CS EE World
https://cseeworld.wixsite.com/home
May 2023
30/34
A

Submitter Info:
Email: dhrumangupta06 [at] gmail [dot] com

# Investigating the Effectiveness of a Neural Network to Detect and Mitigate a Distributed Denial of Service Attack

## To what extent can a Feed Forward Neural Network successfully mitigate an HTTP-flood distributed denial-of-service attack?

Computer Science Extended Essay

---

Personal Code: jgc887

Word Count: 3901

## Table of Content


1. Introduction 1
   1.1 Worthiness 2
   1.2 Scope 3
2. Background Information 4
   2.1 Machine Learning and its Types 4
   2.2 Chosen Machine Learning Model 5
   2.3 Feed Forward Neural Networks 5
3. Experiment Methodology 7
   3.1 Generation of Data Sets 7
   3.2 Processing the datasets for use 11
   3.3 Dependent Variables 14
   3.4 Programming of the Feed Forward Neural Network 14
   3.5 Experimental Procedure 15
   3.6 Hypothesis 16
4. The Experimental Results 16
   4.1 Accuracy over epochs 16
   4.2 Accuracy on testing data 17
   4.3 Time taken for classification 18
5. Analysis 18
   5.1 Analyzing Accuracy 18
   5.2 Analyzing Performance 19
   5.3 Making sense of the drop in accuracy 19
   5.4 Computational Costs 20
6. Conclusion 20
6. Further Research Opportunities 20
   6.1 Investigating a change in the preprocessing of the data 20
   6.2 Utilizing different machine learning models 21
   6.3 Extending to different DDoS attacks 21

Works Cited 22

Appendix 24
   1: Code for simulation of DDoS 24
   2: Code for pre-processing data, training network, and running tests 33
   3: Screenshot of raw data 40
   4: Screencast of model evaluation 40
   5: List of Figures and Tables 40

## 1. Introduction

Cyber-attacks can maliciously disable machines, steal data, or use an infected machine as a point for other attacks. Distributed Denial of Service (DDoS) attacks are a type of attack that send fake requests to a machine and flood it by overloading the system. This leads to the machine crashing or being rendered unusable. Such attacks can cost firms large sums of money since they lose money for every second their server malfunction.

For example, the first few months of 2022 saw an unexpected increase in the number and duration of DDoS attacks, predominantly due to Russia's invasion of Ukraine[1]. Although not all DDoS attacks have political implications, they are a powerful tool for cyber warfare. This type of cybercrime has become increasingly common and can be used to attain nefarious goals.

A DDoS (distributed denial-of-service) attack is one of the most dangerous attacks in which the attacker aims to make a resource or server unavailable to its intended users. There are multiple ways to perform this, such as queuing requests, creating unterminated sessions, overloading packet sizes, etc. The attack is so dangerous because these requests are initiated by compromised devices, making it harder to distinguish between genuine and malicious requests – an IP or device rate limit does not prevent it. A recent example includes Cloudflare – one of the largest content delivery networks responsible for delivering over 7,000,000 websites – which was the target of one of the most significant DDoS attacks in history. Matters like this make it pressing to find ways to detect and mitigate DDoS attacks efficiently.

HTTP flooding is a common form of a DDoS attack. The compromised systems make continuous requests to a web server, using up its resources and preventing users from accessing

[1] Hacken. "How to Detect a DDoS Attack? - 5 Red Flags - Hacken." Hacken, 8 Aug. 2022, hacken.io/discover/how-to-detect-a-ddos-attack/. Accessed 13 Aug. 2022.

them[2]. In addition, the requests are usually sent to endpoints that require many resources to process – such as querying and processing large amounts of data from a database – to increase each request's overall impact. These attacks are necessary to identify, as they consume large amounts of bandwidth and computing power and deny access to them to genuine users. Hence, it is necessary to identify them automatically.

A DDoS request is hard to identify – no definite factors can be used to define it. While they target endpoints with a high packet size and processing time, it is hard to identify whether an individual request is compromised or not. Instead, they must be identified from the pattern of the incoming requests while considering various factors.

Continuous monitoring is a popular tool used for detecting DDoS attacks, since it can be used to automatically deploy safety measures and alert the IT team when there is an anomaly in the requests. While it may speed up the mitigation process, it also requires more manual labor and may be ineffective if monitored too strictly[3]. However, since the patterns of each client can be analyzed to know whether they are malicious, neural networks, due to their pattern recognition ability, pose as a potent tool for detecting such attacks because they can detect patterns.

This paper seeks to investigate further the extent to which a trained feed forward neural network can detect an HTTP flood DDoS, specifically upon receiving live data when used as a proxy.

### 1.1 Worthiness

The vulnerability and impact of a DDoS increases as the number of web applications increases hourly. From small businesses that have just launched their application to large-scale companies, this research could be fruitful. By correctly classifying the type of a request and

[2] "What Is a Distributed Denial-of-Service (DDoS) Attack?" Cloudflare, 2023, www.cloudflare.com/learning/ddos/what-is-a-ddos-attack/. Accessed 11 July 2022.
[3] Hacken. "How to Detect a DDoS Attack? - 5 Red Flags - Hacken." Hacken, 8 Aug. 2022, hacken.io/discover/how-to-detect-a-ddos-attack/. Accessed 13 Aug. 2022.

recognizing a pattern, a neural network can be used as a proxy to intercept every request, verify it, and only forward legitimate ones. This could save firms thousands of dollars and small businesses from such attacks, too – those that can cause them to shut down. Moreover, the data and analysis from this research could be extended to be applied to different DDoS attacks – SYN floods, UDP floods, and ICMP floods.

This investigation also aims to record and compare the performance and timings of the neural network against the time taken to fulfill the request to understand the impact on the neural network on the server.

### 1.2 Scope

DDoS datasets, particularly for HTTP floods, usually contain sensitive data about the users and the server in use – which are often part of the requests – and hence are often found on malicious platforms that have to be accessed using the TOR network. This can be unethical to use for such an investigation. Thus, to conduct this experiment, a dataset will be generated by simulating a DDoS attack.

DDoS attacks include thousands of devices, so, they are very costly and complicated to simulate and require extensive hardware access. Thus, an experiment simulating a small-scale DDoS will be conducted to answer the posed research question and achieve the paper's aim. Firstly, large data sets will be generated consisting of DDoS emulations, including data about the HTTP requests (user IP, endpoint, time taken, packet size, etc. After the data sets are generated, an ANN will be trained to recognize DDoS patterns.

Since the performance and results of a neural network are heavily dependent on its hyperparameters used, the hyperparameters used in this investigation will be selected by analyzing and understanding the preprocessed data. Moreover, to further measure the extent to

which a neural network can detect the attacks, the model will be trained with a different number of hidden layers. The results of these different models with then be compared and evaluated.

## 2. Background Information

### 2.1 Machine Learning and its Types

Pattern recognition is based on machine learning, which is the study of programming computers to do tasks they haven't been directly programmed to do[4]. These networks are then trained to recognize, analyze, and learn from data and perform complex tasks (i.e., identifying patterns and predicting events). They can be trained via multiple strategies – unsupervised, supervised, reinforcement, and more – and each has advantages and disadvantages. Supervised learning allows a network to identify and create mappings between the features and data classification[5].

There are several types of machine learning algorithms as well. Still, pattern recognition requires classification, which involves the computer learning the relations between the data and their labels. So, for example, a machine could be given a collection of texts grouped by their language, and the classification network would analyze the features of those texts to attempt to relate specific visual characteristics to certain languages. This is what is referred to as "training." If successful, the network would eventually be able to accurately predict the language of texts it has not seen before by identifying relations between the features of the text to a language, which it made during training. Pattern recognition algorithms are trained similarly with thousands of labeled entries. Since the main goal of this investigation is to classify network requests, and because labeled data is present, supervised training will be used for this research.

[4] Ng, Andrew. "Supervised Machine Learning: Regression and Classification." Coursera, 2022, www.coursera.org/learn/machine-learning. Accessed 13 Aug. 2022.
[5] Salian, Isha. "NVIDIA Blog: Supervised vs. Unsupervised Learning." The Official NVIDIA Blog, 2 Aug. 2018, blogs.nvidia.com/blog/2018/08/02/supervised-unsupervised-learning/. Accessed 28 Aug. 2022.

**2.2 Chosen Machine Learning Model**

DDoS attacks can be detected using many different types of machine learning models. Feed forward neural networks, support vector machines (SVM), and random forest are some of the most common and popular methods for detecting them[6].

Feed forward networks are generally applicable to most sorts of pattern detection scenarios. Since I have briefly worked with these networks, I will be further studying them and using a feed forward network for the sake of this investigation. The inner workings of a feed forward network are discussed below.

**2.3 Feed Forward Neural Networks**

Feed forward neural networks consist of layers of neurons: the input layer, the output layer, and the hidden layers. Each layer identifies certain patterns within the data. Inspired from how brains function, neural networks are made up of neurons. The purpose of an artificial network is to receive inputs, perform calculations, and give an output – and pass it onto the next layer of neurons. The input is represented as the first layer of neurons, and they continue to activate consecutive layers until the output layer is activated, which represents the output itself[7].

