**Name: Rashmi Patil**

**Student Number: 16110226**

**Programme: MSc Internet Engineering**

Throughput Prediction

# Introduction:

In this assignment I aim to build a prediction system that predicts the throughput for any given url or an IP address from the internet.

Data was collected from Common crawl – A community that regularly crawls the web and collects the data in various formats. Data Is free for research purposes and I have downloaded samples from the dataset that was crawled in January 2020[1]

A CNN and Bi-LSTM architecture is trained using around 14k examples and tested on 3k examples. The results have a mean squared error (MSE) of around 0.05 on the final test set.

Following is the structure of the document. I talk about data collection, Reason for using python request library, model selection, overfitting and complexity then talk about evaluation and results.

Code base is store in git hub. Link to git hub is provided in **Appendix** section at the end.

# Data Collection:

Raw data is collected from Common crawl. Common crawl maintains a repository of web crawled data and is universally accessible for analysis.

I found this resource to be ideal for this project as I instantly will have access to a huge number of URLs. Common crawl crawls the web and creates an archive on a monthly basis.

I aimed at the data that was crawled and collected in the month of January 2020. Following the link to the same <https://commoncrawl.org/2020/02/january-2020-crawl-archive-now-available/>

This data has around “3.1Q billion web pages or 300 TB of uncompressed content, crawled between January 17th and 29th 2020. It includes page captures of 960 million URLs not contained in any crawl archive before.”

All the archived files are hosted on amazon s3 and download instructions are mentioned in the link above. Additionally, there is an index file <https://commoncrawl.org/2015/04/announcing-the-common-crawl-index/> with every repository that is created monthly. The index file is a small file, quick to download and has all the filenames related to that month's repository.

For this project I downloaded around 10 such files. Each file contains structured information of the following kind

ch,posterkoenig)/wp-content/uploads/2013/11/eigermoench\_hansuelikrapf\_wikimedia-670x320.jpg 20200121211044 {"url": "https://posterkoenig.ch/wp-content/uploads/2013/11/EigerMoench\_HansueliKrapf\_wikimedia-670x320.jpg", "mime": "image/jpeg", "mime-detected": "image/jpeg", "status": "200", "digest": "ND6O2S5S2EY6RXSK24GJPIJTTDAOLYJU", "length": "99813", "offset": "632080031", "filename": "crawl-data/CC-MAIN-2020-05/segments/1579250605075.24/warc/CC-MAIN-20200121192553-20200121221553-00054.warc.gz"}

The above structure has 3 parts, and the third part is the structured json that has required information. Following are the fields that are worth considering

* URL
* Status
* Mime-detected

“Mime-detected” field carries information whether the crawled URL is of type Image, Video or Pdf. There are few more types including the generic html, but I restricted to the three types mentioned above.

Writing a shell script was the easiest and fastest way to extract URLs of the mentioned types. With aim of extracting as many URLs as possible and equal amounts that cover all three types of URLs, I could extract 18654 records. Records are stored as /data/raw\_data.csv

# Reason for using python request Library:

URLs from the rawdata.csv was used and throughput was calculated for every IP address in the rawdata.csv.

I started off with using Socket programming. Following code snippet is added in git hub as “Socket.py”. However, this code was returning “404 bad request, header required” and also it wasn’t handling request to CDN (Content delivery network) and requests with “301 permanently moved error”

Issues with socket programming were delaying the data collection process and hence python requests library was used instead.

Requesturl\_bv.py file can be found on github.

This program performs following tasks:

* Import requests library to be able to use request function
* Perform request.get(url) on the list of URLs.
* Calculate the data length of the downloaded image, pdf or video file
* Track the elapsed time = (StartTime-EndTime) of the request. Throughput is calculated using data length and elapsed time. Throughput calculated is in Mega-bytes per second
* The Ip address is also converted to 32-bit vector for easy representation and for further use in neural network

In order to make sure the IP address follows tcp slow start. Multiple requests were made (around 5 requests) to see how throughput varies each time. It was observed that throughput increases slowly at the start and then drops slightly and becomes nearly constant. It was also observed that few Ip addresses follow the sawtooth pattern meaning throughput increases and decreases alternatively.This was performed in order to get a stabilised throughput value.

