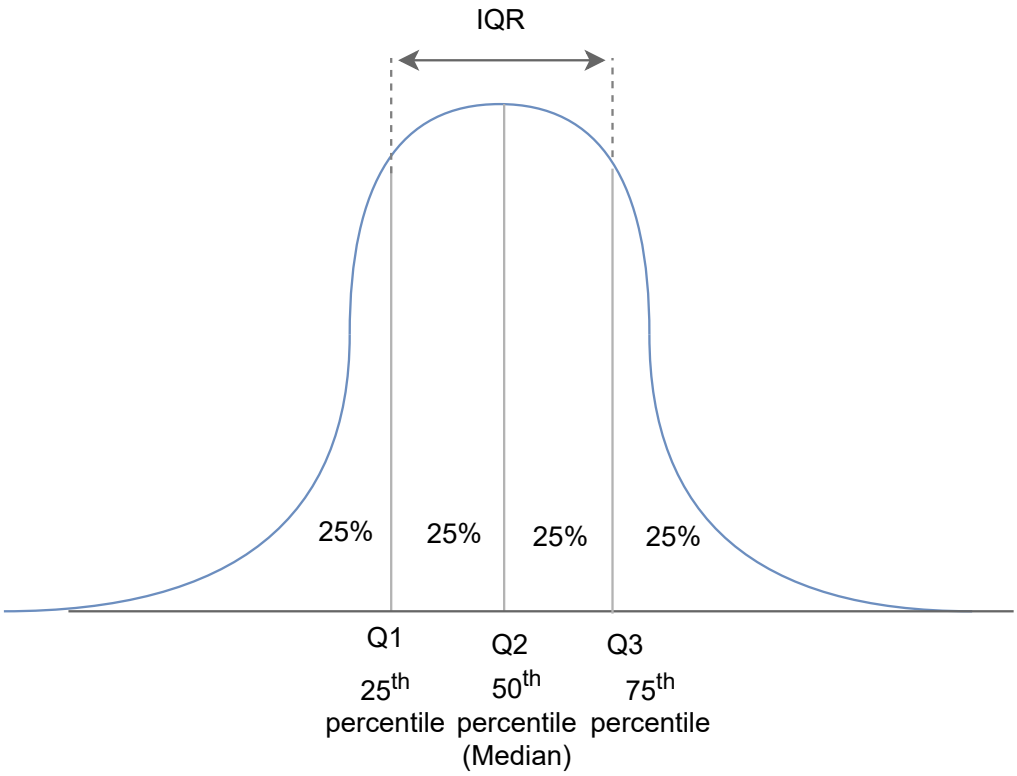

Figure 4.16. Quantile intervals.

To identify potential outliers using quantile intervals, a common practice in statistical analysis is to consider any data points that fall outside of the range Q1 - 1.5 IQR to Q3 + 1.5 IQR as potential outliers. Here, Q1 - 1.5 IQR is considered as the lower bound and Q3 + 1.5 IQR as the upper bound. The value of the multiplier (i.e., 1.5) can be varied depending on the distribution. As IQR considers only a range of values in the middle 50% of the dataset, the other extreme values outside this range do not have an impact. Standard deviation on the other hand is more sensitive to outliers than quantile intervals because it takes into account all of the data points in a dataset when calculating the variance in the data. This means that outliers, which are by definition extreme values that are far from the other observations in the dataset, can have a significant impact on the value of the standard deviation. Thus, IQR is better suited for distributions that are not normal.

As the data obtained for this project indicates that it is not normally distributed based on the results from normality tests, using IQR to detect outliers is considered more suitable. The results of eliminating outliers using both IQR and fixed quantiles are presented below to provide additional evidence as to why IQR was selected for this project. The data that is collected to train the models is segregated based on the location, technology, and MTU size and is saved into individual .csv files, which are merged into a single dataset during training. It is important to note that the outlier removal techniques are applied on each file as well as the combined final dataset. This is because the outliers present in the individual files may not appear as outliers when merged together, leading to inaccurate relationships between the data. When applying fixed quantiles for outlier elimination, a 95% quantile range was applied for each of the files, with the remaining 5% of the data

potentially being classified as outliers and on merging all files, a 99% quantile range was applied, while the rest of the 1% data being considered outliers. The figure 4.17 shows the results after applying fixed quantile to the dataset.

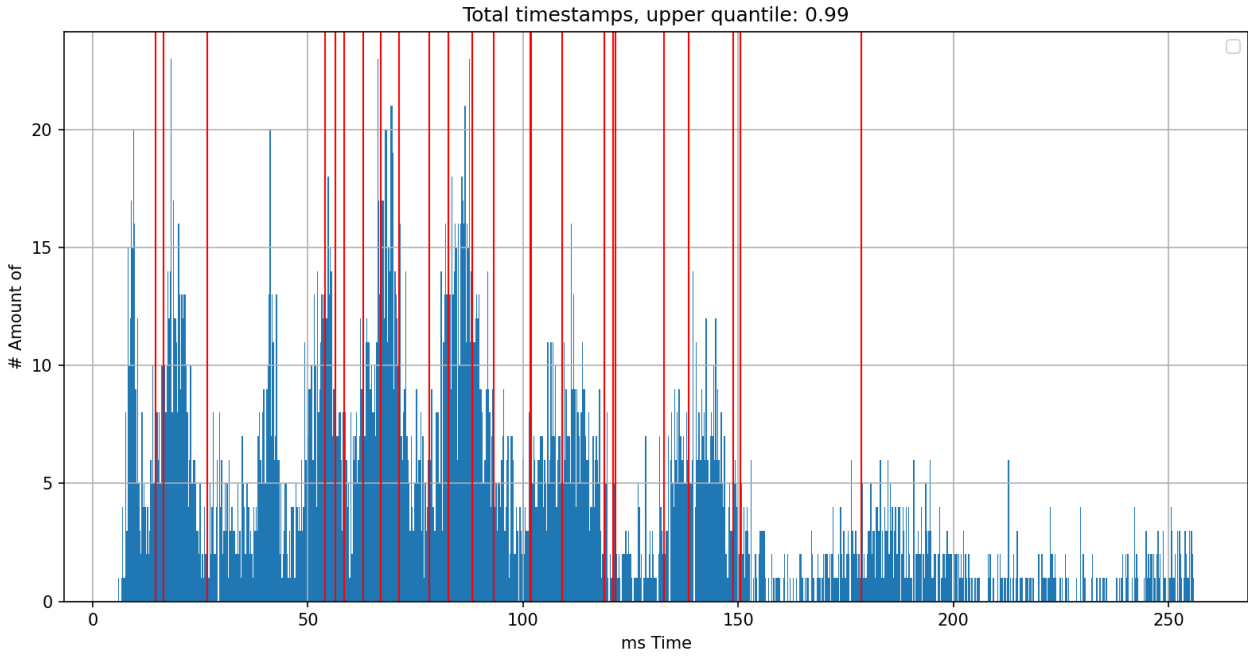

Figure 4.17. Data set after using fixed quantiles to eliminate outliers, with the red lines representing the means of the individual .csv files

Next, the IQR method was applied both before and after combing the files, and the figure 4.18 shows the results for the same.

