UNIVERSITY COLLEGE LONDON (UCL)

ELEC0018

Software for Network Services and Design Assignment

Student Names
DENNIS GOH JIA WANG

Student Numbers
16002077

Module Lecturer: Dr. MIGUEL RIO

May, 2020

Contents

1 Introduction **2**

2 Data Collection **2**
2.1 Problems 2
2.2 Web Scraping 4
2.2.1 Selenium 5
2.2.2 Query and Countries 9
2.2.3 Convert URLs to IP addresses 9
2.2.4 Throughput Calculations 10
2.2.5 Convert IP addresses into 32-bit vector 11
2.2.6 Multiprocessing 12

3 Machine Learning Model **13**
3.1 Data Pre-processing 13
3.2 Neural Network Architecture 14
3.2.1 Number of Hidden Layers 15
3.2.2 Number of neurons 18
3.3 Hyperparameter Tuning 20
3.3.1 Types of optimisation algorithms 20
3.3.2 Activation Functions 21
3.3.3 Weight Initialiser 22
3.3.4 Batch sizes and number of epochs 24
3.3.5 Learning Rate 25
3.3.6 Regularization 25
3.4 Finalised Neural Network Model 27

4 Evaluation **27**
4.1 Result and Discussion 27
4.2 Suggested Solution 29
4.2.1 Large Dataset 29
4.3 Future Improvement 29
4.3.1 Rank Correlation 29
4.3.2 Convolutional Neural Networks (CNN) 29

5 Conclusion **29**

1 Introduction

The hierachical structure of Internet IP addresses can be exploited due to its predictive capabilities that might bring some practical application for network agents such as service selection, user-directed routing, resource scheduling and network inference. The IP address are uniquely assigned and distributed follows the allocation policies [1] which in turn means that IP addresses provide geographic locality.

The physical distance between the machines has a significant effects on network latency and throughput. Surprisingly, these effects are correlated to the distance such that the longer the distance, the higher the latency and the lower the throughput.

Network throughput forecasting using IP address space is a challenging foundation for networking. However, there are many state-of-the-art machine learning approach in today are very robust and successful in complex and dynamic environment that can easily recover from infrequent predictive errors. In this experiment, our goal is to build a machine learning model particularly neural network that able to predict network throughput.

This report comprised into 3 main parts which are data collection, machine learning model and performance evaluation. Firstly, we discussed the problems and procedure of obtaining data for this experiments. Secondly, we discussed the pipeline of the machine learning model involve data pre-processing and hyperparameter tuning for obtaining the best neural network architecture. Finally, we evaluated the performance of the model using several metrics to understand how accurate the predictions made.

2 Data Collection

The data are collected from image URLs by using web scraping which will discussed further in section 2.2. However, there are several problems such as TCP slow start, blocking requests, geographical spread and content delivery networks. These will significantly affect the the latency/throughput of the established connection due to the files sizes of the content and the round trip of data transmission.

2.1 Problems

1. TCP Slow Start

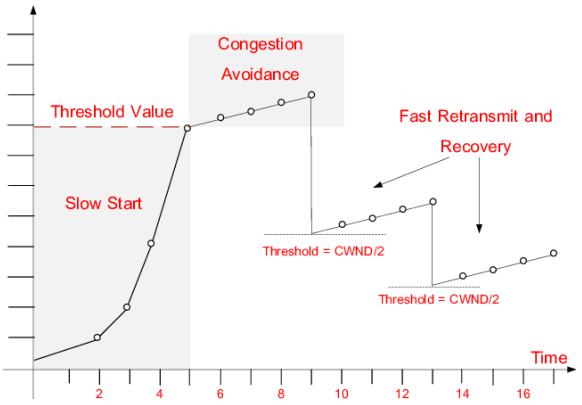

Figure 1: TCP Slow Start [2]

Transmission Control Protocol (TCP) slow start is a congestion-avoidance algorithm used to detect the available bandwidth for packet transmission and balances the speed of network connection. It has a congestion window (*cwnd*) to set an upper limit on the amount of data that can be transmitted from a source over the network before receiving an acknowledgement. The slow start threshold (*ssthresh*) determine the activation of slow start. When a new connection is established, *cwnd* is initialized to one TCP data and waits for acknowledgement packet. Once that acknowledgement packet is received, the congestion window will increase gradually until it reach *ssthresh*. Then, the congestion window will increase slowly until congestion occur as shown in figure 1

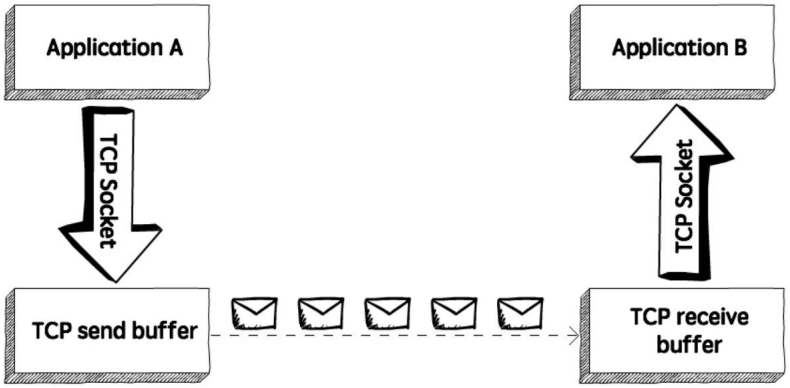

Figure 2: Congestion Control in action

This negotiation of connection between a sender an receiver by defining the amount of data can be transmitted with each packet allowed

it to reach the network maximum carrying capacity. As a result, it prevent the overloaded buffer to dump the incoming packets on both sender and receiver shown in figure 2.

In this experiment, the congestion window is the limiting factor that determines the number of bytes sent over a network. The throughput of a link is solely depended on the congestion window size and therefore throughput become the limiting factor. The TCP slow start does not indicate the average throughput but maximum throughput of the server. In order to overcome this problem, a link should be opened long enough for congestion window to saturate more than $ssthresh = 65535\ bytes$ by downloading large files.

2. Content Delivery Network

   Content Delivery Network (CDN) is a geographically distributed network of servers work together to provide fast delivery of internet contents and services spatially relative to the end users. The popularity of CDN services seem to grow and deploy widely over multiple principal data route that served majority web traffics today. This is because it can improves website load times, reduces bandwidth cost, improves web security and increase content availablity and redundancy. However, this can lead to data homogeneity collected from web scraping due to the similar contents accessible in multiple servers.

3. Geographical Spread

   The ideal dataset collected should have IP addresses geographically spread across multiple countries. This can be very useful for carry out analysis for network throughput forecasting.

4. Blocking Request

   Majority websites block web scraping and crawling based on the static IP of user server or hosting provider but some do not have anti-scraping mechanisms. Apart from that, search engines policies might prohibit user for web scraping to protect from viruses and malicious content found on the websites. This can be a huge problem to collect large amount of data for machine learning purposes.

