Digital Twin for Latency Prediction in Communication Networks

Master's Thesis

Adham Taha, Linette 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 for Latency Prediction in Communication Networks

Project type

Master's Thesis

Project period

Spring 2023

Participants

Adham Taha
Linette Anil
Magnus Melgaard

Supervisors

Andreas Casparsen
Fabio Saggese
Petar Popovski

Number of pages: 84
Date of completion: May 31, 2023

Abstract

This report investigates the problem of latency in a wireless network scenario, and proposes the idea of using Neural Network models in a Digital Twin in order to predict the latency in real-time.

Different Digital Twin structures were proposed, including different amount of Neural Network models as well as different inputs. To accompany the Digital Twin, a Physical Twin with a client-server file transmission use case was developed, in order to obtain values of latency. Based on data available before a transmission, such as the physical location of the server and file size used, the Digital Twin was trained to predict.

It was found that the Digital Twin was capable of making predictions in under a millisecond by implementing the Neural Network model as TensorFLow Lite models. This was significantly faster than Physical Twin in all scenarios, including when the observed latency was the lowest. The latency prediction itself was successful, and a number of future considerations for more accurate predictions were proposed. These considerations include how to accommodate for temporal characteristics in the observed latency.

Nomenclature

Abbreviations

AAU	Aalborg University
AI	Artificial Intelligence
AMC	Adaptive Modulation and Coding
AWS	Amazon Web Services
CDF	Cumulative Distribution Function
DT	Digital Twin
E2E	End-to-end
GUI	Graphical User Interface
IQR	Interquartile Range
LSTMs	Long Short-Term Memory Networks
MAE	Mean Absolute Error
MCS	Modulation and Coding Scheme
ML	Machine Learning
MSE	Mean Squared Error
MSS	Maximum Segment Size
MTU	Maximum Transmission Unit
NN	Neural Network
OFDM	Orthogonal Frequency Division Multiplexing
PDF	Probability Density Function
PT	Physical Twin
QAM	Quadrature Amplitude Modulation
QoE	Quality of Experience
QPSK	Quadrature Phase Shift Keying
LEO	Low Earth Orbit
ReLU	Rectified Linear Unit
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
<ataha18@student.aau.dk>

Linette Anil
<lanil21@student.aau.dk>

Magnus Melgaard
<mmelga16@student.aau.dk>

Contents

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 small delay would not be problematic, but a video call where the voice and video do not 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 real-time, can help define to what extend latency is accepted or not. A system with a soft real-time 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 real-time 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 real-time constraints, is included on figure 1.1.

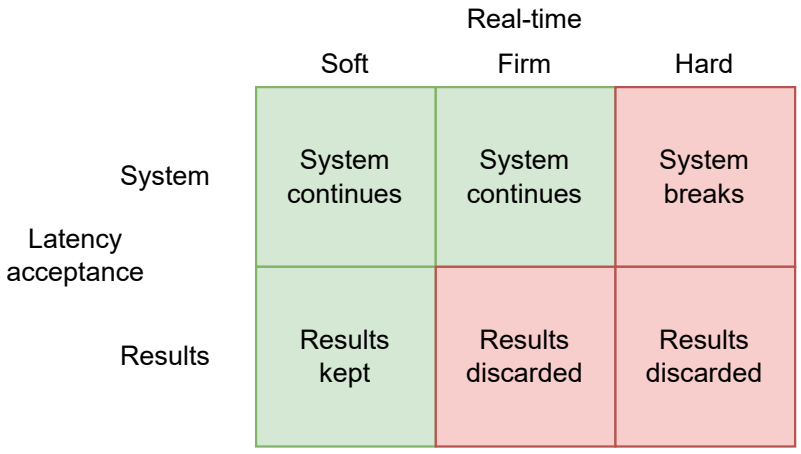

Figure 1.1. Confusion matrix showing three types of real-time constraints and how latency may affect the system and its results. The green cells show when the system or results will not be discarded by latency, and red cells when they will.

However, even if a soft real-time task is not discarded by delays, the value of the results may increase if they are timely. For example, a common soft real-time 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 [Liu et al., 2022].

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

A possible method to isolate the sources of delay is the use of Digital Twins, an emerging and increasingly popular technology in the Machine Learning fields. By simulating a system, the Digital Twin will simulate the same process, and thus give an insight into what the expected result or behavior should be. This can be used to figure out at which step of the process it goes wrong, resulting in unexpected latency. Additionally, by implementing a Neural Network, 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 real-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. Based on the real-time constraints of the system, this can potentially be disruptive or even damaging for the usage and results. The idea of implementing a Neural Network (NN) in a Digital Twin (DT) to predict the latency based on prior data is proposed. This requires that the latency can be isolated in the first place, which raises an initial problem formulation:

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

This chapter goes into detail about the State of the Art in latency, including measurement, prediction, and compensation for said latency. In addition, the concept of Digital Twins and how a Neural Network may be used as one is explored. To accomplish this, it is necessary to investigate specific implementations of Neural Networks and how they work.

2.1 State of the Art

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

2.1.1 Latency and potential sources

Briscoe et al. [2014] describes latency as the measure of responsiveness in a system, i.e. how instantaneous an application feels. It is 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 hindrances in achieving a seamless experience for many applications, and the author 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 the client's endpoint, also have an impact on the latency at the client's end.

Endpoint delay

These delays occur due to the end-to-end (E2E) protocol setup. These protocols can include transport protocols that are used for various control interactions before the data is 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 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 channel access, the latency can become unpredictable even though the transmission error is low. Furthermore, the number of collisions may also increase as the number of users increase, which results in more retransmissions, thus introducing more overhead, and as a result, more 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 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 Measuring latency

A general method of measuring latency is through the use of a timer, either through hardware or software. However, this comes at the cost of having to time the individual components or actions, as well as the potential overhead provided by a software timer function.

Some more precise methods have been proposed to measure latency, one of which is `Mine's technique`, where the time between the activation of two photo-diodes is quantified utilising an oscilloscope [Mine, 1993]. This is done by positioning the first diode on a pendulum, in order to traverse a dim light while the other diode registers it. In order to ensure consistency between the measured position on the swinging pendulum and its visual depiction, precise adjustment is required, since this technique takes measurements at fixed physical positions. A more advanced technique is presented by Ellis et al. [1999], where a driven pendulum is used in order to quantify latency among various tracker devices. However, both techniques necessitate the establishment of specialised infrastructure and hardware. Another method by He et al. [2000] is proposed, involving the utilisation of a camera to count the frames between observable points of change of an object in motion. Nevertheless, this method is susceptible to inaccuracies in recognition of the turning points in the motions, in addition to being time-consuming [Steed, 2008].

However, precisely quantifying the latency is challenging due to a number of factors. These include network complexity, clock synchronisation, dynamic variables, the addition of overhead, and hardware limitations. Taking these factors into account, it is worthwhile to consider the concept of latency prediction.

2.1.3 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 (QoE). 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 (ML) 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 authors use the data set 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 see which suits the data set best, and which produces better results in order to predict latency between device and a server.

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, and can only provide theoretical upper-bounds. Many applications do not have enough knowledge upon which scientific models can be built, and therefore, these mechanisms have their limitations. The author also states that in such cases, ML methods such as Artificial NNs can be used for prediction, and can be substituted for the other methods that alleviate their limitations.

2.1.4 Addressing the predicted latency

Addressing the predicted latency depends on how critical it is to the considered use case. In the case of firm or hard real-time, an application should be able to take action proactively. 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 are widely applied in the gaming industry. Liu et al. [2022] carried out a survey on the different latency compensation techniques used for online computer games. These techniques are software algorithms that are used on either the game server or client side, in order to minimise 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 of the 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 has yet to cause any changes on the actual server and for other players.

The author, however, only discusses latency and compensation techniques for video games and does not consider other use cases.

2.2 Digital Twins and Physical Twins

The concept of using NNs as a predictor, and therefore as a model of the system, prompts an analysis of a relevant concept known as a DT. DTs are 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 a physical entity in all of its complexity. Using a DT allows for monitoring, control, and even optimisation of the physical entity. Depending on the implementation of the DT, it may even be capable of running in real-time alongside an ongoing physical process.

The idea of DTs was initially applied by NASA in 1970 for 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 evolved while going through many different names, such as mirrored spaces, virtual spaces, digital copies, and digital mirrors. It was first in 2002 that the name *Digital Twin* was properly introduced for the concept, by professor Michael Grieves [Zhang et al., 2021]. Later, Grieves [2014] standardised the use of a DT as a system encompassing three components, 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 digital counterpart is the DT which hosts and processes PT data, and performs a number of tasks e.g data visualisation, system troubleshooting, and simulation. 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.

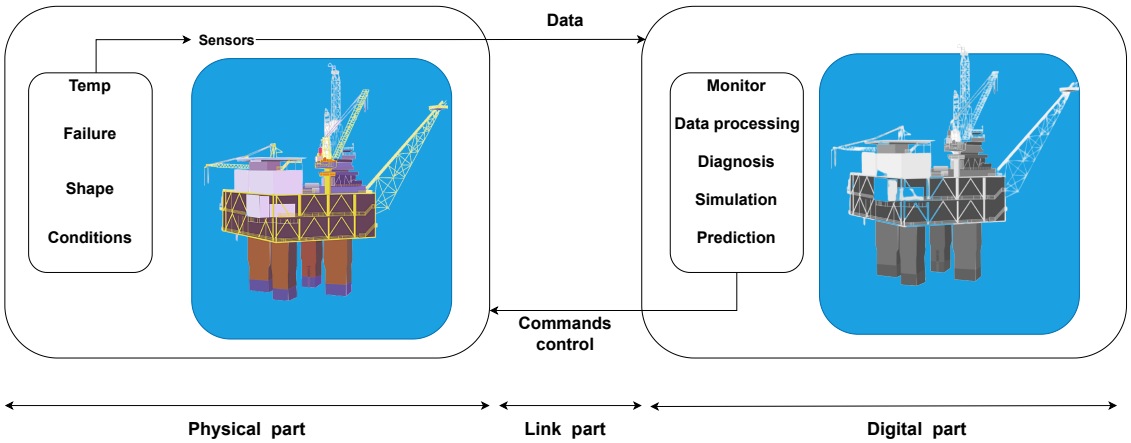

Figure 2.1. DT system parts with an example of usage.

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

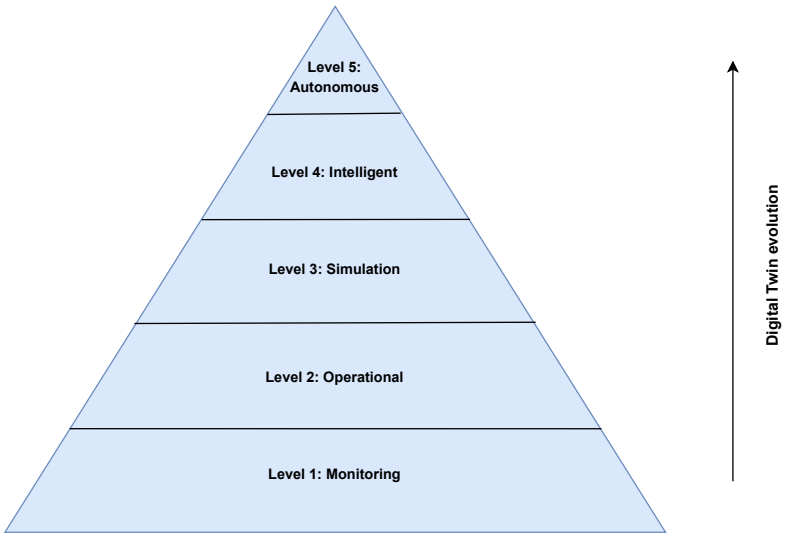

Figure 2.2. Levels of sophistication of DT.

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 is an operational DT, which incorporates making 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. These provide 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 using Artificial Intelligence (AI). 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.

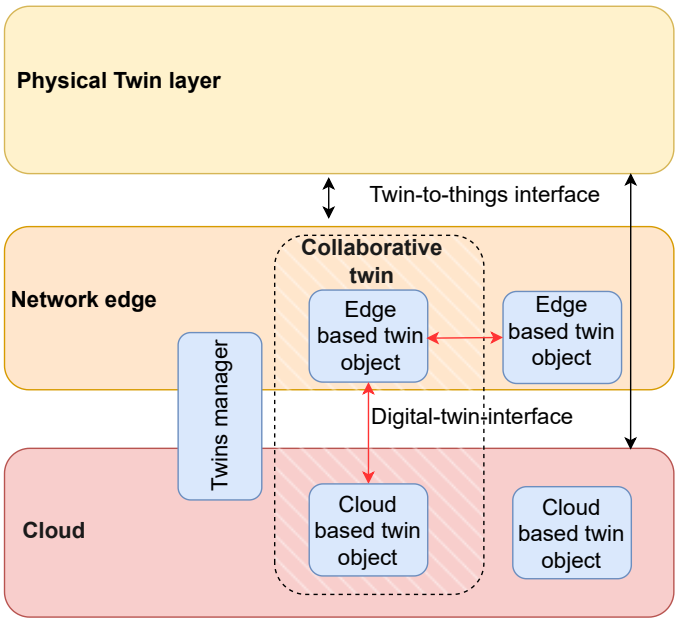

Figure 2.3. Different DT architectures.

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 more powerful computers. 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 ML 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.

As this project revolves around the concept of predicting a value, the intelligent DT structure is used going forwards. This first requires an introduction to what ML is, and how it can be used, as it is part of this type of DT.

2.3 Machine Learning

In the last few years, ML has been extensively researched and used in various applications including image/speech recognition, product recommendation, and fraud detection [IBM, 2023]. ML is a field in AI, where the objective is to build a model using data and algorithms, for the purpose of making decisions with minimum human assistance. Primarily, ML 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 a reward for every correct action, and a penalty or negative feedback for every incorrect action. 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 student performances (pass or fail) based on their grades.

Supervised training 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.

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

2.3.1 The data processing phase

In the data processing phase, the data is commonly split into training and testing sets. The model usually trains itself using the training set, which allows it to estimate parameters and evaluate the model performance. 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 data set 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 or testing data. In order to avoid overfitting, the model must be trained on enough data that are representative of all of the inputs, but also not trained too long on the same data [AWS, 2023c].

2.3.2 Neural Networks

NNs are a subset of ML, which can have varying structures depending on the intended usage. The general structure of a NN is described below.

Neural Network structures

A NN 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 NN model can be seen in figure 2.4.

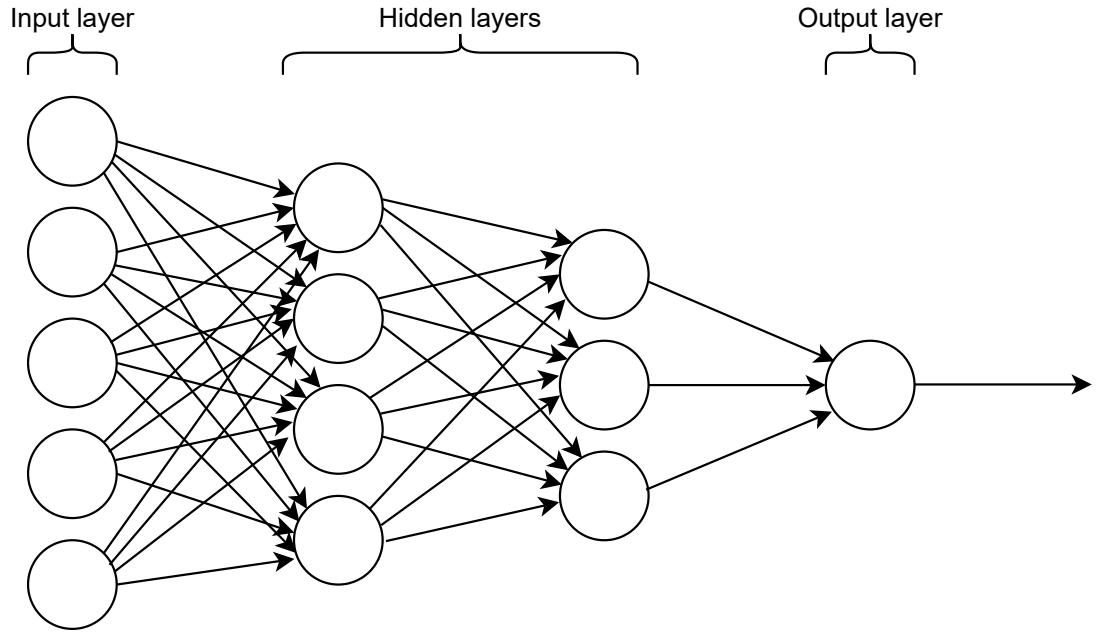

Figure 2.4. Example of a four-layer NN with two hidden layers.

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 is the input for other nodes in the next layer. The NN may consist of a single or multiple hidden layers which are located between the input and the output layers. A NN with more than three different layers in total is classified as a Deep Learning NN. These intermediate layers are responsible for extracting different hidden features and patterns in the data. Finally, the output layer produces predictions or classifications based on the inputs from the previous layers.

Each of these nodes are connected to the next layer of 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 a node in the next layer. The associated weight of the edge is multiplied with the connection's input value, and passed through an activation function.

The activation function decides whether a node should be activated or not, referring to whether the output of the node is used, and thus whether it is significant or not. A node that is not activated would simply give a 0. An activation function adds non-linearity to the network by transforming the summed weighted inputs to the node into an output value. Additionally, a special kind of weight 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.

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 in 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 for the purpose of obtaining a probability. The function is represented in equation (2.3.1) along with its Probability Density Function (PDF) in figure 2.5.

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

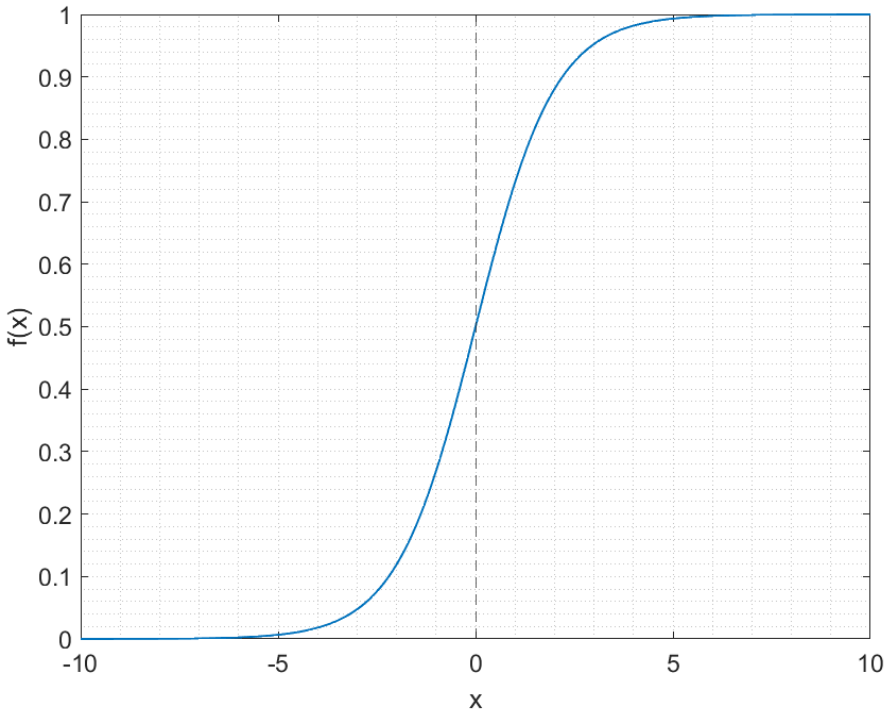

Figure 2.5. The sigmoid function.

