NTNU – Trondheim
Norwegian University of
Science and Technology

Classifying Router Types and Network Path Changes in Local Networks Using Ping

Martin Breivik Smebye

Submission date: August 2019
Responsible professor: Steinar Bjørnstad, IIK
Supervisor: Steinar Bjørnstad, IIK

Norwegian University of Science and Technology
Department of Telematics

Title:	Classifying Router Types and Network Path Changes in Local Networks Using Ping
Student:	Martin Breivik Smebye

Problem description:

Delay measurements in a network are mostly used for measuring the distance between two network components. They allow the user to see how much time it takes for an IP-packet to reach its destination, and are typically measured in milliseconds. The accuracy of the measurements depend on the tool used. Generally we can say that higher accuracy tools are more expensive and less accessible for users.

In this thesis we focus on a widely used and widely accessible measurement tool, the ping program. The ping program is pre-installed by most modern operating systems, and is accessible in their terminal/command line. We focus on two applications of the ping program in this thesis. Firstly, we explore pings capabilities for recognising various router types. Can we identify a router by its ping response? Secondly we examine pings capacity for detecting changes in the network, such as adding additional component. How accurately can we detect small changes in a network path?

Responsible professor:	Steinar Bjørnstad, IIK
Supervisor:	Steinar Bjørnstad, IIK

Abstract

When ping was written in 1983, it was seen as a tool for estimating the distance between the client running the program, and another node in a network the client was connected to. Since pings inception, the Internet Engineering Task Force has specified that all hosts in the internet has to implement functionality for receiving and responding to ping package, making it widely available. By measuring the time it takes from a client sends a packet, until it receives the corresponding response, you are given an indication of the distance between the client and the responder. This, along with measuring the perceived traffic along the network path by looking at packet loss, has been the main application of the ping program. In this thesis we explore additional applications of the ping program.

By gathering measurements and applying our analysis tools in a series of experiments, we examine pings capabilities to detect small changes in local networks, and its capacity to classify router types by looking at series of their ping responses. To analyse our data we have utilised two different methods, Signal Processing and deep learning. Signal Processing is a well established field in engineering that focuses on analysing, modifying and synthesising signals. Deep learning is a type of machine learning, which is an approach to Artificial Intelligence. In addition to evaluating the capabilities of ping for our set tasks, this thesis also discusses the practicality of these two methods.

Sammendrag

Da ping ble skrevet i 1983, ble det hovedsaklig sett på som et verktøy for å estimere avstanden mellom klienten som kjørte programmet, og en annen node i nettverket klienten var koblet til. Siden den gang har Internet Engineering Task Force spesifisert at alle nettverksverter må implementere funksjonalitet for å motta og svare på ping pakker, som gjør ping til et meget tilgjengelig verktøy. Ved å måle tiden det tar fra man sender en pakke til man mottar det korresponderende svaret får man en indikasjon på hvor langt det er fra klienten og svareren. Sammen med å måle pakketap for å få et innblikk nettverkstrafikken har dette vært hovedbruken til ping. I denne oppgaven ser vi på nye bruksområder for ping.

Ved å samle inn måledata og anvende analyseredskapene våre i en serie eksperimenter vil vi utforske pings egnethet til å oppdage små endringer i lokale nettverk, samt ets egnethet til klassifisering av rutere ved å se på serier av ping-svar. For å analysere dataen vår har vi brukt signalprosessering og *deep learning*. Signalprosessering er et veletablert felt innen ingeniørvitenskap som fokuserer på å analysere, modifisere og syntetisere signaler. Deep learning er en type maskinlæring, som igjen er en fremgangsmåte under kunstig intelligens. I tillegg til å vurdere pings egnethet for våre satte utfordringer vil denne oppgaven også vurdere egnetheten til de to analysemetodene.

Preface

This thesis has been submitted to fulfil the graduation requirements of the MSc in Communication Technology at the Norwegian University of Technology and Science (NTNU). The main research and writing were carried out between March and August 2019.

The main objective of this study is to explore the capabilities of the ping program in terms of recognising routers and path changes close to the client in a network.

Special thanks go to Steinar Bjørnstad for his contribution to the project.

Martin Breivik Smebye

Trondheim, 12th of August 2019

Contents

List of Figures

List of Tables

List of Acronyms

ANN Artificial Neural Network.

CNN Convolutional Neural Network.

E2E End-to-End.

ICMP Artificial Neural Network.

IP Internet Protocol.

LSTM Long Short-Term Memory.

MLP Multi-Layered Perceptron.

NTNU Norwegian University of Science and Technology.

RFC Request for Comments.

RNN Recurrent Neural Network.

RTT Round Trip Time.

TCP Transmission Control Protocol.

UDP User Datagram Protocol.

Chapter 1
Introduction

This chapter describes the background and motivation for this project. We, in this chapter, define the primary research questions, illustrate the methodology used and conclude with an overview of the subsequent chapters.

1.1 Motivation

Leased lines are, with the ever increasing demands for faster and more reliable network connections, becoming a more popular solution for people and organisations who want to connect networks across circuits that are free of contenting network traffic of outside users. Leased lines are dedicated connections between two network points, and are therefore not subject to the frequent rerouting of regular broadband connections. Detecting changes due to for example rerouting can, however, be of interest. Anomalies could be an indicator of malfunctioning routers or attacks such as prefix highjacking [BFZ07].

Rerouting on the IP-layer can be discovered fairly easily using a well established tool such as *traceroute*. Rerouting in the data link and physical layer is, however, not detectable using this tool. Rerouting on the lower layers in most cases does change the minimum delay time. This is interesting, because the minimum delay for a network path will always remain the same where the network path does not change. This means that determining the minimum delay time between two nodes in a network and being able to detect when it changes can reveal rerouting. Both hardware based tools and software based tools can be used to measure delay. Software tools are less accurate than hardware tools, as they are susceptible to delays linked to the processor on the system they run on. They are, however, more accessible to regular users. The most widespread software tool available today is the ping program [Muu16], and it is this tool we utilise in this project.

There is no way of determining whether a packet achieved minimum delay time by inspecting it in isolation. Delay measurements are also susceptible to various sources

of noise. Multiple delay measurements therefore have to be made to determine a likely minimum delay time. We want, in this work, to use regular signal processing and machine learning to analyse the measurement data to determine which is the best fit for the task. Machine learning methods can require more data to be efficient. These methods can, however, also find correlations in the data that are not obvious upon manual inspection. To address this we established **RQ1** and **RQ2** as given in section 1.2.

The most obvious use of delay measurements for network packets is to measure the delay at two end points. The processing required at the end points can also tell us something about them. We observed, during the initial phases of our study, that the ping responses we received from different routers varied more than we expected from the change in travel distance alone. We were curious whether this was consistent behaviour, and if so, could we identify routers by their ping response? To address this, we established a 3rd research question, **RQ3**.

1.2 Research Questions

- **RQ1:** Can we train a machine learning algorithm to recognise changes in a local network path with limited nodes, that are undetectable using regular signal processing?
- **RQ2:** Given that we can recognise rerouting, what are the minimum changes we can recognise?
- **RQ3:** Can we identify a router by looking at the latency pattern when pinging it?

To answer the first two questions, we first looked at delay measurements on shorter paths, such as within the NTNU campus, before attempting measurements on longer paths with more noise. To answer the last question, different routers were pinged, and the resulting measurement distributions were analysed.

1.3 Outline

This thesis is divided into 5 chapters, including this introduction.

Chapter 2: Theory. This chapter reviews the different components that make up network delay, and the ways delay can be measured. We describe the measurement tool we intend to use (ping), before covering the statistical parameters that will be used in signal processing. The last section describes some machine learning essentials, before specifying the techniques we intend to use.

Chapter 3: Experiments. We present in this chapter the experiments that we conducted in our project. We describe the general setup for all the experiments before going into more detail on each experiment.

Chapter 4: Analysis/Key Findings. A presentation and analysis of the key findings of the experiments described in Chapter 3.

Chapter 5: Summary/Conclusion. A summary of the thesis, and suggestions for future work.

Chapter 2
Theory

2.1 Network delay

Our project focuses on measuring delay. It is therefore important to know what contributes to network packet delay, and how we can measure it. We are not aware of any studies that have utilised delay measurements to recognise changes in and classify components of normal behaviour in networks. Delay measurements have, however, been used to uncover specific router behaviour on network paths [LW].

2.1.1 Delay components

A data packet traversing through a packet switched network is affected by various types of delay on each node along its path. We divide this delay into the four components described below. Figure 2.1 shows an example of the node delay of Router A connected to two clients and another router.

- *Processing delay*, the time used by the router to transmit a packet from the incoming packet interface to the outgoing packet interface.
- *Queuing delay*, the time used waiting for other packets to be dispatched to a transmission link. This will be zero if there are no other packets in the router when a packet arrives.
- *Serialisation delay*, the time it takes to push all the bits of the packet onto the transmission link. Sometimes also called transmission delay.
- *Propagation delay*, the time required to transmit a bit through a transmission link. This is not strictly a part of the node, but rather the link between the nodes. The speed of propagation is 2/3 the speed of light in optical fibre.

It is important, in the context of this study, to keep in mind the different types of delay, as some will introduce noise to our delay measurements. Propagation delay

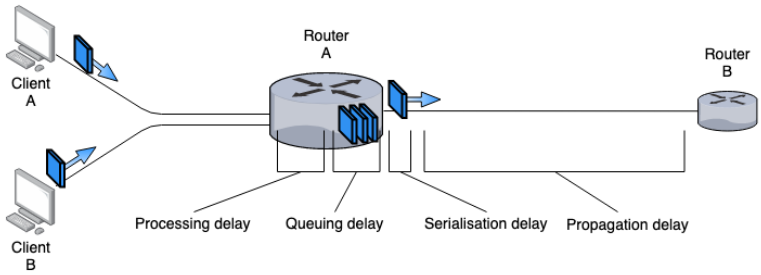


Figure 2.1: The nodal delay of a router.

and serialisation delay will remain the same for packets of the same size, processing and queuing delay can vary and queuing delay can vary from packet to packet in the same node, as it depends on traffic at the node.

Processing delay depends on the operation the node is performing on the packet. Intermediate nodes mainly perform 'fast switching' [LW], meaning that only dedicated hardware is used to forward the packets. This introduces relatively small amounts of delay, and minimal noise. The routing processor has to, however, be involved in more advanced operations that cannot be handled in hardware. This adds considerable delay and noise compared to simple forwarding.

Involving the routing processor introduces noise because processing takes place in the router's software. It may therefore have to wait for other processes within the router. Noise and delay varies between routers, as these are dependent on the routers' processing power. A typical routing architecture is given in figure 2.2

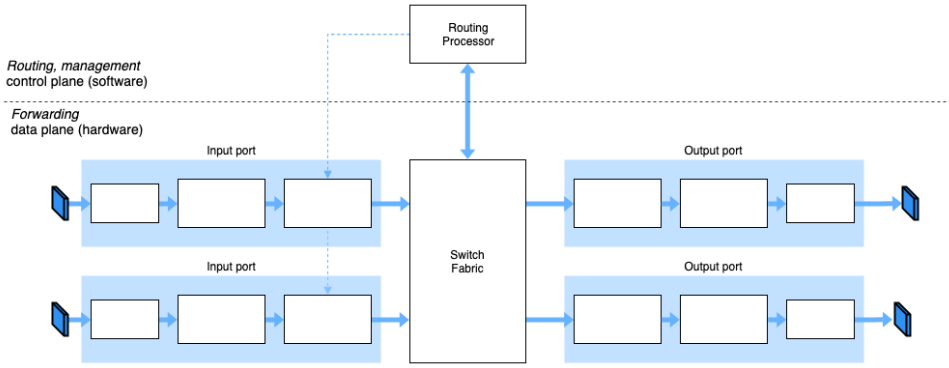


Figure 2.2: Router architecture.

Going forward we can conclude that, even though all four components are part of the total delay, that queuing and processing delay are the components that introduce

noise and affect the accuracy of our delay measurements. By denoting the processing, queuing, serialisation and propagation delay as d_{proc}, d_{queue}, d_{trans} and d_{prop}, we can denote total node delay as follows:

$$d_{node} = d_{proc} + d_{queue} + d_{serial} \tag{2.1}$$

The delay in a transmission link is simply given by the the remaining component:

$$d_{link} = d_{prop} \tag{2.2}$$

The propagation speed depends on the medium of the transmission link, but will generally be around $2 * 10^8 m/s$ [KR13]. This means that the d_{link} will be close to 5 microsecond per km.