2.2 Web Scraping

Web scraping also known as web extraction or web harvesting is a technique used for extracting data from the World Wide Web (www) and save

it to database or file systems. These data can later be used for analysis or retrieval. Web data are commonly scrapped utilizing Hypertext Transfer Protocol (HTTP) or via web browser. Fortunately, Python contains a lot of web scrapping modules such as **urllib**, **BeautifulSoup** and **Selenium**. In this project, we accomplished the data collection task by using **Selenium** to automate web browser.

Selenium is chosen because it is an open-source automated testing framework supports various browser such as Chrome, Firefox, Safari, Internet Explorer, Opera and Edge. The compatibility across multi-browser support make it convenient to write a single script for all. In this project, Google Chrome browser was chosen because Google Search Images contain advanced settings that allows user to filter the image size and different countries as search queries. This functionality solved the TCP slow start and geographical spread data mentioned earlier in section 2.1. Besides, Selenium is also closely imitated human behaviours make it possible to emulate a web browser with Python.

2.2.1 Selenium

The web scraping program begin by first defining the option for web browser and the chromedriver file path.

```
# Define the Chrome options to open the window in incognito
    mode
option = webdriver.ChromeOptions()
option.add_argument('--incognito')
option.add_argument('--ignore-certificate-errors-spki-list')
option.add_argument('--ignore-ssl-errors')

# Find the ChromeDriver path
SNS_dir = os.path.abspath(os.curdir)
Scrap_dir = os.path.join(SNS_dir, 'Scraping')
DRIVER_PATH = os.path.join(Scrap_dir, 'chromedriver')
```

Listing 1: WebDriver file path and browser options

After that, we write a script to fetch each URLs in google search images as shown in the snippet below. The function *fetch_img_urls* takes 4 input parameters:

1. query: Search term for example food.
2. country: Search countries
3. img_to_fetch: Number of links to collect

4. wd: instantiated webdriver

```
def fetch_img_urls(query, country, img_to_fetch, wd):
    def scroll_to_end(wd):
        wd.execute_script("window.scrollTo(0, document.body.scrollHeight);")
        time.sleep(2)

    # Build a search query for picture more than 4 Megapixels
    search_url = 'https://www.google.com/search?safe=off&site=&tbm=isch&source=hp&q={q}&oq={q}&cr=country{c}&gs_l=img&tbs=isz:lt,islt:8mp'
    wd.get(search_url.format(q=query, c=country))

    img_urls = set()
    img_count = 0
    results_start = 0

    while img_count < img_to_fetch:
        scroll_to_end(wd)

        # Get all the image thumbnail results
        thumbnail_results = wd.find_elements_by_css_selector("img.Q4LuWd")
        num_results = len(thumbnail_results)

        print(f"Found: {num_results} search results. Extracting links from {results_start}:{num_results}")

        for img in thumbnail_results[results_start:num_results]:
            # Clicking the resulting thumbnail to get real image
            try:
                img.click()
                time.sleep(0.5)
            except Exception:
                continue

            # Extract the image urls and get rid of the encrypted gstatic links
            real_img = wd.find_elements_by_css_selector('img.n3VNCb')
            for x in real_img:
                if x.get_attribute('src') and 'http' in x.get_attribute('src') and not 'gstatic' in x.get_attribute('src'):
                    img_urls.add(x.get_attribute('src'))

```

```
            # Total number of images urls extracted
            img_count = len(img_urls)

            # Set the limit for retrieved image urls
            if img_count >= img_to_fetch:
                print(f"Image links found: {img_count} ... DONE!!!")
                break
        else:
            print("Looking for more image links ...")
            time.sleep(30)
            return
            show_more_results = wd.find_element_by_css_selector(".mye4qd")
            if show_more_results:
                wd.execute_script("document.querySelector('.mye4qd').click();")

        # move the result startpoint further down
        results_start = len(thumbnail_results)

    return img_urls
```

Listing 2: Fetch URLs script

The images link URLs are collected using the script above. For understanding what the above script does, we further break down each section underneath it.

```
# Build a search query for picture more than 4 Megapixels
search_url = 'https://www.google.com/search?safe=off&site=&tbm=isch&source=hp&q={q}&oq={q}&cr=country{c}&gs_l=img&tbs=isz:lt,islt:8mp'
wd.get(search_url.format(q=query, c=country))
```

Listing 3: Search query and countries

The listing 3 show the query and country allow user to input search term and country interested to search. The term *islt:8mp* in **search_url** indicate the browser will only display 8 megapixels large pictures.

```
# Get all the image thumbnail results
thumbnail_results = wd.find_elements_by_css_selector("img.Q4LuWd")
num_results = len(thumbnail_results)
```

Listing 4: Finding thumbnails using CSS selector

The listing 4 is used to find the thumbnails. It locate and inspect the web element and find the class attribute with value of **img.Q4LuWd**.

```
for img in thumbnail_results[results_start:num_results]:
    # Clicking the resulting thumbnail to get real image
    try:
        img.click()
        time.sleep(0.5)
    except Exception:
        continue

    # Extract the image urls and get rid of the encrypted gstatic links
    real_img = wd.find_elements_by_css_selector('img.n3VNCb')
    for x in real_img:
        if x.get_attribute('src') and 'http' in x.get_attribute('src') and not 'gstatic' in x.get_attribute('src'):
            img_urls.add(x.get_attribute('src'))

    # Total number of images urls extracted
    img_count = len(img_urls)

    # Set the limit for retrieved image urls
    if img_count >= img_to_fetch:
        print(f"Image links found: {img_count} ... DONE!!!")
        break
```

Listing 5: Extract real images URLs

The listing 5 imitated human clicking on each thumbnail and extract the real image URLs. Besides, it also get rid of encrypted gstatic links because sometimes Google had already recognized the has been deleted might causes web browser to freeze. Finally, we also set the limit of links to be retrieved.

```
else:
    print("Looking for more image links ...")
    time.sleep(30)
    return
    show_more_results = wd.find_element_by_css_selector(".mye4qd")
    if show_more_results:
        wd.execute_script("document.querySelector('.mye4qd').click();")

# move the result startpoint further down
results_start = len(thumbnail_results)
```

Listing 6: Search for more results

The listing 6 allow browser to imitate human action for scrolling down the page and search for more results.

2.2.2 Query and Countries

To ensure the data are geographically spread as mentioned earlier, we have a list of chosen countries. The query chosen in this project is food as it is very common topic in every countries.