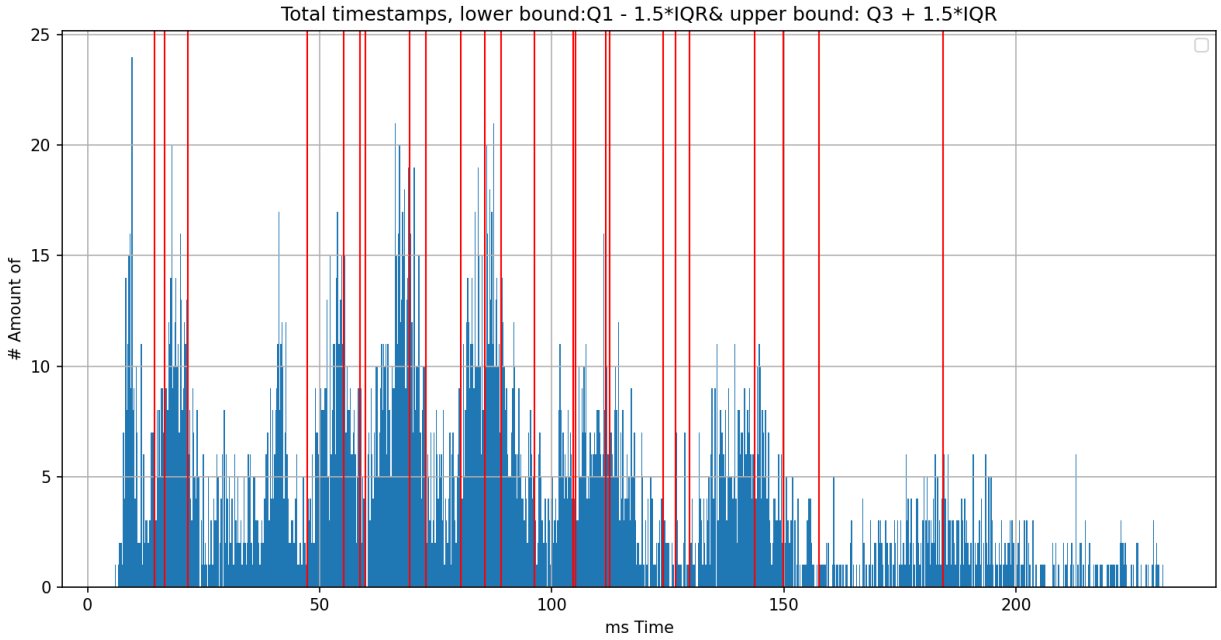

Figure 4.18. Data set after using fixed quantiles to eliminate outliers, with the red lines representing the means of the individual .csv files

Table 4.11. MSE and MAE obtained for different outlier removal techniques.

Outlier removal tenhnique	**MSE**	**MAE (ms)**
Fixed quanitle	0.640	26.50
IQR	0.631	24.01

Therefore, figures 4.17 and 4.18 show that the IQR method is only considering data points up to 230 ms as non-outliers while the fixed quantile method is considering data points up to 250 ms as non-outliers, this shows that the IQR method is more stringent in removing outliers and is potentially removing some data points that are not considered outliers by the fixed quantile method. Further, the results in table 4.11 show that model performs marginally better when applying the IQR method as compared to using fixed quantiles, with a reduction of 2.5ms in the absolute error measurement. Therefore, this project will be using the IQR method on the dataset to detect and eliminate outliers.

Performance Evaluation and Optimisation 5

This chapter expands on the design decisions described in chapter 4 and details how all prior knowledge is used to create the final models. After this chapter, these models are tested in order to see how they compare, both in time prediction accuracy and in execution time.

The first section includes details on how a hyperparameter tuner was used to get the most out of the model training. After that, an optimisation section is included where additional common methods to improve NN models and data are attempted. Finally, the third section compares how the choice of data preprocessing affects the model prediction.

At the end of the chapter is a summary section which briefly mentions all the discoveries and how the final models are made.

Similar to chapter 4, the values given in results and tests for this chapter are based on preliminary models and methods.

5.1 Model Tuning and Performance

Part of the keras training includes setting a number of so-called "hyperparameters", which are parameters initialised prior to the actual training. These parameters include the learning rate, quantity of nodes and hidden layers, loss functions, activation functions, optimization functions, batch size, and epochs. The choice of the hyperparameters have an impact on not only the time it takes to train each model, but also how the model will be expected to perform during and after the training. These hyperparameters were introduced and explained in section 2.3.1.

An example of the terminal for the model fitting, e.g. training, is included on figure 5.1. This shows the process and results for the first 6 out of 25 epochs, including how much time has elapsed, what the loss and metrics are for the training data, and similarly for the validation data (hence the val_ prefixes). The latter validation results are not used for training itself. In addition, the "113/113" represents the 113 batches that had to be parsed for training before the entire training set was used.

```
Epoch 1/25
113/113 [==============================] - 3s 18ms/step - loss: 0.5700 - mean_squared_error: 0.5700 - val_loss: 1.0392 -
val_mean_squared_error: 1.0392
Epoch 2/25
113/113 [==============================] - 2s 18ms/step - loss: 0.5178 - mean_squared_error: 0.5178 - val_loss: 0.8620 -
val_mean_squared_error: 0.8620
Epoch 3/25
113/113 [==============================] - 2s 16ms/step - loss: 0.5098 - mean_squared_error: 0.5098 - val_loss: 0.7253 -
val_mean_squared_error: 0.7253
Epoch 4/25
113/113 [==============================] - 2s 15ms/step - loss: 0.5194 - mean_squared_error: 0.5194 - val_loss: 0.6072 -
val_mean_squared_error: 0.6072
Epoch 5/25
113/113 [==============================] - 2s 18ms/step - loss: 0.5168 - mean_squared_error: 0.5168 - val_loss: 0.4921 -
val_mean_squared_error: 0.4921
Epoch 6/25
113/113 [==============================] - 2s 19ms/step - loss: 0.5126 - mean_squared_error: 0.5126 - val_loss: 0.3738 -
val_mean_squared_error: 0.3738
```

Figure 5.1. Screenshot of the Tensorflow training process, showing the loss value change across multiple epochs. The loss and validation loss results are shown twice due to the setup with Tensorflow.

While it may seem obvious to have as many epochs as possible to keep training the model, there may be a time during the training where the validation loss value will stop improving, and possibly get even worse than the training loss values. This phenomenon is called overfitting, and occurs when the model gets too biased by being exposed to the same data too much, meaning it will perform worse on new and different data. This also means that it loses the ability to generalise.