2.1.2 End-to-End delay and Round Trip Time

The previous section discussed the delay components for a single node and transmission link. We will now, in this section, consider the total delay from host to receiver. Accumulating the total node delay and link delay for every node and transmission link on the path between the host and the receiver gives the *End-to-End delay* (E2E). Figure 2.3 illustrates an example with 3 routers between the packet host and receiver. Here the E2E delay is given by $3d_{node} + 4d_{link} + 2d_{end}$, where d_{end} denotes the processing delay at the end points. Generally we can denote the E2E delay for a path with N nodes between the end points as follows:

$$d_{E2E} = Nd_{node} + (N + 1)d_{link} + 2d_{end} \tag{2.3}$$

The E2E delay between two nodes is the best way to represent the delay between them. It can, however, be difficult to measure. The clocks at both end points must, however, be synchronised for an E2E measurement to give valid delay times. This is certainly possible in a controlled environment, but not when measuring the delay between your client and a server you have no control over. We can overcome this problem by instead measuring the *Round Trip Time* (RTT).

RTT measures the time between a message being sent from the host until the host receives the response message. This is the method used by the ping tool, which we will look closer at in section 2.2. RTT essentially doubles the E2E delay, while circumventing the synchronisation problem. The first main disadvantage of this method is that the packages have to travel twice as far, exposing the delay measurements to more noise. A second disadvantage is that it renders it impossible to detect any asymmetry between the delay from the host to the receiver and the delay from the receiver to the host.

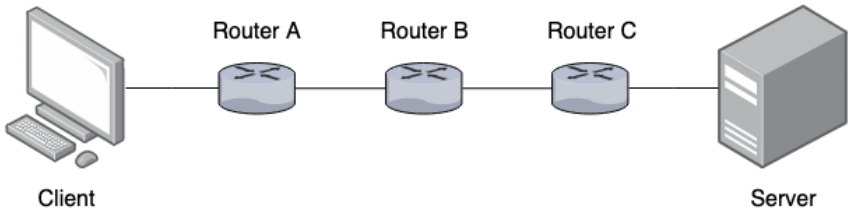


Figure 2.3: Illustration of E2E delay.

2.1.3 Lucky packets

There will always be a potential minimum delay time, the shortest possible time a packet can use between two nodes, even though there will be noise when measuring the delay between two nodes in a network. Minimum delay time occurs where the data packet does not meet any traffic on it's path and there is zero queuing delay. We call these packets *lucky packets*.

Lucky packets are interesting when measuring delay, because the minimum delay time will always be the same where the network path remains unchanged. This means that if the minimum delay time between two nodes in a network changes, then this indicates that the path between the nodes has changed. So how can we determine whether a measured delay is the minimum delay time of a lucky packet?

The minimum delay time would be fairly easy to derive if there was no additional noise to the queuing delay. Every packet without queuing delay would have the same delay, which means that we most likely would have a minimum value that would also be the mode. As we do have noise in addition to the queuing delay, we need to consider the distribution of the lowest measured delays, and use this as an approximation of the minimum delay time.

2.2 The ping program

The code[Muu83] for the ping program was written by Mike Muuss in 1983[Muu16], and is today a widely used and widely available tool for measuring RTT. The tool was first included in Berkeley UNIX[WCL+88], but has since been ported to other platforms such as Microsoft Windows and MS-DOS. The tool operates by sending timed ICMP packets to a target host, and then waiting for the mandatory response from that host. It is commonly used to test the availability of a host on an IP network, and to measure the distance to other nodes in a network. Before going deeper into how the program works, we will first look at the protocol it utilizes, ICMP.

2.2.1 Internet Control Message Protocol (ICMP)

The Internet Control Message Protocol (ICMP) for IPv4[Pos81b] was defined in RFC 792 [Pos81a] in 1981, and is the protocol utilised by ping. It acts as a support protocol and is used by network hosts and routers to communicate network layer information to each other. It is most frequently used in error reporting, for example when a router is unable to find a path to a packet's destination IP.

RFC 791[Pos81b] shows ICMP and IP on the same level in its illustration of protocol hierarchy as shown in figure 2.4, ICMP is in fact above IP, as ICMP packets are carried within the IP payload, as they also are within TCP and UDP packets. The demultiplexing of the IP payload for TCP and UDP packets therefore also takes place for ICMP.

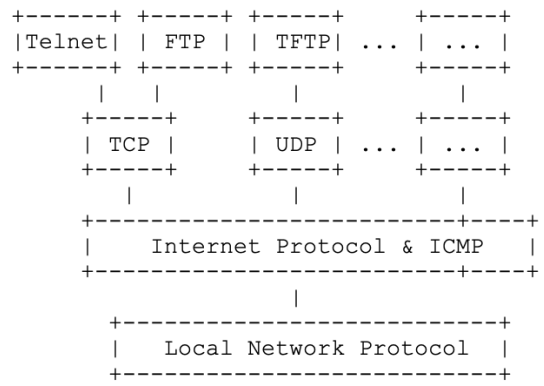


Figure 2.4: Protocol hierarchy.

An ICMP packet consists of an 8 byte header and data of various sizes. The first four bytes have a fixed format, while the last four bytes are often unused. This,

however, varies with the type and the code set in the first two bytes. Figure 2.5 shows the structure of the `ICMP ECHO_REQUEST` and `ICMP ECHO_REPLY` messages, which are the message types ping utilises. A description of the 6 fields used in echo messages is given below.

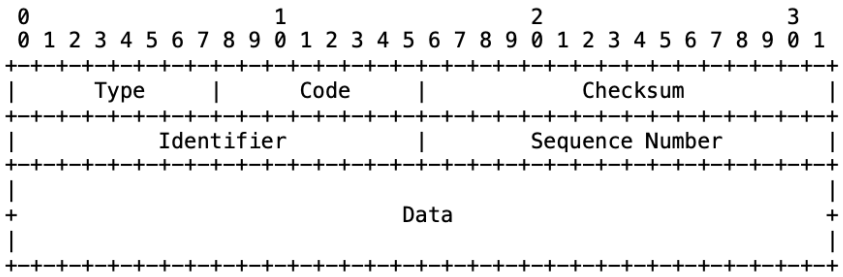


Figure 2.5: ICMP Echo message.

- *Type* is an 8-bit field that identifies the control messages of ICMP. It is set to 8 for `ICMP ECHO_REQUEST` and 0 for `ICMP ECHO_REPLY`.

- *Code* is an 8-bit field that identifies the subtype of the ICMP control messages. It is set to 0 for both echo messages.

- *Checksum* is a 16-bit error checking field, calculated from the ICMP header and data. It uses *The Internet Checksum* specified in RFC 1071[BBP88].

- *Identifier* is a field to aid the matching of requests and replies. Ping uses the UNIX process ID.

- *Sequence Number* is another field to aid the matching of requests and replies. For ping, it is an ascending integer starting at 0. As the field is only 16 bits, ping goes back to 0 after reaching the $2^{16} - 1 = 65535th$ packet.

- *Data* in an ICMP packet can be of various lengths. There does not have to be any data at all in echo messages. However, you need to include at least 16 bytes of data if you want to get the RTT from a ping. This is because ping uses the first few bytes to hold a UNIX timeval struct, which is necessary for getting the RTT. RFC 1122[Bra89] specifies that the data received in an echo request message must be included entirely in a reply message. This is unless the echo reply requires unimplemented intentional fragmentation, in which case the data should be shortened to maximum transmission size. The default data size for ping is 56 bytes, making the whole ICMP message 64 bytes.

2.2.2 Additional functionality

ICMP does most of the work that is required to link outgoing echo requests to incoming echo replies. The ping program, however, adds one essential function to every packet, namely timing the RTT. As mentioned in section 2.2.1, this is added by including a UNIX timeval struct in the data field of the ICMP packets. The value of the timeval struct is then subtracted from the current time when the ICMP reply arrives, giving the RTT. Ping also provides basic statistics for all packages when a ping call contains multiple packets. These include the minimum delay, maximum delay, mean, and standard deviation.

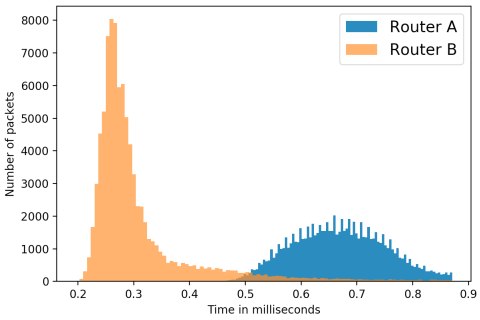
The ping options give the user the ability to control other aspects of the packages, particularly packet size and interval timing. The options we will be using are listed below.

- *[-c <count>]*, the number of packets sent in a single ping call.
- *[-i <interval>]*, the time between each packet, stated in seconds.
- *[-s <size>]*, the size of the ICMP data field. Must be at least 16 bytes to include the timestamp used for delay measurement. Default size is 56 bytes.

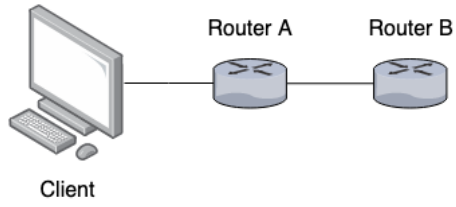
2.2.3 Delay and noise

Ping introduces a delay at the two end points of a measurement, which is in addition to the delays mentioned in section 2.1.1. Ping is a software tool. It is therefore dependent on the process scheduler of the system it is operating on. The process scheduler's prioritisation is beyond the user's control, and can add various amounts of delay. Varying delay adds an extra source of noise in the measurements. However, as long as ping is run on the same system, and enough measurements are made in each measurement series, then the noise profile should remain the same.

he receiver of the ICMP echo request will also have to carry out processing, regardless of whether it is a router or an end node. In routers, processing will be in the control plane as shown in figure 2.2, and the noise generated will depend on the processing power and process scheduler of the router. This can vary substantially between different network nodes, potentially giving network nodes that are further away a lower RTT than closer nodes. An example of this can be seen in figure 2.6 below.



(a) Delay measurement spread



(b) Setup

Figure 2.6: Example of router measurements in a low traffic network.

In this example, two routers were pinged 100 000 times each. The spread of the resulting delay measurements can be seen in the histogram in figure 2.6a. Outliers above the distribution have been cut off in both measurement series, as they are most likely affected by other network traffic. Packets affected by other network traffic are not of interest to us, as they do not convey any information about the router or the network path. The interesting aspect to note is that the delay is lower for Router B, even though it is further away from our client.

2.3 Signal Processing

'The technical field of Signal Processing encompasses all forms of sampled-data manipulation where the data (or signal) has a physical origin, or destination.' This is the definition of Signal Processing given by Lars E. Thon in his article *50 years of Signal Processing at ISSCC* [Tho03]. We, in our project, use Signal Processing to analyse the distribution of our measurement series. We want to group every measurement series made on the same endpoint to see if we can find distinct differences in their distributions, when comparing two groups of measurement series on different endpoints. The statistical parameters of the distributions in each group are calculated to detect any differences. This is then used to find the minimum and maximum value for each parameter for the group as a whole. This will give us a range for every parameter in each group. If any of these ranges are disjunct, then we should still be able to classify every measurement series of the two groups correctly.

2.3.1 Statistical parameters

The statistical parameters we will use for describing our distributions are the **mean**, the **standard deviation**, the **skewness** and the **kurtosis**. The mean and standard

deviation should be familiar to all who have delved into statistics. We therefore will give a brief introduction to the two other parameters.

Skewness is the measure of asymmetry from the mean in a normal distribution. It can be negative or positive depending on the way the distribution skews, and will be zero for a symmetric normal distribution. An example of a negatively and positively skewed distribution are given in figure 2.7.