ReLU

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

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

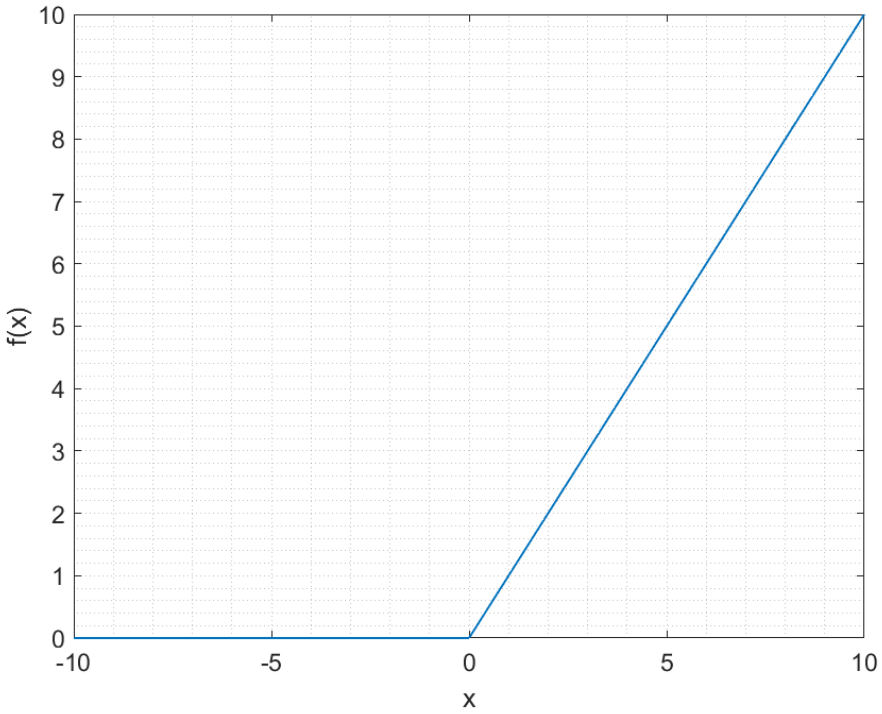

Figure 2.6. The ReLU function.

Having established the general terms and features of a NN model, different types of NNs are introduced.

Types of NN models

NN models can be categorised into different categories, where two common examples are Recurrent NNs and Feedforward NNs. 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 period of time, before stimulating the neighbouring nodes in the previous and next layers, which also become active for a period 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 NNs, 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 NNs are not considered further.

Feedforward algorithm

Essentially, a feedforward NN 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 NN is given a weight `w`, which is a random small number. Furthermore, at each node, the bias `b` is set to 1, with a corresponding small weight. At every single layer, the inputs are multiplied by the w and summed together with b. Next, the resulting sum is passed through an activation function f. This process is illustrated in figure 2.7.

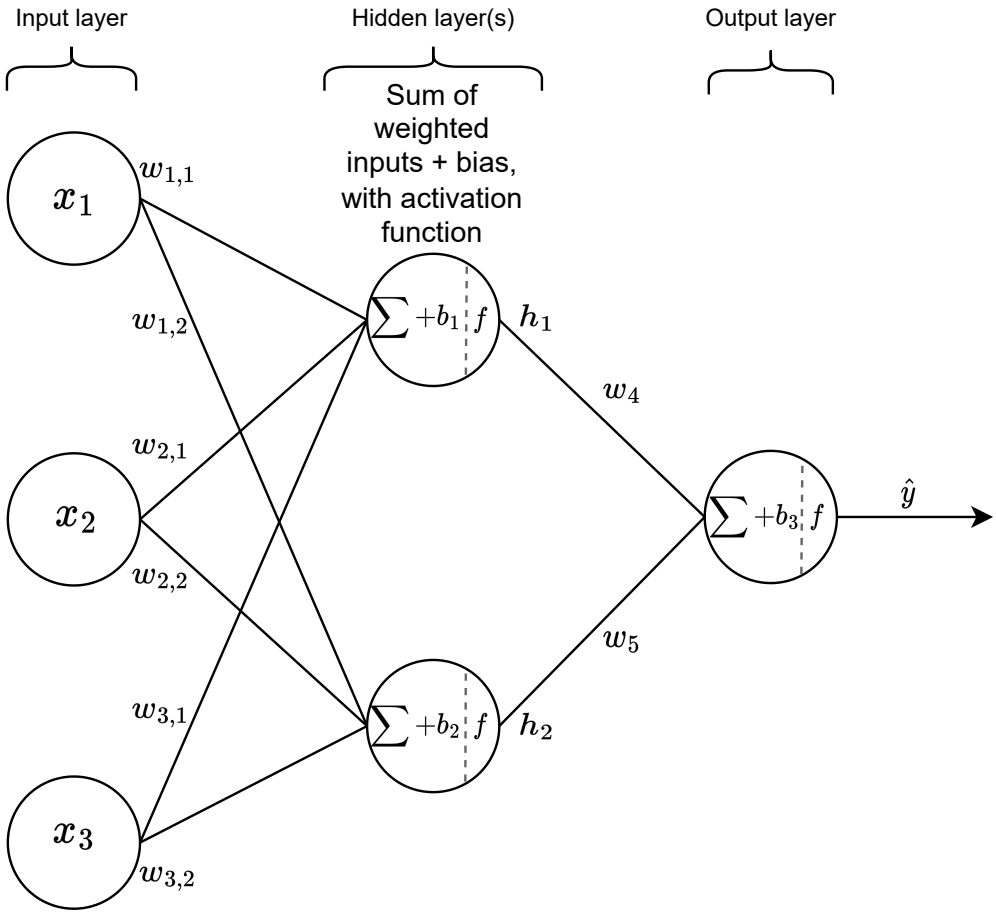

Figure 2.7. Example of a NN with one hidden layer.

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 h_j is the outputs of the hidden layer nodes, 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:

$$\hat{y} = f(h_1 \cdot w_4 + h_2 \cdot w_5 + b_3) \tag{2.3.4}$$

where y is the output value.

2.3.3 The 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, which are different for classification and regression problems.

In classification problems like binary classification, which categorises data into one of two classes, cross-entropy is commonly used as a cost function. This 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-based NNs, on the other hand, 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, which 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, $\hat{y_i}$ are the predicted values and y_i are the actual values.

Backpropagation

Once the loss function is obtained, the goal is to minimise it. 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. This is done 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 2.7 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 the overall loss can be minimised. The stochastic 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 2.8, 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.

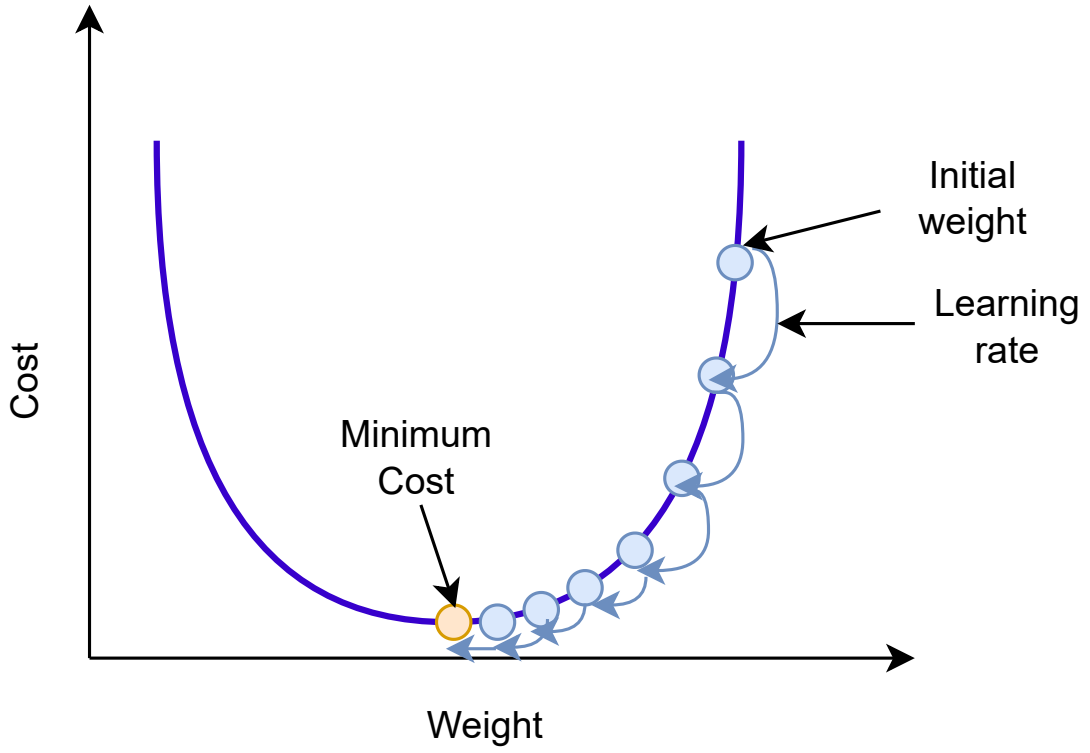

Figure 2.8. Graph showing the cost function with respect to weight.

Returning to the example in figure 2.7, for w_4, the partial derivative of the loss function in terms of w_4 cannot be computed directly, since w_4 is embedded deep inside the output function, namely in $\hat{y}$. Thus, the gradient with respect to w_4 is calculated by applying the chain rule, as shown in equation 2.3.8:

$$\frac{\partial L}{\partial w_4} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_4} \tag{2.3.8}$$

It is important to mention that the equation of $\hat{y}$, as can be seen in equation 2.3.4, is a result of 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_4, the chain rule is applied, so that it results in the following expression:

$$\frac{\partial \hat{y}}{\partial w_4} = \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial w_4} \tag{2.3.9}$$

where

$$z = h_1 \cdot w_4 + h_2 \cdot w_5 + b_3 \tag{2.3.10}$$

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:

$$\frac{\partial \hat{y}}{\partial w_4} = \frac{1}{1 + e^{-z}} \cdot (1 - \frac{1}{1 + e^{-z}}) \cdot \frac{\partial z}{\partial w_4} = \frac{1}{1 + e^{-z}} \cdot (1 - \frac{1}{1 + e^{-z}}) \cdot h_1 \tag{2.3.13}$$

Inserting equation 2.3.13 into equation 2.3.8 gives:

$$\frac{\partial L}{\partial w_4} = (\hat{y} - y) \cdot \frac{1}{1 + e^{-z}} \cdot (1 - \frac{1}{1 + e^{-z}}) \cdot h_1 \tag{2.3.14}$$

By computing equation 2.3.14, a numerical value is obtained and then used in equation 2.3.15 in order to find the new value of w_4. 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.

$$w_{4(update)} \rightarrow w_4 - \eta \cdot \frac{\partial L}{\partial w_4} \tag{2.3.15}$$

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_5 is calculated through the following equation:

$$\frac{\partial L}{\partial w_5} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_5} \tag{2.3.19}$$

$$\frac{\partial L}{\partial w_5} = (\hat{y} - y) \cdot \frac{1}{1 + e^{-z}} \cdot (1 - \frac{1}{1 + e^{-z}}) \cdot h_2 \tag{2.3.20}$$

Next, the updated value of w_5 can be found through:

$$w_{5(update)} \rightarrow w_5 - \eta \cdot \frac{\partial L}{\partial w_5} \tag{2.3.21}$$

After updating the nearest weights (w_4 and w_5) 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 gradient descent 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

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

$$\frac{\partial \hat{y}}{\partial h_1} = \frac{1}{1+e^{-z}} \cdot (1 - \frac{1}{1+e^{-z}}) \cdot w_4 \tag{2.3.25}$$

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

Thus equation 2.3.23 can be written as follows:

$$\frac{\partial L}{\partial w_{1,1}} = (\hat{y} - y) \cdot \frac{1}{1+e^{-z}} \cdot (1 - \frac{1}{1+e^{-z}}) \cdot w_4 \cdot x_1 \tag{2.3.27}$$

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 gradient descents 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 every batch, the loss function and subsequently the weights are calculated, allowing for finer tuning. After this has been done 20 times, meaning all 1000 data points have been used, the epoch is completed.

2.3.4 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. How the NN is trained and subsequently used is detailed in the next chapter, starting with the proposed PT structure to obtain the data needed for the DT.

2.4 Final Problem Statement

This chapter included a technical analysis of DTs 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 PT is created and run alongside a trained DT model, feeding the timestamps from the PT as inputs. However, if the DT is to predict the E2E latency in real-time, it needs to be faster than the PT. 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?

Design and Implementation 3

This chapter contains the design considerations as well as the implementation of the Physical Twin and Digital Twin used in the system.

The initial design sections introduce the idea of different Digital Twin models, with their own set of data inputs related to the Physical Twin process. This leads into a section which investigates some of the sources of these inputs and how they can be used to get an idea of latency. Before this data can be used as an input for the Digital Twin, it needs to be preprocessed, which is covered in the last section of this chapter.

The prediction results shown in figures and tables in this chapter are not representative of the finalised Physical Twin and Digital Twin stages, and were obtained using preliminary data and models. These data are included for the sake of comparison.

The code for the project is found on:

https://gitlab.com/adhamtaha/neural-networks-for-latency-prediction.git

3.1 Initial System Proposal

To predict using a NN DT, it requires a source of data both for training and for testing. This chapter details what kind of PT could be created for this purpose, and how the DT can be structured.

In chapter 2, latency was described as the effect of the many individual steps and processes that take place for example during communication. As suggested in the final problem statement, examining E2E latency would entail analysing the overall 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 an E2E latency that can be observed. This is necessary both to compare the DT with data from a real scenario, 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.

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

A diagram of the system encompassing the PT and DT is seen in figure 3.1, where the PT has 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 time each component takes to provide the outputs are given by t_{PT} and t_{DT} accordingly.

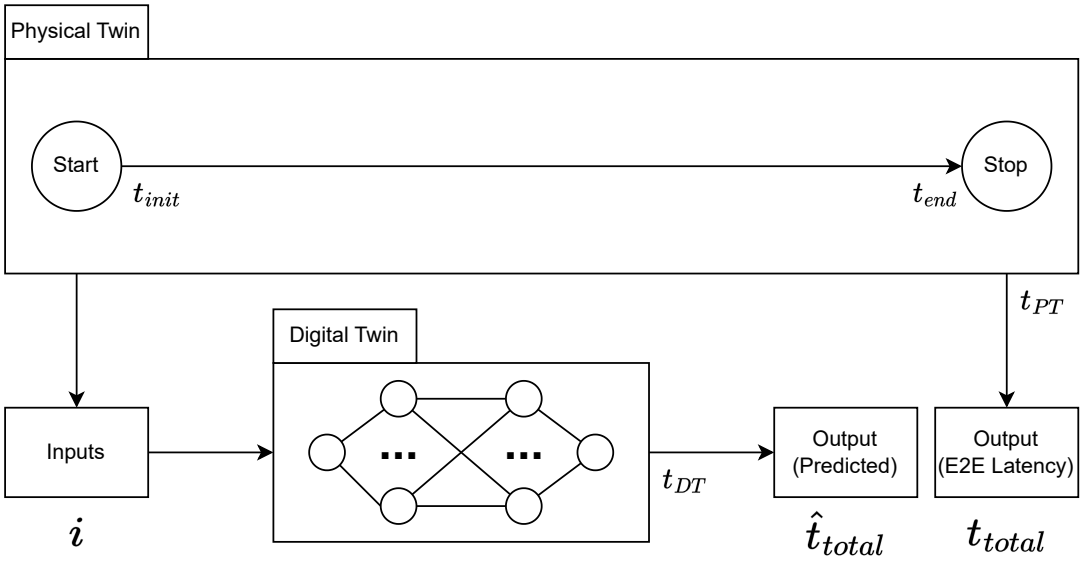

Figure 3.1. Diagram encompassing both the PT and the DT.

The DT will have to be trained prior to usage, using both inputs and output from the PT. 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.

3.1.1 The Physical Twin

In order to obtain the training data necessary for the DT, and to have a real process to compare the results, a 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. A diagram representing this use case can be seen in figure 3.2.

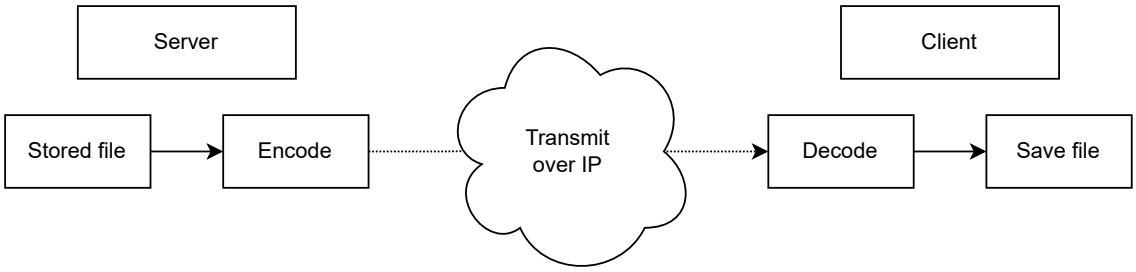

Figure 3.2. A server-client architecture including the encoding/decoding of files before and after transmission.

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 the connection is closed. This figure discounts the elements of decoding the data.

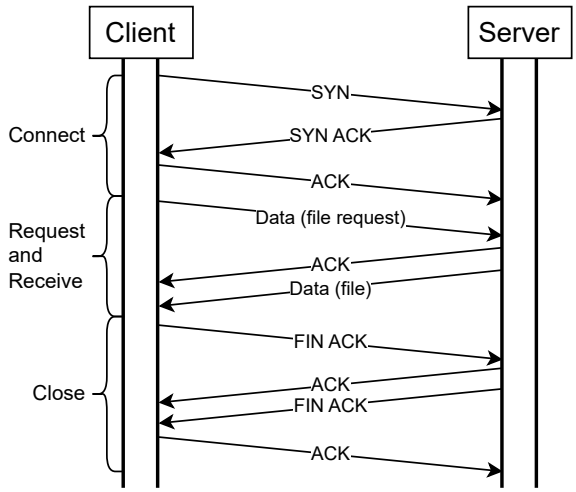

Figure 3.3. A sequence chart showing the process from establishing a connection, requesting and receiving file, and finally closing the connection.

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 in figure 3.4.

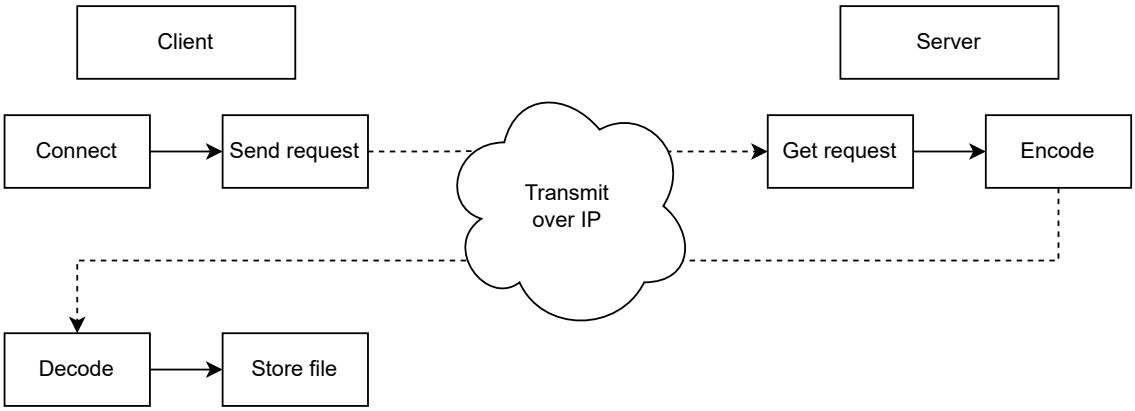

Figure 3.4. An updated server-client architecture, where the client first sends a request.

3.1.2 Creating the Physical Twin

In order for the client and server components of the PT to give a realistic idea of the latency, these will be created individually to be run on different machines. This also allows for scenarios such as the server being deployed at different locations or the client communicating far away from an access point, to see how it affects the observed latency.

Server

The server was created and then configured in the form of a virtual machine, using the Strato [Strato, 2023] cloud environment hosted by Aalborg University (AAU). 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.

The server is run using a Python script. The script establishes sockets with 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. Next, the file name request from the client is parsed, and the targeted file is selected from the server database, encoded, and sent 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 TCP and UDP. The client downloading a file encompasses the following main processes; **connection**, **request and receive**, **store**, and **decode**. The individual latencies of the aforementioned processes are taken as timestamps denoted by t_1, t_2, t_3, t_4 respectively, and t_{total} is the sum of these processes corresponding to the E2E latency.