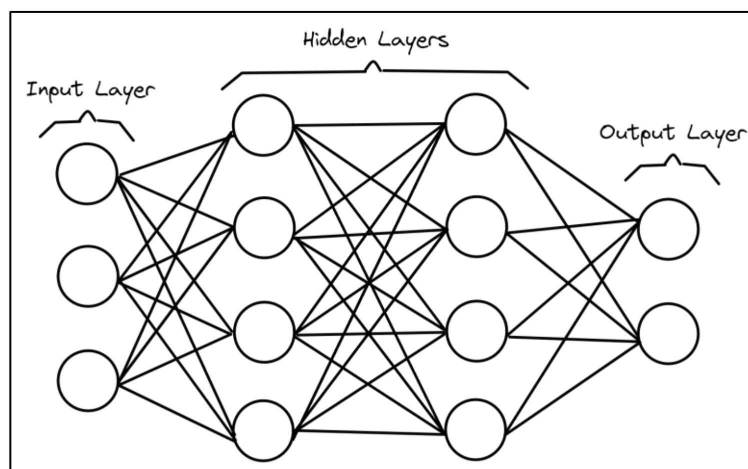


*Figure 1: Structure of a feed forward neural network (self-made)*

[6] Aytac, Tugba, et al. "Detection DDOS Attacks Using Machine Learning Methods." Electrica, vol. 20, no. 2, 15 June 2020, pp. 159–167, https://doi.org/10.5152/electrica.2020.20049. Accessed 7 Sept. 2022.
[7] Sanderson, Grant. "Neural Networks - YouTube." YouTube, 2019, www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi. Accessed 9 Oct. 2022.

To receive and pass data, neurons rely on links, referred to as *weights*. The value of the neuron, which it passes forward, is called its *activation*[8]. All neurons in layers after the input layer rely on the previous layer for input, which is simply the activation of all the neurons in the layer before multiplied by the weights of the neuron with each of the neurons. Moreover, each neuron also has a *bias*, which indicates how much the neuron is activated in general (a negative bias means the neuron usually has low activation, whereas a positive bias means the neuron usually has a high activation). The activation for the *nth* neuron in the layer *j* with *N* neurons, which is preceded by the layer *i* is given by:

$$A_{j_n} = \sigma\left(\sum_{p=1}^{N}\left(w_{j_n}^{j_p} \times A_{i_p}\right) + b_{j_n}\right)$$

*Equation 1: Equation for the activation of a neuron in layer i, given consecutive layers j and I*[9]

Here, $A_{i_n}$ represents the activation of the *nth* neuron in layer *i*, $\sum_{p=1}^{N}\left(w_{i_n}^{j_p} \times A_{j_p}\right)$ represents the sum of the products of the weights and activations of the neurons in the layer before, and $b_{j_n}$ is the bias of the neuron. These biases and weights are assigned randomly using a *seed* when a network is created and are changed as the model trains itself. $\sigma$ represents the *activation function*, which is a mathematical function that gives an output from an input and is used for performance and efficiency reasons[10].

The "output", or result, or the network are the neuron activations in the output layer. Each neuron represents a unique answer, and its activation represents the probability that it is correct[11]. The answer with the highest probability is assumed to be the correct one. While training, the network compares the output with the real answer, using which a "cost" is

[8] Sanderson, Grant. "Neural Networks - YouTube." YouTube, 2019, www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi. Accessed 9 Oct. 2022.
[9] IBID
[10] Sanderson, Grant. "Neural Networks - YouTube." YouTube, 2019, www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi. Accessed 9 Oct. 2022.
[11] IBID

determined. The cost represents the error of the network (the lower, the better) and can be calculated using a range of different methods, such as mean square error, mean absolute error, and root mean square error[12]. Moreover, because the network is essentially a mathematical function, the cost can also be represented by a function. The error function can then be minimized to increase the accuracy of the network, effectively training the network. This is done by calculating the gradient of the cost function at the current weights and biases and descending downwards (to minimize it) by tweaking them. As the cost is further minimized during training, the network becomes better at predicting the answer, given a set of features[13].

Hence, neural networks can be looked at as a function which gives a certain number of outputs given specific inputs and parameters[14].

## 3. Experiment Methodology

Primary experimental data sets are the main sources of data for this paper. An experimental methodology was chosen because there was limited secondary data to conclude this paper, and this method provides freedom to train and test the model with primary data.

### 3.1 Generation of Data Sets

The data sets have been generated by "simulating" a DDoS. This was done by writing two programs – a client and a server – responsible for mocking a server and a user (compromised or genuine) (refer to appendix 1 for code). The clients were deployed in 35 different virtual private servers over ten different geographical locations worldwide, as seen in figure 2 below. They were deployed using Linode and DigitalOcean (computing providers).

[12] Saini, Hrithik. "7 Types of Cost Functions in Machine Learning | Analytics Steps." Www.analyticssteps.com, www.analyticssteps.com/blogs/7-types-cost-functions-machine-learning. Accessed 18 Aug. 2022.
[13] IBID
[14] IBID

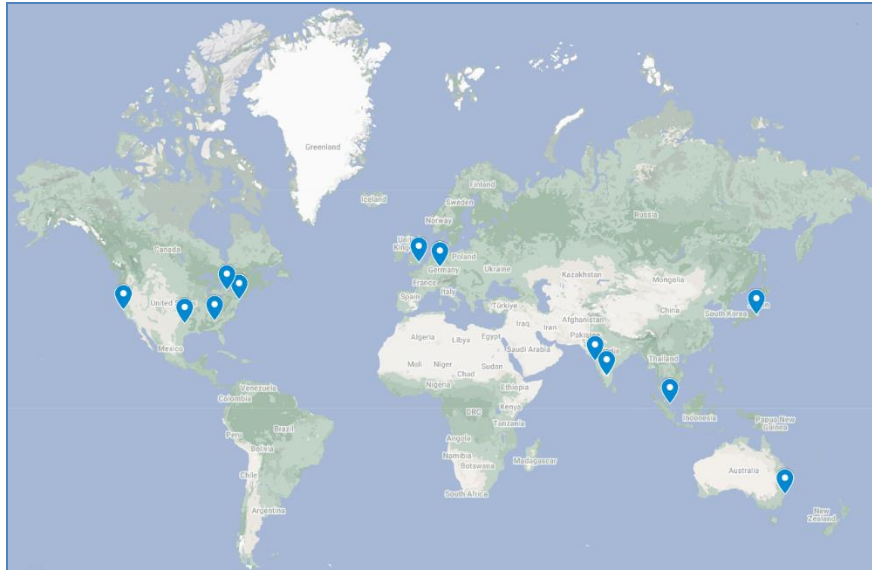

*Figure 2: Geographical locations of clients used in experiment*[15]

The client is structured as shown in figure 3.1 below, exposing two endpoints – compromised and user – giving the researcher control over the client. The same client structure is replicated on all 35 VPSs worldwide to act as a distributed system. While there were multiple clients, the server was kept constant.

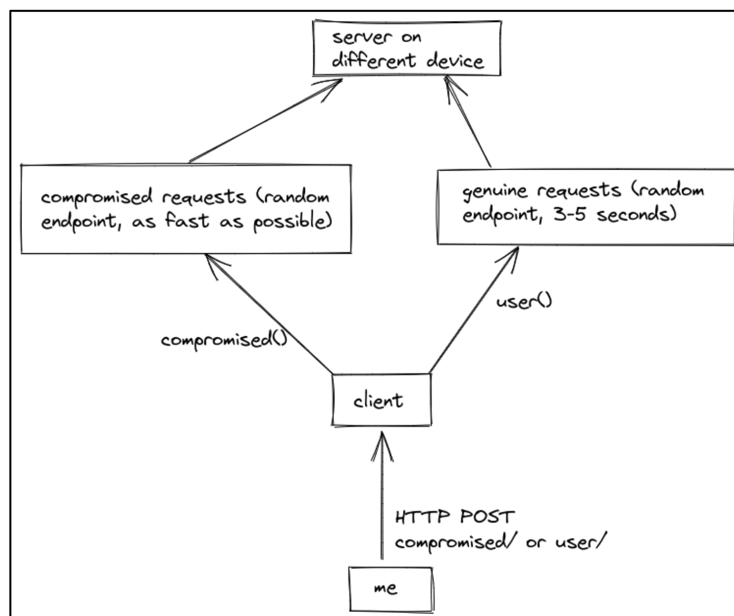


*Figure 3.1: Client-side structure and logic*

[15] Self-made using Google My Maps, mymaps.google.com. Accessed 19 Sept. 2022.

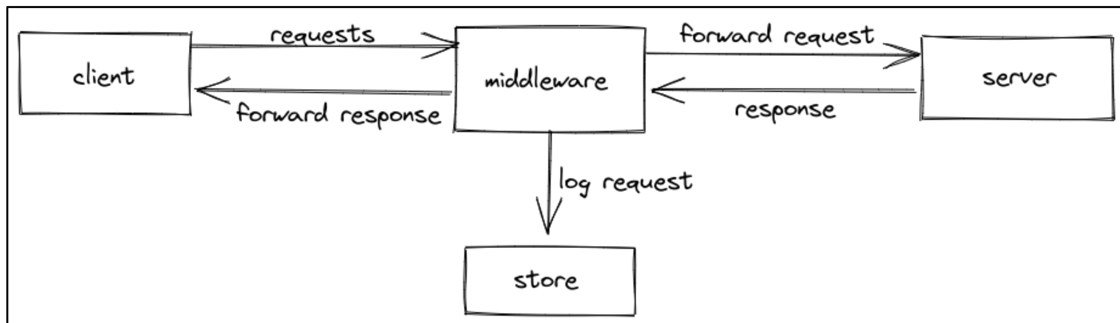


*Figure 3.2: Server-side structure*[16]

The server is designed to keep the logical layer oblivious to the logging process, allowing the codebase to remain maintainable and helping maintain strict conditions for the ANN to be trained with (i.e., acting as middleware or firewall).