In order to prevent overfitting, methods like early stopping and `Dropout` layers are commonly utilised. Early stopping may require some manual training and expectancy of what the results should reach, but can also be implemented with a early stopping callback. If given a `patience = ""` argument, it will prompt the model training to stop if no improvement was had over a certain amount of epochs. Finer adjustments can also be set, where it should reach a certain loss value before stopping, or even return to previous weight values. Dropout layers on the other hand can be implemented using `tf.keras.layers.Dropout()`, which randomly sets the inputs of nodes to 0, effectively dropping them. This has the benefit mitigating nodes correlating to the same result.

However, when attempted on NN2 from Model 4 and 5, the addition of dropout layers with a drop rate of 50% resulted in worse results, increasing the validation MSE from 0.1427 to 0.2183, while 20% instead increased it to 1.6110. However, it did result in the training MSE never surpassing the validation MSE, meaning that overfitting and bias was eliminated. Because of this worsened performance, dropout layers are not used going forward.

An alternative to using these aforementioned methods is to also "train" which hyperparameters are used.

5.1.1 Hyperparameter tuning

While it is common to manually choose the parameters and then adjust them after observing the model performance, it is also a possibility to automate the process, for example using the `keras-tuner` module for Python. The process for initialising and getting the final model can be seen on figure 5.2.

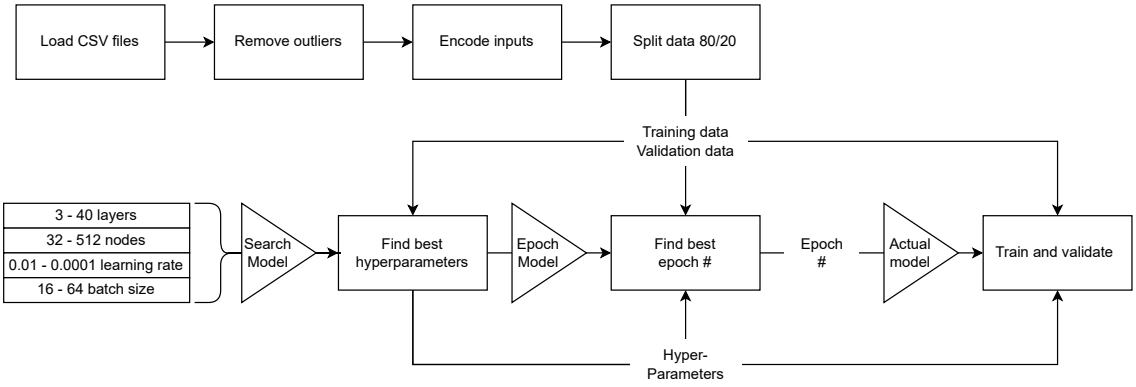

Figure 5.2. A diagram showing the process from preparing the test data to using in for model tuning and training.

`keras-tuner` training can be initiated by creating a *Search Model* with the `RandomSearch()` tuner call, which randomly tries different combinations of hyperparameters given in the model, such as 0.01, 0.001 or 0.0001 for the learning rate. The amount of different unique combinations of hyperparameters depends on the given `trials` argument, and each trial can be tested X number of times using the `executions` argument. The best outcome of any of these executions is then used to compare one trial with another, in order to find the best random combination of hyperparameters.

The code used for creating the hypertuning models is shown in snippet 5.1, which initializes a tuner with 3 to 10 layers of 32 to 512 nodes, has a learning rate of 0.01, 0.001, or 0.0001, and a batch size of 16 to 128. For this project the metrics, optimisation, and activation functions are static.

```
class MyHyperModel(kt.HyperModel):
    def build(self, hp):
        model = keras.Sequential()
        model.add(Flatten())
        for i in range(hp.Int('num_layers', 3, 10)):
            model.add(Dense(units=hp.Int('units_' + str(i),
                                         min_value=32,
                                         max_value=512,
                                         step=32),
                            activation='relu')
                      )
        model.add(Dense(1, activation="linear"))
        model.compile(
            optimizer=keras.optimizers.SGD(
                hp.Choice('learning_rate', [1e-2, 1e-3, 1e-4])),
            loss="mse",
            metrics=["mean_squared_error"],
        )
        return model

    def fit(self, hp, model, *args, **kwargs):
        return model.fit(
            *args,
            batch_size=hp.Choice("batch_size", [16, 32, 64, 128]),
            **kwargs,
```

```
        )
```

Snippet 5.1. Python code for a Keras Tuner model

When a `kt.RandomSearch()` model is initiated given the above criteria, it can be set to `tuner.search()` for the best parameters, by randomly training with the possible parameters. Based on the best loss value observed, the hyperparameter set is chosen, as seen in the example snippet 5.2.

```
Search: Running Trial #1

Value            |Best Value So Far |Hyperparameter
6                |6                 |num_layers
224              |224               |units_0
448              |448               |units_1
64               |64                |units_2
0.01             |0.01              |learning_rate

...

Trial 1 Complete [00h 00m 37s]
val_mean_squared_error: 0.3339455723762512

Best val_mean_squared_error So Far: 0.3339455723762512
Total elapsed time: 00h 00m 37s

Search: Running Trial #2

Value            |Best Value So Far |Hyperparameter
4                |6                 |num_layers
32               |224               |units_0
64               |448               |units_1
160              |64                |units_2
0.0001           |0.01              |learning_rate
480              |32                |units_3
192              |32                |units_4
288              |32                |units_5
16               |16                |batch_size
```

Snippet 5.2. Example of how the Keras Tuner finds the best hyperparameters.

When presented with a large amount of possible hyperparameter combinations, it makes sense to set a suitably large amount of trials to more likely find the best combination. However, due to the nature of the previously mentioned `RandomSearch()`, it is not necessary to test every single combination, as more often than not it may wind up finding a "good enough" combination that only improves the loss ever so slightly when the process was continued. This was further observed when testing a possible 1500 unique combinations lead to a MSE result that was 0.05 higher than a MSE score found with 50 trials. It was also found that the `RandomSearch()` may sometimes repeat previously attempted parameter combinations, even if the search algorithm for Keras Tuner supposedly should prevent this.

When doing searches with the data and models presented in this report, it was generally found that the Tuner results would converge towards a number of hidden layers under 6, with 256 nodes per layer, a learning rate of 0.01, and a batch size of 32. Attempts to

make the amount of layers large resulted in progressively worse results, with 20+ layers generally not progressing at all. Similarly, an amount of nodes per hidden layers that was larger or smaller than 256 would result in slightly worse results, albeit with a much smaller influence than the number of layers. The learning rate would only reach the seemingly "best" performance when set to 0.01, where anything lower would result in significantly slower changes that did not reach the minima loss even after 100 epochs at times. Finally, the batch size gave best results of 32, but performed much faster at higher numbers. As this is a one-time process for each model, the batch size is kept.