The script starts with a `time.perf_counter()` to initialise 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 t_1 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 t_2 is recorded. Afterwards, the client decodes the received data and records the decoding timestamp t_3, followed by the decoded data being saved, while also recording the corresponding timestamp t_4.

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 while taking the timestamps for each of these separate operations.

3.1.3 The Digital Twin

Alongside this PT, a DT would be run. As the objective is to 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, so t_1 to t_4.

However, this method of predicting using timpestamps as 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, t_1 in combination with the size of the transmitted file could be used in order to try and compensate for not having t_2. This would be possible, as the time to transmit a file would depend both on the connection and the amount of Bytes to transfer. This requires an investigation of which exact inputs and factors affect the PT processes, which is included in section 3.2.

Another aspect is how these inputs are modelled in the DT. Five different models are proposed in figure 3.5. Model 1 represents the PT, and how the timestamps are taken and added up. Every subsequent model is a derivative of this, with Model 2 using the timestamps from the PT fed into a NN as a proof of concept, and Models 3 through 5 using different information known prior to the transmission as inputs.

An important distinction between Models 3 through 5, is that while Model 3 only aims to obtain $\hat{t}_{total}$, Models 4 and 5 also estimate the timestamps $\hat{t}_1$ through $\hat{t}_4$. Using these estimated timestamps, Model 4 uses the sum like Model 1, and Model 5 uses a NN like Model 2.

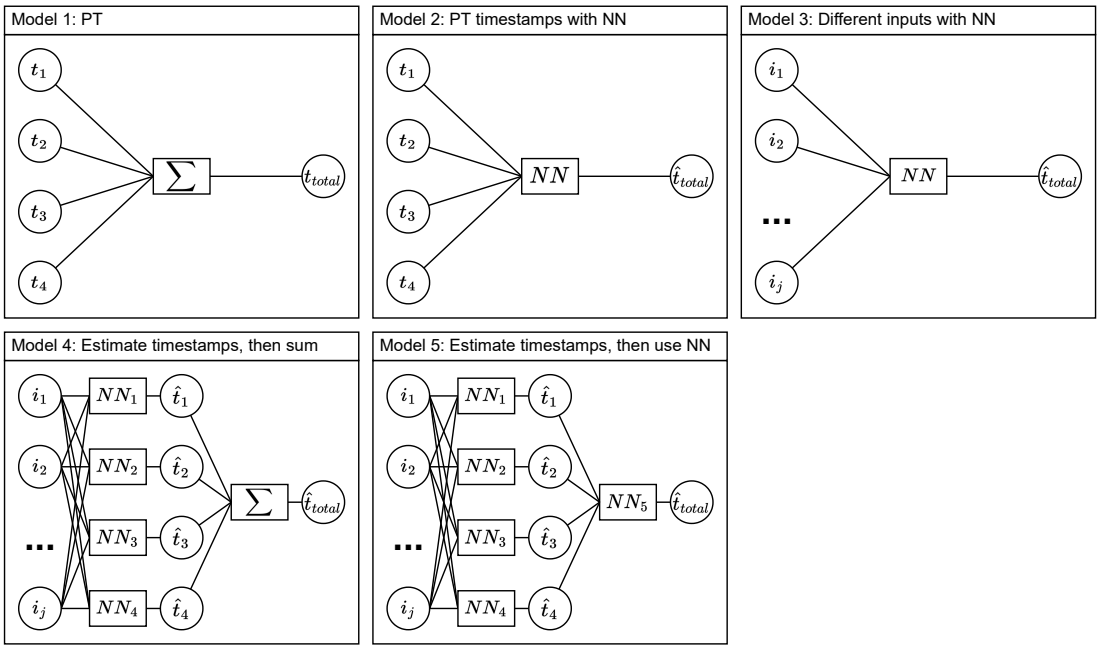

Figure 3.5. Model 1 (PT) and different models proposed for the DT.

On top of analysing how each of these models would perform in terms of getting a $\hat{t}_{total}$ close to t_{total}, the t_{DT} that the different models take to get a prediction can also be investigated.

3.1.4 Creating the Digital Twin

In order to implement the DT, the NNs shown on figure 3.5 will have to be created.

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. The `model.compile` method is then used to set the loss and optimisation functions for the NN. Given that this is a regression problem and not a classification problem, accuracy is not a possible metric, as the result will be how close the predictions are to the ground truth, rather than a correct or incorrect answer. Because of this, MSE is set as the loss function, and Stochastic Gradient Descent as the optimisation function.

When the basic features of a NN are set, *model.fit* trains the model using a set of training input(s), training output(s), and parameters such as batch size and the 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 testing. This requires that it has not been used prior for training. The `train_test_split()` Scikit method accomplishes this by randomly selecting 80% of each input and output data for training and the remainder 20% for testing. In addition to randomly selecting the data, it is also randomly shuffled in advance.

As for the epochs and batch size, those may be chosen based on trial and error, while taking in mind not to overfit or underfit the model. The exact parameters used in a NN are tested and specified in chapter 4.

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 mention that these models need to be trained using offline data. This creates a need to obtain enough data before the model can learn the patterns and make adequate predictions. A diagram that shows the relationship between offline training and the DT can be seen in figure 3.6.

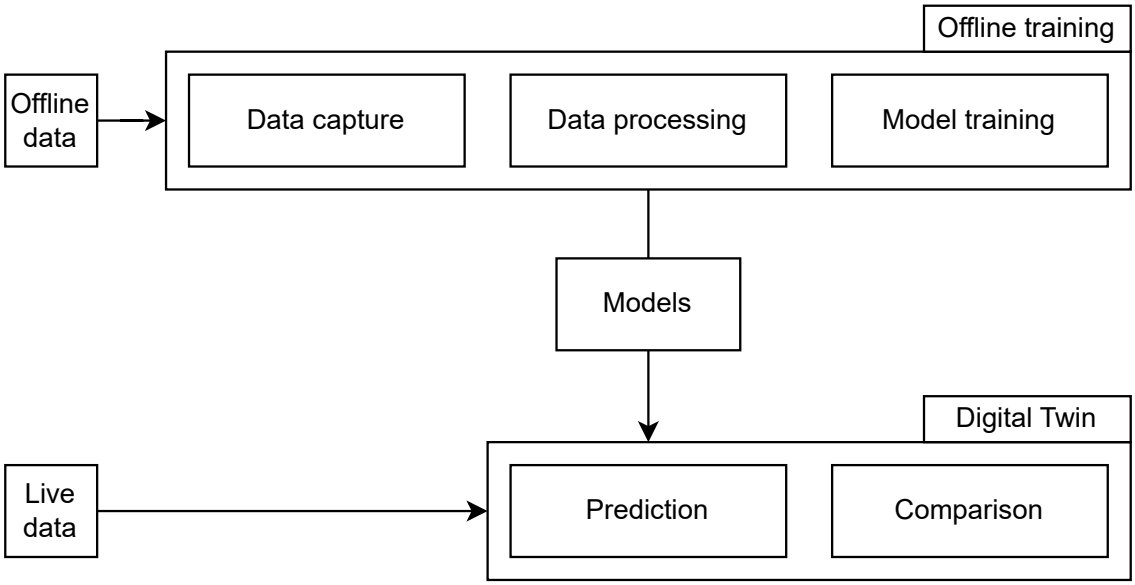

Figure 3.6. A diagram of the offline training procedure and the DT which requires the complete trained models from the latter to function.

The offline training includes three different phases: 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 selected, but also what impact they have on the latency. This comprises an analysis of what information can be extracted from the PT 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 builds on top of the `TensorFlow` basics described previously. This includes how the parameters for the model training, 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.

3.2 System Inputs

Each of the four main processes in the PT may include multiple sub-processes, and examining all of these is not the focus of this work. As such, the processes are mostly treated as a black box. Rather, the factors that influence these processes are discovered and analysed, in order to be used as inputs in the DT. These factors are listed in table 3.1, which are later employed as inputs to the NNs 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 section 3.1. These factors were found to have an impact on the latency, and individual factors are analysed in the subsections below. Note that in this section, the graphs display the average values of a number of iterations corresponding to each test.

Table 3.1. Factors 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 the following tasks: connect to the server, send a TCP request that includes the file name to be downloaded, receive the 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 the server, while maintaining a constant file size of 10 kB. The server is physically situated in Aalborg, Denmark, while the client is positioned at different locations for each test. In addition, at each distance, the client communicates with the server via WiFi.

The graph in figure 3.7 shows a clear positive correlation between distance and t_{total}, with t_1 and t_2 increasing as the distance gets larger. The latencies do not appear significantly different at small distances, which can be observed 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 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.

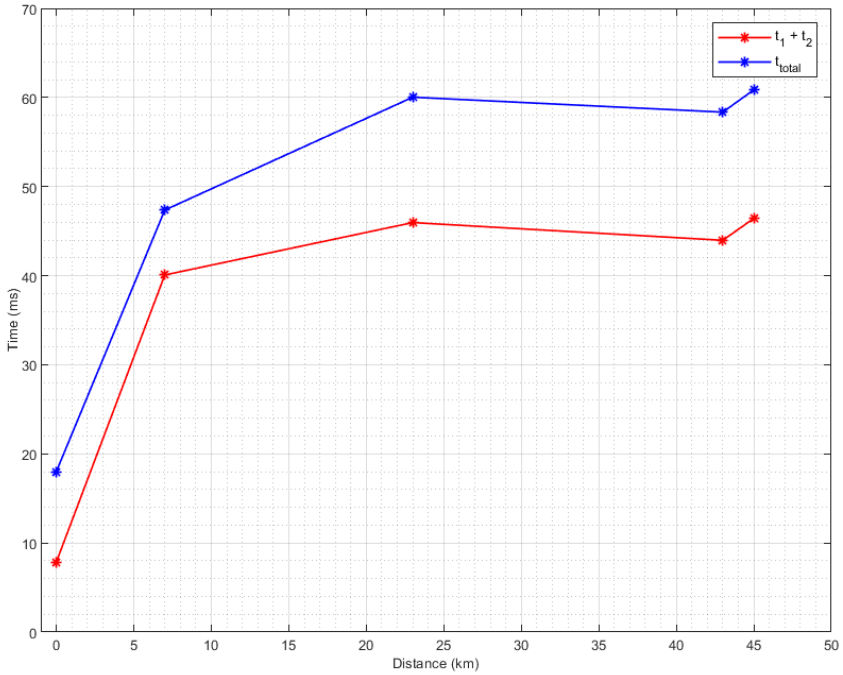

Figure 3.7. Graph showing the impact of the latency on t_{total} and on $t_1 + t_2$, where the averages of 10,000 iterations are computed for distances at 0, 7, 23, 43 and 45 km.

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 involves measuring t_1, t_2, and t_{total} of the client to the server when located at different distances, as seen in table 3.2. 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 3.2. Input information for cities and the corresponding distances 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 3.8, 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 constraints of network communication becomes more apparent.

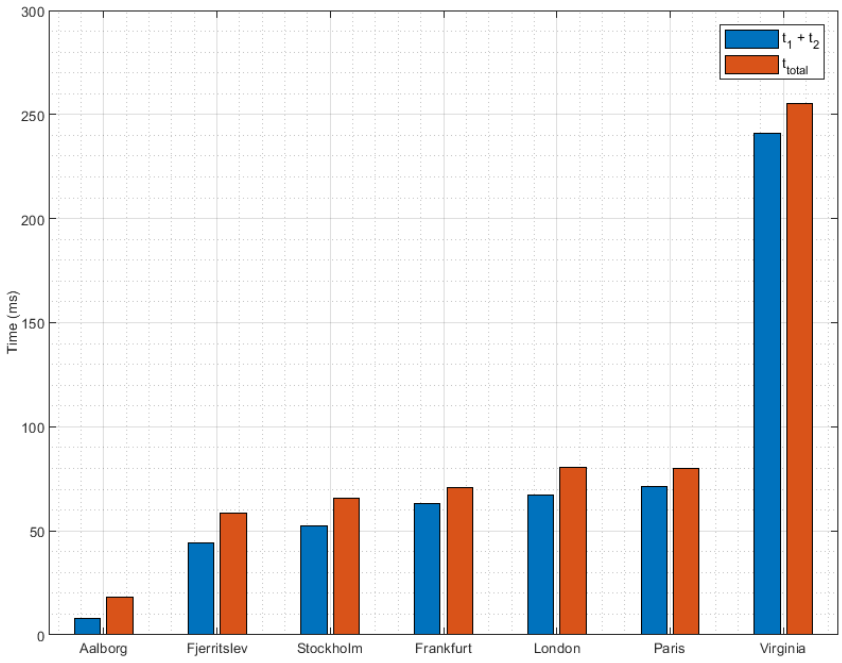

Figure 3.8. Graph showing the impact of the distance on t_{total} and on $t_1 + t_2$, where the averages of 10,000 iterations are computed for various cities arranged in ascending order of distance between the client and the server.

It was decided that only Aalborg, Fjerritslev, Frankfurt, and Paris would be kept as input data, as the change in latency over distance was more apparent than the others. As for Virginia, it was not kept because it was significantly different, due to being in a different continent, where other potential factors than distance may have factored in.

Technology

A number of tests have been conducted where the client repeatedly downloaded files of 10 kB at different distances using either WiFi or 4G. As seen in figure 3.9, 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.

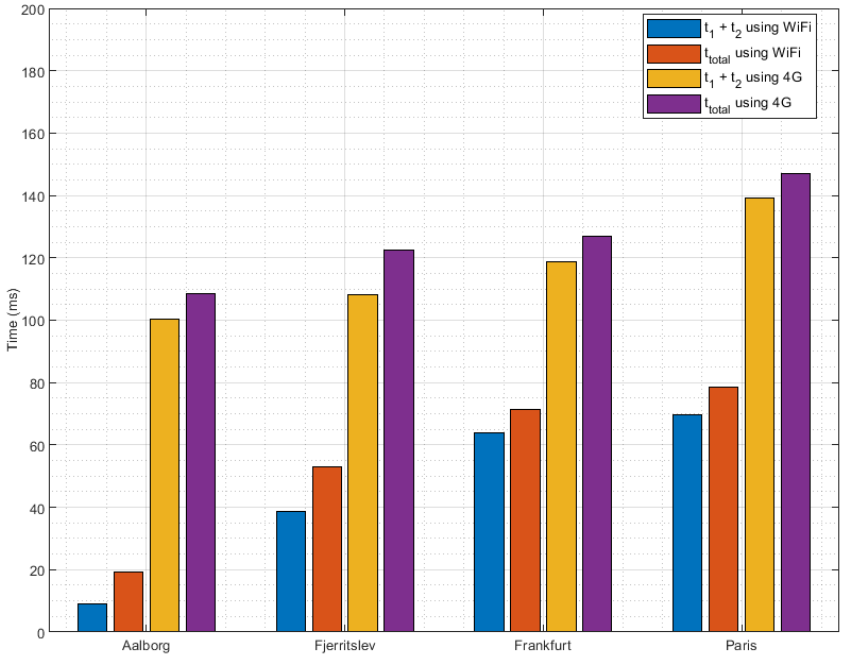

Figure 3.9. Graph showing the impact of the different technologies (WiFi and 4G) on t_{total} and on $t_1 + t_2$, where the averages of 10,000 iterations are computed for various cities arranged in ascending order of distance between the client and the server.

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 WiFi is used, both the client and the server are on the same network, resulting in a decreased number of packet hops for transmission, as can be seen in figure 3.10. However, 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 comprise multiple hops.

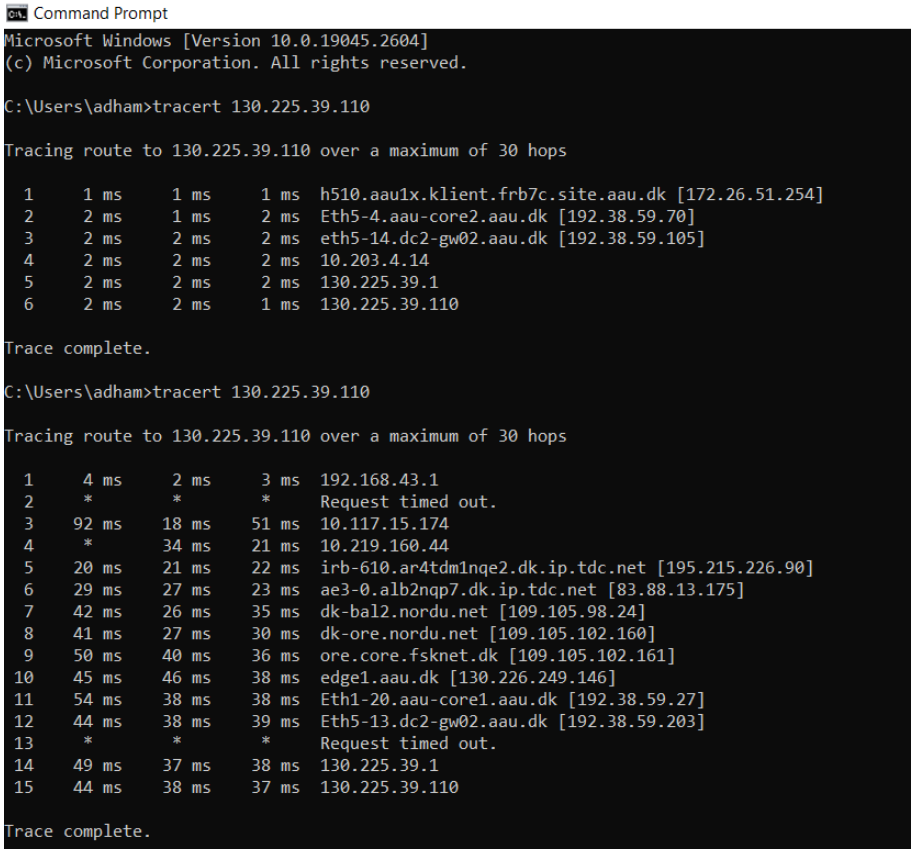

Figure 3.10. Tracert output displaying two different paths and latency of data packets through various network nodes - from the top, one for WiFi and the other for 4G.

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 3.11, 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 sizes cause 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 sizes 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 more latency. 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.

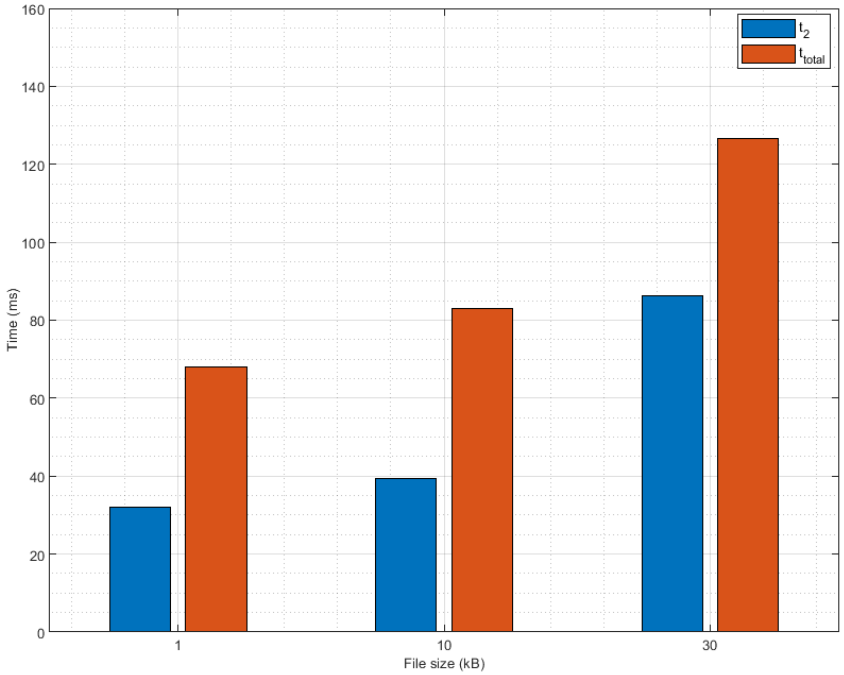

Figure 3.11. Graph showing the impact of the file size on t_2 and on t_{total}, where the averages of 1,000 iterations are computed for the server located in Frankfurt.