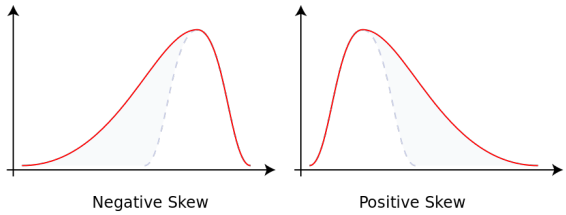


Figure 2.7: Negative and positive skewness.

The formula for calculating the skewness is given by equation 2.4.

$$\frac{\sum_{i=1}^{N}(X_i - \bar{X})^3}{(N-1)\sigma^3} \tag{2.4}$$

Where X_i is the i^{th} element of the distribution, $\bar{X}$ is the mean, σ is the standard deviation, and N is the number of values in the distribution.

Kurtosis is the measure of 'tailedness'. It tells us how heavy the tails of our distribution are. Kurtosis is often given as **excess kurtosis**, which is the difference in kurtosis from the normal distribution. A normal distribution has a kurtosis of 3. You therefore simply subtract 3 from your calculated kurtosis to get excess kurtosis. Higher values of kurtosis means that the tail of the distribution is heavier. An illustration of negative, normal, and positive kurtosis can be seen in figure 2.8.

The formula for calculating the kurtosis is given by equation 2.5.

$$\frac{\sum_{i=1}^{N}(X_i - \bar{X})^4}{N\sigma^4} \tag{2.5}$$

Where X_i is the i^{th} element of the distribution, $\bar{X}$ is the mean, σ is the standard deviation, and N is the number of values in the distribution.

We will also include as parameters for our Signal Processing the **minimum delay** and **packet loss** of every measurement series, and the **range** of our distribution.

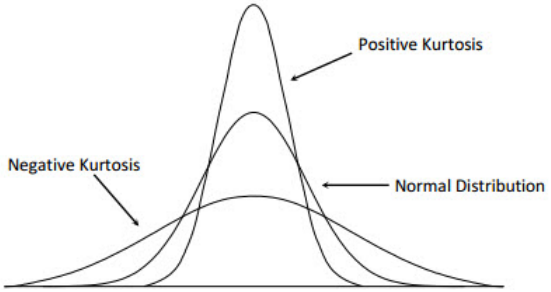


Figure 2.8: Negative and positive kurtosis.

2.4 Deep learning

Deep learning will be used to analyse our data. Deep learning is a subfield of machine learning which has seen rapid progress in recent years. We want to use this method with Signal Processing to see if it can uncover patterns in our data that Signal Processing alone is unable to detect.

2.4.1 Machine learning basics

We use machine learning to analyse the measurement series obtained from the ping measurements. Machine learning is a branch of artificial intelligence and is based around the concept that systems can learn from data by building mathematical models. The mathematical models can then be applied to solve future tasks that the system has not been specifically designed to solve. The type of machine learning we intend to use is called *deep learning*, which will be covered in section 2.4.2 and 2.4.3. This section provides some machine learning basics that are relevant to our project.

In his 1997 book *Machine Learning*, [Mit97] Tom Mitchell provides the following definition of a computers ability to learn:

"A computer program is said to learn from experience ***E*** *with respect to some class of tasks* ***T*** *and performance measure* ***P****, if its performance at tasks in* ***T****, as measured by* ***P****, improves with experience* ***E****."*

The tasks in this project are *classification* tasks, while performance is measured by *accuracy*. The measurement data gathered from pinging is the experience that is used to train the models.

Classification tasks require the computer program to determine which of k categories an input belongs to. A classification model will usually take an input vector x and assign it to a category. There are, however, also classification algorithms that

output probability distributions for the different categories. Handwriting recognition is an example of a classification task, the input being an image, and the output being a category corresponding to a letter.

Accuracy is the ability of the model to categorise input correctly. We determine accuracy by calculating the proportion of examples for which the output is correct. The goal of the model is to perform as well as possible on unseen data. We therefore evaluate the accuracy using a *test set*. The test set is different from the *training set*. The training set is the data which was used to generate the model.

Experience refers to the data the algorithm sees when building its model, this data being the training set. The amount of experience a machine learning model accumulates is typically measured in **epochs**. Epochs are the number of times a model sees the entire training data set. There are no rules for how many epochs a model should use in training. As many epochs as it takes for the performance measure to stagnate are normally used.

Machine learning algorithms are categorised into two categories. Which category algorithms are categorised into depends on the type of experience they are allowed to accumulate during the training process. The two categories are; *supervised learning* and *unsupervised learning*. Supervised learning links each training data point with a *label* or *target*, while unsupervised does not. The label or target for classification tasks typically corresponds to one of the categories the computer program sorts input into. We use labelled data in our project. The learning method we use is therefore supervised.

It is common to **one-hot encode** the labels when performing classification tasks. One-hot encoding is a technique that encodes categorical integer columns to matrices, and is best explained by an example. Lets say you have a data set that contains a group of people, the label being their favourite beverage. Machine learning algorithms deal with numerical data. We therefore need to convert the favourite drink of each member of the group to numbers the algorithm can accept. We define the following values: 1 = Water, 2 = Coffee, 3 = Milk, 4 = Soda. The problem with this encoding is that it implies an increase or decrease in value as the label changes. If our model was to internally calculate an average of milk and water, that would equal coffee (3+1/2 = 2). A relation is assumed where there is none, which can cause problems. To solve this, we expand the categorical columns to the number of categories it encompasses, and only use the values 1 and zero. Figure 2.9 illustrates how a column of 4 categories could be one-hot encoded to multiple columns.

There are plenty of options to chose from when choosing a classification algorithm for a problem [Agg14]. The *no free lunch theorem* [Wol96] for machine learning states that every classification algorithm has the same error rate when classifying previously

...	1
...	4
...	3
...	2
...	4

...	1	0	0	0
...	0	0	0	1
...	0	0	1	0
...	0	1	0	0
...	0	0	0	1

Figure 2.9: Categorical column one-hot encoded to multiple columns.

unobserved points, where averaged over all possible data generating distributions. This means that no algorithm is universally better than any other. Choosing the correct type for your task is therefore important. We have chosen, in this project, to use deep learning, as this is more accurate for larger amounts of data [GBC16], as illustrated in figure 2.10.

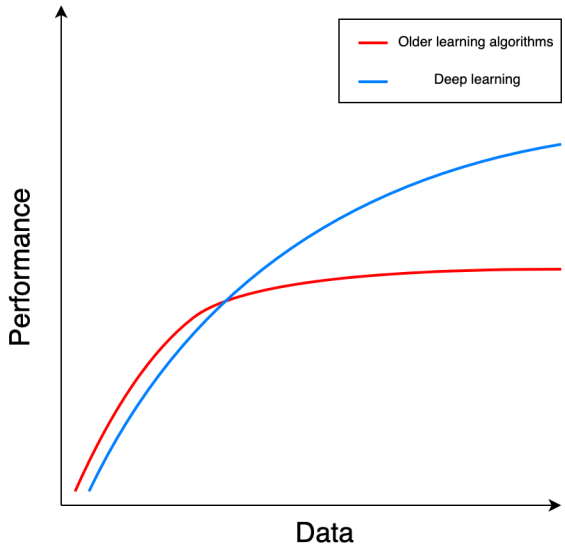


Figure 2.10: Performance of deep learning.

2.4.2 Multilayered perceptrons

Multilayered perceptrons (MLPs), sometimes also called *deep feedforward networks* or *feedforward neural networks*, are the underlying models for deep learning. The architecture is loosely inspired by the human brain, and is sometimes referred to as *artificial neutral networks* ANNs. Neutrons in the cerebral cortex in the biological brain are connected via axons. A neuron signals other neutrons over these axons when enough of it's own input signals are activated. This is simple at the small scale,

but becomes a complex system when taking into consideration the billions of neurons in a human brain. MLPs do not try to perfectly model a biological brain. The basic concepts of the two are, however, similar. As the name suggests, MLPs consist of multiple layers of perceptrons, each perceptron layer being made up of groups of activation units. These activation units resemble the neurons in the biological brain in that they are connected to multiple other activation units, while having their own individual activation threshold.

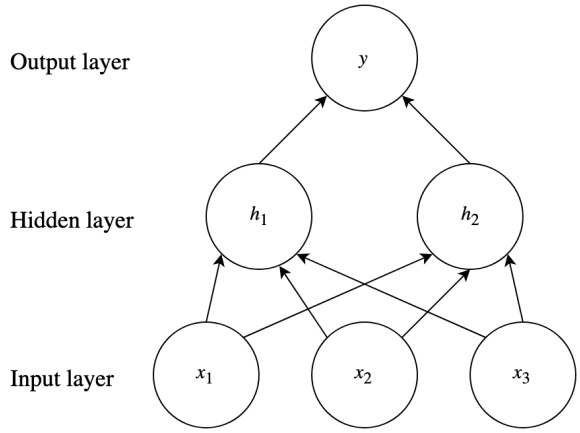


Figure 2.11: Example of a small multilayered perceptron.

Figure 2.11 illustrates a small MLP. The edges between the nodes can be weighted differently. This means that this is a more complex system than simply counting the input incoming signals from other activation units. For example, assume h_1 has an activation threshold of 0.8, and the edges between it and the activation units are as follows: $(x_1, h_1) = 0.9$, $(x_2, h_1) = 0.5$ and $(x_3, h_1) = 0.4$. This then means that a signal from x_1 would be enough to activate h_1, but for both x_2 and x_3, another activation unit would also have to signal h_1 for it to activate. As the figure also shows, the layer between the input and output layer is called the *hidden layer*. MLPs often have multiple hidden layers, and they are typically vector-valued. They are called hidden because their desired output is not shown while training the model. The number of hidden layers determines the depth of the model.

Deep learning is a subset of machine learning, deep and machine learning therefore being closely related. There is, however, a key aspect of deep learning that sets it apart from classic machine learning algorithms. The key aspect is how the features of the input that the algorithms use to build the model are defined. Classic machine learning is unable to influence how features are defined. Deep learning, however, relies on this ability. This is an advantage when it is difficult to know what features should be extracted from a dataset and can help show correlations that are not

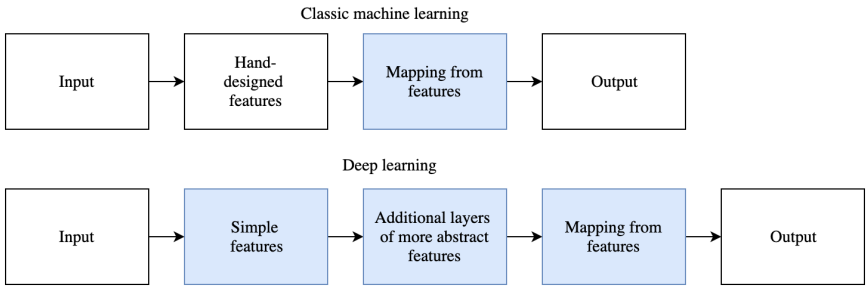


Figure 2.12: Deep learning compared to classic machine learning. Boxes that are coloured blue are able to learn from data.

obvious to a data scientist. For example, suppose a person has a rash on their arm, and we want to use machine learning to determine the cause just by inspecting it. Classic machine learning methods would require a doctor to inspect the rash, and then hand-design the features they deemed relevant. This requires more from the doctor, and could mean that important information is left out. Deep learning, however, allows an entire image to be passed in, the deep learning algorithm then determining what the important features are. This is illustrated in figure 2.12. Deep learning does, however, require more processing power and data to train the model. Technological advancement is, however, making processing power cheaper every day.

2.4.3 Recurrent neural networks

Recurrent neural networks (RNNs) are a family of neural networks that are closely related to MLPs. As we mentioned in section 2.4.2, MLPs are sometimes called feedforward neural networks, or deep feedforward networks. This is because the signals of the activation units in the network only go one way, forwards, like an asyclic graph. Adding *feedback connections* to an MLP gives a recurrent neural network.

A RNN is often considered to be a specialised neural network, whose main purpose is to process sequential data. Feedback connections enable an RNN to process much longer sequences than neural networks, which are not specifically designed to handle sequences.

LSTM

Only the behaviour of the previous time step is fed into the current step when training an RNN model. This means that the behaviour of more recent time steps affect the current time step more than the behaviour of earlier time steps. This might not be a wanted behaviour for some sequences. Hochreiter and Schmidhuber in 1997, and to combat this, released their paper on *long short-term memory* (LSTM) [HS97].