Continents	Countries	Country Codes
Asia	China, India, Japan, Korea, Taiwan	CN, IN, JP, KR, TW
Europe	Great Britain, France, Italy, Germany, Russian	GB, FR, IT, DE, RU
Africa	Nigeria, Egypt, Ethiopia, Tanzania, Zimbabwe	NG, EG, ET, TZ, ZA
Oceania	New Zealand, Australia	NZ, AU
South America	Mexico, Colombia, Argentina, Brazil, Chile	MX, CO, AR, BR, CL
North America	United States, Canada	US, CA

Table 1: Country List

2.2.3 Convert URLs to IP addresses

```
def get_ipaddr(img_urls):
    # Display the result in dataframe
    url_list = []
    ip_list = []

    # Break the URL into component and get the IP address of URL
    for urls in img_urls:
        website = urlparse(urls)
        ip_addr = socket.gethostbyname(website.netloc)
        print(f"URL: {website.netloc} and IP: {ip_addr}")
        url_list.append(website.netloc)
        ip_list.append(ip_addr)

    url_ip_dict = dict(zip(url_list, ip_list))
    dataframe = pd.DataFrame([[keys, values] for keys, values in url_ip_dict.items()]).rename(columns={0:'URL', 1:'IP address'})

    return dataframe, ip_list
```

Listing 7: Convert URLs to IP addresses

The listing 7 return a pandas dataframe of the converted URL into IP addresses. This can be done by using **urllib.urlparse** and *gethostbyname* in **socket** modules. The **urllib.urlparse** module defines a standard interface to break up the URL into components. The *gethostbyname* function retrieve host information corresponding to host name and translate to IPv4 format as shown in Figure 3.

	URL	IP address
0	www.littlegreenduckie.com	185.151.30.163
1	hips.hearstapps.com	199.232.56.155
2	www.boxfituk.com	172.67.69.52
3	www.cda.eu	172.67.131.51
4	s2.r29static.com	199.232.57.179
5	www.alchemylive.london	35.197.204.225
6	baxterstorey.com	185.164.44.9

Figure 3: URL to IP address example

2.2.4 Throughput Calculations

After all the URls are parsed, we can calculated using **time** and **Request** modules. The throughput is defined as data transfer from source to destination within the given timeframe.

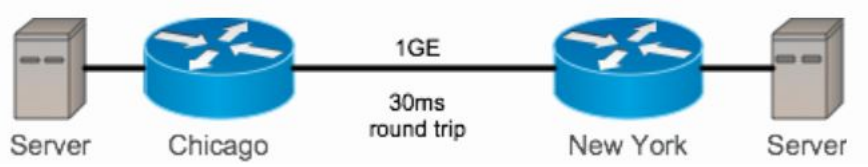

Figure 4: Network Throughput Example from http://bradhedlund.com/2008/12/19/how-to-calculate-tcp-throughput-for-long-distance-links/

$$Throughput = \frac{TCP\ received\ window\ size}{Round\ trip\ time\ latency} \tag{1}$$

```
1 def calc_througput(ssthresh, img_urls):
2     data = 0
3     throughput = 0
4     throughput_list = []
```

```
5
6       # Calculating throughput and set timeout for very large
    files
7       for urls in img_urls:
8           try:
9               start_time = time.time()
10              img_content = len(requests.get(urls, stream=True)
    .content)
11              if img_content > ssthresh:
12                  data += img_content
13          except Exception as e:
14              print(f"ERROR - Could not download {urls} - {e}")
15              data = 0
16
17          end_time = time.time()
18          throughput = data/(end_time - start_time)
19          throughput_list.append(throughput)
20
21      return throughput_list
```

Listing 8: Calculate throughput

From listing 8, the *time.time()* is used to record the round trip time latency while retrieving image content with *request.get().content*.

2.2.5 Convert IP addresses into 32-bit vector

The IP addresses are converted into 32-bit vector using listing 9 for machine learning purpose that will discuss further in section **??**. In machine learning, we uses Neural Network regression to predict network throughput. Therefore, it is necessary to transform into 32-bit dimensional feature space where each bit corresponds to a dimension and throughput as the output labels.

```
1 # Convert IP address into 32 dimension input space
2 def convert2bin(ip_list):
3     bin_ip = []
4
5     for ip in ip_list:
6         ip_vector = format(int(ipaddress.ip_address(ip)), '
    032b')
7         bin_ip.append(ip_vector)
8
9     return bin_ip
```

Listing 9: Convert IP addresses to 32-bit dimension vector

2.2.6 Multiprocessing

Figure 5: Data Extraction Pipeline

The data extraction pipeline is described in figure 5 and script in listing 10. The main reason of making this pipeline is that it can be executed in parallel manner for efficiency.

```
# Pipeline of the image scraping function
def pipeline(country):
    # Argument is tuple of (Continent, [list of countries])
    # Retrieve image urls
    with webdriver.Chrome(executable_path=DRIVER_PATH, options=option) as wd:
        start_timer = datetime.now()
        img_urls = fetch_img_urls('food', country, 50, wd)
        time_elapsed = datetime.now() - start_timer
        print("Time elapsed (hh:mm:ss.ms) {}".format(time_elapsed))

        # Get unique IP address of each images urls and convert to binary 32 bits
        df, ip_addr = get_ipaddr(img_urls)
        ip_binary = convert2bin(ip_addr)

        # Create a duplicated countries list to map the IP address and throughput
        cr_list = [country] * len(img_urls)

        # Calculating throughput
        throughput = calc_througput(65535, img_urls)
        dataframe = pd.DataFrame(list(zip(cr_list, ip_addr, throughput, ip_binary)), \
                                 columns=['Country', 'IP address', 'Throughput', 'Binary IP'])

        # Splitting binary 32 bits IP address into multidimensional columns for machine learning
        for i in range(32):
            dataframe['B'+str(i)] = dataframe['Binary IP'].str[i]

```

```
27      return dataframe
28
29  if __name__ == '__main__':
30      # Multiprocesses to fetch image by parsing different
        countries
31      country_dict = {'Asia': ['CN', 'IN', 'JP', 'KR', 'TW'],
32                      'Europe': ['GB', 'FR', 'IT', 'DE', 'RU'],
33                      'Africa': ['NG', 'EG', 'ET', 'TZ', 'ZA'],
34                      'Oceania': ['NZ', 'AU'],
35                      'South America': ['MX', 'CO', 'AR', 'BR',
        'CL'],
36                      'North America': ['US', 'CA']}
37
38      # Convert dictionary values to list for multiprocessing
        pool
39      country_list = [(values) for values in country_dict.
        values()]
40      merge = list(itertools.chain(*country_list))
41      print(merge)
42
43      # max number of parallel process
44      with Pool(processes=4) as pool:
45          results = pool.map(pipeline, merge)
46
47      result_df = pd.concat(results)
48      print(result_df)
49
50      result_df.to_csv('dataset.csv', index=False)
```

Listing 10: Multiprocessing web scraping

Next, multiprocessing is used to speed up the web scarping. The multiprocessing architecture shown in figure 6 uses module *multiprocessing.Pool* that allows multiple worker to execute the web scraping pipeline in parallel fashions.

3 Machine Learning Model

This section mainly discussed how to obtain the overall design of the neural network architecture. The neural network can be constructed using open-source neural netowrk library **KERAS** written in Python.