The best hyperparameters are then used for the *Epoch Model*, which tries to fit the training data with a number of epochs. Just like with regular model training, hyperparameter tuning may also be subject to issues such as overfitting. Because of this is, it is convenient to include ways to either stop early, or save information on previous model epochs that may perform best. In this case the result metric for each epoch is saved, so that the epoch with the best result metric can be chosen. For example, the number of epochs with the lowest MSE would be chosen.

When both the hyperparameters, best number of epochs, and training data is prepared, the actual model can be trained. This process is used for every single individual model for the different combinations described in the use case in chapter 3.

5.1.2 Validation

When a model has been tuned and later trained, it is ready to be validated. During the training, the model can be fed the validation data set in order to see how it performs on data that it was not previously trained on. This is particularly useful for seeing if the model has become biased towards the training data, by seeing if the training and validation results are significantly different. However, it is also relevant to see exactly how the model would perform when actually given the task of predicting the latency. To do this, the `model.predict()` method can be used, which given an input `x` with the same shape and types as it was trained on, will provide the desired outputs.

In the event that data has been preprocessed, it must be reverted before it can be compared with real data, as the preprocessing is only required during the actual training. As is the case with this implementation, some of the inputs/outputs are standardised during training in order to get better model performance, meaning that this should also be done for the real predictions with the final model. Rather than obtaining new mean and standard deviation values during the live prediction, the ones obtained during the training can be used, provided the circumstances for the data are the same as when the training data were obtained.

As an example, the NN1 from Model 4 and 5 as explained in chapter 3 has been created. This NN gives an output $\hat{t_1}$, and when compared to the true time t_1, it gives a result like in figure 5.3. Here it can be seen that as NN1 only has the location and technology as inputs, meaning eight different possible combinations (four locations, two technologies), very few values of predictions are possible. These therefore also do not give a good representation of how much t_1 actually varies. The transmission protocol is also considered an input, as seen in table 4.4, but when UDP is used, there will be no connection process, and t_1 would

as a result be 0.

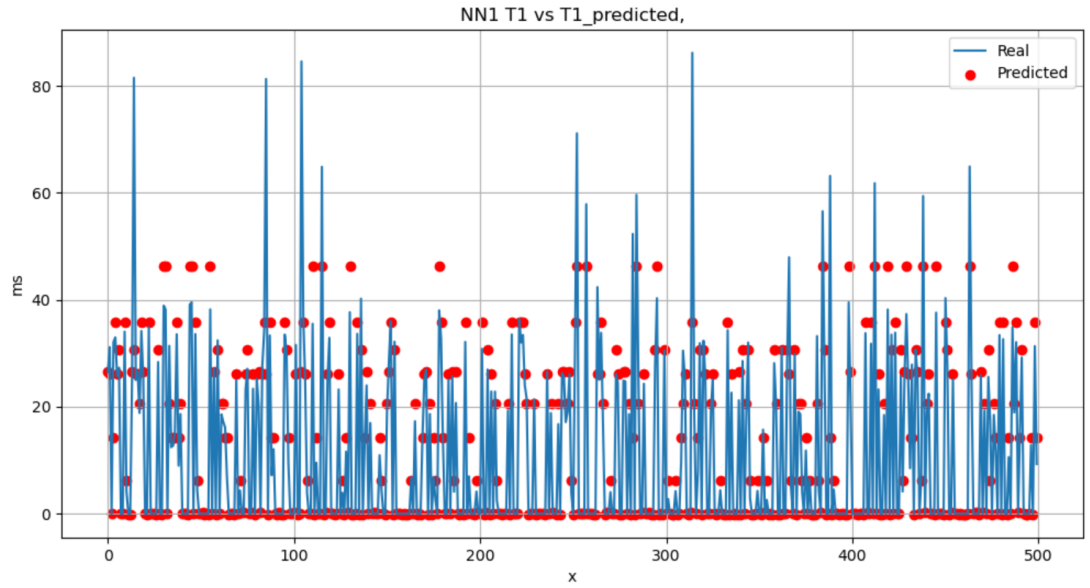

Figure 5.3. Graph showcasing the real t_1 measurements (blue line) and predictions $\hat{t_1}$ (red dots).

5.1.3 Amount of inputs used

The initial tests of the NN1 performance showed that the small amount of categorical inputs lead to only eight possible prediction times, resulting in poor predictions. However, with NN2 from Model 4 in section 3.1.3 there is the possibility of having up to nine different inputs as shown in table 4.4, as NN2 is the most significant part of the PT, encompassing both the request and receive processes. Therefore, owing to the larger number of inputs involved, the NN2 model is deemed the most appropriate to demonstrate the effect of input quantity on the loss function and the predictive ability.

Below, an example of t_2 and the predicted $\hat{t_2}$ being graphed for NN2 when only four out of the total nine possible inputs from table 4.4 are used to train the model is shown in figure 5.4. The four inputs used are distance, technology, file size and MTU.

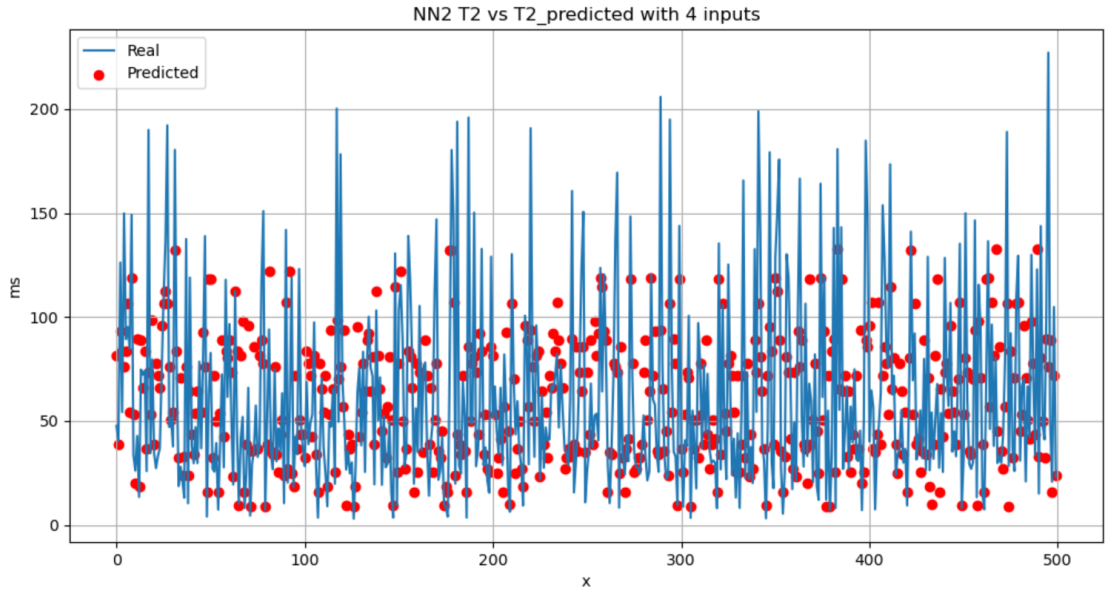

Figure 5.4. Graph showcasing the real t_2 measurements (blue line) and predictions $\hat{t_2}$ (red dots) when only four inputs are chosen.