The method below was used to conduct the simulation and generate the datasets:

1) 30 out of the 35 devices initially act as “real” clients, making requests to random endpoints every 3-5 seconds. This means five clients are to act *only* as compromised devices, not making real requests.
2) 5-6 minutes later (randomly chosen), 15 out of the 35 (including the five inactive ones mentioned in point 1) devices start simulating “compromised” clients by making requests to random endpoints to their maximum load. This emulates the situation where the attacker starts the DDoS, and the compromised devices flood the server with requests. During this, the other 20 devices continue to act as “real” users, as would in the real world.
3) After 5-7 minutes (randomly chosen), the malicious clients stop the DDoS and are terminated by terminating the ongoing request from the researcher’s machine to the client on its compromised endpoint.
4) The rest of the clients continue to make requests as normal users

[16] Figure 2.1 and 2.2 are self-made using excalidraw. “Excalidraw.” Excalidraw, excalidraw.com/.

5) The data logged earlier is then transferred to the researcher's local machine in a CSV format by using the SCP protocol
6) The process is repeated four more times, generating a total of 5 large datasets, each emulating a unique DDoS request pattern

Since each client knows if it is acting as a malicious or real user, the request is labelled as compromised or not when it is made itself.

Each experiment iteration was conducted automatically, using python 3.9, which controlled the clients and their states (refer to appendix 1 for code). Table 1 showcases a few data points of the data collected as a sample.

| ip | endpoint | headers | time | packet_size | time_taken | compromised |
|---|---|---|---|---|---|---|
| 139.144.44.193 | / | Host: 165.232.182.157 User-Agent: python-requests/2.28.1 Accept-Encoding: gzip, deflate Accept: */* Connection: keep-alive Content-Type: application/json | 1674598718 | 100 | 0.057705225 | FALSE |
| 170.187.139.144 | /endpoint-2 | Host: 165.232.182.157 User-Agent: python-requests/2.28.1 Accept-Encoding: gzip, deflate Accept: */* Connection: keep-alive Content-Type: application/json | 1674598721 | 359 | 0.051176134 | FALSE |
| 157.245.104.1 | /endpoint-4 | Host: 165.232.182.157 User-Agent: python-requests/2.28.1 Accept-Encoding: gzip, deflate Accept: */* Connection: keep-alive Content-Type: application/json | 1674599070 | 5347 | 0.161323308 | TRUE |

*Table 1: Sample raw data. A screenshot of the entire data can be seen in Appendix 3*[17]

[17] The descriptions and units for the fields can be seen in Table 2.1

The packet size and time taken of the requests can be plotted against the time they were received to visualize the DDoS simulation. Since there are a very large number of requests, a random sample of 0.2% of the requests is plotted below.

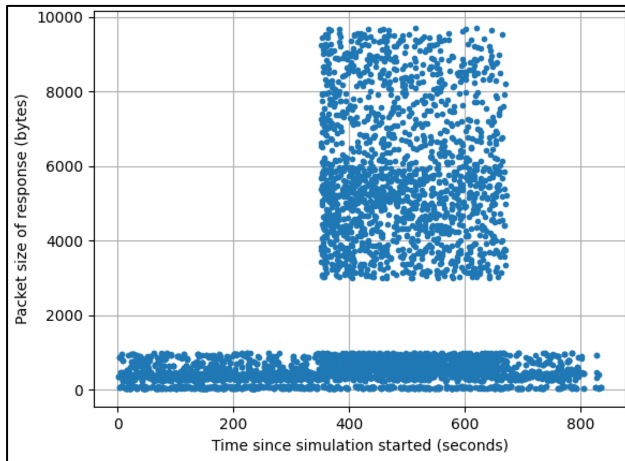


*Figure 4.1: Packet Size of Each Request*

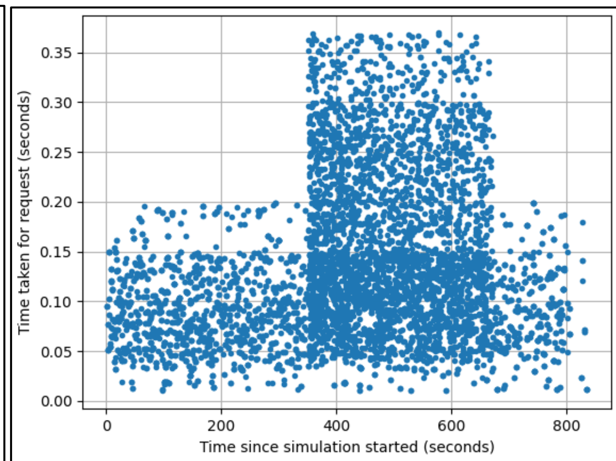


*Figure 4.2: Time Taken for Each Request*[18]

The compromised requests have a higher time taken per request and packet size. When looking at the pattern as a whole, an attack can be recognized. However, it is much harder to recognize whether an individual request is compromised just by looking at the information above.

### 3.2 Processing the datasets for use

The datasets generated must be preprocessed before they can be fed into the neural network as parameters. Feature extraction is an essential process, since it allows more specific information from the data to be found, which allows the network to find more intricate patterns. Moreover, by eliminating variables that are not needed, the number of features is greatly reduced, which reduces the time for the network to learn and generalize[19].

Features such as average packet size and request time for the past 5 requests and requests in the past 5 seconds by each client were calculated for each request. Moreover, data such as the IP address and headers were removed as they do not contain information that can be used to

[18] The diagrams are self-made using matplotlib and python 3.9

[19] Chatterjee, Sampriti. "What Is Feature Extraction? Feature Extraction in Image Processing." Great Learning Blog: Free Resources What Matters to Shape Your Career!, 29 Oct. 2021, www.mygreatlearning.com/blog/feature-extraction-in-image-processing. Accessed 13 Oct. 2022.

recognize a DDoS. The five processed data sets were merged and then shuffled. Lastly, the datasets were split into training and evaluation data, with a 4:1 split ratio[20] – 80% of the data was used for training, while 20% was used for evaluation). The preprocessing was done using python 3.9 and the scikit-learn, pandas, and numpy libraries, as seen in the figure below.

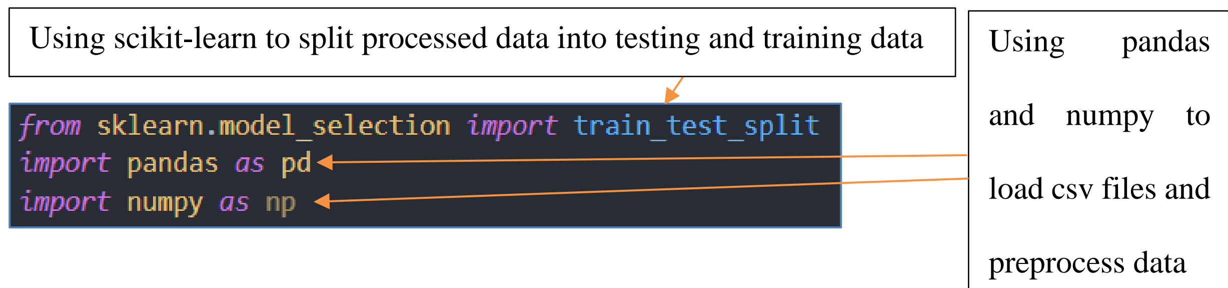


*Figure 5: Python library imports used for preprocessing*

Table 2.1 showcases the fields of the raw data, which was collected during the simulation, and Table 2.2 showcases the fields of the processed data, which will be used as input for the neural network.

| Field | Description |
|---|---|
| ip | The IP address of the client which made the request |
| endpoint | The HTTP endpoint of the request |
| headers | The HTTP headers of the request |
| time | The time at which the request was received by the server, in seconds since epoch |
| packet_size | The size of the response packet sent by the server, in bytes |
| time_taken | The time taken for the server to fulfill the request |
| compromised | The nature of the request: whether it was compromised or not |

*Table 2.1: Fields for raw data*

[20] Tokuç, A. Aylin. "Splitting a Dataset into Train and Test Sets | Baeldung on Computer Science." Www.baeldung.com, 14 Jan. 2021, www.baeldung.com/cs/train-test-datasets-ratio. Accessed 21 Oct. 2022.

| **Field** | **Description** |
|---|---|
| packet_size | The size of the response packet sent by the server, in bytes |
| num_past_requests | The number of past requests received from the same client in the past 5 seconds |
| average_packet_size | The average packet size of the responses sent to the same client in the past 5 requests |
| time_taken | The time taken for the server to fulfill the request in seconds |
| average_time_taken | The average time taken for the server to fulfil the request of the same client in the past 5 requests, in seconds |
| compromised | The nature of the request: whether it was compromised or not |

*Table 2.2: Fields for processed data*

Since the processed data has aggregative and accumulative fields, each request contains information about some previous requests. Hence, the processed data is effectively able to describe the raw data. Furthermore, more relationships can be found within the data, such as the average time taken and packet size of each request.