Following figure shows tcp slow start for one of the IP addresses

A picture containing kitchen, white

Description automatically generated

# Data analysis

Raw data has the following rows:

* IP address
* 32-bit vector
* Throughput in mb
* URL
* data length in bytes
* URL type: image or pdf or video.

Following is the data distribution:

A picture containing red, food

Description automatically generated

Throughputs that are above 80mb are clearly outliers and values between 40mb and 80mb are somewhat outliers compared to the values below 40mb.

Following is distribution of the data by url type:

|  |  |
| --- | --- |
| Pdf | 9120 |
| Image | 7707 |
| Video | 429 |
| Other | 1398 |

To get rid of outliers, I clean up the data and get rid of rows that have throughputs above 30 mb and below 1 mb. After clean-up we ignored 3697 rows out of 18654 that is around 20%. The new data has 14957 rows

A picture containing food, red

Description automatically generatedA close up of a logo

Description automatically generated

1. (b)

Figure (a) Throughput distribution after cleaning. (b) Density chart showing that throughput is denser between 0 and 10 and has a long tail.

Above figures show that distribution is non-trivial, with multiple bumps and a long tail. Creating neural architecture to fit the data looks challenging.

I call the cleaned dataset as “data.csv” and this becomes the total data set for the complete experiment.

"data.csv” is shuffled and split into 80% development set and 20% test set with following rows in the each set

* Development set: 11966 rows
* Test set: 2991 rows

Test set is set aside and will not be used to parameter tuning and model selection.

# Data normalisation:

Throughputs in “data.csv”(**can be found on git hub**) range between 1 and 30 mb, with this range a loss function is likely to be governed by a single large throughput. Normalization is essential so that each throughput value contributes equally to the loss of the regression model, I rescaled the throughputs in the range of 0 to 1. Rescaling is based on Min-Max feature scaling that turns a normal distribution to a distribution where the values range between 0 and 1.

This will help interpret the model predictions better and it helps gradient descent to converge much faster.

2 new files “devset\_normalized.csv” and “testset\_normalized.csv” are created ( **can be found on git hub**). These files will have an extra column that contains rescaled throughput values.

Model selection and evaluation is done on both versions of the files and I will explain more about the advantage of normalization on the loss function in the evaluation section.

# Model Selection:

I have tried the following model:

### **CNN with bidirectional LSTM followed by 3 dense layers**

A screenshot of text

Description automatically generated

The intuition behind using CNN-LSTM layer was that, Convolutional layer would act as feature extractor and LSTM layer would act as a sequence predictor. 4 digits of IP address are converted to 32-bit vector representation with 8 bits for each of the 4 numbers respectively. The 32-bit vector is fed into the above network as input data.

**Layer 1**: CNN layer is the 1-dimensional input layer that takes the 32 bits as input. This layer has a filter of size 16 and has 128 parameters. This totals to 32-16+1 \* 128 = 2176 parameters. Inputs-filters+1 \* number\_of\_parameters. I experimented with different filter settings ranging from 4, 8 and 16. Having 16 as the filter size gave me good results.

**Layer 2**: This a 1D max pooling layer that reduces the size of the representation thereby reducing the parameters in the network. This will also reduce the computations in the network. Again, I have experimented with different pooling sizes ranging from 2, 4 and 8. Having 8 as the pooling size gave me good results. Max pooling considers blocks of 8 parameters and takes the max value of the 8 parameters.

**Layer 3**: This layer is a Bidirectional-LSTM layer that would act as a sequence predictor. Ip address is sequential information and this layer is to learn that hidden sequence information. I used the standard suggested 'glorot\_uniform' initializer.

**Layers 4** to Layers 7: These are stack of Dense layers each with 160 parameters, and kernel initializers set to 'normal'.

**Layer 8**: This is a Dropout layer to help reduce the overfitting in the model. Without this layer, the model performed better on the training set and bad on the validation set. I will talk more about overfitting in the following sections. I have experimented with different Dropout settings ranging from 0.2, 0.4 and 0.5. Having 0.5 gave me better results.