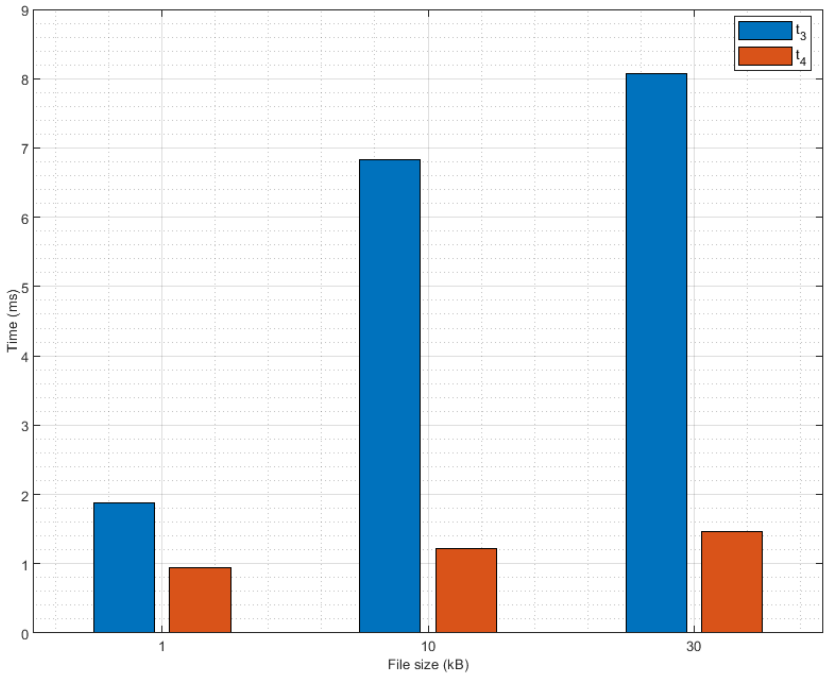

Figure 3.12. Graph showing the impact of the file size on t_3 and t_4, where the averages of 1,000 iterations are computed for the server located in Frankfurt.

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 have been 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 10 kB file from the server located in Frankfurt. Furthermore, the RSSI values 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 3.13, 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 values (the blue line).

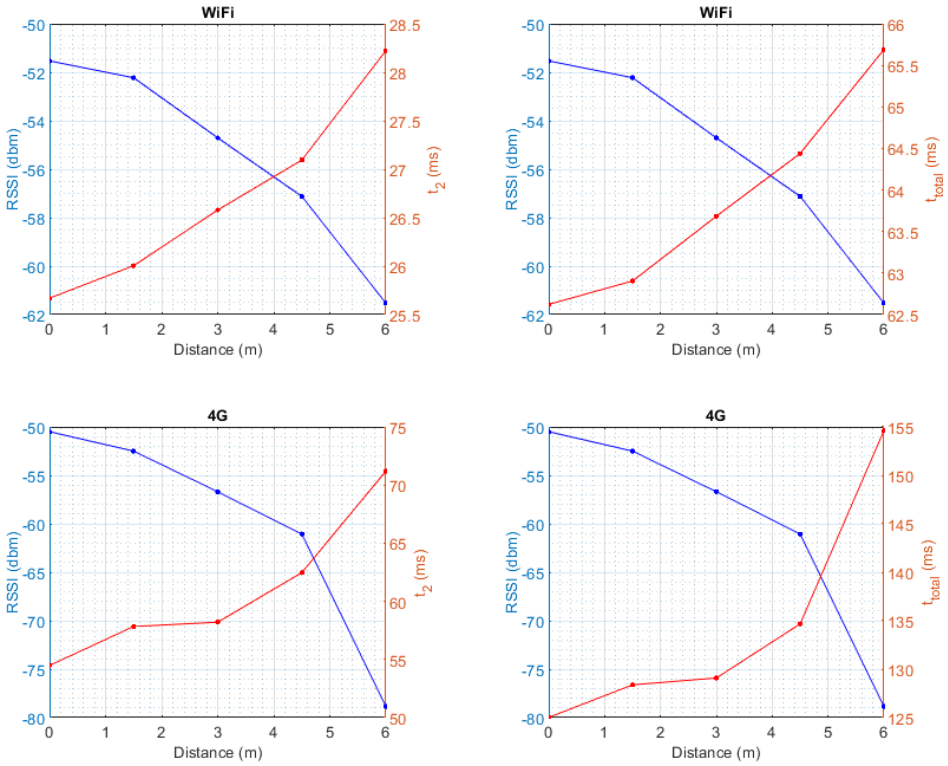

Figure 3.13. Graphs showing the relationship between RSSI and the distance between the client and the access points for both t_2 and t_{total}. This is done for 200 iterations, at each point, to the server located in Frankfurt with a file size of 10 kB.

Furthermore, it was also noticed that when using a 4G hotspot, the RSSI 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 4G, as can be seen in figure 3.14. It was anticipated that this trend would occur since a 4G hotspot has smaller antennas compared to a WiFi router, which resulted in lower bandwidth as well as a 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 section 3.2 contribute to higher latency when using a 4G hotspot.

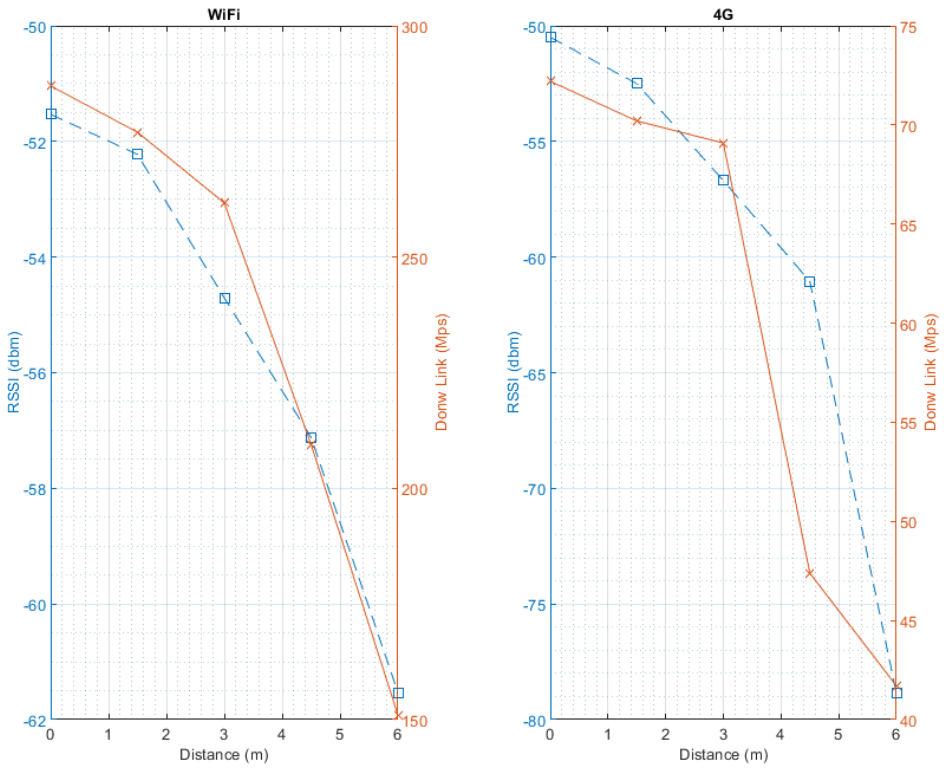

Figure 3.14. Graph showing the relationship between RSSI and the distance between the client and access points at different downlink rates. This is done for 200 iterations, at each point, to the server located in Frankfurt with a file size of 10 kB.

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 3.14, 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), which uses Orthogonal Frequency Division Multiplexing (OFDM) as a primary modulation scheme. For OFDM, 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 [Bappy et al., 2010]. In addition, each sub-carrier is modulated using the same or different modulation schemes depending on the channel condition. For example, at larger distances where the channel conditions are 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 means that at smaller distances, there will also be lower latency. Therefore, the downlink rate is relevant to include as an input, as it directly affects the file request process, that is t_2.

Transmission protocol

Both TCP and UDP have been tested in order to assess 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 in figure 3.15. 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.

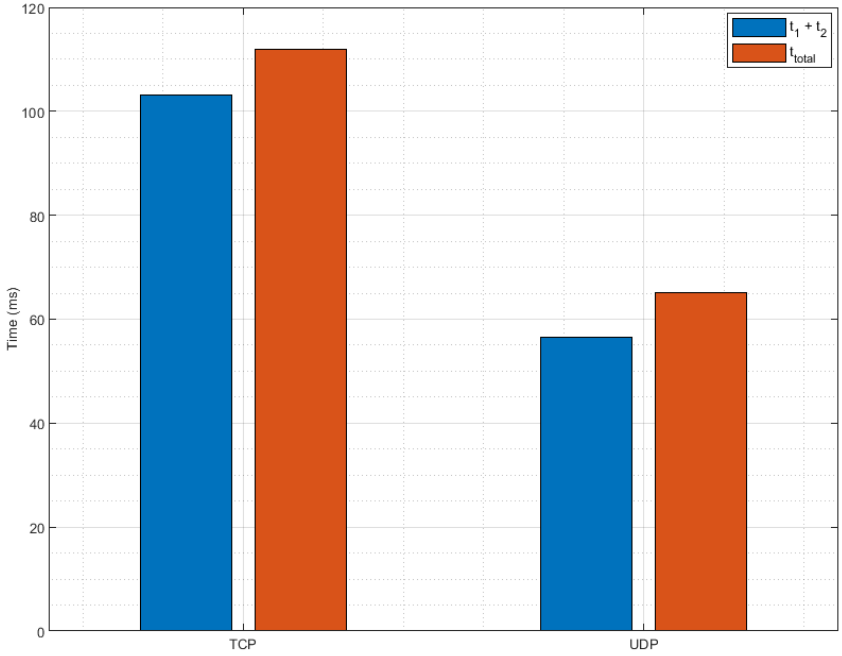

Figure 3.15. Comparison of TCP and UDP latency for downloading a 30 kB file, where the averages of 1,000 iterations are computed for the server located in Frankfurt.

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 packet. The MSS 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 done to ensure that the payload of each packet is optimised with the intention of achieving a reliable and efficient data transmission. The MSS is controlled indirectly through the MTU, which is the maximum amount of data that can be transmitted within a single packet over the network. As shown in figure 3.16, the MTU comprises the MSS in addition to 40 Bytes, which are as the IP and TCP headers. For instance, to attain an MSS of 700 Bytes, the MTU should be adjusted to 740 Bytes.

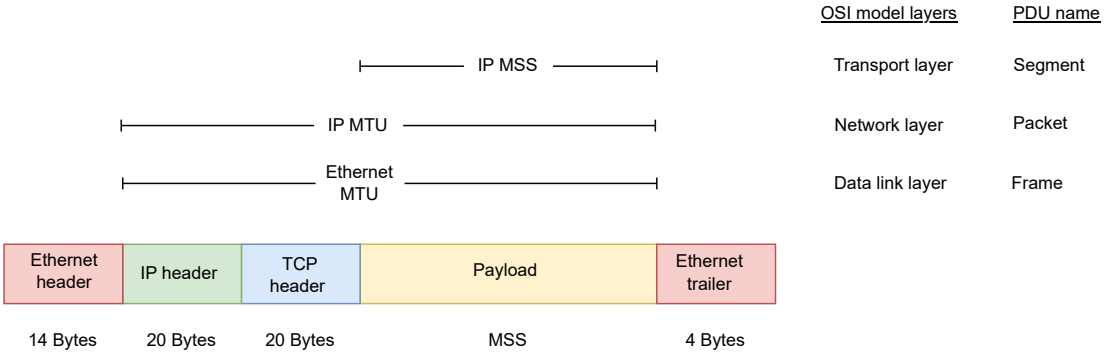

Figure 3.16. Comparison of MSS and MTU for IP and Ethernet protocols in the context of the OSI model layers and their corresponding Protocol Data Unit (PDU) names.

In order to assess the impact of the MSS on t_2 and t_{total}, three different MSSs have been tested, namely 300, 700, and 1300 Bytes. During these tests, the client consistently requested the same file size of 10 kB. As observed in the graph in figure 3.17, the findings indicate that decreasing the MSS 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, more transmissions are required to send the entire file. Figure 3.18 shows 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}.

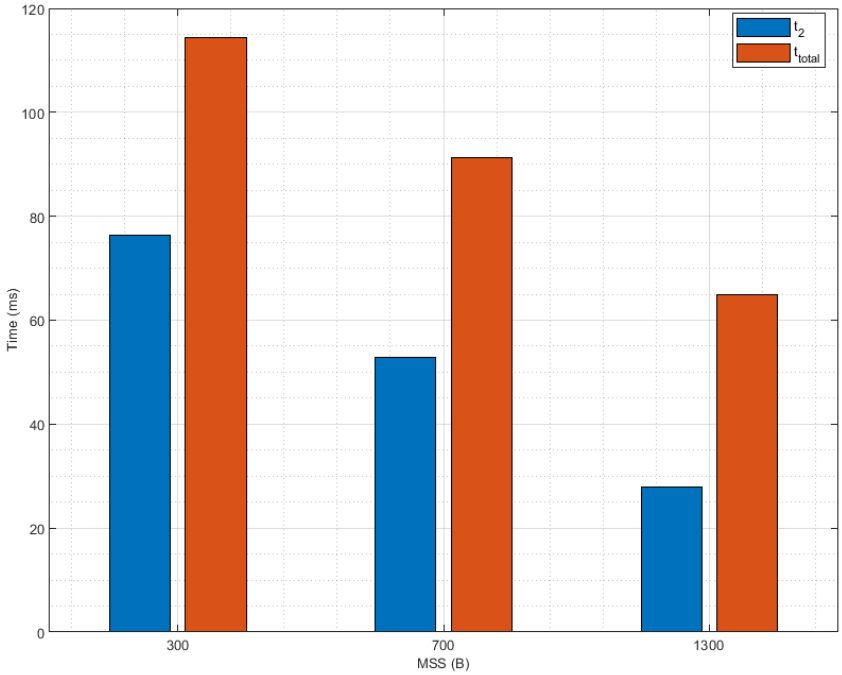

Figure 3.17. Graph showing the impact of the MSS on t_2 and eventually on t_{total}, where the averages of 1,000 iterations are computed for the server located in Frankfurt.

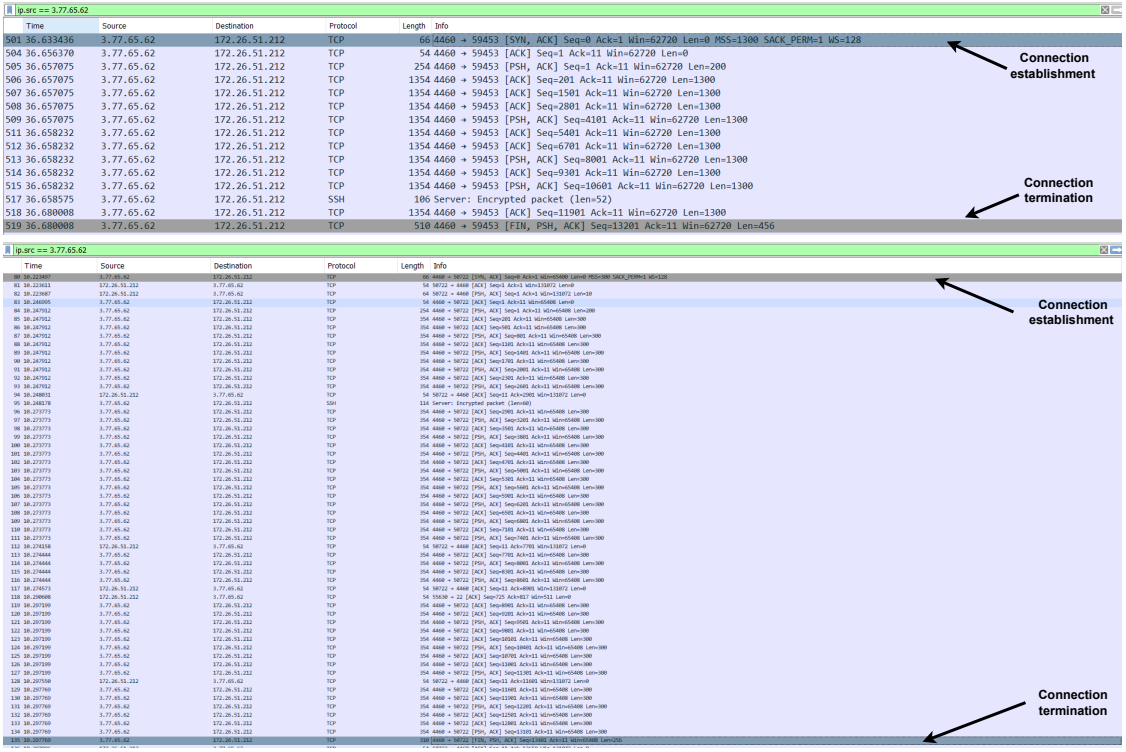

Figure 3.18. Comparing packet transmission count to complete a single 10 kB file download using a MSS of 1,300 (top block) and 300 (bottom block).

When using UDP, the MSS does not apply, since no connection negotiation is applied between the client and the server. Thus, to ensure a fair comparison between UDP and TCP when selecting a MSS 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.

Encoding algorithm

Before transmitting a file over any network connection, it is crucial to encode it. This has many purposes, including increasing the transmission efficiency and enhancing security. There are many encoding techniques, each of which have their own advantages and disadvantages. These include the data size and type, the purpose of encoding, whether the hardware or software supports it, as well as the available resources. To assess how different encoding schemes may affect t_2, t_3 and eventually t_{total}, two encoding schemes have been tested, namely Base64 and binary. As seen in figure 3.19, 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 bit in the file is represented as a Byte 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 latency since more data should be encoded at the transmitter, then transmitted and finally decoded at the receiver.

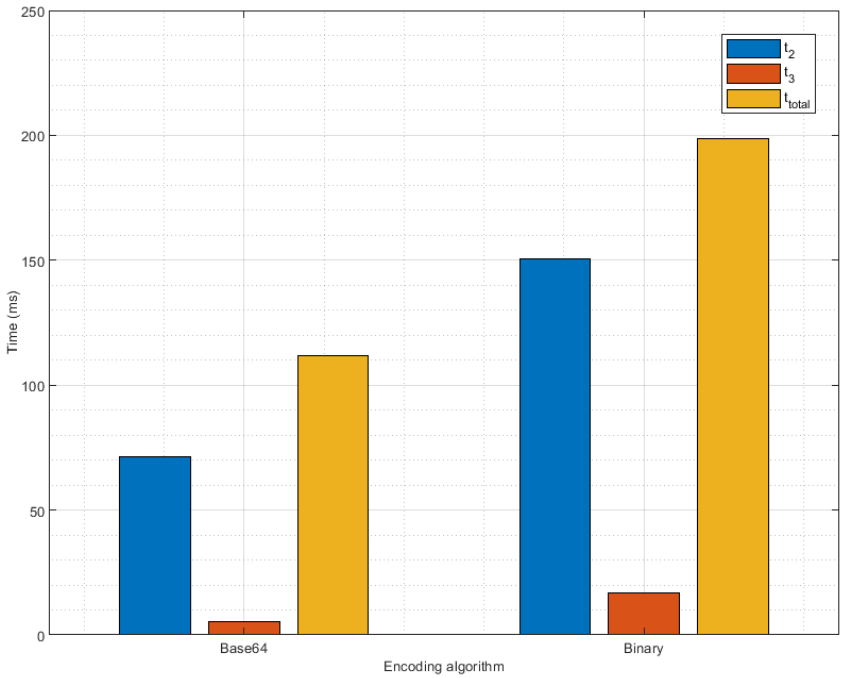

Figure 3.19. Comparison of t_2, t_3, and t_{total}, using Base64 and binary encoding for file transmissions of 30 kB.