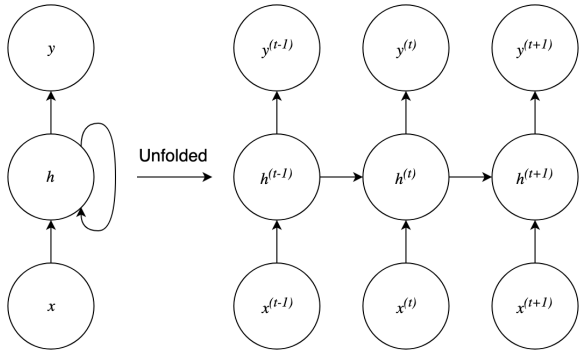


Figure 2.13: Example of recurrent neural network. We can see that an activation loop with a feedback connection essentially is feeding it's output to future versions of itself.

This introduces a cell in the neural network that maintains separate short-term and long-term states, which helps the network take earlier time steps into account.

2.4.4 Convolutional neural networks

Convolutional neural networks (CNNs) [LBD+89] are, just like the RNNs discussed in section 2.4.3, a specialised kind of neural network. CNNs specialise in processing data with a known grid-like topology [GBC16]. They are most frequently used to analyze visual imagery, as this can be represented as a 2D grid of pixels. It can also be applied to time-series measured in regular intervals, as they can be seen as 1D grids.

CNNs are inspired by the animal visual cortex, cortical neurons only responding to stimuli in a restricted area of the animal's visual field. CNNs use a technique called sparse connectivity to replicate this in an artificial neural network. Sparse connectivity entails fewer connections between each layer of the neural network than the MLPs and RNNs. MLPs and RNNs use matrix multiplication, which means that all neurons on each layer are connected to every neuron on the adjacent layers. A comparison is given in figure 2.14 and figure 2.15 below.

Sparse connectivity allows a neural network to handle larger tasks more efficiently, as it divides large grids into smaller units. Let us, as an example, consider a 256x256 pixel image in which we want to recognise objects using an input layer, an output layer and 3 hidden layers. The job of the first hidden layer is to recognise the edges of any shapes in the image by inspecting the pixels in the input layer. If every pixel takes all other pixels into consideration when determining whether it contains an edge, then this would give 65536 parameters taking input from 65535 other parameters. If

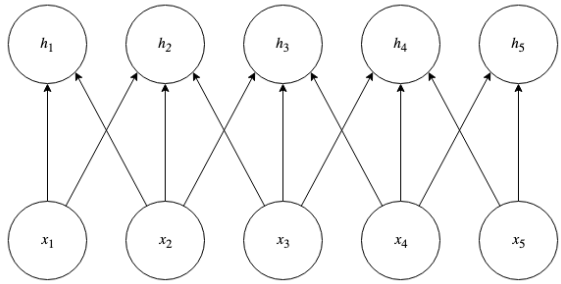


Figure 2.14: Example of the connections in a CNN.

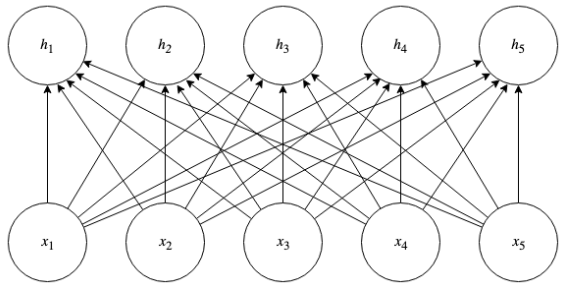


Figure 2.15: Example of the connections in a typical MLP.

we instead divide the picture into squares of e.g. 8x8 pixels, then 1024 parameters each take input from 64 parameters, so substantially decreasing the complexity. The next hidden layer can then look at the detected edges and try to recognise shapes such as corners, circles, etc. by considering detected edges that are in proximity to each other. The 3$^{\text{rd}}$ hidden layer can, finally, look at the recognised shapes, and try to match them to an object. An illustration of this example is given in figure 2.16.

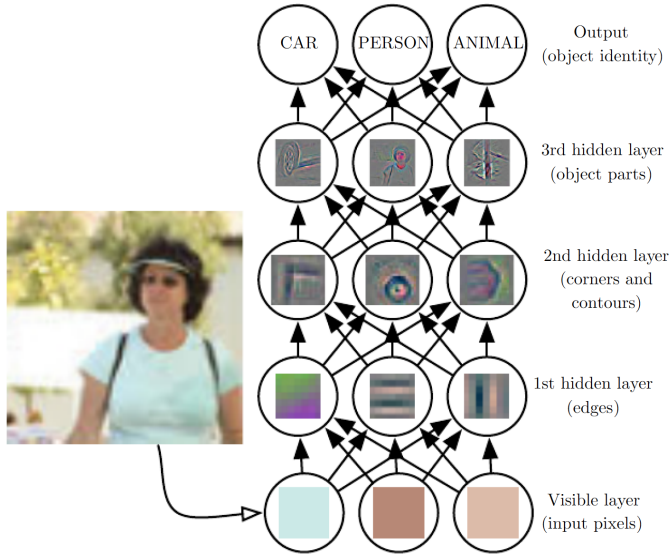


Figure 2.16: Object recognition in an image using CNNs.

Chapter 3
Experiments

This chapter describes the execution of the experiments of the thesis. As we will be conducting two different types of experiments, the experiments will be split into two parts. First we will cover experiments concerning the latency distributions mentioned in **RQ3**. Then we will cover experiments that concern the discovery of network changes introduced in **RQ1** and **RQ2**. As is evident by the number of experiments in each section, we have chosen to focus on the experiments related to **RQ3**.

3.1 General setup and procedures

This section covers the setup and procedures used by all the experiments. The specific details of each experiment are provided in section 3.2 and 3.3.

3.1.1 The data gathering phase

An early 2015 13" MacBook Pro running macOS Mojave version 10.14.4 [Inc17]. was used for data gathering. This computer model does not have an ethernet port. A *Thunderbolt to Gigabit Ethernet Adapter* was therefore required to connect to a wired network. The client computer's position in relation to the routers was set specifically for every experiment. A short python script was written to gather the data, and is given in appendix D. The script extracted the latency values from the ping response, and saved them to a `.csv` file. The script set the response time of a packet to 0 when a ping packet did not receive a response or timed out. We chose 0 because it's outside the range of the other measurements, and hopefully would therefore be interpreted as a special case by our deep learning model.

3.1.2 The deep learning phase

The *TensorFlow* [AAB+15] and *Keras* [C+15] libraries were used for deep learning. library which is specifically geared towards deep learning. It is written in Python and functions as a high-level API for neural network building and configuration. This

makes the process of experimenting with neural networks easier and faster. It is capable of running on top of multiple lower-level libraries such as TensorFlow, CNTK [SA16], or Theano [The16]. We chose to use TensorFlow, as it is the most common library used for deep learning and at the same time provides the functionality we need. TensorFlow is also an open source software library and was originally developed by the Google Brain team within Google's Machine Intelligence Research organization for machine learning and deep learning research. It is, however, general enough to be applied in other domains. The release versions we used were Keras 2.2.4 and TensorFlow 1.14.0. Keras 2.2.4 does not support Python 3.7, so it was run in a Python 3.6.8 environment.

Our data essentially is groups of time series. We therefore need to use a model that is efficient at handling time series classification. Karim, Majumdar, Darabi Chen in 2018 published a paper [KMDC18b] on LSTM-FCN models [KMD], which can handle this problem well. LSTM-FCN combines RNNs with CNNs, and it has been demonstrated that the architecture can achieve high time series classification performance. This is therefore the model architecture we chose for this project. The code for building our model has mostly been taken from the GitHub-repository [KMDC18a] referred to in the aforementioned paper, and can be found in appendix B.

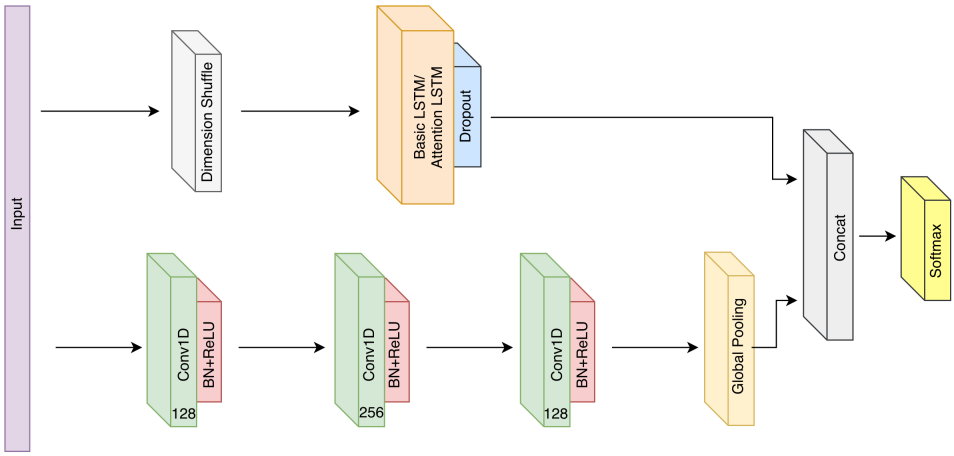


Figure 3.1: LSTM-FCN architecture. [KMDC18b]

We, after setting up the model, have to prepare the data so it has a format our model can work with. The first step was to add a one-hot encoded label to the data of every measurement series. Then we split the data into a training and test set, 70% of the data being assigned to the training set, the remaining 30% to the test set.

We, after gathering the data, labelled every series with their router type by

adding a final element to every series containing the router type. The labels were then one-hot encoded and the data was then split into training data and test data.

3.1.3 The signal processing phase

We used the `print_and_return_statistics()` method found in appendix A to calculate the statistical parameters mentioned in section 2.3 and the packet loss in each series. 2.3 as well as the packet loss in each series, we used the `print_and_return_statistics()` method found in appendix A.

`calculate_and_print_statistics()` consists of three main steps. In the first, it goes through all the measurement series in our data set and counts the zeroes. It then divides the zeroes in each series by the total number of measurements in that series, to find the packet loss.

We then removed the upper outliers of each series. These are most likely a result of traffic from other sources, and will not represent the normal ping response of the pinged node. They are not representative and huge outliers will also distort the remaining parameters. Most of our data are measurements centred around 1 ms. Outliers above 100ms can therefore have a great impact on the mean, standard deviation and range of our distribution. The outliers are removed by recursively calculating the interquartile range (IQR) and mean ($\bar{X}$), and then removing the values higher than $1.5 * IQR + \bar{X}$. This is repeated until the mean and IQR stagnates.

Thirdly we calculate the rest of the parameters for every series using SciPy [JOP$^+$]. We keep track of the minimum and maximum value for every parameter for each group of measurement series, and print them to the console.

3.2 Ping signatures

The purpose of these experiments is to discover whether we can identify routers or router types by analysing the measurement series we receive when sending a series of ping packets. We have elected to call this hypothetical identifiable ping response a *ping signature*, and will use this term in this thesis. By router types we mean models such as the *Nokia 7750 SR-s*, the *Huawei NE9000* or the *Ericsson Router 6675*.

To perform the experiments, we have been given insight into the router types in a large network. We will be working with 4 different router types, two samples of each model. For security reasons we will not disclose the router types or their respective IP addresses in this paper, but use a general denotation. The relation between the denotation used for different routers and router types is given in table 3.1.

Type A	Router 1
	Router 2
Type B	Router 3
	Router 4
Type C	Router 5
	Router 6
Type D	Router 7
	Router 8

Table 3.1: Routers and router types.

Routers 1-7 are all within a 500m radius of the client, while router 8 is approximately 2 km away. This will add between 2 and 10 microseconds of propagation delay for the different routers.

We will use deep learning (section 2.4) and Signal Processing (section 2.3) to analyse our measurements. Training and testing deep learning models is computationally expensive. We therefore want to compare its accuracy with that of regular signal processing methods, and so evaluate its necessity.

3.2.1 Preliminary experiment: Delay distribution

This experiment was not originally intended to be a part of this paper. It is, however, included as a result of the impact it had on the direction of the project as inspiration for **RQ3**. The approach and procedure is therefore different to the rest of the experiments in this section.