The MSE obtained for the model along with the MAE in ms is also provided in table 5.1.

Table 5.1. MSE and MAE obtained for NN2 from model 4 with four inputs.

No. of inputs considered	**MSE**	**MAE (ms)**
4	0.5130	23.32

Next, the same model is tested on all the inputs available for NN2, namely: distance, technology, file size, MTU, protocol, encoding, RSSI, downlink rate, and number of transmissions. The addition of these five inputs brings the total number of inputs to nine, compared to the previous number of four. The results from this test is shown below in figure 5.5.

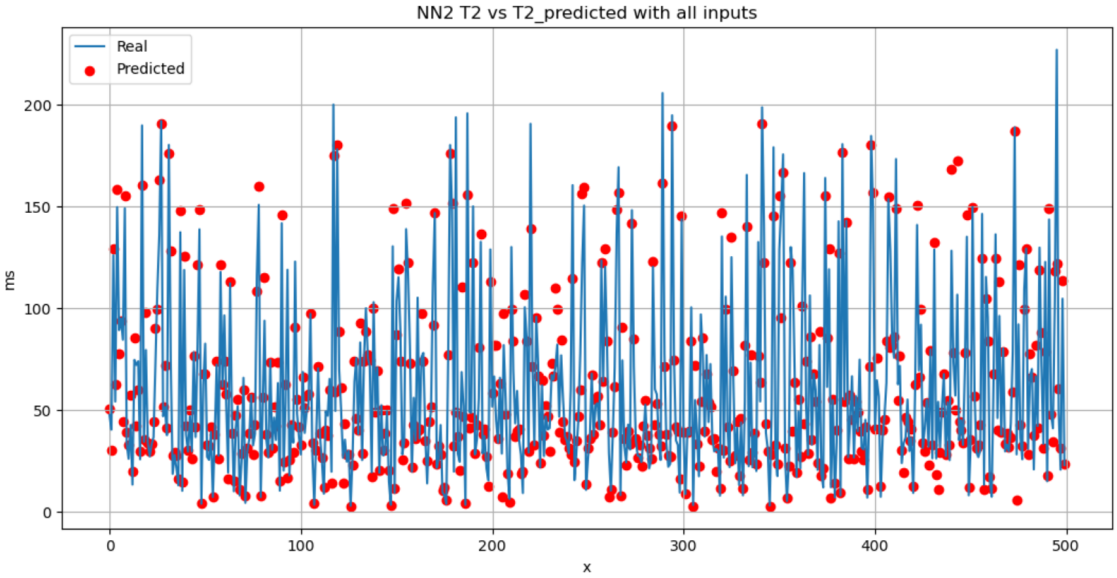

Figure 5.5. Graph showcasing the real t_2 measurements (blue line) and predictions $\hat{t_2}$ (red dots) when seven inputs are chosen.

The MSE obtained for the model along with the MAE in ms is also provided in table 5.2.

Table 5.2. MSE and MAE obtained for NN2 from model 4 with 4 inputs.

No. of inputs considered	**MSE**	**MAE (ms)**
9	0.139	10.93

The aforementioned findings indicate a positive correlation between the number of inputs and the model performance of the NN. The absolute error has decreased from 23.32 ms to 10.93 ms, indicating a significant decrease of approximately 53% in the absolute error on adding more inputs. This is because having more inputs would mean that model has more information available to detect any patterns or relationship within the data, leading to more precise predictions.

In the course of predicting $\hat{t_2}$, one potential input option that was considered was t_1, given that it would already be available and could offer insights into the connection latency. However, subsequent testing revealed that incorporating $t1$ as an input did not yield any noticeable improvements, with the results largely mirroring those obtained with the eight input variables. Consequently, it was decided to exclude $t1$ as an input from further analysis.

It is crucial to ensure that the inputs provided to the NN are pertinent to the problem being solved, and not added arbitrarily for the purpose of increasing the number of inputs. In the case of the project, the inputs should be relevant to the factors that influence the overall latency. Including irrelevant inputs can potentially confuse the model and hinder its ability to establish meaningful relationships between the inputs and the output. Moreover, such inputs may also result in poor generalisation performance, causing overfitting, which performs well on the training data but has a suboptimal performance on unseen data.

Therefore, it is crucial to carefully select and incorporate inputs that align with the objectives of the NN.

5.2 Analysis of Input Data Behavior

Aside from the training of the models and amount of inputs used, how the inputs relate to each other should also be considered.

5.2.1 Multicollinearity

Multicollinearity in this context is the occurrence of high correlation among input variables of a predictor. In a regression model, multicollinearity is an issue which can affect the performance as well as the accuracy of the model. This manifests when the optimiser (i.e. the gradient descend) checks the contribution of the individual input in order to evaluate in which direction the weight of a particular node should be updated. To provide further explanation, it is helpful to consider the example given in equation 5.2.1.

$$output = variable_1 \cdot W_1 + variable_2 \cdot W_2 \tag{5.2.1}$$

In the event that $variable_1$ and $variable_2$ are highly correlated, when one of these variables change, the other would change as well. Consequently, their individual effect on *output* cannot be isolated in order to be later assessed. Therefore, it is crucial to detect and address multicollinearity.

Initially, a potential multicollinearity that may occur through encoding process can be addressed by removing the first column with dummy encoding, as detailed in 4.3.2. Next, after encoding the inputs, multicollinearity can be evaluated by measuring the variance of the regression coefficient. This is by calculating Variance Inflation Factor (VIF), where a value of one signifies no multicollinearity, a value larger than one shows increasing level of multicollinearity, and a value of five or greater indicates high multicollinearity [Investopedia, 2023]. As can be seen in figure 5.6, the test shows high level of multicollinearity between RSSI and the other inputs.

	VIF Factor	features
0	2.153412	Location_Fjer
1	2.118699	Location_Frankfurt
2	2.097250	Location_Paris
3	2.836969	Technology_WiFi
4	5.443713	RSSI
5	2.325487	Protocol_UDP
6	1.546378	DownLink_norm
7	1.691869	Filesize_norm
8	2.254816	Transmission_norm
9	1.250530	MTU_norm
10	2.636805	Encoding_binary

Figure 5.6. Results of VIF test conducted on the encoded inputs.