3.1 Data Pre-processing

The listing 11 uses **scikit-learn** library *StandardScaler* to normalise the data. This transformation is very important because it can speed up the conver-

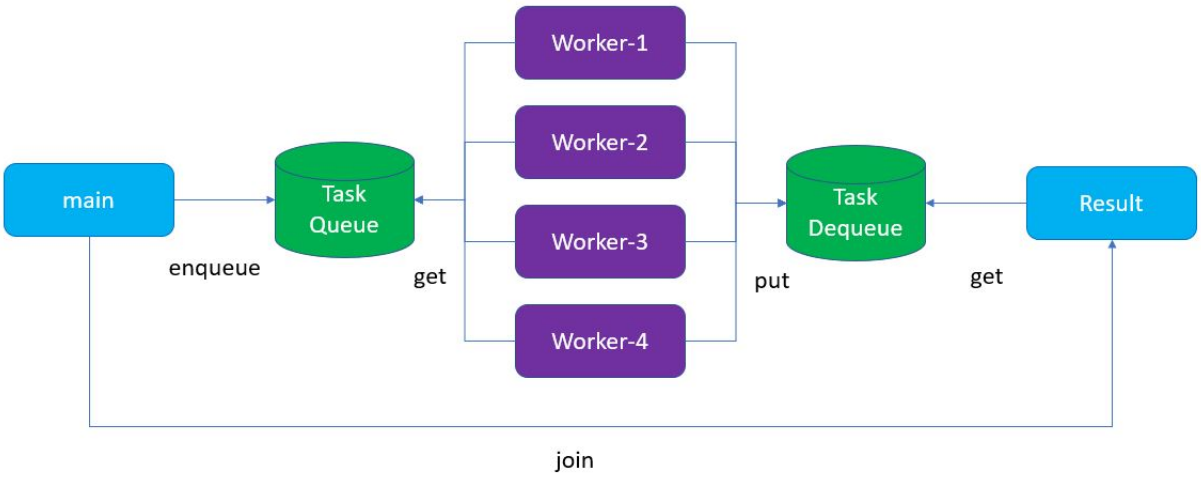

Figure 6: Multiprocessing architecture for data extraction

gence rate o gradient descent during training phase. The data are splitted into 80% training set and 20% test set with randomized order.

```
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

# Read the csv file of the scraped image urls
dataframe = pd.read_csv("dataset.csv", header=0)

# Shuffle the dataset and split into train and test set
X = dataframe[dataframe.columns[4:36]]
Y = dataframe[['Throughput']]
X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
    test_size=0.2, random_state=42)

# Standardised the training and test set
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
Y_train = scaler.fit_transform(Y_train)
Y_test = scaler.transform(Y_test)
```

Listing 11: Data Pre-processing

3.2 Neural Network Architecture

A neural network comprised of simple element called neurons to analyse complex problems, emulate any complex functions and predict the out-

puts. A simple neural network architecture consists of three layers: input layer, hidden layer and output layer.

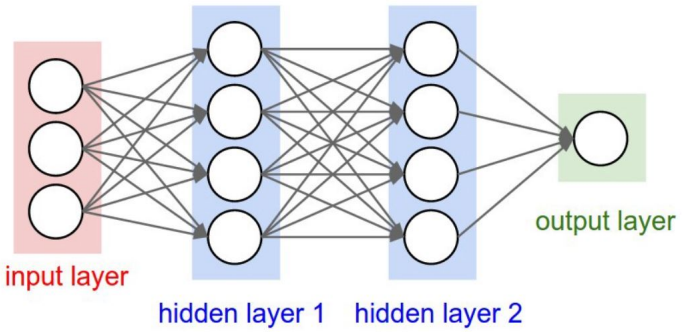

Figure 7: Deep Neural Network Model

A deep neural network shown in figure 7 consists of more than one hidden layers is used in this project because it increases the complexity of the model and significantly improved the prediction. Each hidden layer contain an arbitrary number of neurons. The input layer has number of neurons correspond to the input features. The output layer contain only single neurons as if it is a regression problem and more than one if it is a classification problem. There are two main hyperparameter control the architecture and topology of the network: number of hidden layers and number of neurons per hidden layer.

3.2.1 Number of Hidden Layers

To begin with, we first have to examine the number of hidden layer that result the smallest mean square error. Then, the chosen architecture will be used further for hyperparameter tuning. The different network architectures shown in listing 12 are designed based on the rule of thumb below:

1. 0 hidden layers - Only capable of representing linear separable functions or decisions.

2. 1 hidden layers - Can approximate any function that contains a continuous mapping from one finite space to another.

3. 2 hidden layers - Can represent an arbitrary decision boundary to arbitrary accuracy with rational activation functions and can approximate any smooth mapping to any accuracy.

```
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout

# Define function prototype to create neural network for
    pipelining later
# 3-layers multiperceptrons
def NN_3_mlp():
    # Create a sequential model
    model = Sequential()

    # Add neurons dense layers (specify the input shape in
    first dense layer)
    # In these layers, the ReLu activations functions is used
    # Kernel initializer start some weights and update them
    through backpropagation
    model.add(Dense(32, input_dim=32, kernel_initializer='
    normal', activation='relu'))
    model.add(Dense(16, kernel_initializer='normal',
    activation='relu'))
    model.add(Dense(8, kernel_initializer='normal',
    activation='relu'))
    model.add(Dense(1, kernel_initializer='normal'))

    # Display summary of the model
    model.summary()

    # Compile the model using ADAM (Adaptive learning rate
    optimization)
    model.compile(loss='mean_squared_error', optimizer='adam'
    , metrics=['mse'])

    return model

# 3-layers multiperceptrons with dropout layers
def NN_3_mlp_dropout():
    # Create a sequential model
    model = Sequential()

    model.add(Dense(32, input_dim=32, kernel_initializer='
    normal', activation='relu'))
    model.add(Dense(16, kernel_initializer='normal',
    activation='relu'))
    model.add(Dense(8, kernel_initializer='normal',
    activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(1, kernel_initializer='normal'))

    # Display summary of the model
```

```
39      model.summary()
40
41      # Compile the model using ADAM (Adaptive learning rate
        optimization)
42      model.compile(loss='mean_squared_error', optimizer='adam'
        , metrics=['mse'])
43
44      return model
45
46  # 2-layers multiperceptrons
47  def NN_2_mlp():
48      # Create a sequential model
49      model = Sequential()
50
51      model.add(Dense(32, input_dim=32, kernel_initializer='
        normal', activation='relu'))
52      model.add(Dense(16, kernel_initializer='normal',
        activation='relu'))
53      model.add(Dense(1, kernel_initializer='normal'))
54
55      # Display summary of the model
56      model.summary()
57
58      # Compile the model using ADAM (Adaptive learning rate
        optimization)
59      model.compile(loss='mean_squared_error', optimizer='adam'
        , metrics=['mse'])
60
61      return model
62
63  # 2-layers multiperceptrons with dropout layers
64  def NN_2_mlp_dropout():
65      # Create a sequential model
66      model = Sequential()
67
68      model.add(Dense(32, input_dim=32, kernel_initializer='
        normal', activation='relu'))
69      model.add(Dense(16, kernel_initializer='normal',
        activation='relu'))
70      model.add(Dropout(0.5))
71      model.add(Dense(1, kernel_initializer='normal'))
72
73      # Display summary of the model
74      model.summary()
75
76      # Compile the model using ADAM (Adaptive learning rate
        optimization)
77      model.compile(loss='mean_squared_error', optimizer='adam'
        , metrics=['mse'])
```

```
78
79      return model
80
81  # Single-layer multiperceptrons
82  def regressor():
83      # Create a sequential model
84      model = Sequential()
85
86      model.add(Dense(32, input_dim=32, kernel_initializer='
        normal', activation='relu'))
87      model.add(Dense(1, kernel_initializer='normal'))
88
89      # Display summary of the model
90      model.summary()
91
92      # Compile the model using ADAM (Adaptive learning rate
        optimization)
93      model.compile(loss='mean_squared_error', optimizer='adam'
        , metrics=['mse'])
94
95      return model
```

Listing 12: Different Neural Network architecture

Kfold cross validation can be performed to evaluate the effect of the number of hidden layers. The figure 8 shows the result of the mean squared error score versus the training examples. It can be seen that all the architectures show similar trend but neural network with 2 hidden layers and dropout layer had the least mean squared error shown in table 2.