Purpose

The purpose of this experiment was to get an idea of the distribution of the ping measurements. We knew that the software in the client and the routers would introduce differing amounts of processing delay for each ping packet. The goal was therefore to get an understanding of the magnitude of these processing delays. We inspected the magnitudes of the processing delays to get an impression of the accuracy we could expect from the ping measurements. The accuracy would indicate how large a network change would have to be, for it to be detected using ping. We pinged two different routers on the same path (i.e. the packets to the second router went through the first) to see if we were able to distinguish them from each other.

Setup

The client computer running the ping program was connected to the two routers as shown in figure 3.2.

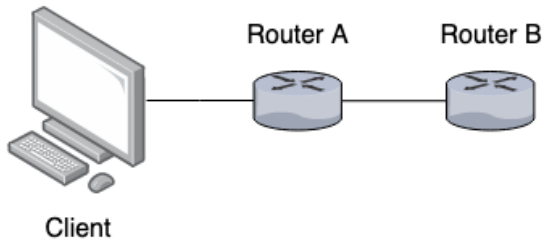


Figure 3.2: Setup for preliminary experiment.

Procedure

The experiment was performed by sending 100 000 ping packets to each router, at 100 packets per second (p/s). Packet size was the standard 56 byte payload + 8 byte header. Outliers were filtered out by finding the router with the highest minimum delay, and then discarding all values more than twice this value. After the outliers were filtered out, a histogram was plotted using python's matplotlib library [Hun07]. The code for filtering and plotting is given in appendix A, in methods `remove_outliers()` and `plot_files()`.

Results

The histogram showing the distribution of the two filtered measurement series is shown in figure 3.3. We found it interesting that Router B had significantly lower delay even though Router B was further away from our client than Router A. The minimum delay for Router A was 0.436 ms, while Router B had 0.196 ms minimum delay.

The width of the distributions for Router B were also noticeably different. Every packet to Router B had to go through A. We can therefore see that processing delay is the biggest factor when sampling routers as close to the client as these. This means that the distribution shown in figure 3.3 gives us a better picture of the capacity of the router that handles the ping request than distance between the client and router. We wanted to explore this further to see whether similar routers behaved similarly, and how accurately we could classify routers based on a series of ping measurements. The remainder of the experiments in section 3.2 therefore adress this.

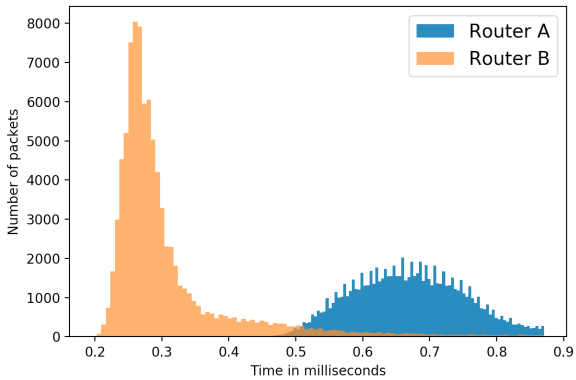


Figure 3.3: Results of preliminary experiment.

3.2.2 Experiment 1: Separating two routers

Purpose

In this experiment, we compare the measurement series from two different router types, **Model C** and **Model A**, to see how accurately we could classify each measurement series using both deep learning and signal processing.

Setup

The client computer running the ping program was connected to the two routers as shown in figure 3.4. The routers denoted with an X are unknown types of routers between the client and the target router.

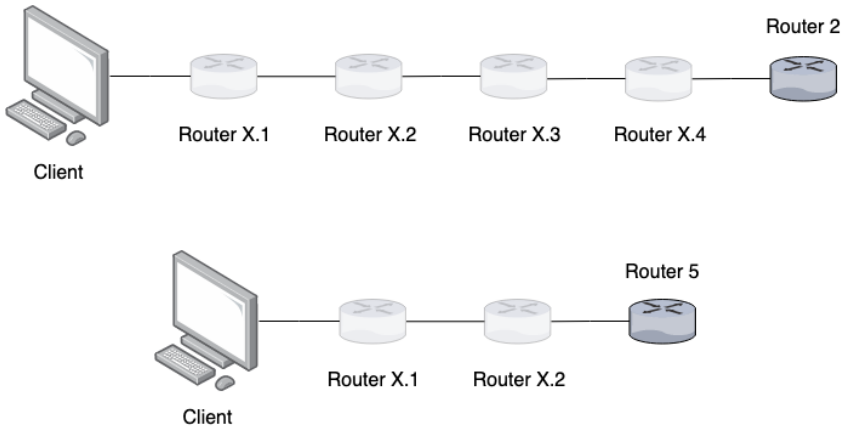


Figure 3.4: Setup for experiment 1.

Procedure

1000 measurement series each of 5000 measurements were gathered for both routers at a rate of $1000p/s$. Each of the 1000 series a 56 bytes (standard) payload. The data, after being prepared, was passed into our LSTM-FCN model for training as a 3-dimensional array in a (1400, 1, 5000) shape. 5, 2 and 1 epochs were used to train the model, to determine the necessary epoch count to achieve specific accuracies. The general setup and procedure described in section 3.1.3 were used for signal processing analysis.

Results

The accuracy of the epochs are listed in table 3.2. It's clear that our model was able to classify our measurement series using relatively low amounts of training.

Epochs	5	2	1
Accuracy	100%	100%	94.17 %

Table 3.2: Results of deep learning classification for experiment 1.

Table 3.3 shows the parameters for signal processing. The deep learning approach provided 100% accuracy. It is, however, fairly easy to classify the two routers by looking at the signal processing parameters in 3.3. We would also achieve 100% accuracy in this experiment if we implemented a rule that classified every series with a minimum delay lower than 600ms as Router 2, and the rest as Router 5. We can also observe that the router furthest away from the client had the lower minimum delay seen in the preliminary experiment.

Looking at the minimum delay is a straightforward way of classifying the routers in this experiment. This is, however, a parameter that can not be depended upon. Router processing delay affects minimum delay. However, so do other components in the network between the network and the client. In our example, Router 2 could have minimum delay values that were similar to Router 5 if it was positioned further away from the client. Minimum delay could therefore just be a measurement of the routers position in relation to the client. We can reject the use of minimum delay as a parameter in signal processing. We can not, however, affect the parameters the deep learning model is allowed to use. Consequently we need different data to determine whether the deep learning model is still able to classify the measurement series correctly without looking at minimum delay.

Parameters		Router 2	Router 5
Packet loss	Min	0.0%	0.0%
	Max	0.0%	0.035%
Minimum delay	Min	0.409ms	0.651ms
	Max	0.503ms	0.752ms
Range	Min	0.152ms	0.092ms
	Max	0.287ms	0.188ms
Skewness	Min	-0.904	-0.606
	Max	0.630	0.204
Kurtosis	Min	-0.329	-0.288
	Max	2.638	3.476
Mean	Min	0.589ms	0.800ms
	Max	0.630ms	0.814ms
Standard deviation	Min	0.022ms	0.013ms
	Max	0.047ms	0.021ms

Table 3.3: Statistical parameters of the measurement series from both routers for experiment 1.

3.2.3 Experiment 2: separating two routers with similar minimum delay

Purpose

The purpose of this experiment is to determine whether we are still able to classify a measurement series of routers with a similar minimum delay.

Setup

The data gathering setup was similar to experiment 1 (Figure 3.4), as the same routers were used.

Procedure

This experiment, as in experiment 1, gathered 1000 series of 5000 measurements at $1000p/s$. The payload of these measurement series was, however, set to 100 bytes. The payload size was changed to see if this impacts any of the parameters in a significant way. We wanted to simulate a situation where the measurement series on two different router types had similar minimum delay values. No such series was, however, available to us. We therefore had to manipulate the data we already had. We found minimum delay for both groups of measurement series and subtracted this from every measurement, then added 0.100ms. The series were then prepared in the

way described in section 3.1.2. We trained the model on the training set until we achieved 100% accuracy on the test set.

Results

The increasing accuracy of the model for each epoch can be seen in table 3.4. This experiment, however, required 1 epoch more to achieve 100% accuracy than experiment 1. 3 epochs is light training and did not take more than 15 minutes on our system.

Epochs	1	2	3
Accuracy	50.67%	99.67%	100%

Table 3.4: Results of deep learning classification for experiment 2.

The two routers cannot be classified based on the minimum delay in table 3.5. The range of the means and standard deviations, however, still provides ways to classify every measurement series with perfect accuracy. Every series in our data set with a mean higher than 0.270ms or a standard deviation higher than 0.021ms will always be Router 2.

Parameters		**Router 2**	**Router 5**
Packet loss	Min	0.0%	0.0%
	Max	0.0%	0.0564%
Minimum delay	Min	0.100ms	0.100ms
	Max	0.204ms	0.201ms
Range	Min	0.151ms	0.092ms
	Max	0.259ms	0.191ms
Skewness	Min	-0.913	-0.634
	Max	0.334	0.126
Kurtosis	Min	-0.055	0.079
	Max	3.066	3.678
Mean	Min	0.288ms	0.250ms
	Max	0.301ms	0.266ms
Standard deviation	Min	0.022ms	0.013ms
	Max	0.031ms	0.020ms

Table 3.5: Statistical parameters of the measurement series from both routers for experiment 2.

It is interesting to note that Router 2 range of means is higher than that of Router 5, even though the Router 2 range of minimum delay was lower. As shown in

figure 3.4 Router 2 is further away from the client, but still had a lower minimum delay. This indicates that even though Router 2 is faster than Router 5, Router 5's measurements are more compact, and have a lower spread than those of Router 2. A reflection of this can be seen in figure 3.5.

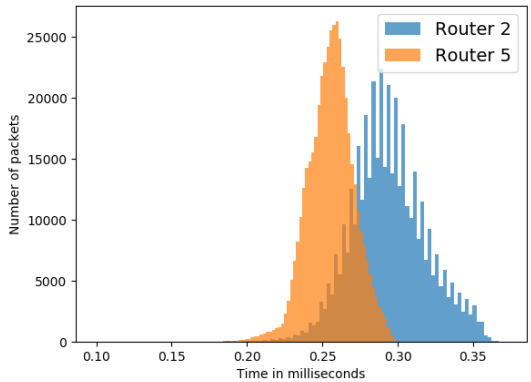


Figure 3.5: Distribution of measurement series used in experiment 2.

3.2.4 Experiment 3: classifying unseen routers

Purpose

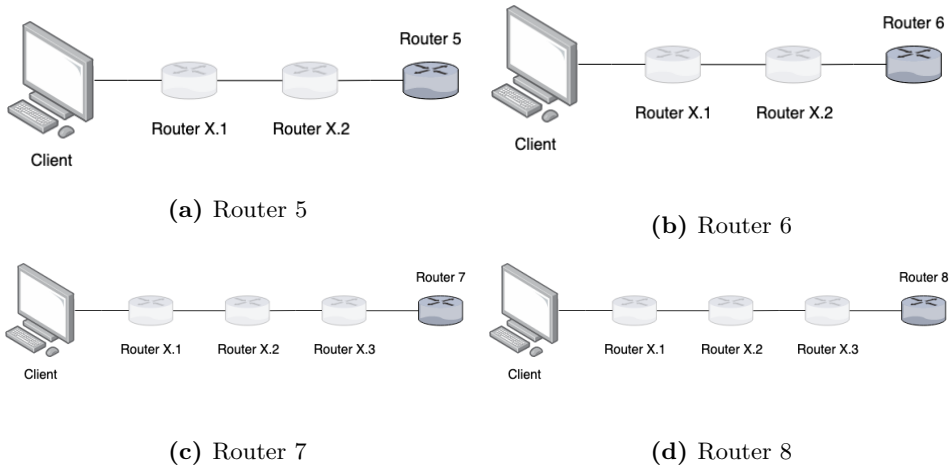
We have so far only used two routers in our experiments, our training and test data also being gathered on the same routers. The test data has therefore belonged to a router that the model has already seen data from. We however wanted to see in this experiment, if we can classify two unseen routers by examining two different routers of the same type. The router types chosen were those with most similar histogram distributions on manual inspection.