One approach to address this issue is by dropping RSSI, however dropping an input may be problematic since the model loses information in which it may affect the model performance. Therefore, it must be observed whether this has a positive or negative impact on the model performance. To carefully address multicollinearity and take into consideration the model performance, an evaluation test should be performed with and without the variable RSSI. It should be noted that after dropping RSSI, the hyperparameter tuning is re-performed before evaluating the change in the model. It can be seen from table 5.3 that dropping RSSI, which was causing multicollinearity, improves system performance with an approximate of a 0.2 ms reduction in MAE.

Table 5.3. MSE and MAE obtained for NN2 from model 4 before and after dropping RSSI.

RSSI	**MSE**	**MAE (ms)**
With	0.139	11.01
Without	0.137	10.35

5.2.2 Data sparsity

One of the most crucial challenges in data science is sparsity in the data [Nasiri et al., 2017]. This issue can be defined as the condition or the situation where a dataset includes large number of unique combinations with a low occurrence rate, or no occurrences altogether. This can result in an increased risk of overfitting, since the model may be fitted according to these rare occurrences rather than learning from the underlying features [Prakash, 2022]. To quantify how sparse the data is in the training data set, an analysis of the number of all the unique combinations occurrence was performed. Since there is no strict rule regarding how many times a combination of features should occur in the data set in order to be considered useful, a sparsity threshold of ten is used as a baseline for a number of tests.

As a result of the analysis, it appeared that 92.7% of unique input combinations appeared less than ten times in the dataset used for training. One of the many approaches to mitigate the sparsity is to decrease the number of unique combinations, which was done to the two inputs with the largest amount of distinct values. These inputs include the number of transmissions with 17 possible values, and the downlink rate with 44 possible values. By using mapping as a method to discreticise the data, the number of transmissions was then put in bins of 6, and downlink rate in bins of 10, which results in the number of unique combinations dropping from 1870 to 1844. As a result, the proportions of combinations with less than ten appearances dropped to 81.5%.

To assess whether this method resulted in an improvement to the model, the model accuracy was tested before and after reducing the sparsity level in the dataset. As can be seen from table 5.4, an improvement has been achieved where the MSE as well as MAE are reduced by 0.0128 and 1.59 ms respectively. However, when mapping the inputs into even fewer bins, meaning transmissions were set to 4 bins and downlink to 5, the MSE and MAE would both *increase* from the initial pre-bin values. This then shows that generalising the inputs to a too high degree may have a negative impact on the performance of the model.

Table 5.4. MSE and MAE performance for Model 5 NN2 before and after reducing the number of unique combinations.

Downlink values	Transmission values	% of infrequent combinations	MSE	MAE (ms)
44	17	92.78	0.1372	10.35
10	6	81.50	0.1243	8.84
5	4	61.48	0.1396	11.14

5.3 Evaluation of the preprocessing techniques for different models

In this section, the performance of the five different models discussed in 3.1.3 is evaluated with respect to the implementation of the various preprocessing techniques outlined in section 4.3. Since the first model represents the true output and does not rely on a NN to generate predictions, only models 2 to 5 are subject to testing. This evaluation aims to determine the effectiveness of each model when coupled with different preprocessing techniques. These techniques are intended to enhance the quality of the data, and ultimately improve the accuracy of the models. The results from this evaluation is crucial in selecting the best preprocessing strategy for the final tests.

Model 2

Model 2 requires four inputs, namely t_1, t_2, t_3, and t_4, which are represented as numerical values. Consequently, only numerical preprocessing techniques can be applied to these inputs to enhance their quality and improve the performance of the model.

In this evaluation, the performance of Model 2 is assessed using both normalised and standardised inputs, as described in chapter 4. The results of these evaluations are presented below on table 5.5, where the MSE is used as a metric to evaluate the performance of the different preprocessing techniques.

Table 5.5. Model 2 performance with different numerical preprocessing techniques.

Preprocessing technique	MSE	MAE (ms)
Normalisation	0.00151	7.11
Standardisation	0.0010	1.18

Based on the analysis of the results obtained for Model 2, it can be seen that the use of standardised inputs leads to a better performance in terms of MSE as compared to the normalised inputs. As a result, it has been decided to use standardisation as the preferred preprocessing technique for the final tests of Model 2, which will be evaluated in the chapter 6.

Model 3

Model 3 differs from Model 2 such that it incorporates additional inputs beyond just timestamps. These inputs include numerical and categorical data, meaning that both

numerical and categorical preprocessing techniques should be used. For numerical preprocessing, both standardisation and normalisation are tested once again, and for categorical preprocessing, one-hot encoding and target encoding are tested. As a result, Model 3 is evaluated using the four different combinations of preprocessing methods. The results for each of these four combinations are presented below in table 5.6.

Table 5.6. Model 3 performance with different preprocessing techniques.

Numerical encoding		**Categorical encoding**		
Standardisation	**Normalisation**	**One-hot**	**Target**	**Results**
X	-	X	-	MSE: 0.1111 MAE: 11.07 ms
-	X	X	-	MSE: 0.0067 MAE: 12.25 ms
X	-	-	X	MSE: 0.1242 MAE: 11.78 ms
-	X	-	X	MSE: 0.0069 MAE: 12.93 ms

Based on the results presented in the table 5.6, it appears that there is a contradiction between the MSE and the MAE values. Specifically, the combination of variables that produces the better MSE result, which includes normalised values, has a lower MAE result. Conversely, the combination that produces the better MAE result, which includes standardised values, has a lower MSE result. This discrepancy could potentially be explained by the fact that the range of the normalized values is between 0 and 1, while the standardized values range from -1 to 1.

Given that the purpose of this project is to provide accurate predictions of latency, it is important to select an appropriate metric for evaluating the performance of the model. While both MSE and MAE are commonly used metrics, for this project, the MAE will be used as the primary metric for the final predictions. This is because MAE provides a clearer insight into how close the predictions are to the actual values, which is crucial for determining the model's overall performance. Therefore, on using MAE as the metric, it is observed that the combination with standardised numerical inputs and one-hot encoded categorical inputs performed the best with a MAE of 11.07 ms. This indicates that for models that involve mixed inputs, this combination of preprocessing technique may be preferred. Similarly, Models 4 and 5 also have both numerical and categorical inputs, and the same preprocessing technique of standarisation for numerical data and one hot encoding for categorical data can be concluded to be the preferred choice based on the results.

However, while MAE is the primary metric for evaluating the performance of the model in the final predictions, MSE will still be used in the training process of the model. During the training process, the weights of the model are adjusted based on the MSE obtained.