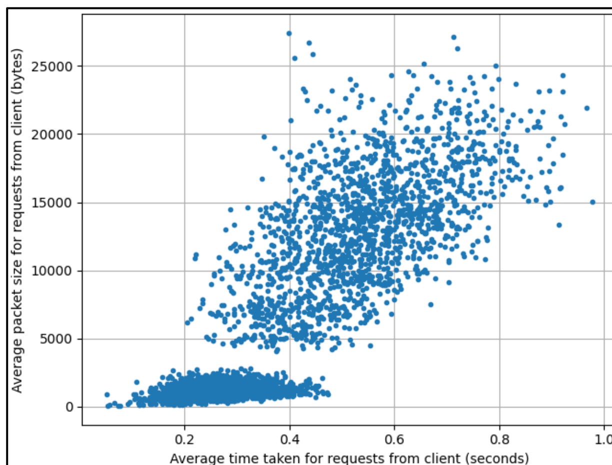


*Figure 6.1: Average packet size vs average time taken*

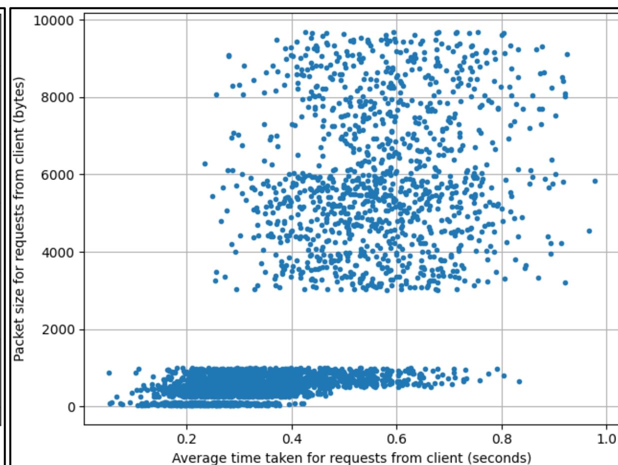


*Figure 6.2: Packet size vs average time taken*[21]

A relationship between the time taken and packet size for requests is visually evident, as seen in figure 6.1 and 6.2. The compromised and genuine requests form "clusters", which can also be used by unsupervised models[22]. Overall, the processed data is effectively able to resemble

[21] The diagrams are self-made using matplotlib and python 3.9
[22] Mishra, Sanatan. "Unsupervised Learning and Data Clustering." Medium, Towards Data Science, 19 May 2017, towardsdatascience.com/unsupervised-learning-and-data-clustering-eeecb78b422a. Accessed 14 Oct. 2022.

and enhance the features of the raw data, which would benefit the network in effectively classifying the requests.

### 3.3 Dependent Variables

The variable being measured in this paper is the classification success rate of the ANN – the percentage of requests it correctly classifies as malicious or real – and the time taken to classify the request.

#### Time

The time measured in this case is the time taken for the network to classify a request instead of the time taken to train the model. Python's `time.time()` method was used to calculate the time before and after the call to the network.

#### Accuracy

The accuracy measured was the accuracy of the network in classifying requests correctly with evaluation data sets. The accuracy of the network is the number of correct classifications divided by the total number of classifications.

### 3.4 Programming of the Feed Forward Neural Network

The feed forward network was programmed using python and TensorFlow. It took 5 inputs, as seen in Table 2, omitting the label, and had one output neuron, whether the request was compromised. The structure of the network is illustrated below.

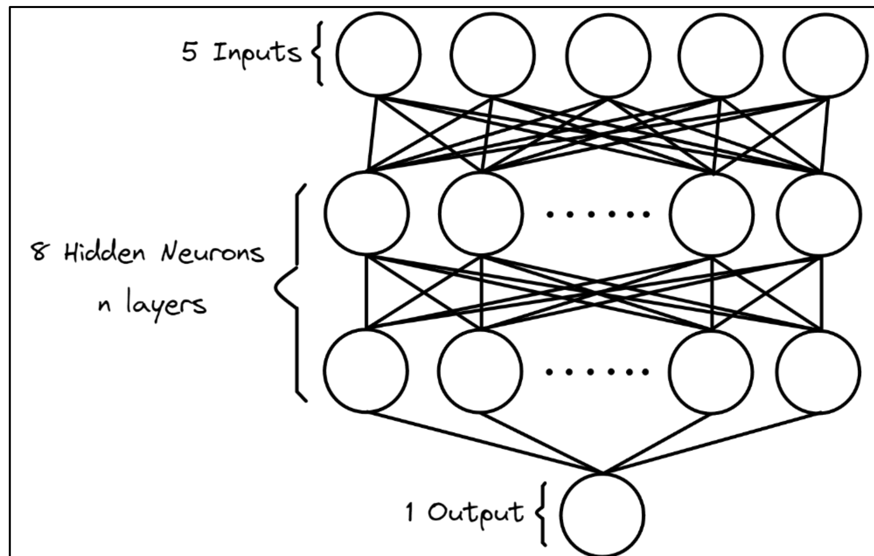


*Figure 7: The structure of the programmed feed forward neural network, where n is the number of layers*[23]

The dimensions of the input layer are equal to the features of the dataset. As discussed before, the model would be trained with a different number of layers. The dense input layer will be processed by these hidden layers, and the output layer flattens the activation of the hidden layers into a single neuron.

The models were trained using TensorFlow's `Sequential.train()` function. Since the data has a large number of features, the model becomes more prone to descending to a local minimum instead of the global minimum. Hence, the model was trained with a low batch size of 8, which allows it to generalize the pattern better and have higher accuracy[24].

### 3.5 Experimental Procedure

4 networks were configured with an input layer with 5 neurons, and an output layer with 1 neuron. Each network was programmed to have a different number of hidden layers – 1, 2, 3, and 4 respectively. They were trained upon the same training data, and their performance and accuracy were then recorded against the testing data.

[23] Self-made using excalidraw. "Excalidraw." Excalidraw, excalidraw.com/.
[24] Keskar, Nitish Shirish, et al. "On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima." ArXiv:1609.04836 [Cs, Math], 9 Feb. 2017, arxiv.org/abs/1609.04836.

Additionally, neural networks are initialized with randomized weights and biases, so their training results may have outliers and inconsistencies. While a low batch size was already chosen to minimize this error, 2 models were also programmed for each configuration to ensure that the patterns observed were not pertaining to the randomized initial state. The models were completely identical, except their seed, which were “12345” and “98765”, and were nicknamed “Model 1” and “Model 2”. Both seeds were chosen arbitrarily.

### 3.6 Hypothesis

I hypothesize that because neural networks specialize in detecting patterns the network will be able to successfully detect a DDoS attack with high accuracy. Moreover, the accuracy would increase as the number of hidden layers are increased, as the network would be able to find more specific patterns.

## 4. The Experimental Results

### 4.1 Accuracy over epochs

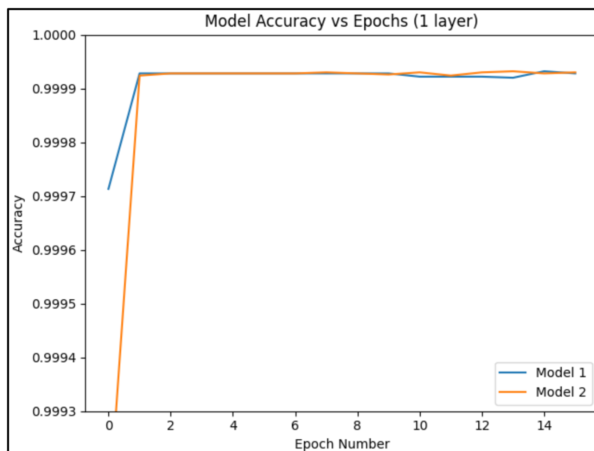

*Figure 8.1: Accuracy vs Epochs with 1 layer*

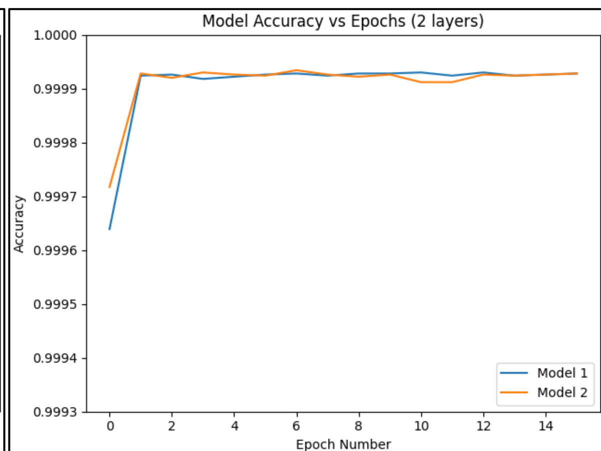

*Figure 8.2: Accuracy vs Epochs with 2 layers*

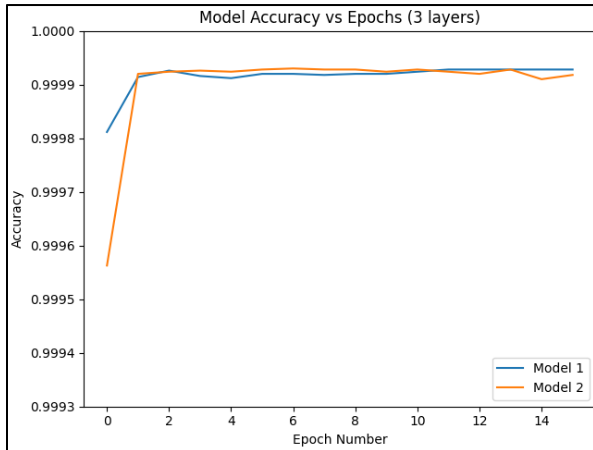


*Figure 8.3: Accuracy vs Epochs with 3 layers*

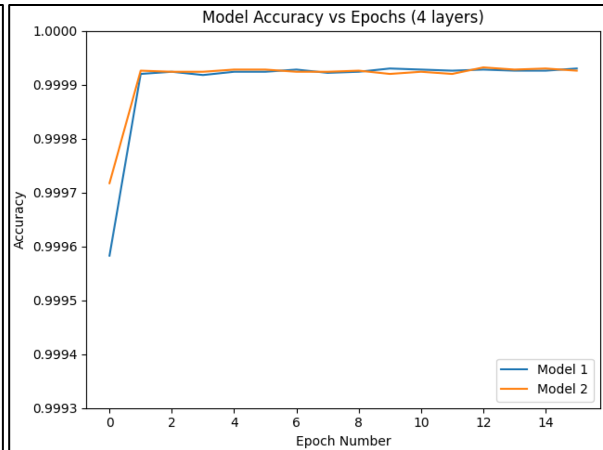


*Figure 8.4: Accuracy vs Epochs with 4 layers*[25]

The accuracy measured while training was the accuracy against the training data itself. The models were trained for 16 epochs, however as seen in the figures above, their accuracy remained similar after the first two epochs.