Another technique that 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, since the largest file size of 30 kB is still very small, compression does not have a notable impact. Nevertheless, when applying the binary scheme to a 30 kB file, the file becomes $\approx$ 246 kB, where applying the file compression technique becomes 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 transmissions can affect the latency in several ways. Each transmission has some manner of overhead, which enables features such as error detection and correction, addressing, and flow control. The more transmissions, the more times the overhead will be added, leading to longer total download time. 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, thereby causing additional delays. In figures 3.17 and 3.18 it was previously shown that t_2 is affected by the MSS, 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 of transmissions. 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 transmissions was derived by testing all of the possible combinations of file sizes, the MTU, and the encoding algorithms. These tests involve downloading the file under all input combinations, and observing how many transmissions are needed to completely download a file. The results of theses tests can be seen on table 3.3.

Table 3.3. 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

3.2.1 Summary

As all of the factors included on table 3.1 now have been analysed, a new table 3.4 has been created to include the type of input as well as what values are possible.

Table 3.4. 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

The table includes the concept of *categorical* inputs, which are inputs that are not numerical. These require further processing to use with a model, which is investigated in section 3.3, alongside how the inputs will be used in general.

3.3 Processing the model inputs

To get the most out of data given as input or output for a model, it is useful or sometimes even required to preprocess it in advance, in order to make it compatible with the model. This can range from making the values lower so that the activation functions are utilised more, or simply making the value range of an input smaller. Depending on if the data is numerical or categorical, e.g., if it is a number or a label, different approaches can be used.

3.3.1 Numerical preprocessing

Numerical preprocessing is the process of transforming raw numerical data into data that is more suitable to build and train a ML 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, which is used for input features with 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 data set. It can also 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, by scaling the data to a fixed 0 to 1 range. An example of this normalisation can be seen in table 3.5, and the equation for normalising inputs is given by:

$$x_{norm} = \frac{x - \min(x)}{\max(x) - \min(x)} \tag{3.3.1}$$

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 data set respectively.

Table 3.5. Example tables showing how different file sizes 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 does not have a Gaussian distribution 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 with 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. An example of applied standardisation is seen on table 3.6, and the equation for standardising inputs is given by:

$$x_{stan} = \frac{x - \bar{X}}{s} \tag{3.3.2}$$

where x_{stan} is the standardised input data, $\bar{X}$ is the estimated sampled mean of the input data and s is the estimated sampled standard deviation of the input data.

Table 3.6. Example tables showing how different file sizes appear when standardised.

Default

File size (kB)
1
10
30

Standardised

File size (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:

$$x = x_{norm} \cdot (\max(x) - \min(x)) + \min(x) \tag{3.3.3}$$

Similarly, standardisation can be reverted with the following equation:

$$x = x_{stan} \cdot s + \bar{X} \tag{3.3.4}$$

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 3.3.3.

3.3.2 Categorical preprocessing

When dealing with categorical inputs, an example for this scenario is the location of the server, 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 locations.

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 3.7.

Table 3.7. Example tables showing how categorical locations are encoded into labels.

Default

Location
AAU
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 location any number, target encoding can be used to give a category the mean value of the target output values, so AAU could become the mean of all of the t_{total} values that include this location. An example of this can be seen on table 3.8.

Table 3.8. Example tables showing how categorical locations are target encoded.

Default

Location	t_{total}
AAU	26.02
AAU	23.84
AAU	25.41
Paris	48.53
Paris	56.23

Target encoded

Location	t_{total}
25.09	26.02
25.09	23.84
25.09	25.41
52.38	48.53
52.38	56.23

However, if a location 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:

$$\text{AAU} = \omega \cdot \mu_{categorical} + (1 - \omega) \cdot \mu_{overall} \tag{3.3.5}$$

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

$$\omega = \frac{n}{n + m} \tag{3.3.6}$$

where n is the amount of appearances for said location, 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 $\mu_{overall} = 32.52$, the locations from table 3.8 and their subsequent appearances n and categorical means $\mu_{categorical}$, would achieve the following values:

$$\begin{aligned} \text{AAU} &= \frac{3}{3+2} \cdot 25.09 + (1 - \frac{3}{3+2}) \cdot 32.52 = 28.06 \\ \text{Paris} &= \frac{2}{2+2} \cdot 52.38 + (1 - \frac{2}{2+2}) \cdot 32.52 = 42.45 \end{aligned} \tag{3.3.7}$$

In essence, this weighs the frequency of appearance for locations. 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 3.9, the location is turned from a location into a vector of either 1 or 0, depending on which is active.

Table 3.9. Example tables showing how categorical locations are one-hot encoded.

Default

Location
AAU
Frankfurt
Paris

One-hot encoded

AAU	**Frankfurt**	**Paris**
1	0	0
0	1	0
0	0	1

An apparent effect of one-hot encoding is that a column such as "Location" would expand into one column for AAU, Frankfurt, and Paris each. 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 locations chosen are much closer than any other possible inputs. Thus, it is ideal to analyse the data for correlation followed by manual removal of one of highly correlating columns.

Even if a single column is removed, the information about it still exists. For example, when Paris is removed, if both AAU and Frankfurt show 0, then it would mean that the location is in fact Paris. This is shown on table 3.10. This removal of a column is also sometimes known as dummy encoding, although it is also common to just drop the first of the columns instead of checking for correlation first [Basanisi, 2019].

Table 3.10. Example tables showing how categorical locations are dummy encoded.

Default

Location
AAU
Frankfurt
Paris

Dummy encoded

AAU	**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 location.

3.3.3 Outlier Analysis

Outliers are data points with values that are considerably different from the rest of the data points. This could be due to a number of reasons, for example measurement errors, heavy-tailed data or noise. 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 3.20 shows the occurrence of outliers for the t_{total} at Frankfurt for 200 measurements using a boxplot. A boxplot identifies outliers by visually depicting data points that fall outside the central range of the data set, using the Interquantile Range (IQR) which is further described below the figure. It can be assumed that the number of outliers in the data set 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 data set.

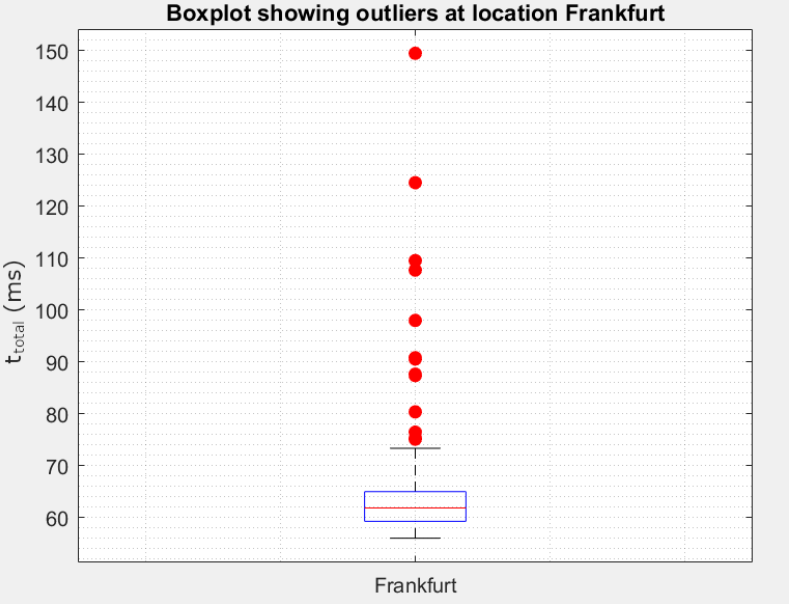

Figure 3.20. Boxplot showing outliers (red dots) in a distribution at Frankfurt.

Quantile intervals are one way of identifying potential outliers in a data set. These are a range of values in a distribution that are attributed with a certain probability of occurring. In order to use quantile intervals to find outliers, we can use the IQR. It is usually defined by two quantiles: Q1 and Q3, which represent the 25th percentile and 75th percentile of the data set, 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 the middle 50% of the entire distribution.

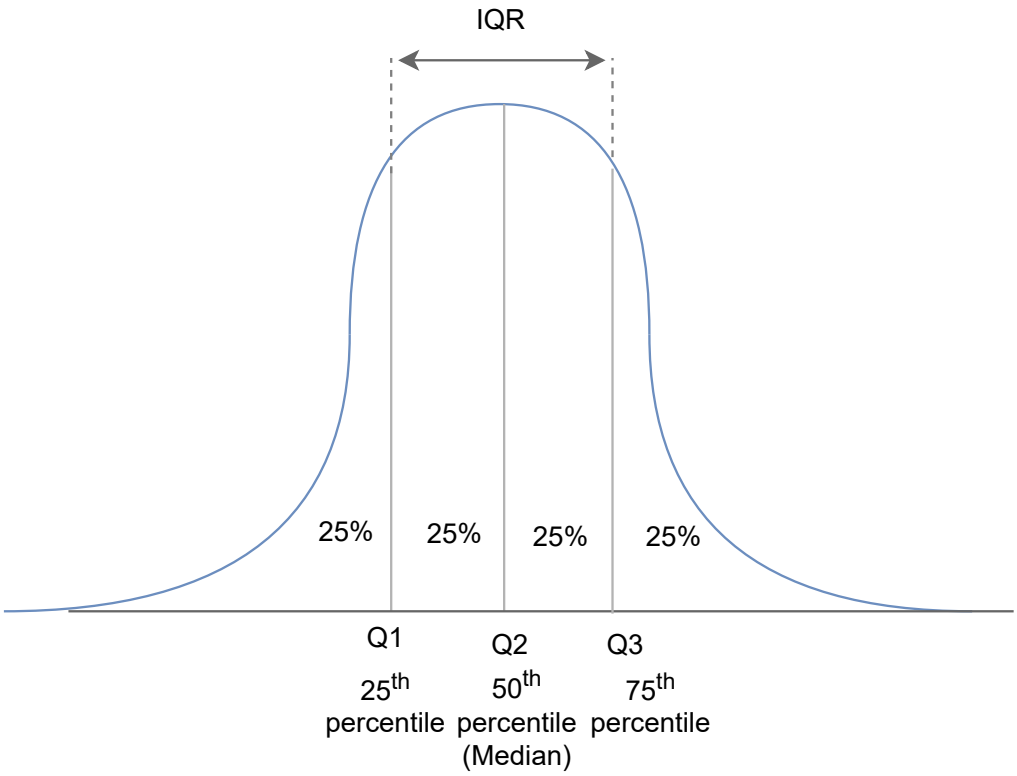

Figure 3.21. Quantile intervals.

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 data set, 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 data set 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 data set, can have a significant impact on the value of the standard deviation. Thus, IQR is better suited for distributions that are not normal.

The data that is collected to train the models are segregated based on the location, technology, and MSS, and are saved into individual .csv files. These .csv files are merged into a single data set during training. It is important to note that the outlier removal techniques are applied on each file as well as the combined final data set. 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. Then, on on merging all files, a 99% quantile range was applied, with the rest of the 1% data being considered outliers.

The figure 3.22 shows the results after applying a fixed quantile to the data set.

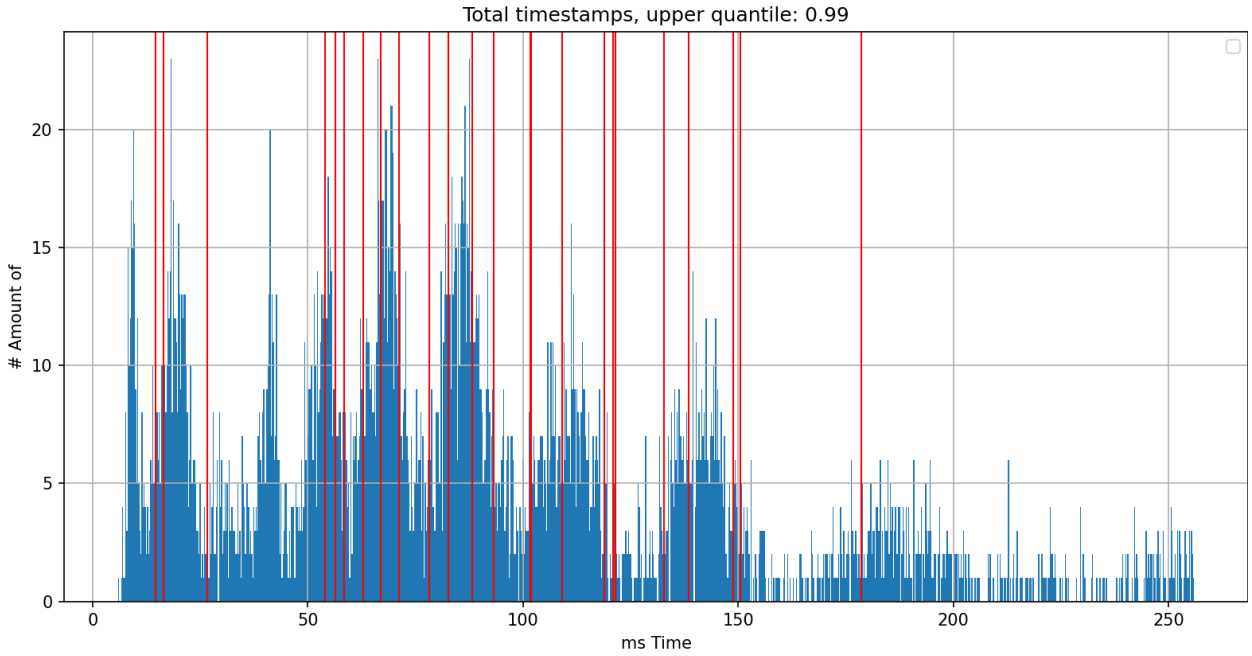

Figure 3.22. Data set after using fixed quantiles to eliminate outliers, with the red lines representing the means of the individual .csv files

Next, the IQR method was applied both before and after combing the files, and the figure 3.23 shows the results for the same.

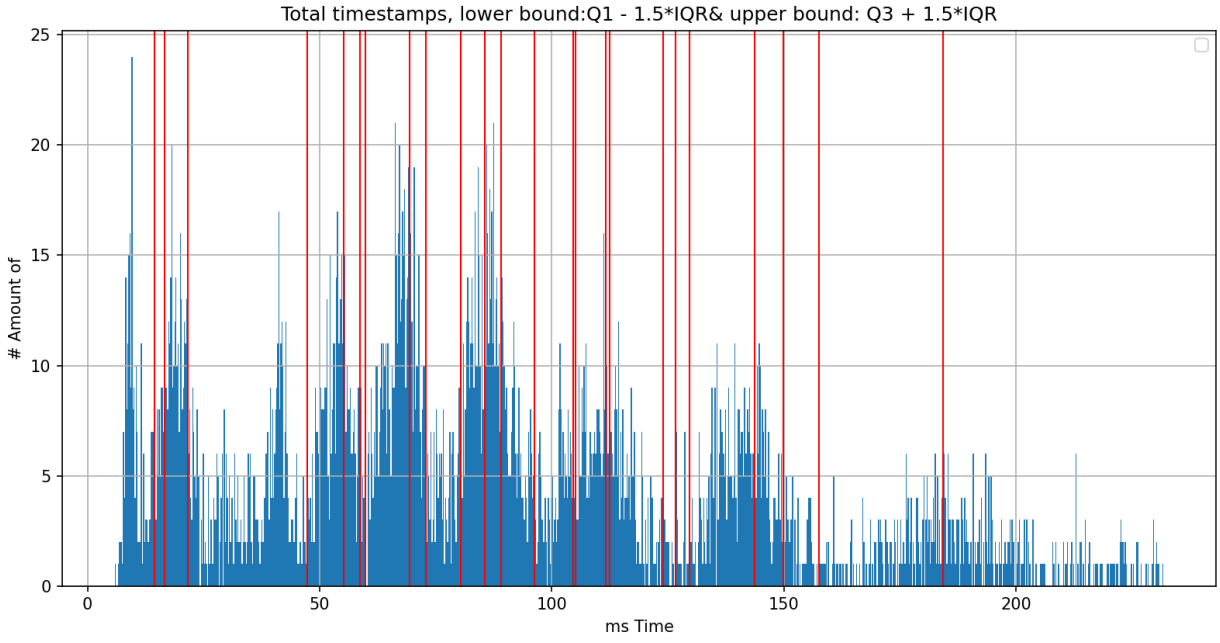

Figure 3.23. Data set after using IQR method to eliminate outliers, with the red lines representing the means of the individual .csv files

The figures show that the fixed quantile method is considering data points up to approximately 250 ms as non-outliers, whereas the IQR method is only considering data points up to 230 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. Furthermore, the results on table 3.11 show that model performs better when applying the IQR method as compared to using fixed quantiles, with a reduction of 2.5 ms in the Mean Absolute Error (MAE). Therefore, this project will be using the IQR method on the data set to detect and eliminate outliers. Additionally, the data obtained for this project indicate that it is not normally distributed based on the results from normality tests, it makes further sense to use IQR.

Table 3.11. MSE and MAE obtained for different outlier removal techniques.

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

Performance Evaluation and Optimisation 4

This chapter expands on the design decisions described in chapter 3, 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, followed by a comparison of how the choice of data preprocessing affects the model prediction. Finally, the impact of the amount of inputs used is observed through some introductory tests.

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

4.1 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, optimisation 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 performs during and after the training. These hyperparameters were introduced and explained in section 2.3.2.

When fitting a model with TensorFlow, e.g. training it, the epochs and the batch size hyperparameters are used. An example of the terminal during this procedure is included in figure 4.1. 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 testing data (referred to as validation, hence the val_ prefixes). The test data 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 4.1. Screenshot of the TensorFlow training process, showing the loss value change across multiple epochs. The loss and test 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 test 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 mitigate overfitting, methods like `early stopping` and `Dropout layers` are commonly utilised.

Dropout layers work by dropping random nodes in the layers, for the purpose of stabilising the training [Holbrook, 2023]. Dropping in this case means that the contribution of a node will become 0, meaning the output will be ignored. This also means that the nodes and weights will be mitigated from converging, which could otherwise lead to overfitting. Dropout layers can be implemented using `tf.keras.layers.Dropout()`, which randomly sets a number of nodes in a layer to 0. However, when attempted on NN2 from Models 4 and 5, the addition of dropout layers with a drop rate of 50% resulted in worse results, increasing the test MSE from 0.14 to 0.21, while 20% instead increased it to 0.16. However, it did result in the training MSE never surpassing the test MSE, meaning that overfitting and bias was eliminated. Because of this worsened performance, dropout layers are not used going forward.

Early stopping, on the other hand, refers to having the method halt the training process early, in order to prevent it from training too much on the same data, which could otherwise cause overfitting. Early stopping may require some manual training and the expected value, but can also be implemented with an early stopping callback from the *Keras* Python module. If a `patience = ""` is specified in this callback, it will prompt the model training to stop if no improvement was observed 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.

Another way to do early stopping is to know at which epoch the model starts to get worse performance, and then stop the training at that point. This can be accomplished through hyperparameter tuning, which is what was done in this implementation.

4.1.1 Hyperparameter tuning

While it is common to manually choose the hyperparameters and then adjust them after observing the model performance, it is also a possibility to automate the process. This is called hyperparameter tuning or hyperparameter optimisation, which is a general concept in NNs [Cloud, 2023].

A way to implement hyperparameter tuning is to simply train the model, then train a second model with different hyperparameters, and see how they compare. It can also be done using existing tools, for example the `keras-tuner` module for Python. The process for initialising and getting the final model can be seen in figure 4.2.