5.4 Summary

In this chapter, it was found through thorough hyperparameter tuning that the models would perform best with a range of 1-6 hidden layers with an average of 256 nodes, a batch size of 32, and a learning rate of 0.01. It was also found that RSSI should be dropped as an input as it had a high multicollinearity, and the input ranges for amount of transmissions and the downlink rate should be reduced. Lastly, categorical inputs performed best when preprocessed with one-hot encoding, and numerical inputs with standardisation.

Thus, all of the models should be trained with these qualities in mind.

Validation and Performance Testing 6

The previous chapters cover the problem of latency, the proposal of using Neural Network models to predict, and how these models are built and trained.

The next step is to set up different categories of tests in order to get the most out of the parameters found and the model training phase. This is followed by thorough performance tests, where the predicted data are compared to the ground truth. In addition to prediction, the time it takes to predict is also considered, as it is relevant to know how feasibly the models can be run in realtime alongside the PT.

After each test segment is a brief summary of what was found and if the results were as expected. Based on the results found in this chapter, the next and final chapter draws conclusions as to whether the proposed solution was a success, and how the findings may be applied to other real use cases.

6.1 Test Overview

The models to be used for the tests are the same as described in chapter 3, however Model 5 is split in two variants: A and B. In Model5 A, the NN5 used is trained with the real timestamps as inputs. In Model5 B, the NN5 is instead trained on estimated timestamps from NN1 to NN4. For reference, a list of the models used and the output values can be found in table 6.1. These models have been tuned, trained, and had data optimised based on the discoveries made in chapters 4 and 5.

Table 6.1. The models used in the test.

Model	Description
Model 2	Real timestamps run through a NN
Model 3	All inputs run through a NN
Model 4	Predict the four different timestamps, then sum them up
Model 5A	Same as model 4, but with a NN instead of sum. NN5 trained on real timestamps as inputs.
Model 5B	Same as model 4, but with a NN instead of sum. NN5 trained on estimated timestamps as inputs.

Each model generates a corresponding $\hat{t}_{tot}$ output, which is to be compared with the ground truth t_{tot} obtained from the PT per iteration of the process. Additionally, Model 4 and

5 generate and use estimated timestamps $\hat{t}_{1-4}$, which will also be compared with the real t_{1-4} data.

The circumstances that the PT is used in mirror that of the previous chapters, meaning the client only communicates with one server at a time, using either WiFi or 4G and one set MTU size. For the loss performance part of the test, a large number of data are collected at once to be used for offline predictions, which allows for outliers and timeouts to be filtered using the same IQR values as during training. For the time performance, all of the models are run alongside the PT as a realtime DT, and the times it takes to predict t_{dt} are compared with the t_{pt}.

6.2 Model Loss Performance

Table 6.2. Models and their respective MAE for the offline tests.

Model	MAE
Model 2	4.71
Model 3	26.41
Model 4	26.74
Model 5A	25.77
Model 5B	23.92

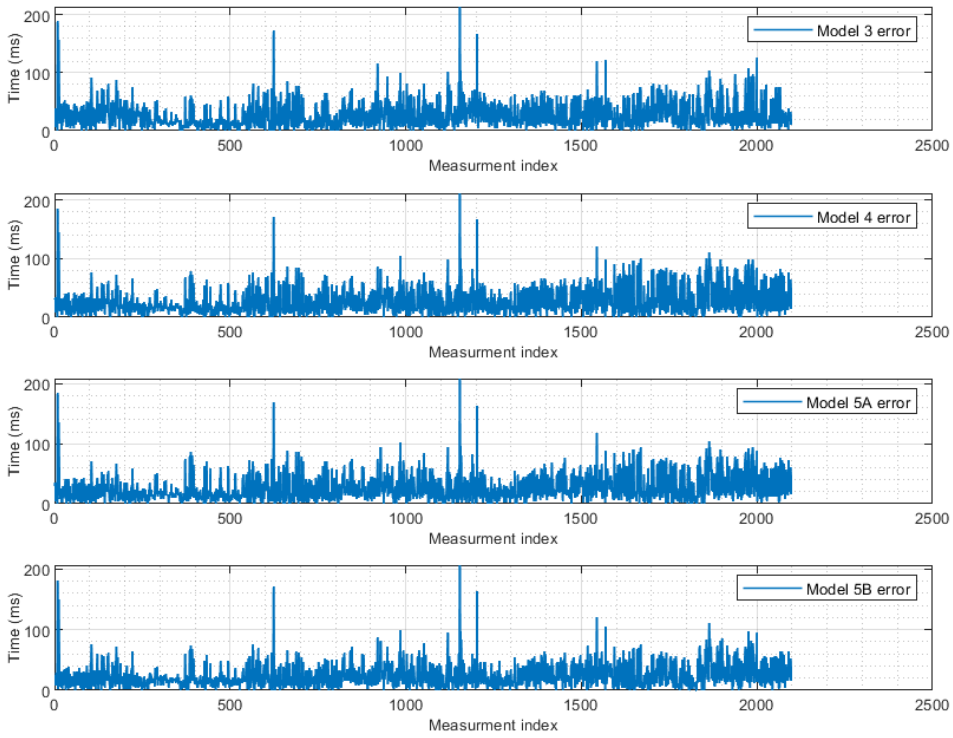

Figure 6.1. .

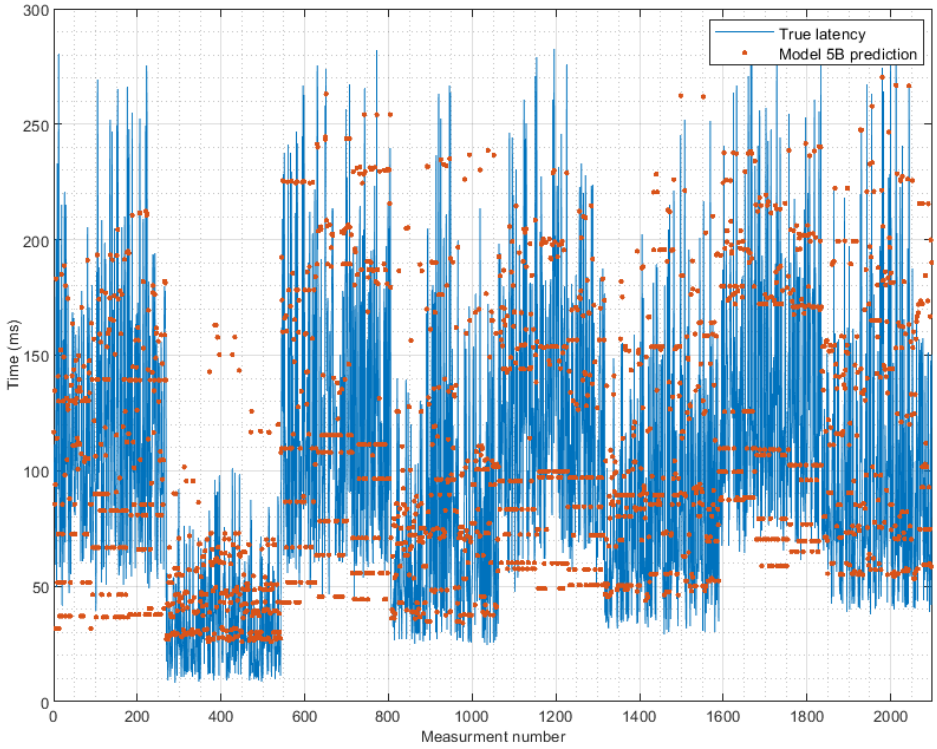

Figure 6.2. Model 5B performance.