## 4.2 Accuracy on testing data

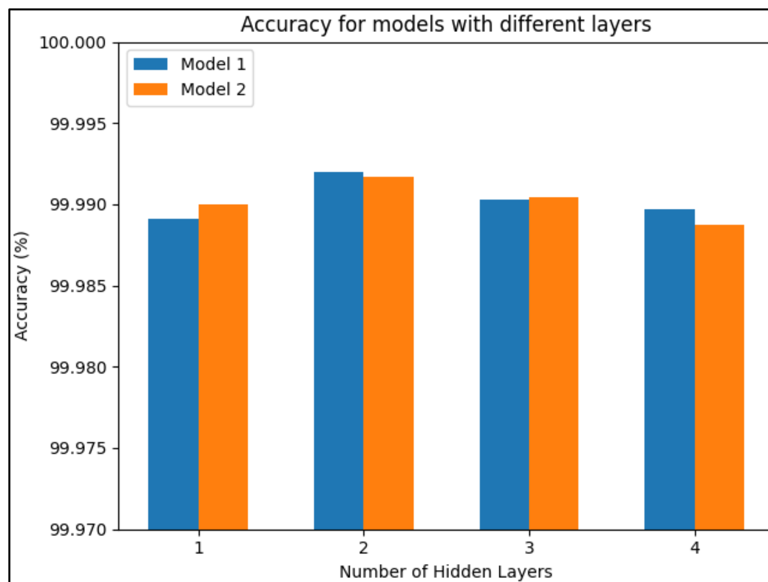


*Figure 9: Accuracy of models on testing data for different layered networks*[26]

The networks had astonishingly high accuracies, of over 99.98%. Both, model 1 and model 2 had a higher accuracy with two hidden layers, and the accuracy gradually reduced as more

[25] Figures 8.1, 8.2, 8.3, and 8.4 were self-made using python and matplotlib
[26] Self-made using python and matplotlib

layers were introduced. This is against my hypothesis, as I predicted that the accuracy would continue to rise.

### 4.3 Time taken for classification

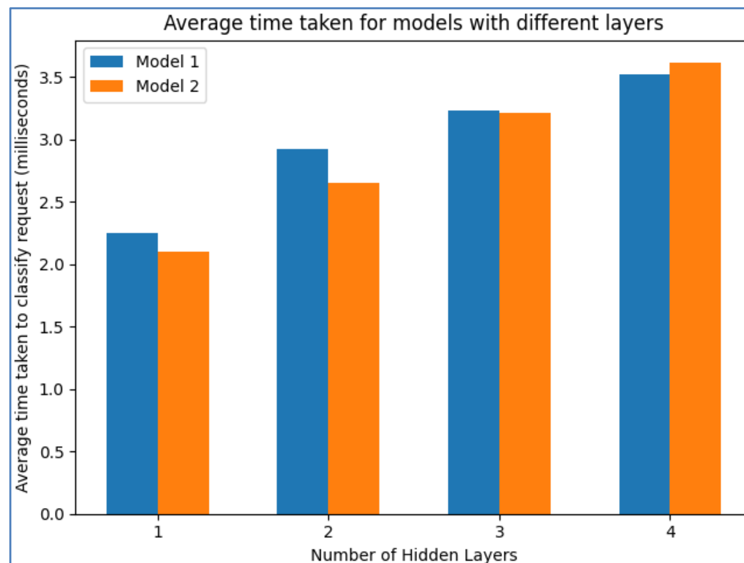


*Figure 10: Average time taken to classify request vs number of hidden layers*[27]

The time taken to classify a request increased for every hidden layer added. The networks took from 2 to 3.5 milliseconds to classify a request, which accounts for the TensorFlow function call overhead.

## 5. Analysis

### 5.1 Analyzing Accuracy

As seen by the 99.99% accuracy with 2 hidden layers, the neural network is highly successful in classifying requests, and hence, can help solve the problem of mitigating HTTP flood DDoS attacks. While this makes it a potent tool in mitigating DDoS attacks, its high accuracy can be accredited to factors that may make the network less effective in the real world. The dataset created was only a simulation of a real DDoS attack. This meant that the data had a significantly lower number of individual clients. Hence, the patterns of the requests were also inherently

[27] Self-made using python and matplotlib

limited. This may cause the data collected to be too narrow, which as a result can cause the network to adjust only for the given dataset, and hence have high accuracy.

### 5.2 Analyzing Performance

In terms of performance, as discussed before, the network is highly efficient and performant. A request is usually classified in a few milliseconds, which is extremely low when compared to the average time taken by an HTTP request: 500ms[28]. Hence, the network would have a negligible impact when implemented as a middleware for servers.

### 5.3 Making sense of the drop in accuracy

As was seen in figure 9 above, the network's accuracy reduced when it was trained upon 3 or 4 hidden layers, which was against my proposed hypothesis. The larger the number of layers, the larger the number of trainable parameters for a network that influence its output. This means that Model 1 with 2 layers is not the same mathematical function as Model 1 with 4 layers. Although the function with 4 layers is more complicated, it does not guarantee better results.

Each layer picks up on the layer before, and finds patterns in that, meaning, as more layers are added, more and more details are picked up which influence the output. However, when too many layers are added, the network overanalyzes patterns and starts considering "noise" – meaningless data[29]. This leads to the model overfitting itself on the training data, and in turn, leading to higher inaccuracy[30].

[28] Saunders, Orde. "How Long Does an HTTP Request Take? | Blog | Decade City." Decadecity.net, 12 Mar. 2014, decadecity.net/blog/2012/09/15/how-long-does-an-http-request-take. Accessed 28 Nov. 2022.
[29] "What Is Noise in ML." Iguazio, www.iguazio.com/glossary/noise-in-ml. Accessed 17 Dec. 2022.
[30] "What Is Overfitting? - Overfitting - AWS." Amazon Web Services, Inc., aws.amazon.com/what-is/overfitting/. Accessed 28 Dec. 2022.

### 5.4 Computational Costs

However, there are computational costs that come along the implementation of a network. Firstly, a machine with extensive resources, especially RAM and CPU, is required to train a neural network. With larger datasets, the training process can take days, if not weeks, and hence, the availability of system resources is essential. Furthermore, since the batch size is small, the training time for the model is further increased.

Moreover, when using the network as a middleware, there are further implications for the resources. As the network must be loaded into RAM, and the CPU would be used while classifying the requests, the network might utilize resources that the server could otherwise use.

## 6. Conclusion

This experiment sought to identify the effectiveness and extent to which a feed forward neural network can identify and mitigate HTTP flood DDoS attacks. The feed forward network setup by me is able to successfully identify DDoS attacks to mitigate them, while being highly performant and having a minimal impact on the request timings. This is in line with and validates my hypothesis.

## 6. Further Research Opportunities

### 6.1 Investigating a change in the preprocessing of the data

The feature extraction for this experiment contained several average and aggregate values. For example, the average packet size was calculated from the past 5 requests. It is intriguing to find the impact on the accuracy of the network when the preprocessing of the data is changed, such as changing the calculations to account for 10 requests, and when more features are added.

### 6.2 Utilizing different machine learning models

As found after preprocessing the data, visual patterns were visible, which can be utilized by different models. Hence, different models can also be used to detect DDoS attacks, and since they differ in the way they recognize patterns, it is compelling to investigate how well other machine learning models can detect DDoS attacks. Moreover, as mentioned before, other models, such as SVM and random forest could also be explored.

### 6.3 Extending to different DDoS attacks

This investigation only examined how HTTP flood attacks can be mitigated and did not look at different types of DDoS attacks such as SYN and UDP floods. These attacks are executed on different network layers, and hence have different input parameters. Hence, it would be interesting to see how neural networks can be used to detect these types of attacks.

## Works Cited

Aytac, Tugba, et al. "Detection DDOS Attacks Using Machine Learning Methods." *Electrica*, vol. 20, no. 2, 15 June 2020, pp. 159–167, https://doi.org/10.5152/electrica.2020.20049. Accessed 7 Sept. 2022.

Chatterjee, Sampriti. "What Is Feature Extraction? Feature Extraction in Image Processing." *Great Learning Blog: Free Resources What Matters to Shape Your Career!*, 29 Oct. 2021, www.mygreatlearning.com/blog/feature-extraction-in-image-processing. Accessed 13 Oct. 2022.

"Excalidraw." *Excalidraw*, excalidraw.com/.

Hacken. "How to Detect a DDoS Attack? - 5 Red Flags - Hacken." *Hacken*, 8 Aug. 2022, hacken.io/discover/how-to-detect-a-ddos-attack/. Accessed 13 Aug. 2022.

Keskar, Nitish Shirish, et al. "On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima." *ArXiv:1609.04836 [Cs, Math]*, 9 Feb. 2017, arxiv.org/abs/1609.04836.

Mishra, Sanatan. "Unsupervised Learning and Data Clustering." *Medium*, Towards Data Science, 19 May 2017, towardsdatascience.com/unsupervised-learning-and-data-clustering-eeecb78b422a. Accessed 14 Oct. 2022.

Ng, Andrew. "Supervised Machine Learning: Regression and Classification." *Coursera*, 2022, www.coursera.org/learn/machine-learning. Accessed 13 Aug. 2022.

Saini, Hrithik. "7 Types of Cost Functions in Machine Learning | Analytics Steps." *Www.analyticssteps.com*, www.analyticssteps.com/blogs/7-types-cost-functions-machine-learning. Accessed 18 Aug. 2022.

Salian, Isha. "NVIDIA Blog: Supervised vs. Unsupervised Learning." *The Official NVIDIA Blog*, 2 Aug. 2018, blogs.nvidia.com/blog/2018/08/02/supervised-unsupervised-learning/. Accessed 28 Aug. 2022.

Sanderson, Grant. “Neural Networks - YouTube.” *YouTube*, 2019, www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi. Accessed 9 Oct. 2022.