Number of hidden layers	Mean squared error	Standard deviation
0	-1.19	+/- 0.5
1	-1.08	+/- 0.49
1 (Dropout)	-1.01	+/- 0.51
2	-1.08	+/- 0.53
2 (Dropout)	-0.99	+/- 0.51

Table 2: Hidden layer cross validation summary

3.2.2 Number of neurons

At first, the network architecture are designed using the rule of thumb below:

1. The number of hidden neurons should be between the size of the input layer and the size of the output layer.

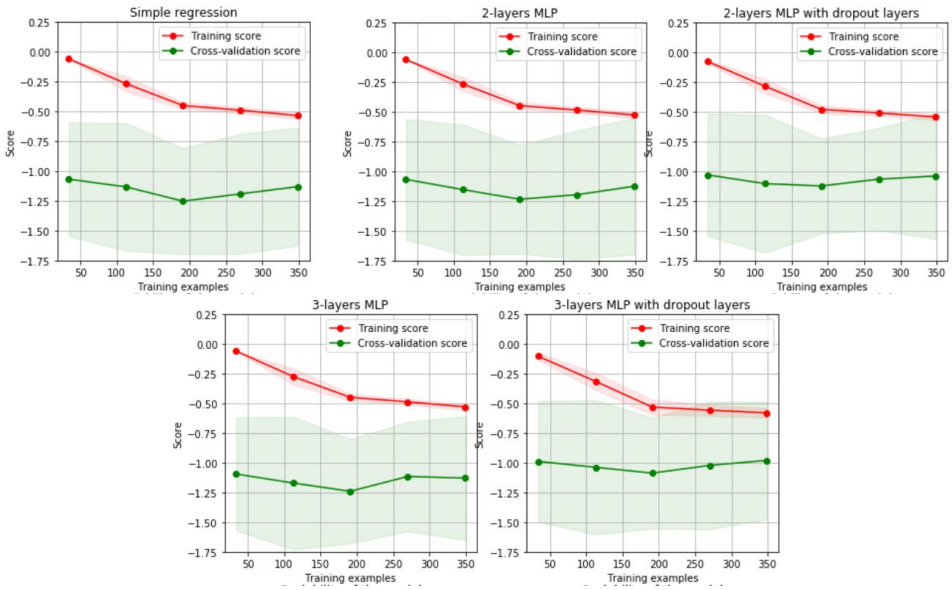

Figure 8: (a) Simple regression model. (b) Single hidden layer. (c) Two hidden layers

2. The number of hidden neurons should be 2/3 the size of the input layer, plus the size of the output layer.

3. The number of hidden neurons should be less than twice the size of the input layer.

In this part, we used the selected architecture earlier and performed **GridSearchCV** on different set of neurons per hidden layers shown in listing 13.

```
# Define different number of neurons per layers as grid search parameters
hidden_1 = [8, 16, 32, 64, 96, 128]
hidden_2 = [8, 16, 32, 64, 96, 128]
param_grid = dict(hidden_1=hidden_1, hidden_2=hidden_2)
model_CV = KerasRegressor(build_fn=NN_3_mlp_dropout, epochs=100, batch_size=5, verbose=1)
grid = GridSearchCV(estimator=model_CV, param_grid=param_grid, n_jobs=-1, cv=3, return_train_score=True)
grid_result = grid.fit(X_train, Y_train)
```

Listing 13: Number of neurons per layer GridSearchCV

The figure 9 show the mean squared error score on training and validation set. It can be seen that the best validation score achieved is -0.9565 with the optimal number of neurons for 1st and 2nd hidden layers are 16 and 8 respectively.

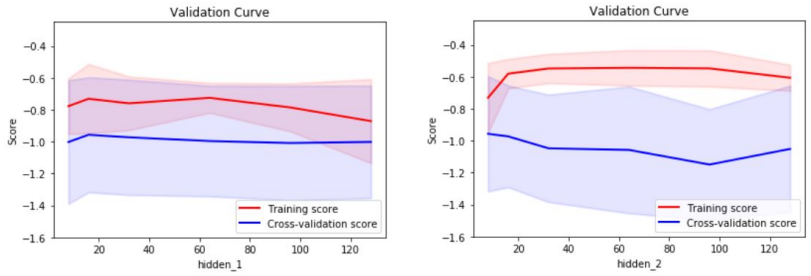

Figure 9: (a) Score on first hidden layer. (b) Score on second hidden layer.

To summarize this section, the overall neural network architecture is 2 hidden layer with 16 and 8 neurons in 1st and 2nd hidden layer respectively.

3.3 Hyperparameter Tuning

After obtaining the neural network architecture, there are several hyperparameters such as types of optimisation algorithms, weight initializer, activation functions, learning rates batch size and number of epochs to be optimised.

3.3.1 Types of optimisation algorithms

There are several optimisation algorithms available in Keras library as shown in listing 14. The optimisation algorithms are iterative learning algorithm that find the a set of internal weights to minimise the error gradient.

```
# Define different optimizer as grid search parameters
optimizers = ['SGD', 'RMSprop', 'Adagrad', 'Adadelta', 'Adam'
    , 'Adamax', 'Nadam']
param_grid = dict(optimizer=optimizers)
model_CV = KerasRegressor(build_fn=NN_3_mlp_dropout, epochs
    =100, batch_size=5, verbose=1)
grid = GridSearchCV(estimator=model_CV, param_grid=param_grid
    , n_jobs=-1, cv=3)
grid_result = grid.fit(X_train, Y_train)
```

Listing 14: Optimisation algorithms GridSearchCV

From the table 3, the adaptive gradient algorithm (Adagrad) had the least mean squared error of -0.9609 compared to other algorithms.

Optimisers	Mean squared error	Standard deviation
SGD	-0.9641	+/- 0.3626
RMSprop	-1.006	+/- 0.3697
Adagrad	-0.9609	+/- 0.3347
Adadelta	-1.012	+/- 0.3618
Adam	-0.9805	+/- 0.3595
Adamax	-0.9713	+/- 0.3282
Nadam	-1.051	+/- 0.3405

Table 3: Optimisation algorithm grid search summary

3.3.2 Activation Functions

The activation function in neural network play a vital role of introducting non-linearity property. From article [3], the final layer activation function are chosen based on the use-cases summarize in table 4.