Setup

The position of the four routers in relation to the client is shown in figure 3.6. The routers denoted X.N are unknown router types between the client and the target routers.

Procedure

We gathered 4000 series of 1000 measurements in this experiment for each router at $100p/s$ The series had this time varying payload sizes, 16 bytes, 56 bytes, 500 bytes and 1000 bytes. The data was then rearranged instead into 1000 series of 4000 measurements, each series consisting of 1000 measurements for every payload.



(a) Router 5

(b) Router 6

(c) Router 7

(d) Router 8

Figure 3.6: Setup for experiment 3.

We used different payloads to see if we could detect router behaviour that deviated with payload, something that could help us to classify them. We again chose to manipulate the gathered data in the same way as in experiment 2, to simulate hard conditions for our analysis tools. We trained the model using the data from router 5 and 7, one of each type.

To determine how well we could classify the routers based on the Signal Processing parameters we got, we wrote a python script. This script assigned scores to each router for every series, before classifying the series as the router with the highest score. The scipt consist of 3 functions: `categorize_unseen()`, `categorize_serie()` and `get_index_of_closest_range()`, found in appendix A.

Results

The results from deep learning are given in table 3.6. We were not able to achieve perfect accuracy in this experiment, although we were close at one point. Table 3.6 shows accuracy initially increasing, before dropping to 47.83% and remaining there. This represents a worse alternative than guessing the same router type for every measurement series, which is obviously a bad result. The reason for this drop is most likely overfitting. Overfitting means that our model is not generalising well, and that it has instead memorised the training data. It will perform well on the training data, but fail at the test data.

We can see, from the statistical parameters in 3.11, that we are no longer able to divide the two routers by any one parameter. This means that the classification

Epochs	10	20	30	40
Accuracy	87.33%	99.50%	47.83%	47.83%

Table 3.6: Results of deep learning classification for experiment 3.

process is no longer as straightforward. We can see a significant difference in the range of the kurtosis. The ranges, however, still overlap. The classifying script managed to classify with 52.65% accuracy, which is slightly better than guessing randomly.

Parameters		**Router 5**	**Router 7**
Packet loss	Min	0.0%	0.0%
	Max	0.0%	0.0%
Minimum delay	Min	0.100ms	0.100ms
	Max	0.188ms	0.168ms
Range	Min	0.126ms	0.110ms
	Max	0.351ms	0.336ms
Skewness	Min	-0.938	-1.019
	Max	0.759	0.831
Kurtosis	Min	-1.158	-1.313
	Max	0.568	0.057
Mean	Min	0.218ms	0.204ms
	Max	0.318ms	0.287ms
Standard deviation	Min	0.018ms	0.021ms
	Max	0.065ms	0.057ms

Table 3.7: Statistical parameters of the measurement series from both routers for experiment 3.

Despite getting close to perfect accuracy, this was the first experiment in which we did not achieve this by deep learning. Manual inspection of the Signal Processing parameters also could find no distinctive features that could be used to distinguish the data of the two routers. The next experiment will also try to solve this, but by using other data to see if we can better our results for these routers.

3.2.5 Experiment 4: Classifying unseen routers using packet loss

Purpose

The purpose of this experiment is to see if we can improve the results from experiment 3 by using different data. The data we used was generated by exposing the routers

to a heavier load than in previous series, increasing their drop rate.

Setup

The setup was the same as for experiment 3, as shown in figure 3.6.

Procedure

We changed the rate of packets in the data gathering phase to 10 000p/s. The payload was set to 500. The measurement series was, however, kept to 1000 measurements, and we collected 1000 series for every router. As in experiment 3, the data from router 5 and router 7 was used for training, while router 6 and router 8 data was used for testing. We only calculated packet loss for Signal Processing.

Results

The results of the deep learning training are shown in table 3.8. As we can see, the packet loss could be used to classify the routers quickly, and we were again able to achieve perfect accuracy.

Epochs	1	2	3
Accuracy	99.64%	99.64%	100.00%

Table 3.8: Results of deep learning classification for experiment 4.

Table 3.9 shows the packet loss parameter values found by Signal Pro- cessing. We see that the ranges of packet loss for Router 5 and Router 7 do not overlap. We should be able to accurately classify an unseen measurement series with a similar load by manually inspecting the packet loss.

Parameters		**Router 5**	**Router 7**
Packet loss	Min	85.9%	42.5%
	Max	92.6%	79.9%

Table 3.9: The packet loss parameter from the measurement series from both routers in experiment 4.

This experiment yielded 100% accuracy. However, relying on the packet loss of routers at certain loads is not desirable. The configuration of the routers can greatly affect packet loss, so making it an unreliable parameter. Most home networks are, however, likely to use the standard configuration. Packet loss in these cases can therefore certainly be of help. We did not use this parameter in the remaining

experiment, as we wanted to explore how accurately we can classify by looking at the relation between actual delay measurements.

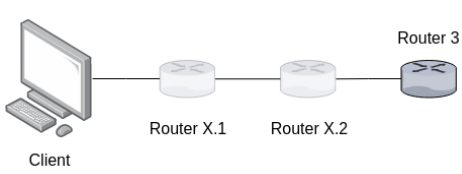
3.2.6 Experiment 5: Classifying 4 different unseen routers

Purpose

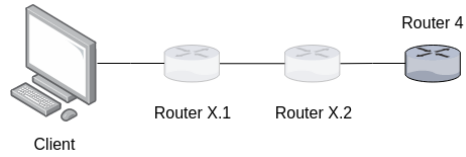
Our final ping signature experiment explores how accurately we can classify all 4 router types we have been given access to.

Setup

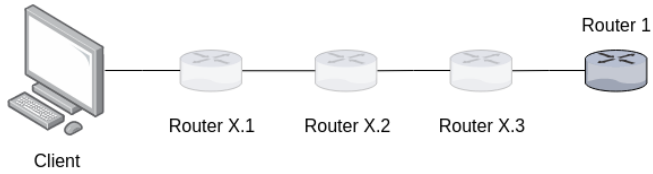
The setup of the routers not illustrated in previous experiments are given in figure 3.7.



(a) Router 3



(b) Router 4



(c) Router 1

Figure 3.7: Addiditional setup for experiment 5.

Procedure

The data gathering phase of this experiment was similar to that of experiment 3. The data used in experiment 3 was also used in this experiment for the routers included in the experiment. Every router was therefore subject to 1000 series of 4000 measurements with 4 different payloads, 16 byte, 56 byte, 500 byte and 1000 byte. The data was gathered at a rate of $100p/s$. We, as in experiment 3 and 4, trained our model on one of the routers of each router type, before testing the model on the remaining unseen routers.

Results

The results of the deep learning training are given in table 3.10. This shows that accuracy was not perfect, but even so shows that we can identify the router type of unseen routers to a high degree of accuracy.

Epochs	10	20	30
Accuracy	78.20%	89.12%	96.15%

Table 3.10: Results of deep learning classification for experiment 5.

Table 3.11 shows the statistical parameters for Signal Processing for the standard payload (56 bytes). The tables for the remaining payloads are given in appendix C. We note that there are no parameters that can separate Router 1 and Router 3 from the remainder of the routers. The ranges of every parameter overlap as in experiment 3.

Parameters		**Router 2**	**Router 4**	**Router 6**	**Router 8**
Minimum delay	Min	0.100ms	0.100ms	0.100ms	0.100ms
	Max	0.208ms	0.209ms	0.224ms	0.170ms
Range	Min	0.215ms	0.247ms	0.222ms	0.104ms
	Max	0.447ms	0.412ms	0.394ms	0.342ms
Skewness	Min	-0.499	-0.767	-0.822	-0.910
	Max	0.502	0.637	0.683	0.789
Kurtosis	Min	-0.640	-0.599	-1.008	-1.375
	Max	0.315	0.568	0.331	0.023
Mean	Min	0.293ms	0.277ms	0.296ms	0.203ms
	Max	0.372ms	0.360ms	0.355ms	0.296ms
Standard deviation	Min	0.038ms	0.047ms	0.045ms	0.019ms
	Max	0.072ms	0.077ms	0.062ms	0.060ms

Table 3.11: Statistical parameters of the measurement series from all routers for experiment 5 for payload of 56 bytes.

This was the most challenging experiment we could invent from the data we had access to. For this reason, we will not attempt any further experiments in this section. The results of our experiments will be discussed in section 4.1.

3.3 Detecting network changes

3.3.1 Experiment 6: adding 1 fast switched component to our network path

Purpose

In this experiment, we want to examine whether we can detect the addition of 1 fast switched component to the path of our ping packets. This is either a switch or a fast switched router.

Setup

The setup for experiment 6 is shown in 3.8. Router 9 and Router 10 are less than 200 meters from each other, the additional transmission delay being around 0.001ms.

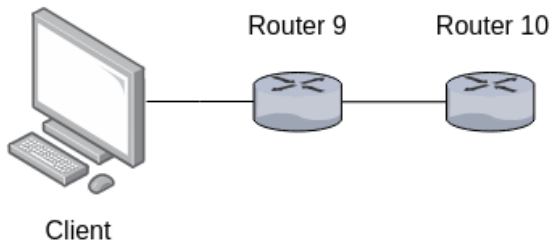


Figure 3.8: Setup for Experiment 6.

Procedure

There were no fast switched components which we could add to our network path. We therefore needed to simulate this condition if we were to carry out this experiment. We therefore used two routers of the same type in series, the added component being the first of the two routers in series. As seen from experiments 1-5, it is essential that the routers in series are the same router type. The classification could otherwise end up separating on the router signature rather than on the difference in the network path.

Results

Epochs	10	20	30
Accuracy	48.75%	53.24%	52.00%

Table 3.12: Results of deep learning classification for experiment 6.

As we can see from the results in table 3.12, our deep learning model was able to classify the changes to a high degree of accuracy. This is also reflected by our Signal Processing parameters seen in table 3.13. We can see that Router 10 generally has higher delay, but the ranges are overlapping, and would be difficult to separate.

Parameters		**Router 9**	**Router 10**
Minimum delay	Min	0.334	0.363
	Max	0.433	0.454
Range	Min	0.195	0.236
	Max	0.520	0.474
Skewness	Min	-0.872	-0.796
	Max	0.765	0.727
Kurtosis	Min	-0.662	-0.750
	Max	0.797	0.649
Mean	Min	0.492	0.525
	Max	0.602	0.619
Standard deviation	Min	0.035	0.047
	Max	0.108	0.102

Table 3.13: Statistical parameters of the measurement series from both routers for experiment 6.

We chose to focus our attention on the ping signature experiments and we therefore carried out no further experiments on detecting network changes using ping. Our results will be briefly discussed in section 4.2

Chapter 4
Results

In this chapter, we present the key findings from chapter 3 and discuss the implications of these findings. The chapter is divided into two sections, one for each type of experiment. Section 4.1 covers experiments 1-5, in which we explored our ability to recognise routers based on the patterns of their ping response. Section 4.2 covers experiment 6.

4.1 Ping signatures

We immediately observed, from our first experiment, that we were able to separate two routers based on their ping responses using both deep learning and Signal Processing. We saw, however, a decrease in accuracy as the perceived difficulty of our experiments increased. All our experiments, when deep learning was used to analyse the data, ended up with an accuracy that we considered to be satisfactory. Signal Processing was sufficient for experiments 1, 2 and 4. However, signal processing experienced difficulties classifying routers when the range of the statistical parameters overlapped, as we saw in experiment 3 and 5.

We consider experiment 3 and experiment 5 to be most relevant to the real world application of our findings. These use our training to identify to a high degree of accuracy the type of previously unseen routers. This can be used to identify routers in an unknown network, where you do not have access to the router types. If these routers have known weaknesses, this could imply an increased security risk, as a potential attacker could exploit known router weaknesses if they are able to identify the router type.

Higher accuracy could have been achieved for experiment 3 and 5, if we had more routers of the same type to train our deep learning model on. More samples of each router type would help our model identify the patterns of the router types rather than the properties of a single data set. This would have made our training less prone to overfitting.

It is worth noting that the classification task becomes more difficult the more router models we add, due to more routers with similar ping signatures. However, even if we cannot classify the specific router correctly, then we should still be able to narrow down the potential router type to which a ping signature can belong.