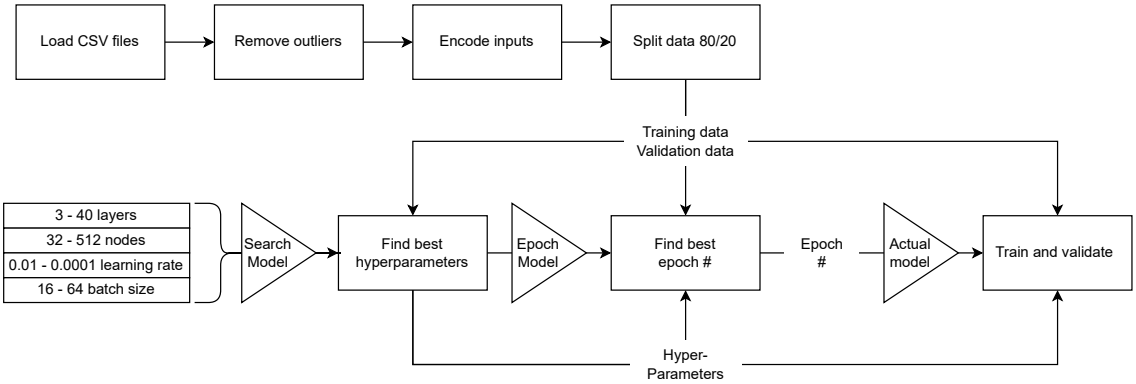

Figure 4.2. A diagram showing the process from preparing the test data to using it in model tuning and training.

The `keras-tuner` training can be initiated by creating a *Search Model* with the `RandomSearch()` tuner call, which randomly tries different combinations of hyperparameters defined during creation, 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. This outcome is based on the result of a model evaluation using this set of hyperparameters. To prevent the previously discussed overfitting, the evaluation is based on the testing data, which it has not been trained on yet, and thus, should not be biased towards.

The code used for creating the hypertuning models is shown in snippet 4.1, which initialises 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.

```python
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 4.1. 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 4.2.