6.3 Model Time Performance

Discussion and Conclusion 7

7.1 Discussion

7.2 Conclusion

7.3 Reflections

Appendix

Bibliography

AWS, **2023a**. AWS. *Start Building on AWS Today*. Online Service, 2023a. URL https://aws.amazon.com/.

AWS, **2023b**. AWS. *What Is A Neural Network?*, 2023b. URL https://aws.amazon.com/what-is/neural-network/. Last visited on 07/02/2023.

AWS, **2023c**. AWS. *What Is Overfitting?*, 2023c. URL https://aws.amazon.com/what-is/overfitting. Last visited on 13/02/2023.

Barricelli et al., **2019**. Barbara Rita Barricelli, Elena Casiraghi and Daniela Fogli. *A Survey on Digital Twin: Definitions, Characteristics, Applications, and Design Implications*. IEEE Access, 7, 167653–167671, 2019. doi: 10.1109/ACCESS.2019.2953499.

Briscoe et al., **2014**. Bob Briscoe, Anna Brunstrom, Andreas Petlund, David Hayes, David Ros, Jyh Tsang, Stein Gjessing, Gorry Fairhurst, Carsten Griwodz and Michael Welzl. *Reducing internet latency: A survey of techniques and their merits*. IEEE Communications Surveys & Tutorials, 18(3), 2149–2196, 2014.

Cisco, **2023**. Cisco. *What is Low Latency?*, 2023. URL https://www.cisco.com/c/en/us/solutions/data-center/data-center-networking/what-is-low-latency.html#~q-a. Last visited on 03/02/2023.

Coffey, **2023**. Joseph Coffey. *Latency in optical fiber systems*, 2023. URL https://www.commscope.com/globalassets/digizuite/2799-latency-in-optical-fiber-systems-wp-111432-en.pdf?r=1. Last visited on 03/02/2023.

Grieves, **2014**. Michael Grieves. *Digital twin: manufacturing excellence through virtual factory replication*. White paper, 1(2014), 1–7, 2014.

IBM, **2021**. IBM. *Supervised vs Unsupervised Learning*, 2021. URL https://www.ibm.com/cloud/blog/supervised-vs-unsupervised-learning. Last visited on 04/05/2023.

IBM, **2023a**. IBM. *What Is Overfitting?*, 2023a. URL https://www.ibm.com/topics/overfitting. Last visited on 13/02/2023.

IBM, **2023b**. IBM. *What Is machine learning?*, 2023b. URL https://www.ibm.com/topics/machine-learning. Last visited on 27/02/2023.

Investopedia, **2023**. Investopedia. *Variance Inflation Factor (VIF)*. Online Service, 2023. URL https://www.investopedia.com/terms/v/variance-inflation-factor.asp.

Kaggle, **2021**. Kaggle. *Target Encoding*, 2021. URL `https://www.kaggle.com/code/ryanholbrook/target-encoding`. Last visited on 21/03/2023.

Khan et al., **2022**. Latif U Khan, Walid Saad, Dusit Niyato, Zhu Han and Choong Seon Hong. *Digital-twin-enabled 6G: Vision, architectural trends, and future directions*. IEEE Communications Magazine, 60(1), 74–80, 2022.

Ali Safari Khatouni, Francesca Soro and Danilo Giordano, 2019. Ali Safari Khatouni, Francesca Soro and Danilo Giordano. A Machine Learning Application for Latency Prediction in Operational 4G Networks. In *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 71–74, 2019.

Liu et al., **2021**. Mengnan Liu, Shuiliang Fang, Huiyue Dong and Cunzhi Xu. *Review of digital twin about concepts, technologies, and industrial applications*. Journal of Manufacturing Systems, 58, 346–361, 2021.

Liu et al., **02 2022**. Shengmei Liu, Xiaokun Xu and Mark Claypool. *A Survey and Taxonomy of Latency Compensation Techniques for Network Computer Games*. ACM Computing Surveys, 54, 2022. doi: 10.1145/3519023.

Max Kuhn, **2019**. Kjell Johnson Max Kuhn. *Feature Engineering and Selection: A Practical Approach for Predictive Models*, volume 25. Chapman and Hall/CRC Data Science, 2019.

Nasiri et al., **2017**. Mahdi Nasiri, Behrouz Minaei and Zeinab Sharifi. *Adjusting data sparsity problem using linear algebra and machine learning algorithm*. Applied Soft Computing, 61, 1153–1159, 2017.

Popovski, **2020**. Petar Popovski. *Wireless Connectivity: An Intuitive and Fundamental Guide*. John Wiley & Sons, 2020.

Prakash, **2022**. Arushi Prakash. *Working With Sparse Features In Machine Learning Models*. Online Service, 2022. URL `https://www.kdnuggets.com/2021/01/sparse-features-machine-learning-models.html`.

Strato, **2023**. Strato. *CLAAUDIA research data services*. Online Service, 2023. URL `https://www.strato-docs.claaudia.aau.dk/`.

Tuegel et al., **2011**. Eric J Tuegel, Anthony R Ingraffea, Thomas G Eason and S Michael Spottswood. *Reengineering aircraft structural life prediction using a digital twin*. International Journal of Aerospace Engineering, 2011, 2011.

Verizon, **2023**. Verizon. *IP Latency Statistics*, 2023. URL `https://www.verizon.com/business/terms/latency/`. Last visited on 03/02/2023.

Wagg et al., **2020**. DJ Wagg, Keith Worden, RJ Barthorpe and Paul Gardner. *Digital twins: state-of-the-art and future directions for modeling and simulation in engineering dynamics applications*. ASCE-ASME J Risk and Uncert in Engrg Sys Part B Mech Engrg, 6(3), 2020.

Yang et al., 02 2004. Ming Yang, X.R. Li, Huimin Chen and Nageswara Rao. *Predicting Internet End-to-End Delay: An Overview.* Proceedings of the Annual Southeastern Symposium on System Theory, 36, 210 – 214, 2004. doi: 10.1109/SSST.2004.1295650.

Zhang et al., 2021. Lin Zhang, Longfei Zhou and Berthold KP Horn. *Building a right digital twin with model engineering.* Journal of Manufacturing Systems, 59, 151–164, 2021.