To conclude, we find from experiments 1-5 that we were are able to identify the router types at our disposal to a high degree of accuracy. Our answer to **RQ3** therefore is that we are indeed able to identify a router by looking at the latency pattern when pinging it. We observed, when comparing Signal Processing with deep learning, that both methods yielded the same results for some of our experiments, but that deep learning was superior as the challenges became greater.

4.2 Detecting network changes

We were unable, in experiment 6, to achieve a high degree of accuracy in the measurements of our data. The change in network path was too small for a tool such as ping to accurately classify the two network paths. A hardware tool for measuring delay would probably be necessary for us to differentiate between the two paths. We could have expanded our experiments to see how many network components we would need to traverse through for ping to provide accurate results for the changing of network paths. However, as mentioned previously, we chose to focus on the experiments related to ping response. Our answers to **RQ1** and **RQ2** are therefore inconclusive.

Chapter 5
Summary

The identificational capabilities of ping on IP-routers have been examined in this thesis. The thesis started out as a project that planned to identify the capabilities of the ping program to detect changes in short network paths. However, the focus of the project changed after initial experiments yielded interesting results. The processing delay we initially had considered to be noise in our measurements, became the focal point of the thesis. The main bulk of our experiments therefore revolved around this. Ping is an old and widely distributed tool. We were not, however, able to find any previous work on this subject, and this thesis does therefore not add to any previous studies.

A python script was written to gather the data we required to perform our experiments. In total over 100 000 000 ping requests were sent, not all being used for experiments. We chose, for some of the experiments, to manipulate some of the measurement series to simulate harder conditions for our analysis methods to operate in. Two methods were used to analyse our data: Signal Processing and deep learning.

The results of our experiments confirmed that we were able to identify different types of routers to a high degree of accuracy. We were also able to land on a preferred method of analysis, that of deep learning. Signal Processing was sufficient for some task analysis. But it never outperformed deep learning. If this project was to be repeated, it would be more efficient to use only deep learning for analysis.

5.1 Suggestions for future work

Future work related to this project could proceed in a number of directions. Firstly, this thesis does not answer **RQ1** and **RQ2** put forth in the introduction. Evaluating the capabilities of the ping program with more experiments is therefore an area that still can be expanded. We suggest that, in this work, the components for which you want to determine whether they can be detected by the ping program, are attach on the client side of the path. This approach will ensure the results of you measurements

are not a result of different ping signatures, but rather the inclusion of the additional components. Ping is not suited for minimal changes to network paths, and higher precision tools should be used.

Secondly, more experiments can be conducted on ping signatures. This project assessed routers in relative proximity to the client. We therefore experienced low amounts of noise. Examining how increasing the distance between the client and the target routers affects accuracy would be of interest to this topic, as this adds unwanted noise from other traffic.

Exploring how different router configurations can affect the ping signature would also be a valuable contribution. The configuration of the router is able to affect the packet loss. Examining whether you can make a router completely unrecognisable by configuring its settings would be interesting. If it were possible to change the router signature of a router completely, it would negate the negative security implications of router signatures.

Finally, there is also room for repeating the ping signature experiments in this thesis with extra routers. Our suggestion would be to group routers that perform the same functionality (e.g. home routers, core routers, gateway routers), and experiment on their ping signatures. These routers would most likely be similar in terms of their processing capabilities, and would therefore be harder to classify.

References

[AAB+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[Agg14] Charu C Aggarwal. *Data classification: algorithms and applications*. CRC press, 2014.

[BBP88] Robert Braden, David A. Borman, and Craig Partridge. Computing the internet checksum. *RFC*, 1071:1–24, 1988.

[BFZ07] Hitesh Ballani, Paul Francis, and Xinyang Zhang. A study of prefix hijacking and interception in the internet. *ACM SIGCOMM Computer Communication Review*, 37(4):265–276, 2007.

[Bra89] Robert T. Braden. Requirements for internet hosts - communication layers. *RFC*, 1122:1–116, 1989.

[C+15] François Chollet et al. Keras. https://keras.io, 2015.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[Hun07] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[Inc17] Apple Inc. MacBook Pro (retina, 13-inch, early 2015) - technical specifications, 2017.

[JOP+] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. [Online; accessed <today>].

[KMD] Fazle Karim*, Somshubra Majumdar*, and Houshang Darabi. Insights into lstm fully convolutional networks for time series classification. *Arxiv*.

[KMDC18a] Fazle Karim, Somshubra Majumdar, Houshang Darabi, and Shun Chen. Lstm-fcn. https://github.com/titu1994/LSTM-FCN, 2018.

[KMDC18b] Fazle Karim, Somshubra Majumdar, Houshang Darabi, and Shun Chen. Lstm fully convolutional networks for time series classification. *IEEE Access*, 6:1662–1669, 2018.

[KR13] James F Kurose and Keith W Ross. *Computer networking: A top-down approach featuring the internet, 6/E*. Pearson Education India, 2013.

[LBD+89] Yann LeCun, Bernhard E. Boser, John S. Denker, Donnie Henderson, Richard E. Howard, Wayne E. Hubbard, and Lawrence D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551, 1989.

[LW] Lars Landmark and Otto Wittner. Measuring packet forwarding behavior in a production network.

[Mit97] Tom M. Mitchell. *Machine learning*. McGraw Hill series in computer science. McGraw-Hill, 1997.

[Muu83] Mike Muuss. iputils. https://github.com/iputils/iputils/, 1983.

[Muu16] Mike Muuss. The story of the ping program, 1983. *Web-page: http://ftp. arl. army. mil/mike/ping. html*, 2016.

[Pos81a] Jon Postel. Internet control message protocol. *RFC*, 792:1–14, 1981.

[Pos81b] Jon Postel. Internet protocol. *RFC*, 791:1–51, 1981.

[SA16] Frank Seide and Amit Agarwal. Cntk: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 2135–2135, New York, NY, USA, 2016. ACM.

[The16] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[Tho03] L. E. Thon. 50 years of signal processing at isscc. In *2003 IEEE International Solid-State Circuits Conference, 2003. Digest of Technical Papers. ISSCC.*, pages 27–28, Feb 2003.

[WCL+88] Robert Wilensky, David N Chin, Marc Luria, James Martin, James Mayfield, and Dekai Wu. The berkeley unix consultant project. *Computational Linguistics*, 14(4):35–84, 1988.

[Wol96] David H. Wolpert. The lack of A priori distinctions between learning algorithms. *Neural Computation*, 8(7):1341–1390, 1996.

Appendix A
signal_processing.py

```
import matplotlib.pyplot as plt
from scipy.stats import skew, kurtosis
import numpy as np
import random

def remove_outliers(series_with_outliers):
    series_with_outliers.sort()

    input_max = max(series_with_outliers)


    list_len = len(series_with_outliers)
    first_quartile_index = int(list_len / 4)
    last_quartile_index = int((3 * list_len) / 4)

    first_quartile_value = round((series_with_outliers
        [first_quartile_index]), 3)
    last_quartile_value = round((series_with_outliers
        [last_quartile_index]), 3)
    IQR = last_quartile_value - first_quartile_value
    upper_threshold = last_quartile_value + (IQR * 1.5)

    series_without_outliers = []

    for e in series_with_outliers:
        if e < upper_threshold:
            series_without_outliers.append(e)
    if max(series_without_outliers) == input_max:
        return series_without_outliers
    elif max(series_without_outliers) < input_max:
```

```
        return remove_outliers(series_without_outliers)
    else:
        print("This␣should␣not␣happen")


def import_files(filenames):
    list_of_files = []
    for filename in filenames:
        file_as_list = []
        with open(filename, "r") as file:
            for line in file:
                split_line = line.split(",")
                float_list = [float(i) for i in split_line]
                file_as_list.append(float_list)
        list_of_files.append(file_as_list)
    return list_of_files


def print_and_return_statistics(unfiltered_files,
    filtered_files, routernames):

    statistics = []

    for index in range(0, len(routernames)):
        print(routernames[index] + ":")

        unfiltered_file = unfiltered_files[index]
        filtered_file = filtered_files[index]
        min_drop_rate = 100.0
        max_drop_rate = 0.0
        for series in unfiltered_file:
            len_with_zeroes = len(series)
            series_without_zeroes = list(filter(lambda x: x != 0,
            series))

            dropped_packages = (len_with_zeroes -
            len(series_without_zeroes))

            drop_rate = dropped_packages/len_with_zeroes
            if drop_rate < min_drop_rate:
                min_drop_rate = drop_rate
```

```
        if drop_rate > max_drop_rate:
            max_drop_rate = drop_rate

min_drop_rate = min_drop_rate * 100
max_drop_rate = max_drop_rate * 100

print("Package loss rate range: " +
    "%.3f" % min_drop_rate + "% - "
    + "%.3f" % max_drop_rate + "%")

min_min_delay = 100000.0
max_min_delay = -100000.0
min_range = 100000.0
max_range = -100000.0
min_skew = 100000.0
max_skew = -100000.0
min_kurtosis = 100000.0
max_kurtosis = -100000.0
min_mean = 100000.0
max_mean = -100000.0
min_std = 100000.0
max_std = -100000.0

for filtered_series in filtered_file:
    min_delay = min(filtered_series)
    if min_delay < min_min_delay:
        min_min_delay = min_delay
    if min_delay > max_min_delay:
        max_min_delay = min_delay

    max_delay = max(filtered_series)
    distribution_range = max_delay - min_delay
    if distribution_range < min_range:
        min_range = distribution_range
    if distribution_range > max_range:
        max_range = distribution_range

    skewedness = skew(filtered_series)
    if skewedness < min_skew:
        min_skew = skewedness
    if skewedness > max_skew:
```

```
                max_skew = skewedness

            kurt = kurtosis(filtered_series)
            if kurt < min_kurtosis:
                min_kurtosis = kurt
            if kurt > max_kurtosis:
                max_kurtosis = kurt

            mean = np.mean(filtered_series)
            if mean < min_mean:
                min_mean = mean
            if mean > max_mean:
                max_mean = mean

            std = np.std(filtered_series)
            if std < min_std:
                min_std = std
            if std > max_std:
                max_std = std

        print("Min delay range: " + "%.3f" % min_min_delay + "ms
␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣" + "%.3f" % max_min_delay + "ms")
        print("Measurement range range: " + "%.3f" % min_range + "ms
␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣" + "%.3f" % max_range + "ms")
        print("Skewedness range: " + str(min_skew) + "
␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣" + str(max_skew) + "")
        print("Kurtosis range: " + str(min_kurtosis) + "
␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣" + str(max_kurtosis) + "")
        print("Mean range: " + "%.3f" % min_mean + "ms
␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣" + "%.3f" % max_mean + "ms")
        print("STD range: " + "%.3f" % min_std + "ms
␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣" + "%.3f" % max_std + "ms")

        statistics.append([(min_min_delay, max_min_delay),
            (min_range, max_range), (min_skew, max_skew),
            (min_kurtosis, max_kurtosis), (min_mean, max_mean),
            (min_std, max_std)])

    return statistics
```

```
def filter_list(unfiltered_list):
    filtered_list = []
    for series in unfiltered_list:
        removed_zeroes = list(filter(lambda x: x != 0, series))
        filtered_series = remove_outliers(removed_zeroes)
        filtered_list.append(filtered_series)
    return filtered_list

def plot_files(file_list, number_of_series_to_be_plotted, routernames):
    number_of_files_to_be_plotted = len(file_list)
    for file in file_list:
        series_to_be_plotted = []
        if number_of_series_to_be_plotted == 0:
            series_to_be_plotted = lambda file:
            [measurement for series in file for measurement in series]

        elif number_of_series_to_be_plotted > 0:
            for example in range(0,number_of_series_to_be_plotted):
                index = random.randint(0,len(file))
                series_to_be_plotted.extend(file[index])
        if number_of_files_to_be_plotted == 1:
            plt.hist(series_to_be_plotted, 100, alpha=1.0)
        elif number_of_files_to_be_plotted > 1:
            plt.hist(series_to_be_plotted, 100, alpha=0.7)

    plt.ylabel("Number␣of␣packets")
    plt.xlabel("Time␣in␣milliseconds")
    plt.legend(routernames, loc=1, prop={'size': 14})
    plt.show()

def get_index_of_closest_range(list, value):
    closest = []
    for tuple in list:
        closest.append(min(abs(value-tuple[0]), abs(value-tuple[1])))

    numpy_array = np.array(closest)

    index = (np.abs(numpy_array-value)).argmin()

    return index
```

```
def categorize_serie(serie, stats):
    scores = [0] * len(stats)
    for parameter_index in range(0, len(serie)):
        parameter = serie[parameter_index]
        routers_in_range = []
        for router_index in range(0, len(stats)):
            router_params = stats[router_index][parameter_index]
            if (router_params[0] < parameter and parameter <
                router_params[1]):

                routers_in_range.append(router_index)

        parameter_range_list = []

        if len(routers_in_range) == 0:
            for router in stats:
                parameter_range_list.append(router[parameter_index])
            index = get_index_of_closest_range(parameter_range_list,
            parameter)

            scores[index] += 5

        if len(routers_in_range) == 1:
            scores[routers_in_range[0]] += 50

        if len(routers_in_range) > 1:
            for router_in_range in routers_in_range:
                scores[router_in_range] += 1

            for idx in range(0, len(stats)):
                if idx in routers_in_range:
                    router = stats[idx]
                    parameter_range_list.append(router[parameter_index])
                elif idx not in routers_in_range:
                    parameter_range_list.append((-1e8, -1e7))
                index = get_index_of_closest_range(
                    parameter_range_list, parameter)

                scores[index] += 2
    return scores
```

```
def categorize_unseen(unseen_files, stats):
    correctly_categorized = 0
    wrongly_categorized = 0

    for index in range(0, len(unseen_files)):

        file = unseen_files[index]
        for series in file:
            min_delay = min(series)
            distribution_range = max(series) - min_delay
            skewness = skew(series)
            kurt = kurtosis(series)
            avg = np.mean(series)
            std = np.std(series)

            series_scores = categorize_serie([min_delay,
                distribution_range, skewness, kurt, avg, std], stats)

            if series_scores.index(max(series_scores)) == index:
                correctly_categorized += 1
            else:
                wrongly_categorized += 1

    print(correctly_categorized /
        (correctly_categorized + wrongly_categorized))




def main(filenames, routernames, unseen_files):

    if len(filenames) != len(routernames):
        return
    else:
        unfiltered_files = import_files(filenames)
        filtered_files = []
        for file in unfiltered_files:
            filtered_files.append(filter_list(file))

        stats = print_and_return_statistics(unfiltered_files,
```

```
        filtered_files, routernames)

    unfiltered_unseen = import_files(unseen_files)

    filtered_unseen_files = []
    for unseen_file in unfiltered_unseen:
        filtered_unseen_files.append(filter_list(unseen_file))
```