Problem Type	Output Type	Final Activation Function	Loss Function
Regression	Continuous value	Linear/ReLU	Mean Squared Error (MSE)
Classification	Binary outcome	Sigmoid	Binary Cross Entropy
Classification	Single label, multiple classes	Softmax	Cross Entropy
Classification	Multiple label, multiple classes	Sigmoid	Binary Cross Entropy

Table 4: Final Activation Function summary

In this project, we are predicting throughput as an output value and it is a regression problem. Therefore, the listing 15 show the grid search only comparing the performance of linear and ReLU activation function.

```
# Define different activation function as grid search parameters
activation = ['relu', 'linear']
param_grid = dict(activation=activation)
model_CV = KerasRegressor(build_fn=NN_3_mlp_dropout, epochs=100, batch_size=5, verbose=1)
grid = GridSearchCV(estimator=model_CV, param_grid=param_grid, n_jobs=-1, cv=3)
grid_result = grid.fit(X_train, Y_train)
```

Listing 15: Activation functions GridSearchCV

From the table 5, the ReLU is chosen as final activation function.

Activation function	Mean squared error	Standard deviation
Linear	-1.004	+/- 0.3988
ReLU	inf	+/- inf

Table 5: Activation function grid search summary

3.3.3 Weight Initialiser

Weight initialisation is an important step for neural network. If all the weights are initialised to zero, then the backpropagation is basically the derivation of zero with respect to loss function. Therefore, convergence to a local or global minima using gradient descent will be impossible.

The article [4] discussed several weight initialisers used to mitigate the chances of vanishing or exploding gradients. An example of 5 hidden layers neural network is used to observe the impact of vanishing and exploding gradient.

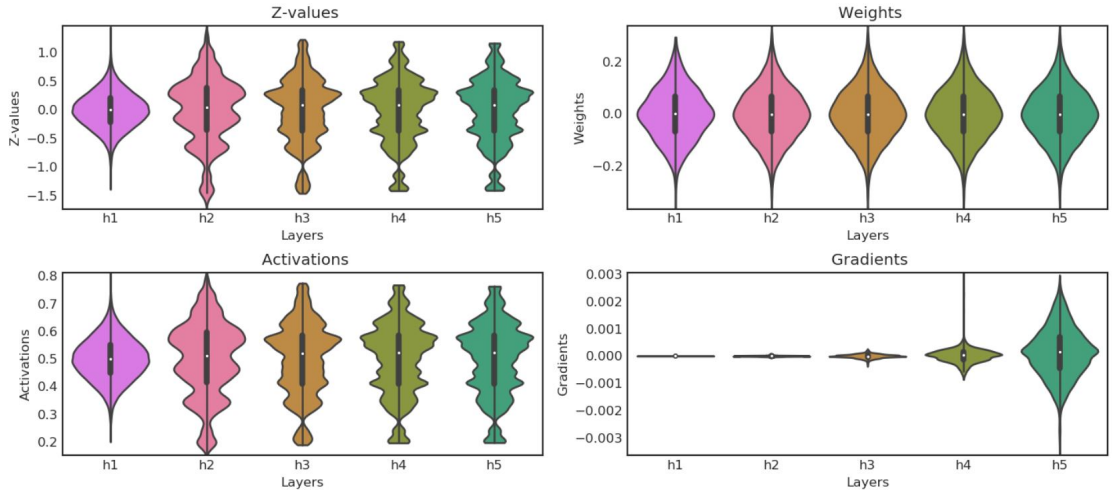

Figure 10: Vanishing Gradient example with normal weight distribution $\sigma = 0.10$ from [4]

From figure 10, the Z-values and activation show smaller range across all hidden layers and the gradient vanished as it backpropagate through the network. This will causes the weights in the first 3 hidden layer to stay constant throughout the learning process.

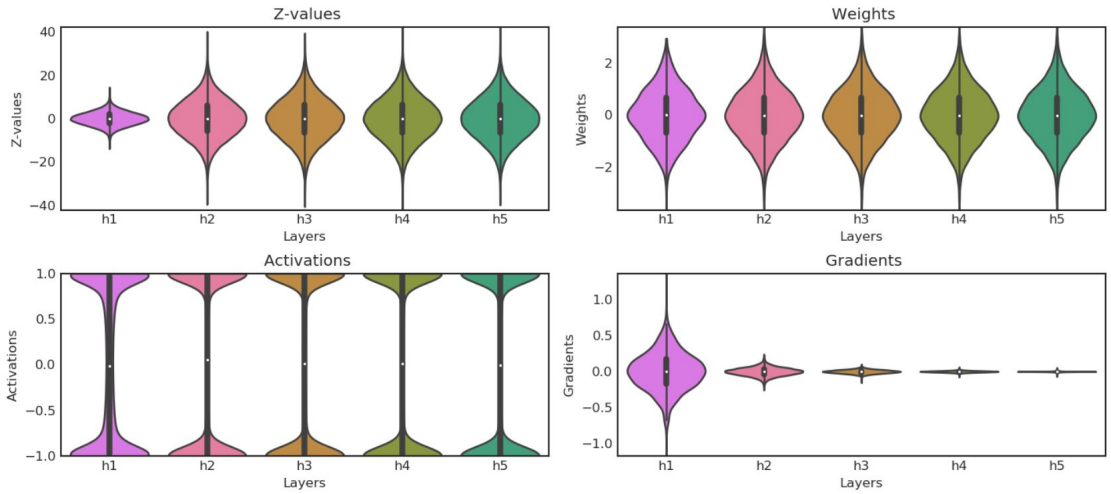

Figure 11: Exploding Gradient example with normal weight distribution $\sigma = 1.00$ from [4]

From figure 11, the activation distribution collapses and Z-values show a wide range of values. Besides, the gradient exploded as it backpropagate through the network. This causes the effect as opposed to vanishing gradient. Therefore, the important to be made here is there is a trade-off between the standard deviation of weight distribution and gradients.