Saunders, Orde. “How Long Does an HTTP Request Take? | Blog | Decade City.” *Decadecity.net*, 12 Mar. 2014, decadecity.net/blog/2012/09/15/how-long-does-an-http-request-take. Accessed 28 Nov. 2022.

Tokuç, A. Aylin. “Splitting a Dataset into Train and Test Sets | Baeldung on Computer Science.” *Www.baeldung.com*, 14 Jan. 2021, www.baeldung.com/cs/train-test-datasets-ratio. Accessed 21 Oct. 2022.

“What Is a Distributed Denial-of-Service (DDoS) Attack?” *Cloudflare*, 2023, www.cloudflare.com/learning/ddos/what-is-a-ddos-attack/. Accessed 11 July 2022.

“What Is Noise in ML.” *Iguazio*, www.iguazio.com/glossary/noise-in-ml. Accessed 17 Dec. 2022.

“What Is Overfitting? - Overfitting - AWS.” *Amazon Web Services, Inc.*, aws.amazon.com/what-is/overfitting/. Accessed 28 Dec. 2022.

*Google My Maps*, mymaps.google.com. Accessed 19 Sept. 2022.

# Appendix

### 1: Code for simulation of DDoS

The following code was used to provision multiple virtual private servers around the globe, deploy the client code on them, and then perform the DDoS simulation. It also has the code for the host server, which received the requests and stored them in files.

Client that acts as a malicious or compromised user:

```
import threading
from dotenv import load_dotenv
load_dotenv()

import os
import random
import time
from time import sleep
from flask import Flask, request
import requests

app = Flask(__name__)

user_endpoints = ['/', '/endpoint-1', '/endpoint-2', '/endpoint-3']
malicious_endpoints = ['/endpoint-3', '/endpoint-4', '/endpoint-5']

def get_url(endpoint):
    return f"http://{os.getenv('SERVER_IP_ADDRESS')}{endpoint}"


def make_request_to_endpoint(endpoint, data):
    url = get_url(endpoint)
    print(f"Making request to {url}")
    requests.post(url, json=data)

user_thread: threading.Thread = None
user_running = False

malicious_thread: threading.Thread = None
malicious_running = False

def run_user(timetorun):
    global user_running
    start = time.time()
    while start + timetorun > time.time():
        if not user_running:
            break
```

```
        make_request_to_endpoint(random.choice(user_endpoints),
{"compromised": False})
        sleep(random.uniform(3, 5))
    user_running = False

def run_malicious(timetorun):
    global malicious_running
    start = time.time()
    while start + timetorun > time.time():
        if not malicious_running:
            break
        make_request_to_endpoint(random.choice(malicious_endpoints),
{"compromised": True})
    malicious_running = False

@app.route('/emulate-user')
def normal():
    global user_thread, user_running
    args = request.args
    timetorun = args.get("time", default=0, type=int)
    if timetorun <= 1:
        return "Invalid time"

    if user_running:
        return "User already running"

    user_running = True
    user_thread = threading.Thread(target=run_user, args=(timetorun,))
    user_thread.start()
    return 'Hello world!'

@app.route('/stop-user')
def stop_user():
    global user_thread, user_running
    if user_running:
        user_running = False
        user_thread.join()
        del user_thread
        user_thread = None
        return "User stopped"
    else:
        return "No user running"

@app.route('/emulate-malicious-user')
def malicous():
    global malicious_thread, malicious_running

    args = request.args
```

```
    timetorun = args.get("time", default=0, type=int)
    if timetorun <= 1:
        return "Invalid time"

    if malicious_running:
        return "malicious already running"

    malicious_running = True
    malicious_thread = threading.Thread(target=run_malicious,
args=(timetorun,))
    malicious_thread.start()

    return 'Hello world!'

@app.route('/stop-malicious-user')
def stop_malicious():
    global malicious_thread, malicious_running
    if malicious_running:
        malicious_running = False
        malicious_thread.join()
        del malicious_thread
        malicious_thread = None
        return "Malicious stopped"
    else:
        return "No malicious user running"


@app.route('/')
def index():
    return 'pong'

app.run(host="0.0.0.0", port=80)
```

Server that receives the requests from the clients and saves them:

```
from datetime import datetime, timedelta
import random
import time
from flask import Flask, request
import pandas as pd
import os
from queue import Queue

app = Flask(__name__)
data_queue = Queue()

columns = ['ip', 'endpoint', 'packet_size', 'headers', 'time', 'time_taken',
'compromised']
```

```
data = pd.DataFrame(columns=columns)
i = 0
filename = f"data_{i}.csv"
while os.path.exists(filename):
    i += 1
    filename = f"data_{i}.csv"

route_data = {
    "/": {
        "packet_size": [30, 100],
        "time_taken": [0.01, 0.1]
    },
    "/endpoint-1": {
        "packet_size": [250, 500],
        "time_taken": [0.03, 0.2]
    },
        "/endpoint-2": {
        "packet_size": [250, 500],
        "time_taken": [0.05, 0.15]
    },
    "/endpoint-3": {
        "packet_size": [500, 1000],
        "time_taken": [0.04, 0.15]
    },
    "/endpoint-4": {
        "packet_size": [3000, 6000],
        "time_taken": [0.1, 0.3]
    },
    "/endpoint-5": {
        "packet_size": [5000, 9700],
        "time_taken": [0.1, 0.37]
    },
}

@app.before_request
def log_request_info():
    if request.path not in route_data.keys():
        return

    req_data = route_data[request.path]

    compromised = request.get_json()["compromised"]


    packet_size = random.randint(req_data["packet_size"][0],
req_data["packet_size"][1])
    time_taken = random.uniform(req_data["time_taken"][0],
req_data["time_taken"][1])
```

```
    data_queue.put([request.remote_addr, request.path, packet_size,
request.headers, time.time(), time_taken, compromised])
    if len(data) % 100 == 0:
        data.to_csv(filename, index=False)

@app.route('/', methods=['POST'])
def hello_world():
    return 'Hello world!'

@app.route('/endpoint-1', methods=['POST'])
def hello_world_1():
    return 'Hello world!'

@app.route('/endpoint-2', methods=['POST'])
def hello_world_2():
    return 'Hello world!'

@app.route('/endpoint-3', methods=['POST'])
def hello_world_3():
    return 'Hello world!'

@app.route('/endpoint-4', methods=['POST'])
def hello_world_4():
    return 'Hello world!'

@app.route('/endpoint-5', methods=['POST'])
def hello_world_5():
    return 'Hello world!'


def save_logs():
    global data
    while True:
        end = datetime.now() + timedelta(seconds=1)
        res = []
        while datetime.now() < end:
            try:
                new_data = data_queue.get(timeout=0.1)
                res.append(new_data)
            except:
                continue

        res_df = pd.DataFrame.from_records(res, columns=columns)
        if len(res_df) > 0:
            data = pd.concat([data, res_df])
            data.to_csv(filename, index=False)
```

```
if __name__ == "__main__":
    import threading
    t = threading.Thread(target=save_logs, daemon=True)
    t.start()
    app.run(host='0.0.0.0', port=80)
```

<u>Code to run the simulation:</u>

```
from datetime import datetime
import random
from time import sleep
import dotenv
import linode as linode_api
import os
import pandas as pd
from sys import argv
import requests
import digital_ocean
from concurrent.futures import ThreadPoolExecutor
import concurrent.futures


csv_path = 'linodes_data.csv'

def debug(msg):
    print(f"{datetime.now().strftime('%H:%M:%S')} {msg}")

def delete():
    linodes = linode_api.get_linodes_raw()
    print(f"[INFO]: Deleting {len(linodes)} linodes")
    for linode in linodes:
        linode.delete()

    digital_ocean.delete_all()


def create():
    regions = linode_api.get_regions()
    print(f"[INFO]: Found {len(regions)} regions")

    linodes = linode_api.get_linodes()
    print(f"[INFO]: {len(linodes)} linodes are running")

    new_linodes = []
    if len(linodes) > 0:
        debug("Linodes already exist.")
        return
```

```
    for i in range(1):
        lin, pwd = linode_api.make_linode(regions[i % len(regions)], 1059469,
{"SERVER_IP_ADDRESS": os.getenv('SERVER_IP')})
        new_linodes.append([lin.id, pwd])

    linodes = linode_api.get_linodes()
    for i in range(len(new_linodes)):
        id = new_linodes[i][0]
        for linode in linodes:
            if linode['id'] == id:
                new_linodes[i].append(linode['ip'])
                break

    do_regions = digital_ocean.get_regions()


    def run_concurrent(func, args):
        e = ThreadPoolExecutor(max_workers=15)
        final_res = []
        futures = [e.submit(func, *arg) for arg in args]
        for future in concurrent.futures.as_completed(futures):
            try:
                res = future.result()
                if res:
                    final_res.append(res)
            except Exception as e:
                print(f"[ERROR] {e}")

        return final_res

    args = []

    for i in range(1):
        args.append((do_regions[i % len(do_regions)], f"Machine-{i}"))

    res = run_concurrent(digital_ocean.create, args)
    new_linodes += res



    df = pd.DataFrame(new_linodes, columns=['Id', 'Password', 'IP'])
    df.to_csv(csv_path, index=False)

def run():
    data = pd.read_csv(csv_path)
    seconds_before_attack = random.randint(5*60, 6*60) # 5-6 minutes
    attack_duration_seconds = random.randint(5*60, 7*60) # 5-7 minutes
```

```
    print(f"http://{data.iloc[0]['IP']}/emulate-
user?time={seconds_before_attack*1000}")

    requests.get(f"http://{data.iloc[0]['IP']}/emulate-
user?time={seconds_before_attack*1000}")
    requests.get(f"http://{data.iloc[1]['IP']}/emulate-malicious-user?time=5")


    if len(data) != 35:
        debug(f"[ERROR]: Expected 35 linodes, found {len(data)}")
        return


    real_clients = []
    compromised_clients = []

    for i in range(30):
        real_clients.append(data.iloc[i]['IP'])

    for i in range(30, 35):
        compromised_clients.append(data.iloc[i]['IP'])

    debug(f"[INFO]: Real clients loaded")
    debug(f"[INFO]: Starting real client simulation, will start attack in
{seconds_before_attack} seconds")

    for ip in real_clients:
        requests.get(f"http://{ip}/emulate-
user?time={seconds_before_attack*1000}")

    debug(f"[INFO]: User requests started, waiting {seconds_before_attack}
seconds before attack")

    sleep(seconds_before_attack)

    debug(f"[INFO]: Starting attack..")

    converted = []
    for i in range(9, -1, -1):
        ip = random.choice(real_clients)
        real_clients.remove(ip)
        converted.append(ip)

    for i in converted:
        requests.get(f"http://{i}/stop-user")
        requests.get(f"http://{ip}/stop-malicious-user")
        compromised_clients.append(i)
```

```
    for ip in compromised_clients:
        requests.get(f"http://{ip}/emulate-malicious-
user?time={attack_duration_seconds*1000}")

    debug(f"[INFO]: Attack started, waiting {attack_duration_seconds} seconds
before stopping attack")
    sleep(attack_duration_seconds)

    debug("[INFO]: Stopping attack..")

    for ip in compromised_clients:
        requests.get(f"http://{ip}/stop-malicious-user")
        requests.get(f"http://{i}/stop-user")

    debug(f"[INFO]: Attack stopped")


options = {
    "delete": delete,
    "create": create,
    "run": run
}


def main():
    dotenv.load_dotenv()
    if len(argv) < 2:
        print(f"No arguments given. Options are {', '.join(options.keys())}")
        return

    command = argv[1]
    if command not in options.keys():
        print(f"Invalid command. Options are {', '.join(options.keys())}")
        return

    options[command]()


if __name__ == "__main__":
    main()
```