```
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
```

Snippet 4.2. 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 increase the chances of finding 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 only find combinations that reduce the loss by a very small margin. This was further observed when testing 1,500 trials, which 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 larger resulted in progressively worse results, with 20+ layers generally not progressing at all. Similarly, an amount of nodes per hidden layer 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.

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, 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 detailed in section 3.1.

4.2 Analysis of Input Data Behavior

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

4.2.1 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 descent) 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 4.2.1.

$$output = variable_1 \cdot W_1 + variable_2 \cdot W_2 \tag{4.2.1}$$

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 3.3.2. Next, after encoding the inputs, multicollinearity can be evaluated by measuring the variance of the regression coefficient. This is done 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 4.3, the test shows a 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 4.3. Results of VIF test conducted on the encoded inputs, for NN2 from Model 4 and 5.

One approach to address this issue is by dropping RSSI, however dropping an input may be problematic since the model loses information, which 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 4.1 that dropping RSSI, which was causing multicollinearity, improves system performance with an approximate of a 0.66 ms reduction in MAE.

Table 4.1. MSE and MAE obtained for NN2 from models 4 and 5 before and after dropping RSSI.

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

4.2.2 Data sparsity

One of the most crucial challenges in data science is sparsity in the data [Nasiri et al., 2017]. This occurs when a data set includes a 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 combination occurrences 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 data set 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.

Initially, the values of these two inputs are mapped into 10 different equidistant intervals, each spanning between the minimum and maximum of their respective values. It should be noted that the choice of 10 intervals was arbitrary and selected beforehand, where its effectiveness and impact in capturing the characteristics of the data was later evaluated. Next, each interval is represented by a bin value from 1 to 10. By performing this process, downlink rate values were discretised into 10 bins, whereas the number of transmissions values were discretised into 6 bins. This deviation is due to the distribution of the the transmission values, which led to fewer bins representing the data. As a result, the number of unique combinations in the data set dropped from 1870 to 1184. Furthermore, the proportions of combinations with less than ten appearances decreased 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 in the data set. As can be seen on table 4.2, an improvement has been achieved where the MSE as well as MAE are reduced by 0.012 and 1.59 ms respectively. However, an attempt to decrease the inputs into even fewer bins is made by selecting the number of intervals to 5. Here the downlink and transmissions values were mapped into 5 and 4 bins respectively, resulting in a decrease of combinations with fewer than ten occurrences to 61.48%. Reducing the sparsity to a small value led to smoothing or removal of features in the data, as can be seen from the higher MSE and MAE in the same table. Therefore, it was decided to keep the discretised values for downlink rate and transmission numbers to 10 and 6 respectively.

Table 4.2. MSE and MAE performance for NN2 from Models 4 and 5 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

4.3 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 3.3. Since Model 1 represents the true output and does not rely on a NN to generate predictions, it is excluded. As for the rest of the models, because the same type of inputs are used, only Models 2 and 3 are tested as it is assumed that Models 4 and 5 would behave similarly.

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.

In this evaluation, the performance of Model 2 is assessed using both normalised and standardised inputs, as described in chapter 3.3. The results of these evaluations are presented below on table 4.3, where the MSE and MAE are used as metrics to evaluate the performance of the different preprocessing techniques.

Table 4.3. Model 2 performance with different numerical preprocessing techniques.

Preprocessing technique	**MSE**	**MAE (ms)**
Normalisation	0.0015	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 lead 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.

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 on table 4.4.

Table 4.4. 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 on table 4.4, it appears that there is a contradiction between the MSE and the MAE values. Specifically, the combination of variables that produces the better MSE, which includes normalised values, has a lower MAE. Conversely, the combination that produces the better MAE, which includes standardised values, has a lower MSE. This discrepancy could potentially be explained by the fact that the range of the normalised values is between 0 and 1, while the standardised 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 standardisation for numerical data and one hot encoding for categorical data can be concluded to be the preferred choice based on the results.

4.4 Validation

When a model has been tuned and later trained, it is ready to be tested. During the training, the model can be given the test data set in order to see how it performs on data that it was not previously trained on. This is particularly useful to see if the model has become biased towards the training data, as bias would cause significantly different training and test results. To reiterate, using the test data during training does not modify the models or affect the calculations of the weights.

However, it is also relevant to see exactly how the model would perform when 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. The same preprocessing that took place during training should also be done during prediction. In the event that the preprocessing was applied on the outputs, as is the case in this project where all of the outputs are numerical and standardised, the preprocessing must also be reverted. This is necessary to compare each t with the predicted $\hat{t}$ outputs. The same estimated means and standard deviation values obtained during training can be applied for the real test, provided the circumstances the data were obtained under are the same.

As an example of a prediction, the NN1 from Models 4 and 5 as explained in section 3.1 has been tested. This NN gives an output $\hat{t}_1$, and when compared to the true time t_1, it gives a result shown in figure 4.4 alongside the absolute error. Only the first 100 points from the test are included for clarity. 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. Therefore, it does not give a good representation of how much t_1 actually varies. The transmission protocol is also considered an input, as seen on table 3.4, but when UDP is used, there will be no connection process, and t_1 would as a result be 0.

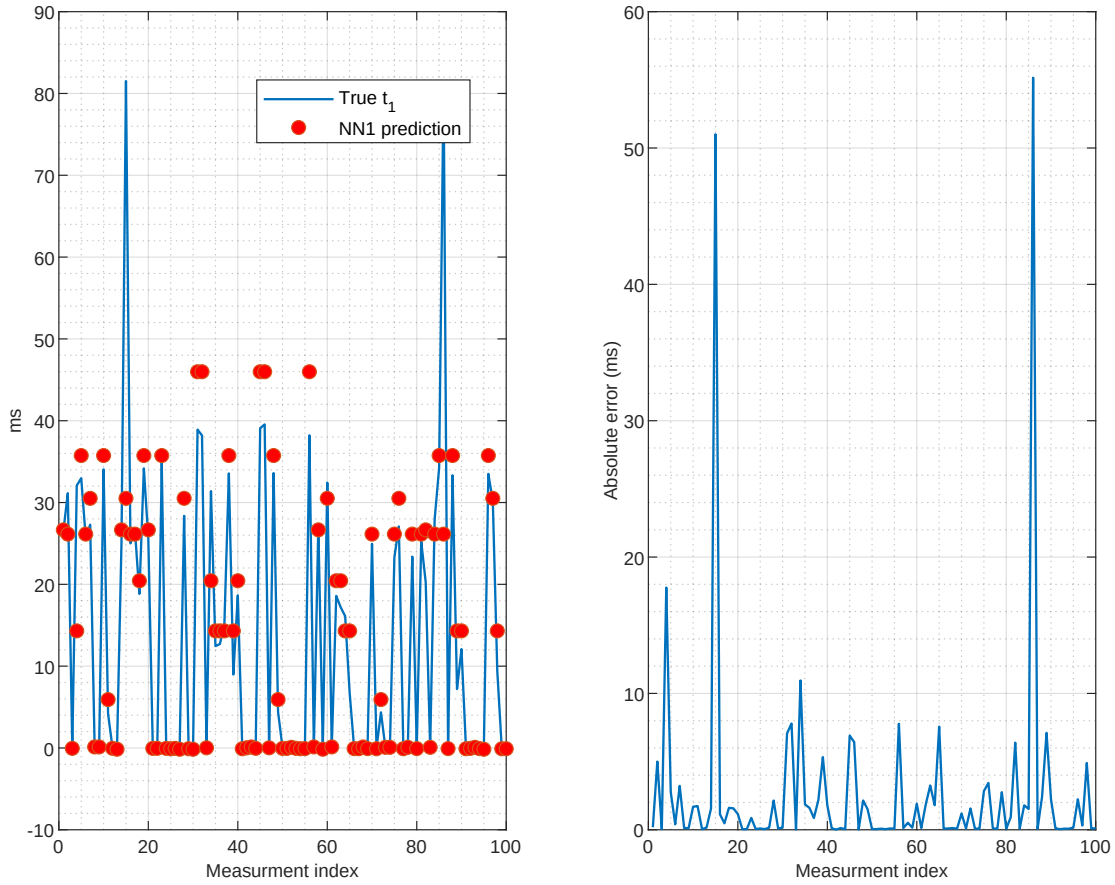

Figure 4.4. (Left) Graph showcasing the real t_1 measurements and predictions $\hat{t}_1$. (Right) The absolute error observed for NN1 in Models 4 and 5.

4.4.1 Amount of inputs used

The initial tests of the NN1 from Models 4 and 5 performance showed that the small amount of categorical inputs led to only eight possible prediction times, resulting in poor predictions. However, with NN2 from Models 4 and 5 in section 3.1.3, there are nine different input categories as shown in table 3.4. This is because 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 from Models 4 and 5 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 $\hat{t}_2$ is shown for NN2. Here, only the distance, technology, file size, and MTU are included as inputs. The first for 100 points of these results can be seen in figure 4.5 with the corresponding absolute error.

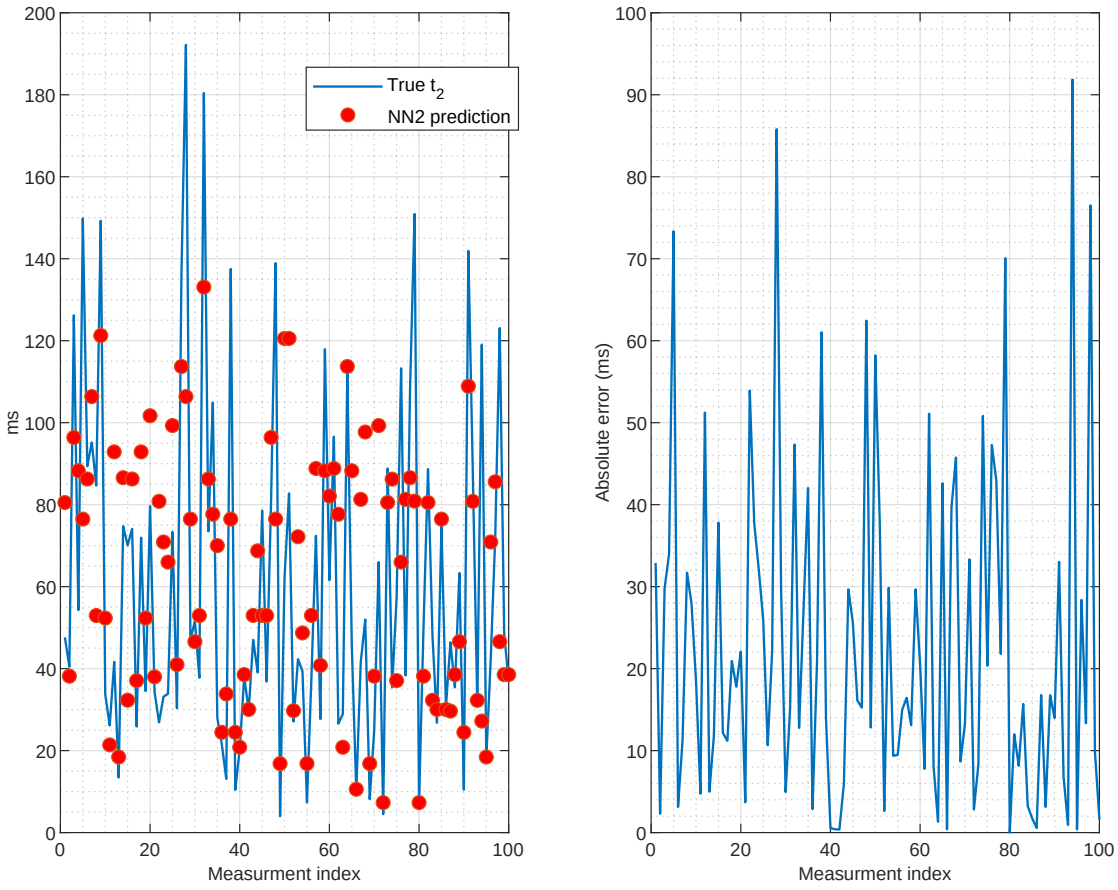

Figure 4.5. (Left) Graph showcasing the real t_2 measurements and predictions $\hat{t}_2$ when four inputs are chosen. (Right) Absolute error observed for NN2 in Model 4 and 5 when trained on four inputs.

The MSE obtained for the model along with the MAE in ms is also provided in table 4.5.

Table 4.5. MSE and MAE obtained for NN2 from Models 4 and 5 with four inputs for 1,918 predictions.

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: distance, technology, file size, MTU, protocol, encoding, downlink rate, and number of transmissions. The addition of these four inputs brings the total number of inputs to eight, compared to the previous number of four. While RSSI was also one of the inputs present in table 3.4, it was shown to introduce multicollinearity in section 4.2, and therefore not included in this test and onwards. The first 100 points of these results are shown below in figure 4.6 with the corresponding absolute error.

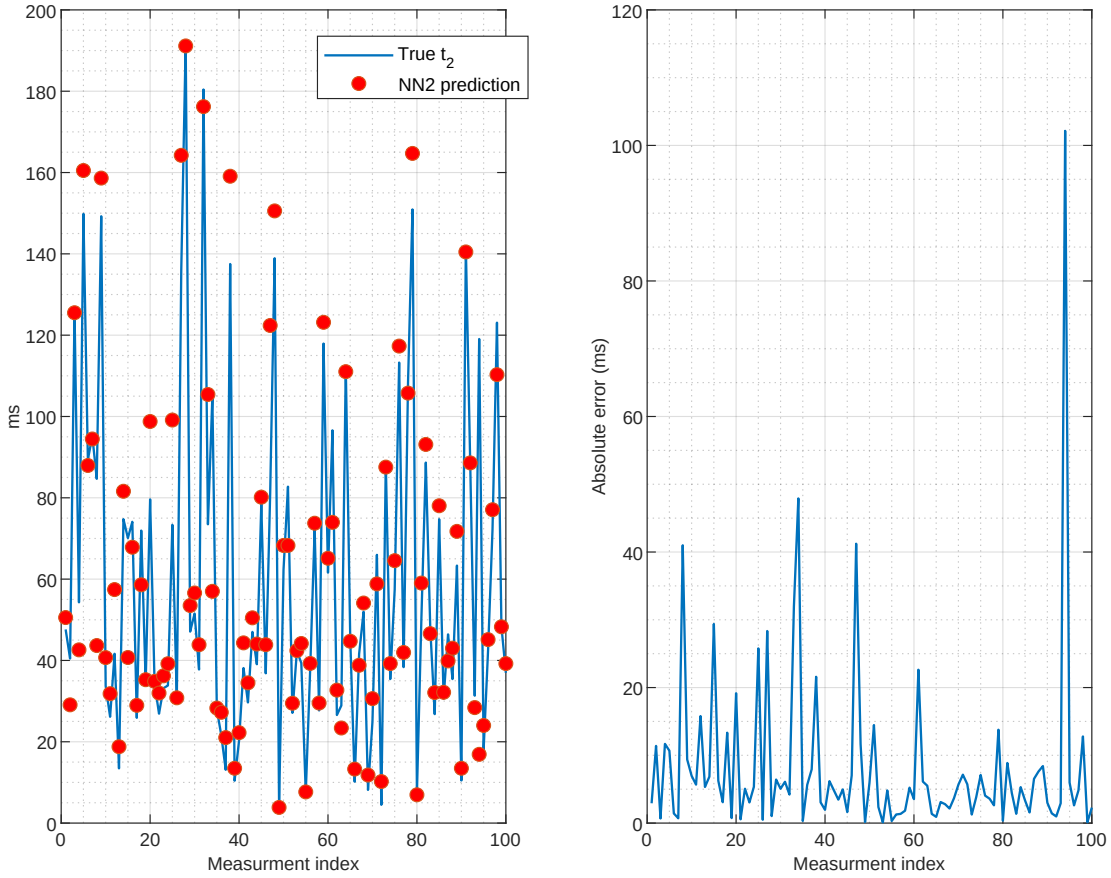

Figure 4.6. (Left) Graph showcasing the real t_2 measurements and predictions $\hat{t}_2$ when all eight inputs are chosen. (Right) Absolute error observed for NN2 in Model 4 and 5 when trained on eight inputs.

The MSE obtained for the model along with the MAE in ms is also provided in table 4.6.

Table 4.6. MSE and MAE obtained for NN2 from models 4 and 5 with eight inputs for 1,918 predictions.

No. of inputs considered	**MSE**	**MAE (ms)**
8	0.131	10.71

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 10.71 ms, indicating a significant decrease of 54% in the absolute error on adding more inputs. This gives the impression that having more inputs would provide the model with more information to detect patterns or relationships within the data, leading to more precise predictions.

However, 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.

Over 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 t_1 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 t_1 as an input from further analysis. While this did not result in worse performance or generalisation for the model, it helped prove that not all information added is beneficial to the model.

4.5 Summary

In this chapter, it was found through 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. Categorical inputs performed best when preprocessed with one-hot encoding, and numerical inputs with standardisation. Lastly, it was found that generally more inputs gave more information, leading to better prediction capabilities.

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

Validation and Performance Testing 5

The previous chapters cover the difficulties 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 real-time alongside the PT.

After each test phase 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.

5.1 Test Overview

The models to be used for the tests are the same as described in section 3.1, however Model 5 is implemented with two different variants: A and B. In Model 5A, the NN5 used is trained with the real timestamps t_1 to t_4 as inputs. In Model 5B, the NN5 is instead trained on estimated timestamps $\hat{t}_1$ to $\hat{t}_4$ from NN1 to NN4. This is to give an idea of whether training on true or estimated data will give better performance, when during the real test, the estimated timestamps are used.

For reference, a list of the models used and the output values can be found on table 5.1. These models have been tuned, trained, and had data optimised based on the discoveries made in chapters 3 and 4.

Table 5.1. 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 $\hat{t}_{total}$, which is to be compared with the t_{total} obtained from the PT per iteration of the process. Additionally, Models 4 and 5 generate and use estimated timestamps $\hat{t}_1$ - $\hat{t}_4$.

The data capture follows the same procedure as during the training phase, meaning the client only communicates with one server at a time, with different technologies, protocols, file sizes, and more. For the prediction part of the test, a large number of data are collected at once to be used for offline predictions. This allows for outliers and timeouts to be filtered using the same IQR values as during training. To test the time to predict, all of the models are run alongside the PT as a real-time DT, and the time it takes to predict t_{DT} are compared with the t_{PT}.

5.2 Model Loss Performance

Each model's performance is evaluated in this section based on the MAE of the predicted $\hat{t}_{total}$. To conduct this evaluation, a new data set has been specifically generated for offline testing. Data collection for this data set is based on location, technology, and MTU. For each location, there are six files - one for each of the three MTU values (340, 740, and 1340) utilising both 4G and WiFi technologies. For instance, data for the Frankfurt location would be separately collected for each of the three MTU values using both 4G and WiFi technologies, resulting in six files for Frankfurt alone. The same is applied for other locations. Considering that there are four locations in total, there are a total of 24 files or combinations. To conduct this evaluation, 100 measurements have been taken for each combination, thereby resulting in 600 measurements for each location.

After collecting the data for offline testing, the IQR method is applied to remove outliers. This process is similar to what was done during the test in Chapter 4, i.e., the IQR method is applied individually on each file and also once on the entire data set after merging all 24 files. Once the IQR method is applied to remove the outliers, the performance of each model is evaluated based on the modified data set. The absolute error is assessed both as the MAE and based on the variance. The results for each model are presented on table 5.2. The performance of each of these models is then compared to each other in order to determine which model performs the best.

Table 5.2. Models and their respective MAE and variance for the offline tests.

Model	**MAE (ms)**	**Var**
Model 2	4.71	15.09
Model 3	26.41	451.66
Model 4	26.74	496.13
Model 5A	25.77	445.51
Model 5B	23.92	360.91

The results show that Model 2 outperforms the other models in terms of both MAE and the variance of the absolute error. For the rest of this chapter, this is referred to as the MAE and variance. However, Model 2's performance was expected, as it utilises the timestamps t_1 through t_4 as inputs. This implies that the model does not predict the actual values but rather estimates the total latency after receiving all the true times as inputs.

This means that while it has a lower MAE and variance, it may not be the most efficient or practical model in real world applications, as it defeats the purpose of predicting the latency before it actually happens. On the other hand, Models 3, 4, 5A and 5B predict the latency based on the available information at the time, instead of waiting for the timestamps. Therefore, although these models have a higher MAE, they are more practical in real life applications and thus, Model 2 is not tested further.

On comparing the performance of Models 3, 4, 5A, and 5B, it is evident from the results that Model 5B exhibits the best performance in terms of MAE and variance. It can also be observed on table 5.2 that as the number of NNs in a model increases, its prediction accuracy tends to improve. This can also be observed in the Cumulative Distribution Function (CDF) graphed in figure 5.1, where Model 5B has the highest cumulative probability compared to other models. Furthermore, in the case of Models 5A and 5B, where both comprise five NNs, Model 5B outperforms Model 5A. This can be attributed to the fact that NN5 in Model 5B is trained on the predicted timestamps $\hat{t}_1$ through $\hat{t}_4$ from NN1 to NN4 which are the same inputs it receives during training. This is as opposed to Model 5A, where NN5 is trained on true timestamps t_1 through t_4 as inputs.

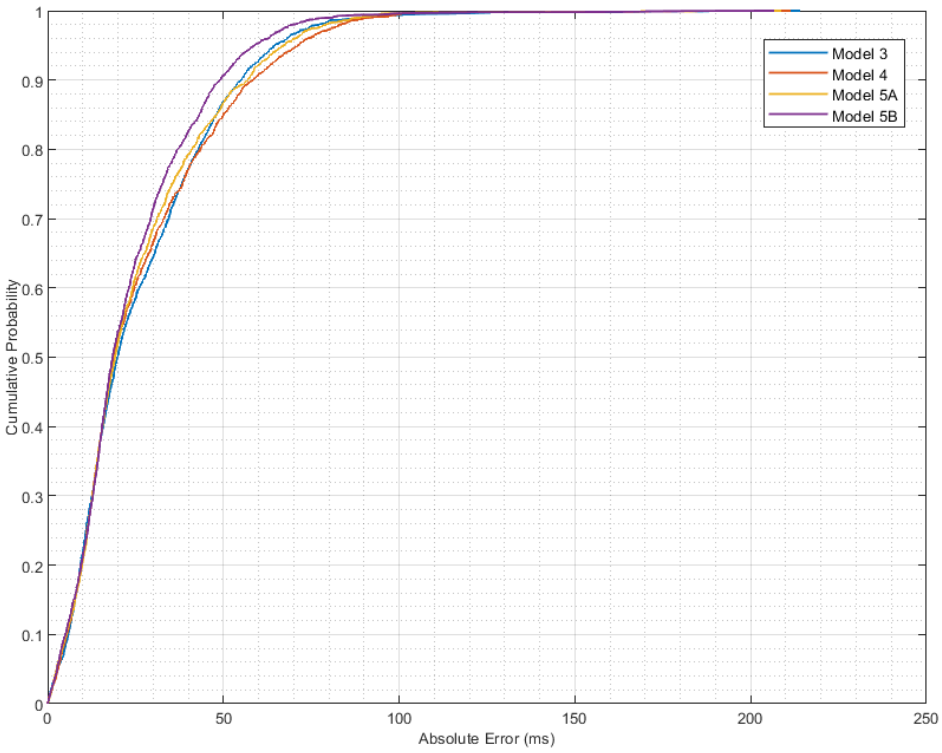

Figure 5.1. CDF of the error between the predictions $\hat{t}_{total}$ and true time t_{total} for Models 3, 4, 5A, and 5B.

5.2.1 Model prediction performance

Although the predictions show to have MAEs of around 25 ms, this may not reflect the individual absolute error values. For this, the $\hat{t}_{total}$ absolute error values for Models 3, 4, 5A, and 5B were saved and graphed on figure 5.2.

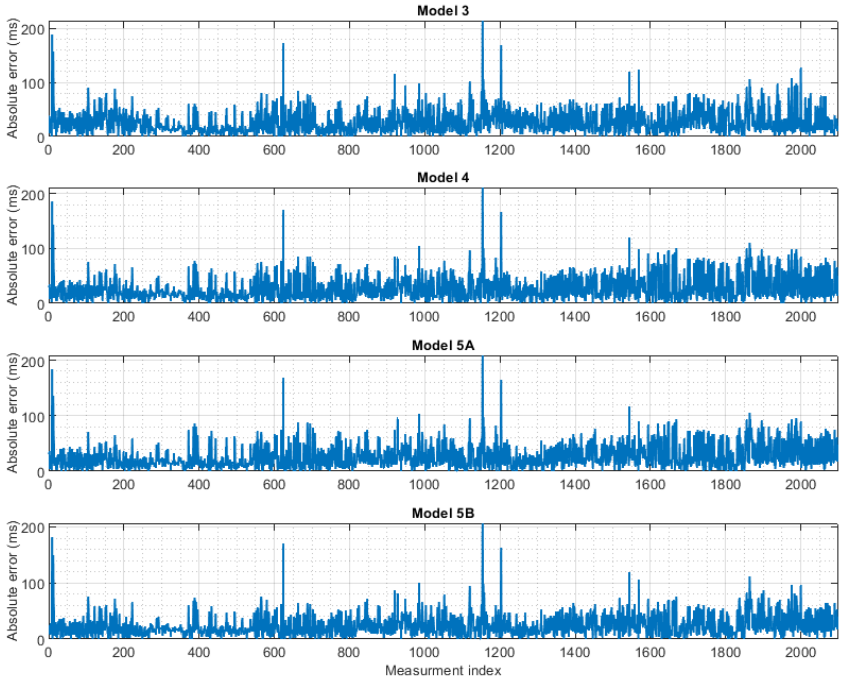

Figure 5.2. The error between the predictions $\hat{t}_{total}$ and true time t_{total} in different measurement instances for Models 3, 4, 5A, and 5B.

As can be seen, the error values frequently reach 50 ms, and sometimes up to 200 ms. Despite these much larger error observations, there are still enough accurate predictions to obtain a lower MAE. It also helps visualise the large variance values observed in table 5.2.

Table 5.3 takes t_{total} from the measurement indices with the largest spikes of error from figure 5.2, and shows the $\hat{t}_{total}$ for each model with the same set of inputs. Ten new data samples were taken with the same conditions and inputs as these indices, and the means of these ten t_{total} were computed.

Table 5.3. Models and their respective MAE for the offline tests.

Index	t_{total} **(ms)**	**Model 3**	**Model 4**	**Model 5A**	**Model 5B**	**Mean** t_{total}
9	232.90	44.67	47.87	48.73	51.65	63.67
12	280.34	124.50	136.50	144.74	130.14	128.18
624	221.93	49.98	50.87	53.00	51.68	64.12
985	153.02	54.02	49.15	50.17	53.30	43.96
1154	279	65.07	67.49	70.29	72.39	74.88
1203	223.29	55.48	57.11	59.60	59.87	69.41
1544	200.69	80.67	80.81	84.33	80.37	82.54
1864	222.48	118.17	111.68	117.81	111.16	120.15

The results show that even with the same inputs, there are large spikes of error that did not get removed by the IQR. That means that the inputs provided to the models are not representative of the entire transmission process, which is a consequence of only having access to data available as a client. Conversely, if there was a way to obtain information about the rest of the link, e.g. congestion, retransmission, load balancing, and queues, it may make the predictions more accurate.

5.3 Isolating the impact of inputs on performance

Following up on the previous observation of variation in t_{total}, the different inputs are isolated to see exactly where the models have trouble predicting a close $\hat{t}_{total}$. This means for all of the test data, all combinations with the different inputs from distance, technology, file size, and protocol are compared to each other.

5.3.1 Distance

These tests single out all of the input combinations with the four different distance parameters, so AAU, Fjerritslev, Frankfurt, and Paris, and predictions are compared.

Table 5.4. Model performances on data from different locations, measured by MAE and variance.

	AAU		**Fjerritslev**		**Frankfurt**		**Paris**	
Model	**MAE**	**Var**	**MAE**	**Var**	**MAE**	**Var**	**MAE**	**Var**
Model 3	23.10	466.67	25.34	449.95	30.18	452.86	41.19	891.13
Model 4	21.69	391.92	26.44	413.33	29.63	488.23	50.83	1,431.2
Model 5A	21.03	327.31	26.77	421.93	27.91	418.81	50.50	1,139.5
Model 5B	21.82	403.09	24.34	344.07	26.44	383.71	44.18	873.04

As can be seen on table 5.4, the MAE and variance increase as the distance gets larger, and increase more significantly once the input combinations from Paris are used. It can be assumed that this is due to a variety of factors that cannot easily be quantified via client-side observations, such as the conditions of the radio access network, load balancing, congestion, and different routes used compared to when the training data was captured. While it is true that latency increases over distance, as documented in section 3.2, the jump in error between Frankfurt and Paris was not expected. Another observation is that inputs captured for AAU used on Model 3 and 5A gave a worse variance than that of Fjerritslev. While not a significant difference, it may be caused by AAU being a university with potentially many students and employees connecting to the network at the same time.

As the Paris inputs performed worst for all of the models, we thought to be relevant to include a small test with a model trained only on the set of inputs from Paris. This was done in order to see if having a specialised model would perform better than the generalised ones. The results of this test can be seen on table 5.5, which shows the MAE and variance for each, as well as the observed percentage improvement for the MAE.

Table 5.5. Comparison between the generalised models versus the specialised Paris model.

	Generalised models		**Specialised models**		**Improvement**
Model	**MAE**	**Var**	**MAE**	**Var**	**MAE**
Model 3	41.19	891.13	33.16	755.73	21.60%
Model 4	50.83	1,431.2	40.56	1,418.4	22.48%
Model 5A	50.50	1,139.5	40.79	1,439.7	21.27%
Model 5B	44.18	873.04	38.30	1,015.9	14.25%

A trend for all of the models is the reduction in MAE by almost 10 ms, showing that using a specialised model is beneficial compared to only using a single generalised model. The variance on the other hand does not always increase with as consistent of a magnitude, and in the case of Model 5A, actually worsens. However, this increased variance could potentially spur from the smaller data given, as only the input sets with Paris are included. As a result, the trained model may have been subject to either overfitting or underfitting. To address this, future work could include attempts to analyse this behavior, and potentially accommodate and overcome the variation seen with more relevant data.

5.3.2 Technology

In this test, the measurements with the WiFi and 4G inputs are tested individually.

Table 5.6. Model performances measured by MAE and variance for WiFi and 4G.

	WiFi		**4G**	
Model	**MAE**	**Var**	**MAE**	**Var**
Model 3	17.46	393.95	30.99	619.37
Model 4	13.97	308.80	33.40	931.0
Model 5A	14.87	307.42	34.20	606.17
Model 5B	14.10	331.72	29.98	469.35

The test results show that the models make better predictions when given input data from WiFi compared to 4G, as seen on table 5.6. This could be due to several reasons, one of which being that when test data set was collected, the 4G network was more congested. Consequently, the packets take additional time to travel to their destination, leading to a higher level of jitter, thereby the models encountered increased difficulty in making accurate predictions. On the other hand, WiFi networks are less likely to experience congestion since it has shorter transmission distances, which can result in more stable and faster data transmission, making it less challenging for the models to make accurate predictions.