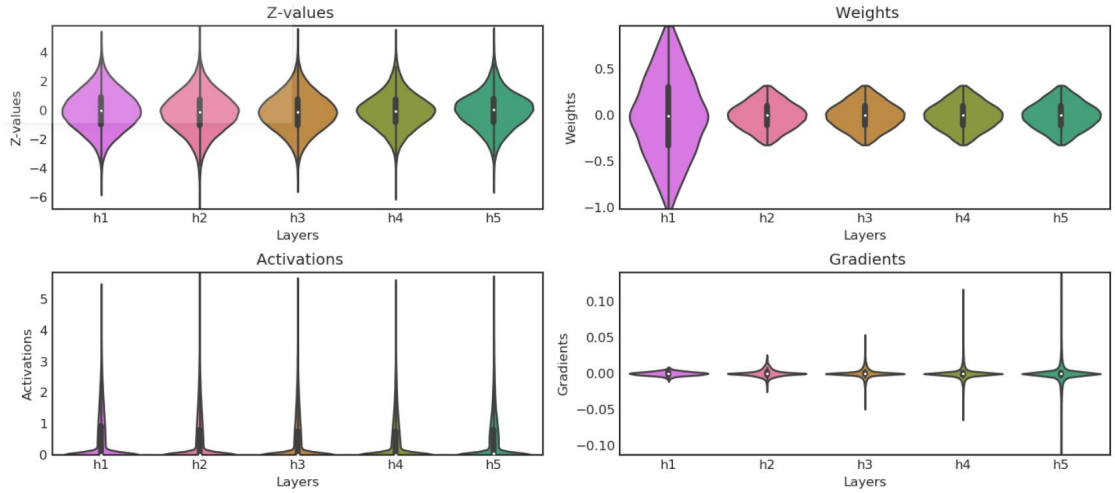

Figure 12: He normal weight distribution from [4]

The He initialisation scheme solved the problem mentioned above and it worked out well with ReLU activation function. From figure 12, the distribution of Z-values are almost similar for all hidden layers. The gra-

dient variance decreases as it backpropagrate through the network but the scales are 10000 times larger than gradient in figure 10.

The listing 16 only uses He initialisation scheme for weight initialiser because it works better for ReLU activation function. The table 6 show that He normal is a better weight initializer scheme for neural network.

```
# Define different weight initializer as grid search
    parameters
init_mode = ['he_normal', 'he_uniform']
param_grid = dict(init_mode=init_mode)
model_CV = KerasRegressor(build_fn=NN_3_mlp_dropout, epochs
    =100, batch_size=5, verbose=1)
grid = GridSearchCV(estimator=model_CV, param_grid=param_grid
    , n_jobs=-1, cv=3)
grid_result = grid.fit(X_train, Y_train)
```

Listing 16: Weight Initializers GridSearchCV

Weight Initialisers	Mean squared error	Standard deviation
He normal	-1.001886	+/- 0.371364
He uniform	-1.012735	+/- 0.324046

Table 6: Weight initialisers grid search summary

3.3.4 Batch sizes and number of epochs

Most of the time, neural networks are trained using mini-batch gradient descent algorithm where the error gradient is used to update the weights based on batch sizes. This hyperparameter control the number of training samples to work through before updating the internal weights. The other hyperparameter is the number of epochs that control the number of complete passes through the training data set to prevent overfitting.

```
# Define different batch size and epochs as grid search
    parameters
batch_size = [5, 10, 20, 40, 60, 80, 100]
epochs = [10, 100, 500, 1000]
param_grid = dict(batch_size=batch_size, epochs=epochs)
model_CV = KerasRegressor(build_fn=NN_optimized, verbose=1)
grid = GridSearchCV(estimator=model_CV, param_grid=param_grid
    , n_jobs=-1, cv=3, return_train_score=True)
grid_result = grid.fit(X_train, Y_train)
```

Listing 17: Batch sizes and epochs GridSearchCV

From Figure 13, the optimal batch sizes is 20. This is consider small batch sizes that can yield noisy updates to the internal weights offering lower generalisation errors. On the other hand, the optimal number of epochs is 100 that lies at the elbow point on the validation curve.

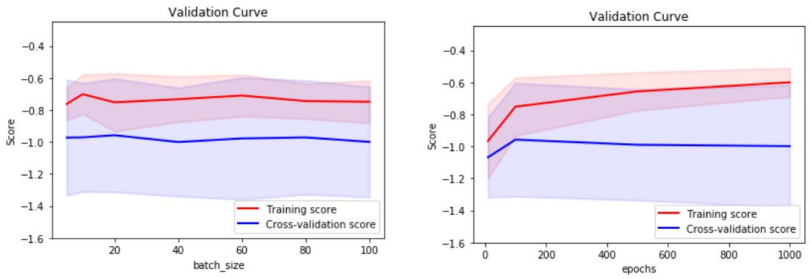

Figure 13: (a) Batch Sizes score. (b) Number of epochs.

3.3.5 Learning Rate

Learning rate is another hyperparameter that determine the speed of convergence to local or global minimal in a non-linear manifold.

```
# Define different learning rate of optimizer as grid search
    parameters
learn_rate = [0.001, 0.01, 0.1, 0.2, 0.3]
param_grid = dict(learn_rate=learn_rate)
model_CV = KerasRegressor(build_fn=NN_optimized, batch_size
    =20, epochs=100, verbose=1)
grid = GridSearchCV(estimator=model_CV, param_grid=param_grid
    , n_jobs=-1, cv=3, return_train_score=True)
grid_result = grid.fit(X_train, Y_train)
```

Listing 18: Learning rate GridSearchCV

From figure 14, the optimal learning rate selected for the optimiser is 0.2.

Snapshot/learning_rate.JPG

Figure 14: Learning rate score

3.3.6 Regularization

A network with large weights are very likely to learn the statistical noise in the training data set. This will possible sign of overfitting model that very

sensitive to the changes of input features. This overfit model will result in poor performance when making prediction of new unseen data.

There are two ways of reducing the overfitting of neural network model on training data using weight regulariser and varying dropout rate. The popular weight regulariser used mostly are L1 and L2 norm. These regulariser set a constraint in the network to force small weights. The L1 norm take the absolute difference of weights and it does feature selection. The The L2 norm take the square difference of the weights shown in equation 2.

$$\sum_{i=1}^{n}(y - \sum_{j=1}^{p}(x_{ij}\beta_j))^2 + \lambda \sum_{j=1}^{p}\beta_j^2 \tag{2}$$

The dropout regularization also reduce overfitting in the networks by randomly dropout nodes in the layer during training phase. As a result, this had enabled network to learn a sparse representation of the input features.

```
# Define different dropout rate and l2 regularizer lambda as
    grid search parameters
dropout_rate = [0.3, 0.4, 0.5, 0.6, 0.7, 0.8]
weight_lambda = [1e-1, 1e-2, 1e-3, 1e-4, 1e-5, 1e-6]
param_grid = dict(dropout_rate=dropout_rate, weight_lambda=
    weight_lambda)
model_CV = KerasRegressor(build_fn=NN_regularized, batch_size
    =20, epochs=100, verbose=1)
grid = GridSearchCV(estimator=model_CV, param_grid=param_grid
    , n_jobs=-1, cv=3, return_train_score=True)
grid_result = grid.fit(X_train, Y_train)
```

Listing 19: L2 regulariser and dropout rate GridSearchCV

From figure 15, the optimal dropout rate and lambda are 0.5 and $1e-5$.