<u>Bash Code for Deploying Client Code on Server:</u>

```
#!/bin/bash
if [ -f /etc/apt/sources.list ]; then
   apt update
   apt -y upgrade
   apt install -y python3-pip git
   apt-get install -y systemd
else
```

```
   echo "Your distribution is not supported by this StackScript"
   exit
fi

if [ ! -d /root/ddos ]; then
        git clone https://github.com/DhrumanGupta/mini-ddos-emulator
/root/ddos
else
        git --git-dir="/root/ddos/.git" pull origin master
fi

pip install -r /root/ddos/requirements.txt

echo "[Unit]
Description=Python DDoS Client
After=multi-user.target
[Service]
Type=simple
Restart=always
ExecStart=/usr/bin/python3 /root/ddos/client/main.py

[Install]
WantedBy=multi-user.target" > /etc/systemd/system/ddos-emulator.service


echo "SERVER_IP_ADDRESS=$SERVER_IP_ADDRESS" > /root/ddos/client/.env

systemctl daemon-reload

systemctl enable ddos-emulator.service
systemctl start ddos-emulator.service
```

**2: Code for pre-processing data, training network, and running tests**

The code below was used to pre-process the data collected before, and train the model. It was running TensorFlow in Visual Studio Code, on a Lenovo Legion laptop.

```
import matplotlib.pyplot as plt
import matplotlib.ticker as mticker
from sklearn.model_selection import train_test_split
import pandas as pd
import numpy as np
import json
import time
import os


EPOCHS = 16
```

```
def get_model(features, num_layers=2):
    import tensorflow as tf
    from tensorflow.python.keras.layers import Dense
    model = tf.keras.models.Sequential()

    normalizer = tf.keras.layers.Normalization(axis=-1)
    normalizer.adapt(features)

    model.add(normalizer)

    model.add(Dense(5, activation="relu", input_shape=(1,)))

    for i in range(num_layers):
        model.add(Dense(8, activation="relu"))

    model.add(Dense(1, activation="sigmoid"))

    model.compile(optimizer="adam", loss="binary_crossentropy",
metrics=["accuracy", tf.keras.metrics.MeanSquaredError()])
    return model


def get_processed_data(num):
    raw_data = pd.read_csv(f"data/data_{num}.csv")

    def process_ip(ip, raw_data):

        data = raw_data.loc[raw_data["ip"] == ip]

        final_df = pd.DataFrame(
            columns=[
                "time",
                "packet_size",
                "num_past_requests",
                "average_packet_size",
                "time_taken",
                "average_time_taken",
                "compromised",
            ]
        )
        past_requests_time = 5
        num_requests = 5

        for i, row in data.iterrows():
            num_past_requests = 0
            packet_requests = 1
            packet_size = row["packet_size"]
```

```
            time_taken = row["time_taken"]

            row_data = final_df.tail(num_requests - 1)
            packet_size += row_data["packet_size"].sum()
            time_taken += row_data["time_taken"].sum()

            num_past_requests = len(
                row_data.loc[final_df["time"] > row["time"] -
past_requests_time]
            )


            final_df.loc[i] = [
                row["time"],
                row["packet_size"],
                num_past_requests,
                packet_size / packet_requests,
                row["time_taken"],
                time_taken / packet_requests,
                row["compromised"],
            ]

        return final_df

    final_data = []
    for ip in raw_data["ip"].unique():
        final_data.append(process_ip(ip, raw_data))

    final_data = pd.concat(final_data)
    return final_data


def preprocess():
    for i in range(5):
        data: pd.DataFrame = get_processed_data(i)
        data.to_csv(f'data/processed_data_{i}.csv')

    res = []

    for i in range(4):
        data = pd.read_csv(f'data/processed_data_{i}.csv')

        res.append(data)

    res = pd.concat(res)
    res.to_csv('data/processed_data.csv')

    target = res.pop("compromised")
```

```
    shuffled = res.sample(frac=1)
    x_train, x_test, y_train, y_test = train_test_split(
        res, target, shuffle=True, test_size=0.2
    )

    x_train.to_csv("data/x_train.csv", index=False)
    x_test.to_csv("data/x_test.csv", index=False)
    y_train.to_csv("data/y_train.csv", index=False)
    y_test.to_csv("data/y_test.csv", index=False)


def train_model(num_layers):
    x_train = pd.read_csv("data/x_train.csv")
    y_train = pd.read_csv("data/y_train.csv")
    model = get_model(features=x_train, num_layers=num_layers)
    history = model.fit(x_train, y_train, epochs=EPOCHS, batch_size=8)
    with open(f"models/history_{num_layers}.json", "w") as f:
        json.dump(history.history, f)

    model.save(f"models/model_{num_layers}.tf")

def train(seed):
    import tensorflow as tf
    tf.keras.utils.set_random_seed(seed)
    for i in range(1, 5):
        print()
        print()
        print(f"Training model with {i} layers")
        train_model(i)
        print()
        print()

def evaluate_model(n=2):
    import tensorflow as tf
    res = {}
    test = pd.read_csv("data/x_test.csv")
    model = tf.keras.models.load_model(f"models/model_{n}.tf")


    target = pd.read_csv("data/y_test.csv")
    target = target["compromised"]
    total = len(test)
    total_time_taken = 0
    wrong = []
    print(total)
    for i in range(len(test)):
        start = time.time()
```

```
        prediction = model(test.iloc[i]).numpy()[0][0]
        end = time.time()
        total_time_taken += end-start
        compromised = prediction > 0.9
        if i % 500 == 0:
            print(f"[{i}/{total}] ({round((i/total)*100, 2)}%):
{total_time_taken/(i+1)}s")
        real = target.iloc[i]
        if compromised != real:
            wrong.append(prediction)

    accuracy = (1-len(wrong)/total) * 100
    print(f"Accuracy: {accuracy}%")

    res["accuracy"] = accuracy
    res["time"] = total_time_taken
    res["average_time"] = total_time_taken / total

    return res

def evaluate():
    res = []

    if os.path.exists("res.json"):
        with open("res.json", "r") as f:
            res = json.load(f)
    else:
        for i in range(1, 5):
            print()
            print()
            print()
            print()
            print(f"######### Model with {i} layers #########")
            print()
            with open(f"models/history_{i}.json", "r") as f:
                history = json.load(f)

            data = evaluate_model(i)

            res.append({
                "num_layers": i,
                "history": history["accuracy"],
                "time": data["time"],
                "average_time": data["average_time"],
                "accuracy": data["accuracy"]
            })
            print()
            print("########################################")
```

```
    with open('res.json', 'w') as f:
        json.dump(res, f)

plt.figure(constrained_layout=True)

bar_width = 0.3

res2 = []
with open('res_old.json', 'r') as f:
    res2 = json.load(f)

accuracy = [res[i]["accuracy"] for i in range(len(res))]
accuracy2 = [res2[i]["accuracy"] for i in range(len(res))]

ax = plt.gca()
ax.set_ylim([99.97, 100])
ax.xaxis.set_major_locator(mticker.MultipleLocator(1))

br1 = [i + bar_width/2 for i in range(len(res))]
br2 = [x + bar_width for x in br1]

plt.bar(br1, accuracy, width=bar_width, label='Model 1')
plt.bar(br2, accuracy2, width=bar_width, label='Model 2')

plt.xticks([r + bar_width for r in range(len(res))],
    [i + 1 for i in range(len(res))])

plt.title("Accuracy for models with different layers")
plt.legend([f"Model 1", f"Model 2"], loc="upper left")
plt.xlabel('Number of Hidden Layers')
plt.ylabel('Accuracy (%)')
plt.savefig(f'diagrams/accuracy.png')
# plt.show()

plt.cla()

bar_width = 0.3

time_taken = [res[i]["average_time"] * 1000 for i in range(len(res))]
time_taken2 = [res2[i]["average_time"] * 1000 for i in range(len(res))]

ax = plt.gca()
ax.xaxis.set_major_locator(mticker.MultipleLocator(1))

br1 = [i + bar_width/2 for i in range(len(res))]
br2 = [x + bar_width for x in br1]
```

```
    plt.bar(br1, time_taken, width=bar_width, label='Model 1')
    plt.bar(br2, time_taken2, width=bar_width, label='Model 2')

    plt.xticks([r + bar_width for r in range(len(res))],
        [i + 1 for i in range(len(res))])

    plt.title("Average time taken for models with different layers")
    plt.xlabel('Number of Hidden Layers')
    plt.ylabel('Average time taken to classify request (milliseconds)')
    plt.legend([f"Model 1", f"Model 2"], loc="upper left")
    plt.savefig(f'diagrams/time_taken.png')



    for i in range(4):
        plt.cla()
        plt.plot(res[i]["history"])
        plt.plot(res2[i]["history"])

        plt.title(f"Model Accuracy vs Epochs ({i+1} layer{'s' if i > 0 else
''})")
        plt.ylabel("Accuracy")
        plt.xlabel("Epoch Number")
        plt.gca().set_ylim([0.9993, 1])
        plt.legend([f"Model 1", f"Model 2"], loc="lower right")
        plt.savefig(f"diagrams/model_{i+1}.png")


    plt.cla()
    import tensorflow as tf
    model = tf.keras.models.load_model("models/model_2.tf")

if __name__ == "__main__":
    # train(98765)
    evaluate()
```