5.3.3 File size

This test isolates only the file size as an input and compares the model predictions for the 1 kB and the 30 kB files.

Table 5.7. Model performances measured by MAE and variance for 1 kB and 30 kB files.

Model	1 kB		30 kB	
	MAE	**Var**	**MAE**	**Var**
Model 3	16.37	264.06	32.69	471.89
Model 4	20.40	271.18	42.82	620.87
Model 5A	15.75	235.14	44.90	634.29
Model 5B	16.54	261.23	38.93	557.79

The results on table 5.7 show that as the file size increases from 1 kB to 30 kB, the MAE and the variance increase by more than double. This suggests that the models predict more accurately for smaller file sizes. This could potentially be because larger files can take longer to transmit as it requires more packets to be transmitted, which increases the probability of packet loss and retransmissions, making it difficult for the model to make close predictions. In addition, larger files can have different characteristics and behavior compared to smaller files, such as different variations in network traffic behaviour. This can make it more challenging for the model to accurately predict the transfer time as this information is not available at the client-side.

5.3.4 Protocol

The following test provides the MAE and the variance for all models when only the protocol is isolated as an input. This is done in order to evaluate the individual performances of both UDP and TCP.

Table 5.8. Model performances measured by MAE and variance for 1 kB and 30 kB files.

Model	UDP		TCP	
	MAE	**Var**	**MAE**	**Var**
Model 3	23.69	615.26	55.44	770.61
Model 4	17.97	315.26	58.59	744.85
Model 5A	19.97	429.59	60.09	810.51
Model 5B	21.60	489.39	53.72	651.54

The results presented on table 5.8 suggest that the models perform significantly better when predicting the latency for UDP as compared to TCP. Across all the models, the predictions for UDP are on average closer to the true value by 32 ms. This difference in performance could be attributed to a higher variance in latencies present in the test data set, that may be a result of a high number of retransmission. This mechanism is a feature in TCP, which occurs when the transmitter has to retransmit the packets in case the receiver does not receive them correctly or does not receive them at all within a certain timeout period. Whereas, for UDP, the packet is simply dropped in case of a timeout, and these cases are removed from the data set. As a result, when there is a retransmission in TCP, the model does not anticipate this increase in latency, leading to inaccurate predictions.

However, although UDP gives a much lower MAE, the variance is still high, which may be attributed to UDP being unreliable in general.

5.3.5 Combining inputs

After analysing the model performance on the individual inputs i.e., distance, technology, file size and protocol, these inputs are now combined to analyse the overall model performances. The combined inputs are divided into two categories: the best case combination and the worst case combination. The best case combination consists of only the inputs from each category on which the models performed the best i.e., AAU for location, WiFi for technology, 1 kB for file size, and UDP for protocol. Similarly, the worst case combination consists of only inputs from each category on which the models had the least impressive performance i.e., Paris for location, 4G for technology, 30 kB for file size, and TCP for protocol. Examining these extreme cases can provide the bounds or thresholds of how well the model performs. Note that these best and worst case combinations are assuming each input is independent. The results for this analysis is presented in a heatmap table in figure 5.3.

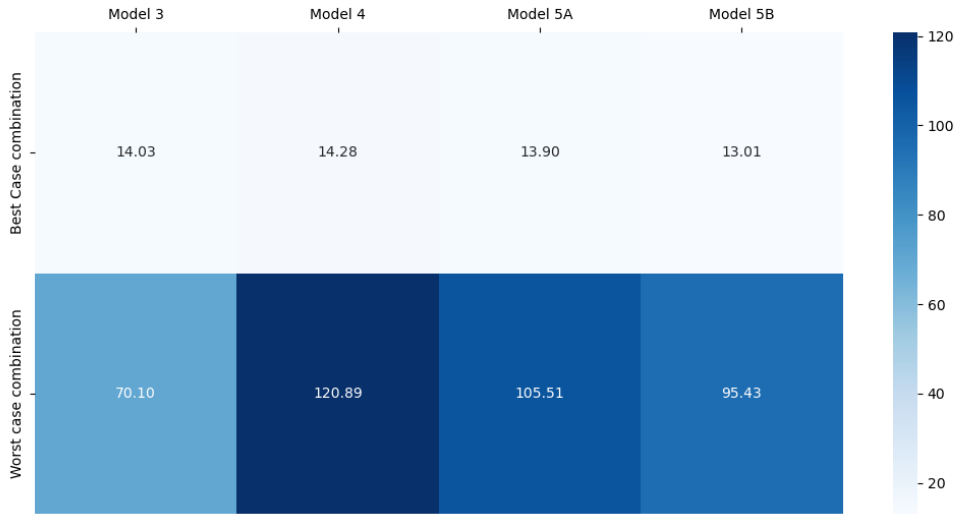

Figure 5.3. MAE in ms for the best and worst case combinations of the technology, protocol, file size, and location inputs.

The results show expected behavior, revealing that all models demonstrate better performance when presented with the best case combinations. This establishes a threshold of the prediction performance based on both the best and the worst case scenarios.

5.4 Live Latency Prediction

In this test, the live performance of the DT is measured by taking t_{DT} for each model. This is done in order to assess the feasibility of the DT in real life applications in real-time. As stated in chapter 3, in order to consider the predictions by the DT useful, t_{DT} should be less than t_{PT}.

This test is initiated by looking at the worst case location scenario, namely Paris. Each time the PT downloads a file from the server, the DT has to make a prediction, which is considered as an iteration. At each iteration, the input data corresponding to the PT transmission are passed to the DT. Models 3, 4, 5A, and 5B are loaded in advance, meaning that the DT can make the predictions immediately. To show the real-time t_{PT} alongside the time to predict t_{DT}, an interactive Graphical User Interface (GUI) was developed. A snapshot of this GUI is provided in figure 5.4 for the test on Paris.

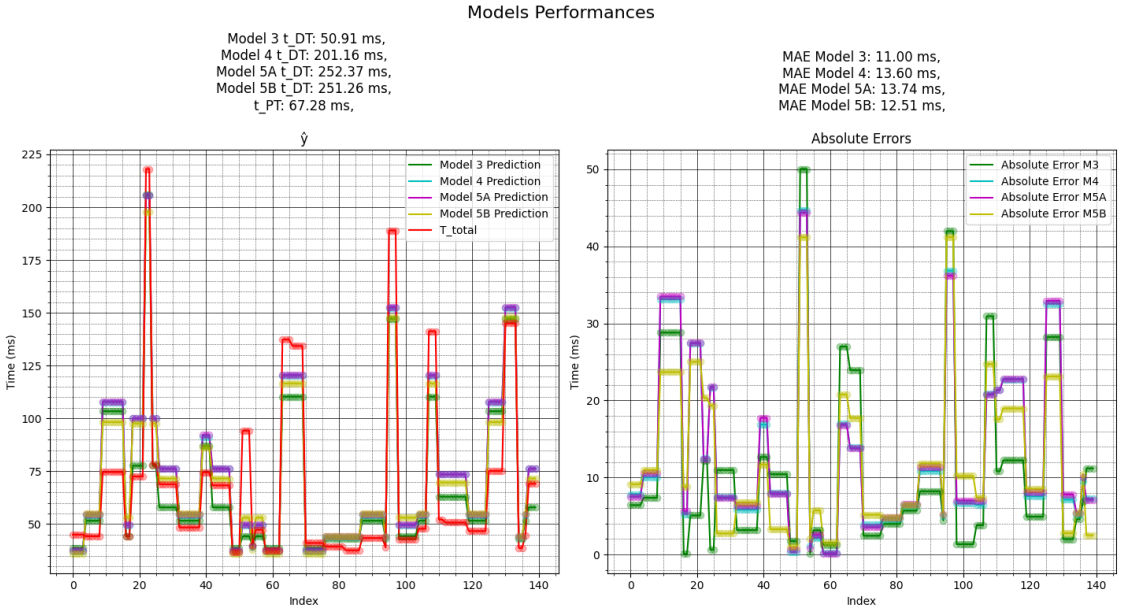

Figure 5.4. A snapshot of the DT making live predictions along with the average time to predict, t_{DT} and the MAE for each model with the server located in Paris.

As can be seen on the GUI, the predictions follow the patterns of the t_{total} observations, however, they are shown to either underestimate or overestimate at each point.

Above the left segment of the figure is a comparison of the t_{DT} observations, and as can be seen, all but Model 3 take at least three times as long to predict the time for Paris at the time of the capture. This already showed that the idea of a live DT using these models and methods may not be feasible. To get a better idea of the feasibility, the mean t_{DT} were taken for all DT models, and the mean t_{PT} for AAU and Paris were found. This can be seen on table 5.9, and gives an idea of the best and worst case scenarios.

Table 5.9. Models and their mean t_{DT} taken over 100 iterations alongside the mean true latency t_{PT} with the server located in Paris.

Model	$t_{DT}(ms)$	$t_{PT}(ms)$ **AAU**	$t_{PT}(ms)$ **Paris**
Model 3	46.65	84.11	134.46
Model 4	187.69		
Model 5A	234.60		
Model 5B	234.80		

These data show on average that only Model 3 would be useful for online prediction, as Models 4, 5A, and 5B are all slower to predict than the mean of a PT execution. However, this table only accounts for the mean time taken, and therefore it is also relevant to consider the best and worst t_{PT} that can be expected. This is included on table 5.10.

Table 5.10. Lowest and highest t_{PT} observed for each location.

	AAU	**Fjerritslev**	**Frankfurt**	**Paris**
Lowest	8.12	24.37	29.43	37.95
Highest	220.42	279.00	281.87	311.68

As it was found on table 5.9 that each NN on average contributes with 45 - 50 ms, it appears as though not even Model 3 with a single NN would be able to do live predictions for any of the best case t_{PT} scenarios. This means that the DT is generally not feasible for live predictions in the current state. Additionally, this makes the models with more than one NN even less feasible, as Models 4, 5A, and 5B had a worse t_{DT} than the mean t_{PT} for Paris. This however sparks the question of whether the use of an NN to predict was the problem, or if it was a result of the implementation. As such, brief tests were conducted with different prediction methods.

5.4.1 Improving the NN implementation

Investigation of different TensorFlow Github issue threads showed that not only were there issues with newer versions of TensorFlow GitHub [2019b], but there were also issues with the `predict()` method [GitHub, 2020]. One of the possible solutions was to try and import an early version of TensorFlow in compatibility mode, however, this did not prove to make the NN execute any faster.

Other solutions included using `model(x)` calls instead of `predict()`, where the input data is directly given to the model as `x`. Another method brought up on some Github issues is `predict_on_batch()`, which supposedly does the same as `predict()`, but only on one batch at a time [GitHub, 2019a]. It appears to still be capable of predicting a larger batch, and returned the same output as the previous method. A final option considered is the conversion of the models to *tflite*, a lightweight TensorFlow implementation meant for mobile and edge devices.

When the model has been converted, it cannot use the usual TensorFlow methods, and instead has to do an inference by setting up an interpreter [TensorFlow, 2023a]. This process infers the relation between the input and output data, although it is still classified as a prediction in the documentation, and returned the same results as the regular TensorFlow model predictions, given the same inputs.

The execution times of Model 3 with `predict()`, `predict_on_batch()`, `tflite`, and the `model()` call are seen on figure 5.5.

Figure 5.5. Time to predict for Model 3 for number of iterations inside a while loop using predict(), predict_on_batch(), model calls, and the tflite conversion.

As can be seen, the `model()` call provided a better t_{DT} than `predict()` for the first number of iterations, but it quickly rose in prediction time. This appears to be caused by a memory leak when the call is used within a loop, as it consistently performed better outside of a loop when doing a quick comparison. However, as the DT relies on the loop, the call method simply is inadequate.

With `predict_on_batch()` on the other hand, not only were the resulting t_{DT} always lower than the previous two methods, it was also consistent despite being executed in a loop. It appeared on average to perform five times faster than `predict()`. Reasons for this difference in performance appears to be due to less overhead, where `predict()` splits the input data given, checks the dimensions of the shape, and supports additional arguments [Stackoverflow, 2018].

Finally, the model conversion to `tflite` proved significantly faster than all of the other methods. Aside from the significantly smaller size of the model and the difference in method used, the way Tensor memory is allocated and reused could further explain this reduction in prediction time [TensorFlow, 2023b]. A comparison of the mean prediction time for the stable methods can be seen in table 5.11.

Table 5.11. Models and their respective mean t_{DT} for 100 iterations.

Model	t_{DT} **(ms) predict**	t_{DT} **(ms) predict on batch**	t_{DT} **(ms) tflite**
Model 3	46.65	9.14	0.24
Model 4	187.69	34.02	0.59
Model 5A	234.60	42.68	0.70
Model 5B	234.80	42.64	0.70

The results of `predict_on_batch()` show that t_{DT} for a single NN goes from roughly 45 - 50 ms to 8.5 to 9.14 ms. While this is a significant improvement, and shows that every model falls under the mean t_{PT} for AAU, the lowest t_{PT} observed in table 5.10 is still faster than a single NN implemented with this method.

The improvements had with `tflite`, however, saw that every model, even Models 5A and 5B with five NNs, can predict with a t_{DT} under one ms. This means that every single model is feasible for live prediction, even for the lowest t_{PT} observed from AAU. At the same time, this method provided the same accuracy as the previous methods. While there are a number of ways listed by TensorFlow on how to improve the performance further [TensorFlow, 2023b], the current models are deemed adequate.

5.4.2 Temporal Variance Observations

During the online execution of the DT, it was observed that the prediction accuracy of the models varied from time to time, as can be seen in figures 5.6, 5.7 and 5.8.

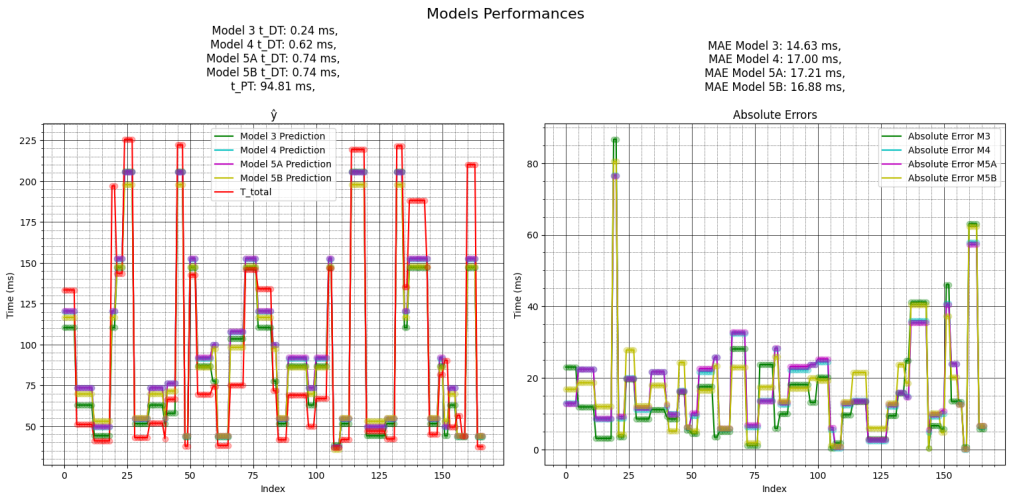

Figure 5.6. A snapshot of the DT making live predictions along with the average time to predict, t_{DT} and the MAE for each model with the server located in Paris on May 18th, 2023 at 11:00.

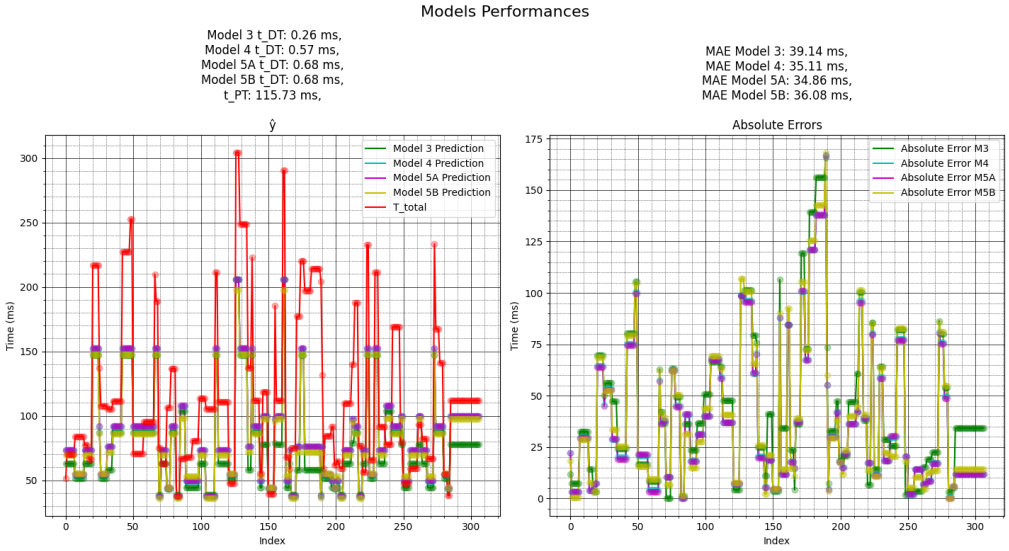

Figure 5.7. A snapshot of the DT making live predictions along with the average time to predict, t_{DT} and the MAE for each model with the server located in Paris on May 19th, 2023 at 9:00.

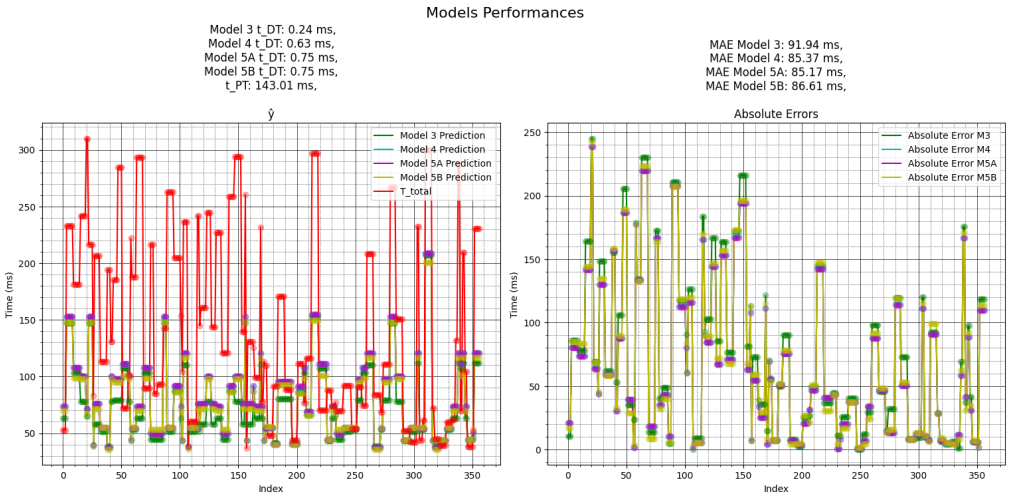

Figure 5.8. A snapshot of the DT making live predictions along with the average time to predict, t_{DT} and the MAE for each model with the server located in Paris on May 19th, 2023 at 10:30.

Later, it was discovered that this significant discrepancy was a result of the link's instability, which changes over time due to a number of factors. Network congestion could be one of these factors, which can be caused by high data traffic resulting from the increasing amount of users and their activities. In addition, suboptimal routing could also be a factor for latency variations, as it can occur due network issues or inefficient routing configurations.

While there is no direct control over such factors, it is still possible to continuously monitor the state of the link using network tools like `pathping` or `traceroute`. By doing so, more information can be gathered, providing insight into routing path and latency-contributing points or likely bottlenecks along the path. This information can then be employed by the models as inputs, in order to alleviate the effect of latency variations on the prediction accuracy.

Based on the observed temporal variations, it is evident that time should be regarded as an important factor, with potential to be employed as an input to the models. This would provide deeper insight into temporal patterns that may influence the latency such as the number of active users, varying user behaviour, network maintenance times, peak hour usage, and more. This may allow the models to take the fluctuation in t_{total} into account, however, it would require a large amount of data to represent the patterns over a large span of time. Another possibility is training the models specifically based on the times of the day, rather than have time as an input.

Conclusion and Reflection 6

6.1 Conclusion

This project examined the possibility of using a Digital Twin in order to predict end-to-end latency as detailed by the 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?

During the course of this project, a use case involving a client downloading a file from a server was proposed. The end-to-end latency of this operation was split into the latencies for the individual parts of the process, which allows for analysis of the contribution of each part. A number of factors like the distance between server and client, transmitted file size, and protocol used were investigated in order assess their impact on these latencies. These factors were employed as inputs to a number of models, involving one or multiple Neural Network components, in order to predict the the end-to-end latency. To get the most out of these models, the data used was preprocessed.

Two types of tests were conducted: an offline test to evaluate the prediction accuracy, and an online test to see if it was viable to run the Digital Twin predictor in real-time. This required that the predictions should be faster than the Physical Twin could finish the task.

It can be concluded that by using a number of inputs which are available prior to the file downloading process, the end-to-end latency can be predicted. To obtain closer predictions, several techniques can be applied such as reducing data sparsity and addressing multicollinearity in the data set.

As a result of this project, it was found that it is possible to have a Digital Twin latency predictor based on Neural Networks run alongside a Physical Twin in real-time. This is provided that the Neural Network implementation is lightweight and fast enough for the task, such as with the TensorFlow Lite library.

It was also found that the accuracy of the predictions varied during the time of the day, which indicated that the actual end-to-end latency changed throughout the day. This could be based on a number of factors such as usage patterns, and means that if a model is trained for one hour of the day, it may not be suitable for the next. A possible solution for this would be to look into including time, which day, or even it is a holiday as inputs for the models, or to have specialised models for each scenario.

Finally, it was found that the composition of the Digital Twin and the amount of Neural Network components used had an impact on the accuracy of the prediction. At the cost of having a higher time to predict, having more Neural Networks did in some scenarios prove to make more accurate predictions. However, in other scenarios, the Digital Twin with just a single Neural Network had the best performance.

In short, the objective of this project was accomplished, and opens up possibilities for future work and implementation.

6.2 Reflections

One of the major findings in this project was the significant improvements achieved by converting the TensorFlow models to tflite. As this was found late in the project, there was not adequate time to do a more thorough investigation of how to get the most out of tflite. For future iterations, if necessary, some of the many optimisation methods listed by TensorFlow could be tested.

Another discovery made during testing was how much variation in the results was found when captured at different times during the day. This means that if training data was obtained in the morning, it may not be representative of test data captured during the afternoon. While it may not be possible to easily obtain information about the state of the link, the latency trends over hours, days, and even months could be relevant to investigate further. If the latency observed during the day is significantly different than during the night, it would make sense to have individual models trained for these circumstances instead of just a generalised model.

Future application of these latency-detecting Digital Twins could potentially be with video games, which have time-critical tasks and a number of existing methods to accommodate for latency [Liu et al., 2022]. For example, if a client is known to have a high latency due to the inputs given, something like Attribute Scaling could be applied in advance. A second potential application is evaluation of the connections chosen based on distances to, e.g., different carriers or even Low Earth Orbit (LEO) satellites, to select the option with the lower of the latencies.

Another case that could be interesting to apply these latency-detecting Digital Twins to is autonomous delivery drones. During their operation, these drones generate large amounts of data which may encompass location updates, sensor reading, videos, images etc. These data require real-time processing in order to ensure secure and seamless operations. To do so, edge computing is applied in which a number of edge points are distributed across the areas where drones frequently operate. Here, the Digital Twin can help adapt the offloading of computation tasks across multiple edge points, in order to ensure a more efficient and balanced real-time data processing. Furthermore, the Digital Twin can play a major role in reducing the network congestion, as well as latency, by optimising the trajectory of the drone paths to areas with better coverage.

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.

Bappy et al., **2010**. DM Bappy, Ajoy Kumar Dey, Susmita Saha, Avijit Saha and Shibani Ghosh. *OFDM system analysis for reduction of inter symbol interference using the AWGN channel platform*. International Journal of Advanced Computer Science and Applications, 1(5), 2010.

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.

Basanisi, **2019**. Luca Basanisi. *How to deal with categorical features*, 2019. URL https://www.kaggle.com/code/lucabasa/how-to-deal-with-categorical-features. Last visited on 04/05/2023.

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.

Cloud, **2023**. Google Cloud. *Overview of hyperparameter tuning*, 2023. URL https://cloud.google.com/ai-platform/training/docs/hyperparameter-tuning-overview. Last visited on 04/05/2023.

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.

Stephen R Ellis, Bernard D Adelstein, S Baumeler, GJ Jense and Richard H Jacoby, *1999*. Stephen R Ellis, Bernard D Adelstein, S Baumeler, GJ Jense and Richard H

Jacoby. Sensor spatial distortion, visual latency, and update rate effects on 3D tracking in virtual environments. In *Proceedings IEEE Virtual Reality (Cat. No. 99CB36316)*, pages 218–221. IEEE, 1999.

GitHub, **2019a**. Keras GitHub. *Repeatedly calling model.predict(...) results in memory leak #13118*, 2019a. URL `https://github.com/keras-team/keras/issues/13118`. Last visited on 04/05/2023.

GitHub, **2020**. Tensorflow GitHub. *model.predict is much slower on TF 2.1+ #40261*, 2020. URL `https://github.com/tensorflow/tensorflow/issues/40261`. Last visited on 04/05/2023.

GitHub, **2019b**. Tensorflow GitHub. *Why is TensorFlow 2 much slower than TensorFlow 1? #33487*, 2019b. URL `https://github.com/tensorflow/tensorflow/issues/33487`. Last visited on 04/05/2023.

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

He et al., **2000**. Ding He, Fuhu Liu, Dave Pape, Greg Dawe and Dan Sandin. *Video-Based Measurement of System Latency. International Immersive Projection Technology Workshop 111 (July 2000), 6*, 2000.

Holbrook, **2023**. Ryan Holbrook. *Dropout and Batch Normalization*, 2023. URL `https://www.kaggle.com/code/ryanholbrook/dropout-and-batch-normalization`. Last visited on 04/05/2023.

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, **2023**. IBM. *What Is machine learning?*, 2023. 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.

Mine, **1993**. Mark R Mine. *Characterization of end-to-end delays in head-mounted display systems.* The University of North Carolina at Chapel Hill, TR93-001, 1993.

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.

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`.

Stackoverflow, **2018**. Stackoverflow. *What is the difference between the predict and predict_on_batch methods of a Keras model?*, 2018. URL `https://stackoverflow.com/questions/44972565/what-is-the-difference-between-the-predict-and-predict-on-batch-methods-of-a-ker`. Last visited on 04/05/2023.

Anthony Steed, 2008. Anthony Steed. A simple method for estimating the latency of interactive, real-time graphics simulations. In *Proceedings of the 2008 ACM symposium on Virtual reality software and technology*, 2008.

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

TensorFlow, **2023a**. TensorFlow. *TensorFlow Lite inference*, 2023a. URL `https://www.tensorflow.org/lite/guide/inference`. Last visited on 022/05/2023.

TensorFlow, **2023b**. TensorFlow. *Optimizing TensorFlow Lite Runtime Memory*, 2023b. URL `https://blog.tensorflow.org/2020/10/optimizing-tensorflow-lite-runtime.html`. Last visited on 022/05/2023.

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.

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.