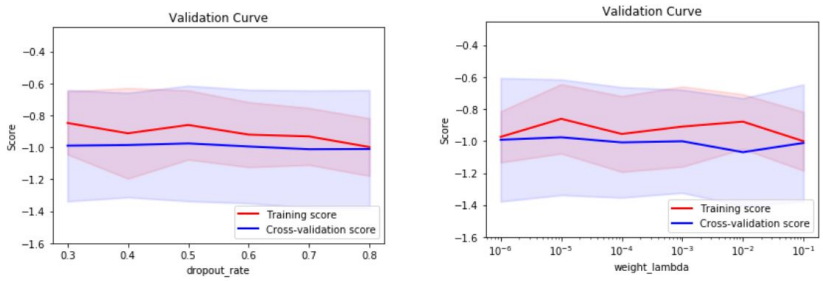

Figure 15: (a) Dropout Rate score. (b) L2 regularizer score.

3.4 Finalised Neural Network Model

The listing 20 show the finalised neural network model with all optimal hyperparameter listed in table 7.

```
# Finalized neural network model
def NN_finalized():
    # Create a sequential model
    model = Sequential()

    model.add(Dense(32, input_dim=32, kernel_initializer='
    he_normal', activation='relu', kernel_regularizer=l2(1e
    -05)))
    model.add(Dense(16, kernel_initializer='he_normal',
    activation='relu', kernel_regularizer=l2(1e-05)))
    model.add(Dense(8, kernel_initializer='he_normal',
    activation='relu', kernel_regularizer=l2(1e-05)))
    model.add(Dropout(0.5))
    model.add(Dense(1, kernel_initializer='he_normal'))

    # Display summary of the model
    model.summary()

    # Compile the model using ADAM (Adaptive learning rate
    optimization)
    optimizer = Adagrad(lr=0.2)
    model.compile(loss='mean_squared_error', optimizer=
    optimizer, metrics=['mse'])

    return model
```

Listing 20: Finalised Model

4 Evaluation

The finalized model is first used to train the dataset and evaluate the validation set. Then, it is used to evaluate the performances of estimator on the test dataset. The metrics used to measure the performances of the model are mean square error loss versus epochs and scatter plot of predicted throughput versus measured throughput.

4.1 Result and Discussion

Figure 16(a) show the model evaluation on test dataset. It can be seen that model does not perform very well on the validation set with mean squared

Hyperparameters	
Number of hidden layers	2
1st hidden layer neurons	16
2nd hidden layer neurons	8
Optimiser	Adagrad
Activation function	ReLU
Weight initialiser	He normal
Batch sizes	20
Number of epochs	100
Learning rate	0.2
Dropout rate	0.5
Lambda	$1e-5$

Table 7: Finalised model summary

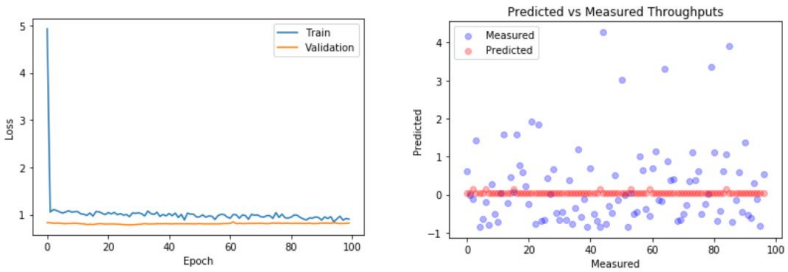

Figure 16: (a) Model evaluation on test data. (b) Model prediction on real data.

error of 1.131. Figure 16(b) show the scatter plot of predicted throughput versus real throughput. For a good predictor, the predicted output will fall along the diagonal axis. Unfortunately, the predicted output is very far off the real output and it indicate that poor performances of the model.

The major reason causing the poor performance of the model is the insufficient amount of dataset due to blocking request. From figure 17(a), the real throughput drawn online contain only 485 data and it violated the assumption made that data must follow normal distribution.

Insufficient amount of data will also lead to unrepresentative dataset. It means that the dataset does not capture the statistical characteristics relative to other dataset drawn from the same domain. This happen when the validation dataset is too small relative to training dataset.

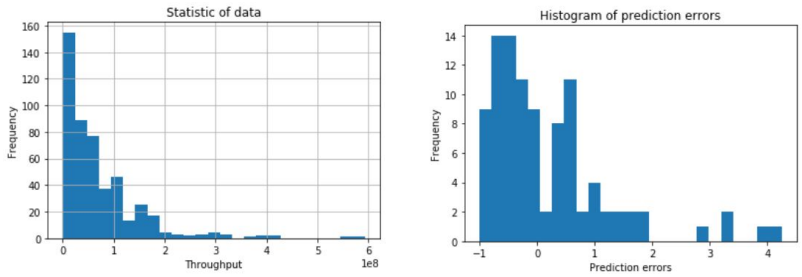

Figure 17: (a) Real throughput distribution. (b) Prediction error distribution.

4.2 Suggested Solution

4.2.1 Large Dataset

In this paper **??**, 30,000 randomly drawn IP latency pair are used to train the Support Vector Machine (SVM) regression model. To overcome the problem, the suggested way is to scrape more data manually from more countries to ensure an even geographical spread.

4.3 Future Improvement

4.3.1 Rank Correlation

Principal component analysis can be used to find the patterns of high dimensional data like 32-bit vector correlated to output values.

4.3.2 Convolutional Neural Networks (CNN)

1D CNN architecture can be explored for this regression problems.

5 Conclusion

To sum all up, the trained neural network model on this dataset is not ideal because of its large mean squared error. The major problem is because insufficient amount of training data and distribution of data is not normal. The suggested way of dealing this problem is to manually scrape more (IP, Throughput) pairs from range of geographical spread. Apart from that, several improvement are also mentioned for future experiment such as 1D CNN and SVM regression method.

References

[1] K. Hubbard, M. Kosters, D. Conrad, D. Karrenberg, and J. Postel, "Internet registry ip allocation guidelines," 01 1996.

[2] M. Danielson, L. Vanfretti, M. Almas, Y. Choompoobutrgool, and J. Gjerde, "Analysis of communication network challenges for synchrophasor-based wide-area applications," 08 2013.

[3] S. Ronaghan. (2018, July) Deep learning: Which loss and activation functions should i use? [Online]. Available: https://towardsdatascience.com/deep-learning-which-loss-and-activation-functions-should-i-use-ac02f1c56aa8

[4] D. Godoy. (2018, July) Hyper-parameters in action! part ii — weight initializers. [Online]. Available: https://towardsdatascience.com/hyper-parameters-in-action-part-ii-weight-initializers-35aee1a28404

[5] R. Beverly, K. Sollins, and A. Berger, "Svm learning of ip address structure for latency prediction," 01 2006, pp. 299–304.

[6] W. R. Stevens, "Tcp slow start, congestion avoidance, fast retransmit, and fast recovery algorithms," *RFC*, vol. 2001, pp. 1–6, 1997.

[7] M. Mirza, J. Sommers, P. Barford, and X. Zhu, "A machine learning approach to tcp throughput prediction," *IEEE/ACM Transactions on Networking*, vol. 18, no. 4, pp. 1026–1039, 2010.

[8] J. Brownlee. (2018, December) A gentle introduction to dropout for regularizing deep neural networks. [Online]. Available: https://machinelearningmastery.com/dropout-for-regularizing-deep-neural-networks