### 3: Screenshot of raw data

| ip | endpoint | packet_size | headers | time | time_taker | compromised |
|---|---|---|---|---|---|---|
| 139.144.4 | / | 100 | Host: 165.232.182.157User-Agent: python-requests/2.28.: | 1674598718 | 0.05771 | FALSE |
| 170.187.1 | /endpoint | 359 | Host: 165.232.182.157User-Agent: python-requests/2.28.: | 1674598719 | 0.09539 | FALSE |
| 139.162.1 | /endpoint | 871 | Host: 165.232.182.157User-Agent: python-requests/2.28.: | 1674598721 | 0.05118 | FALSE |
| 172.105.9 | / | 56 | Host: 165.232.182.157User-Agent: python-requests/2.28.: | 1674598722 | 0.07691 | FALSE |
| 170.187.1 | /endpoint | 470 | Host: 165.232.182.157User-Agent: python-requests/2.28.: | 1674598723 | 0.08613 | FALSE |
| 139.144.4 | /endpoint | 348 | Host: 165.232.182.157User-Agent: python-requests/2.28.: | 1674598723 | 0.14932 | FALSE |
| 139.144.4 | /endpoint | 885 | Host: 165.232.182.157User-Agent: python-requests/2.28.: | 1674598723 | 0.10248 | FALSE |
| 66.175.22 | / | 90 | Host: 165.232.182.157User-Agent: python-requests/2.28.: | 1674598725 | 0.05419 | FALSE |
| 139.162.1 | /endpoint | 971 | Host: 165.232.182.157User-Agent: python-requests/2.28.: | 1674598725 | 0.05789 | FALSE |
| 172.105.9 | / | 90 | Host: 165.232.182.157User-Agent: python-requests/2.28.: | 1674598726 | 0.03681 | FALSE |
| 172.105.1 | /endpoint | 253 | Host: 165.232.182.157User-Agent: python-requests/2.28.: | 1674598727 | 0.06568 | FALSE |
| 139.144.4 | /endpoint | 446 | Host: 165.232.182.157User-Agent: python-requests/2.28.: | 1674598727 | 0.13238 | FALSE |
| 209.97.18 | / | 79 | Host: 165.232.182.157User-Agent: python-requests/2.28.: | 1674598728 | 0.07384 | FALSE |
| 170.187.1 | /endpoint | 790 | Host: 165.232.182.157User-Agent: python-requests/2.28.: | 1674598728 | 0.12131 | FALSE |
| 139.59.77 | /endpoint | 476 | Host: 165.232.182.157User-Agent: python-requests/2.28.: | 1674598728 | 0.11953 | FALSE |
| 139.144.4 | /endpoint | 435 | Host: 165.232.182.157User-Agent: python-requests/2.28.: | 1674598729 | 0.08191 | FALSE |
| 66.175.22 | /endpoint | 422 | Host: 165.232.182.157User-Agent: python-requests/2.28.: | 1674598729 | 0.11246 | FALSE |
| 139.162.1 | /endpoint | 387 | Host: 165.232.182.157User-Agent: python-requests/2.28.: | 1674598730 | 0.06358 | FALSE |
| 209.97.13 | /endpoint | 261 | Host: 165.232.182.157User-Agent: python-requests/2.28.: | 1674598730 | 0.11112 | FALSE |
| 172.105.1 | /endpoint | 335 | Host: 165.232.182.157User-Agent: python-requests/2.28.: | 1674598730 | 0.07229 | FALSE |
| 172.105.9 | /endpoint | 256 | Host: 165.232.182.157User-Agent: python-requests/2.28.: | 1674598732 | 0.0804 | FALSE |
| 209.97.18 | / | 33 | Host: 165.232.182.157User-Agent: python-requests/2.28.: | 1674598732 | 0.03928 | FALSE |
| 170.187.1 | /endpoint | 417 | Host: 165.232.182.157User-Agent: python-requests/2.28.: | 1674598732 | 0.15398 | FALSE |
| 139.144.4 | /endpoint | 610 | Host: 165.232.182.157User-Agent: python-requests/2.28.: | 1674598733 | 0.10205 | FALSE |
| 139.59.77 | / | 49 | Host: 165.232.182.157User-Agent: python-requests/2.28.: | 1674598733 | 0.06756 | FALSE |
| 66.175.22 | / | 59 | Host: 165.232.182.157User-Agent: python-requests/2.28.: | 1674598734 | 0.07142 | FALSE |
| 139.144.4 | / | 47 | Host: 165.232.182.157User-Agent: python-requests/2.28.: | 1674598734 | 0.04411 | FALSE |
| 139.162.1 | /endpoint | 363 | Host: 165.232.182.157User-Agent: python-requests/2.28.: | 1674598734 | 0.06323 | FALSE |
| 209.97.13 | / | 60 | Host: 165.232.182.157User-Agent: python-requests/2.28.: | 1674598734 | 0.03206 | FALSE |
| 172.105.1 | /endpoint | 545 | Host: 165.232.182.157User-Agent: python-requests/2.28.: | 1674598734 | 0.1496 | FALSE |
| 209.97.18 | /endpoint | 300 | Host: 165.232.182.157User-Agent: python-requests/2.28.: | 1674598736 | 0.14174 | FALSE |
| 170.187.1 | /endpoint | 312 | Host: 165.232.182.157User-Agent: python-requests/2.28.: | 1674598737 | 0.12996 | FALSE |
| 139.59.77 | /endpoint | 263 | Host: 165.232.182.157User-Agent: python-requests/2.28.: | 1674598737 | 0.08202 | FALSE |
| 172.105.9 | / | 88 | Host: 165.232.182.157User-Agent: python-requests/2.28.: | 1674598737 | 0.09897 | FALSE |
| 139.144.4 | /endpoint | 941 | Host: 165.232.182.157User-Agent: python-requests/2.28.: | 1674598737 | 0.13083 | FALSE |
| 66.175.22 | /endpoint | 305 | Host: 165.232.182.157User-Agent: python-requests/2.28.: | 1674598737 | 0.16102 | FALSE |
| 139.144.4 | /endpoint | 275 | Host: 165.232.182.157User-Agent: python-requests/2.28.: | 1674598738 | 0.08437 | FALSE |
| 172.105.1 | /endpoint | 272 | Host: 165.232.182.157User-Agent: python-requests/2.28.: | 1674598738 | 0.05136 | FALSE |
| 139.162.1 | / | 30 | Host: 165.232.182.157User-Agent: python-requests/2.28.: | 1674598739 | 0.07184 | FALSE |
| 209.97.13 | /endpoint | 476 | Host: 165.232.182.157User-Agent: python-requests/2.28.: | 1674598739 | 0.13159 | FALSE |

### 4: Screencast of model evaluation

The models with 4 different layers were tested upon the training data, and their performance and accuracy was measured. The screencast of the code can be seen below.

https://youtu.be/b-iVnoBMfyo

### 5: List of Figures and Tables


| Name | Page No. |
|---|---|
| Figure 1: Structure of a feed forward neural network (self-made) | 5 |
| Equation 1: Equation for the activation of a neuron in layer i, given consecutive layers j and I | 6 |
| Figure 2: Geographical locations of clients used in experiment | 8 |
| Figure 3.1: Client-side structure and logic | 8 |

| | |
|---|---|
| Figure 3.2: Server-side structure | 9 |
| Table 1: Sample raw data | 10 |
| Figure 4.1: Packet Size of Each Request | 11 |
| Figure 4.2: Time Taken for Each Request | 11 |
| Figure 5: Python library imports used for preprocessing | 12 |
| Table 2.1: Fields for raw data | 12 |
| Table 2.2: Fields for processed data | 12 |
| Figure 6.1: Average packet size vs average time taken | 13 |
| Figure 6.2: Packet size vs average time taken | 13 |
| Figure 7: The structure of the programmed feed forward neural network, where n is the number of layers | 14 |
| Figure 8.1: Accuracy vs Epochs with 1 layer | 16 |
| Figure 8.2: Accuracy vs Epochs with 2 layers | 16 |
| Figure 8.3: Accuracy vs Epochs with 3 layers | 17 |
| Figure 8.4: Accuracy vs Epochs with 4 layers | 17 |
| Figure 9: Accuracy of models on testing data for different layered network | 17 |
| Figure 10: Average time taken to classify request vs number of hidden layer | 18 |