Appendix B
deep_learning.py

```
from keras.layers import Dense, LSTM, Dropout, Input,
    concatenate, Activation
from keras.layers import Conv1D, BatchNormalization,
    GlobalAveragePooling1D, Permute
from keras.models import Model
from keras.callbacks import ModelCheckpoint, ReduceLROnPlateau
from keras.optimizers import Adam
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split

def get_one_hot(targets, nb_classes):
    res = np.eye(nb_classes)[np.array(targets).reshape(-1)]
    return res.reshape(list(targets.shape)+[nb_classes])

def prepare_data(filenames, unseen_filenames, labels,
    sizes_and_labeled, unseen):

    file_list = []

    #reads in all the csv files needed
    for file in filenames:
        read_files_list = []
        labeled = None
        for size in sizes_and_labeled:
            read_file = pd.read_csv(file + "_" + size +
                ".csv", header=None)
            if ("labeled" in size):
```

```
                labeled = read_file
            else:
                read_files_list.append(read_file, axis=1, sort=False,
                    ignore_index=True)

        read_files_list.append(labeled)
        concated_file = pd.concat(read_files_list)
        file_list.append(concated_file)

    merged_files = file_list[0].append(file_list[1:],
        ignore_index=True)

    for index in range(0, len(labels)):
        merged_files = merged_files.replace(
            to_replace=labels[index], value=index)

    if not unseen:
        # if we are not using unseen routers to test against (Ex 3 - 5),
        train, test = train_test_split(merged_files, test_size=0.3)

    if unseen:
        unseen_file_list = []

        for unseen_file in unseen_filenames:
            unseen_read_files_list = []
            unseen_labeled = None
            for size in sizes_and_labeled:
                unseen_read_file = pd.read_csv(unseen_file
                    + "_" + size + ".csv", header=None)
                if ("labeled" in size):
                    unseen_labeled = unseen_read_file
                else:
                    unseen_read_files_list.append(unseen_read_file,
                        axis=1, sort=False, ignore_index=True)

            unseen_read_files_list.append(unseen_labeled)
            unseen_concated_file = pd.concat(unseen_read_files_list)
            unseen_file_list.append(unseen_concated_file)

        unseen_merged_files = unseen_file_list[0].append(
            unseen_file_list[1:], ignore_index=True)
```

```
        for index in range(0, len(labels)):
            unseen_merged_files = unseen_merged_files.replace(
                to_replace=labels[index], value=index)

            train = merged_files.sample(frac=1).reset_index(drop=True)
            test = unseen_merged_files.sample(frac=1)
                .reset_index(drop=True)

    x_train_df = train.iloc[:, :-1]
    y_train_df = train.iloc[:, -1]
    x_test_df = test.iloc[:, :-1]
    y_test_df = test.iloc[:, -1]

    x_train = x_train_df.values
    y_train = y_train_df.values
    x_test = x_test_df.values
    y_test = y_test_df.values

    y_train_one_hot = get_one_hot(y_train, len(labels))
    y_test_one_hot = get_one_hot(y_test, len(labels))

    train_sequence_length = x_train[0].size
    test_sequence_length = x_test[0].size

    x_train = np.reshape(x_train, (x_train.shape[0],
        1, train_sequence_length))
    x_test = np.reshape(x_test, (x_test.shape[0], 1,
        test_sequence_length))

    return x_train, y_train_one_hot, x_test, y_test_one_hot,
        train_sequence_length, test_sequence_length




def main(filenames, unseen_filenames, labels,
    sizes_and_labeled, unseen, eval):
```

```
x_train, y_train, x_test, y_test, train_sequence_length,
    test_sequence_length = prepare_data(filenames,
    unseen_filenames, labels, sizes_and_labeled, unseen)

#setting up the LSTM-FCN model
ip = Input(shape=(1, train_sequence_length))

x = LSTM(8)(ip)
x = Dropout(0.8)(x)

y = Permute((2, 1))(ip)
y = Conv1D(128, 8, padding='same',
    kernel_initializer='he_uniform')(y)
y = BatchNormalization()(y)
y = Activation('relu')(y)

y = Conv1D(256, 5, padding='same',
    kernel_initializer='he_uniform')(y)
y = BatchNormalization()(y)
y = Activation('relu')(y)

y = Conv1D(128, 3, padding='same',
    kernel_initializer='he_uniform')(y)
y = BatchNormalization()(y)
y = Activation('relu')(y)

y = GlobalAveragePooling1D()(y)

x = concatenate([x, y])

out = Dense(2, activation='softmax')(x)

model = Model(ip, out)

#model has been set up

print(model.summary())
learning_rate = 1e-3

factor = 1. / np.cbrt(2)
dataset_prefix = 5
```

```
model_checkpoint = ModelCheckpoint("%s_weights.h5" %
    dataset_prefix, verbose=1, monitor='val_loss',
    save_best_only=True, save_weights_only=True)
reduce_lr = ReduceLROnPlateau(monitor='val_loss',
    patience=100, mode='auto', factor=factor, cooldown=0,
    min_lr=1e-4, verbose=2)

callback_list = [model_checkpoint, reduce_lr]

optm = Adam(lr=learning_rate)
model.compile(optimizer=optm, loss='categorical_crossentropy',
    metrics=['accuracy'])

accuracy = np.float64(0)
scores = []



if eval:
    model.load_weights("%s_weights.h5" % dataset_prefix)
    scores = model.evaluate(x_train, y_train, verbose=0)

    accuracy = scores[1]

    print("Accuracy: %.2f%%" % (accuracy * 100))
    print("Loss: " + str(scores[0]))

else:

    while True:
        #until we abort training, we train the model and save the b
        model.fit(x_train, y_train, validation_split=0.2,
            callbacks=callback_list, epochs=10, batch_size=64,
            verbose=2)
```

Appendix C
Tables

Parameters		Router 2	Router 4	Router 6	Router 8
Minimum delay	Min	0.100	0.100	0.100	0.100
	Max	0.205	0.196	0.228	0.205
Range	Min	0.269	0.243	0.183	0.098
	Max	0.417	0.433	0.391	0.343
Skewness	Min	-0.558	-0.855	-0.835	-0.907
	Max	0.432	0.587	0.661	0.879
Kurtosis	Min	-0.666	-0.516	-1.073	-1.431
	Max	0.503	0.669	0.350	0.036
Mean	Min	0.291	0.262	0.273	0.233
	Max	0.361	0.354	0.359	0.320
Standard deviation	Min	0.051	0.047	0.030	0.016
	Max	0.071	0.081	0.067	0.074

Table C.1: Statistical parameters of the measurement series from all routers for experiment 5 for payload of 16 bytes.

Parameters		Router 2	Router 4	Router 6	Router 8
Minimum delay	Min	0.100	0.100	0.100	0.100
	Max	0.334	0.219	0.216	0.179
Range	Min	0.236	0.160	0.135	0.103
	Max	1.316	0.415	0.383	0.340
Skewness	Min	-0.476	-1.169	-0.830	-0.831
	Max	0.810	0.814	0.843	0.871
Kurtosis	Min	-0.577	-0.797	-0.951	-1.188
	Max	0.512	2.157	0.530	0.087
Mean	Min	0.359	0.240	0.272	0.209
	Max	0.746	0.357	0.355	0.310
Standard deviation	Min	0.042	0.032	0.025	0.017
	Max	0.338	0.076	0.077	0.059

Table C.2: Statistical parameters of the measurement series from all routers for experiment 5 for payload of 500 bytes.

Parameters		Router 2	Router 4	Router 6	Router 8
Minimum delay	Min	0.100	0.100	0.100	0.100
	Max	0.212	0.238	0.241	0.178
Range	Min	0.269	0.156	0.158	0.105
	Max	75.687	0.424	0.398	0.354
Skewness	Min	-0.573	-1.590	-0.964	-1.006
	Max	0.482	0.765	0.784	0.842
Kurtosis	Min	-1.450	-0.822	-1.030	-1.358
	Max	0.521	3.115	0.932	0.407
Mean	Min	0.307	0.233	0.278	0.203
	Max	36.159	0.354	0.388	0.339
Standard deviation	Min	0.049	0.032	0.024	0.017
	Max	25.755	0.089	0.071	0.067

Table C.3: Statistical parameters of the measurement series from all routers for experiment 5 for payload of 1000 bytes.

Appendix D
gather_ping_data.py

```
def ping_and_save(numberOfDataPoints, ping_adr_list, size,
    packet_count):
    for dataPoint in range(numberOfDataPoints):
        for pingAdr in ping_adr_list:
            pingtimes = []
            timeouts = []
            ping = os.popen('sudo␣ping␣' + pingAdr + '␣-c␣'
                + str(packet_count) + '␣-i␣0.01␣-s␣' + str(size))
            result = ping.readlines()
            icmp_seq_list = list(range(0, packet_count))
            for i in result:
                if ("ttl=" in i):
                    temp = i.split("=")
                    temp2 = temp[3].split("␣")
                    icmp_seq = temp[1].split("␣")[0]
                    if int(icmp_seq) in icmp_seq_list:
                        icmp_seq_list.remove(int(icmp_seq))
                    else:
                        print(icmp_seq)
                    pingtimes.append(temp2[0])

            for missed_icmp_seq in icmp_seq_list:
                pingtimes.insert(missed_icmp_seq, "0")

            file = open(pingAdr.replace(".", "_") + "_size_"
                + str(size) + ".csv", "a")
            for i in pingtimes[:-1]:
                file.write(i + ",")
            file.write(pingtimes[-1] + "\n")
```

```
        file.close()
        if dataPoint%10 == 0:
            print(dataPoint)
```