**Layer 9**: This is a standard output layer with 1 parameter to predict and output the throughput predicted by the network.

**Activation Function**: 'Relu': I have used 'Relu' as the activation function. I have experimented with *tanh* and *Relu* activation functions and I found that 'Relu' is better compared to tanh as it gave me good results. Also 'Relu' accelerated SGD stochastic gradient descent to reach convergence faster. "ImageNet Classification with Deep Convolutional Neural Networks" [2].

**Loss**: I have used 'MSE' as the loss function as this is the function I want to reduce while training the network.

**Optimizer**: I have used 'adam' optimizer. I have tried with SGD Stochastic Gradient descent with different learning rates. Having ‘adam’ as the optimizer gave good results and also ‘adam’ automatically adjusts the learning rate as the learning progresses.

**Parameter size**: I have experimented with various parameter numbers on different layers. However, I have maintained the same parameter numbers in the hidden layers while I had a different number on the input layer. I have started with large numbers like 512 for all layers and reduced to 256, 190, 160 and 128. A large number was overfitting with low loss on training set and high loss of validation set. As I gradually reduced the numbers, I started seeing training loss and validation loss getting closer to each other. I stopped just before the validation loss went lower than training loss. Following section will depict the overfitting effect on one of the 10-fold cross validations.

### Overfitting and model complexity:

I started off with a very complex model with a large number of parameters to begin with. The model had 512 neurons in every layer except the output layer. I ran 25 epochs with batch size equal to 1. With this model settings, I got good results on the training set and bad results on the validation set.

Following picture will depict the training loss and validation loss on 25 epochs. The picture shows that by simplifying the model and reducing the number of neuron’s/parameters I got better results.

A picture containing white, water, table, man

Description automatically generatedA picture containing white, water

Description automatically generated

Image showing overfitting             overfitting is resolved

In the above images, the blue line is “training loss” and the orange line is “validation loss”. x-axis is the number of epochs and y-axis is the ‘mse’ loss. In the first image we can see that the blue line is far away from the orange line and as the epochs grow, the gap between the blue and orange grows. However, in the second image we can see that the blue line is close to the orange line and the gap between them is relatively low. As measures to reduce the overfitting, I have reduced the number of parameters from 512 to 250 and to 160 and then to 128. Also I have added an extra Dropout layer with 0.5% as the value. Both of these measures helped to simplify the model and fix the overfitting to some extent. Although there was an improvement in the overfitting effect, there was a reduction in the overall ‘mse’ loss.

10-fold cross validation:

10-fold cross validation was performed on “devset **(can be found on git hub**) and following settings were adjusted

* Number of parameters in the layers were changed from 512, 256, 192, 160 and to 128
* Number of epochs was changed from 50 to 10 and then set to 25.
* Loss remained the same and was set to ‘MSE’
* Learning rate: I started with SGD as the optimizer and set the values from 0.01 and 0.0001. However, using the ‘adam’ optimizer, I got good results and it also adjusts learning rate as training progresses
* Activation function was set to ‘Relu’. I tried ‘tanh’ and ended up using ‘relu’ as it gave me good results.

Following table depicts the mean squared error for 10 fold’s

|  |  |
| --- | --- |
| Cross validation number | Mean squared error |
| 1 | 0.04040479030808288 |
| 2 | 0.04197440368294558 |
| 3 | 0.03924952800737763 |
| 4 | 0.04260239671184757 |
| 5 | 0.03996116059147406 |
| 6 | 0.04154475790220107 |
| 7 | 0.04080199080481794 |
| 8 | 0.03735164575395976 |
| 9 | 0.04078529242611844 |
| 10 | 0.040870060522810656 |
| Average mean squared error | 0.04055460267 |

Following chart shows the predicted throughput values as compared to the actual throughput values. This chart is for one of the cross-validation outputs. The shaded or blurry part on the chart shows the points where the prediction values are exactly equivalent to the predicted output

A picture containing table, white, large, people

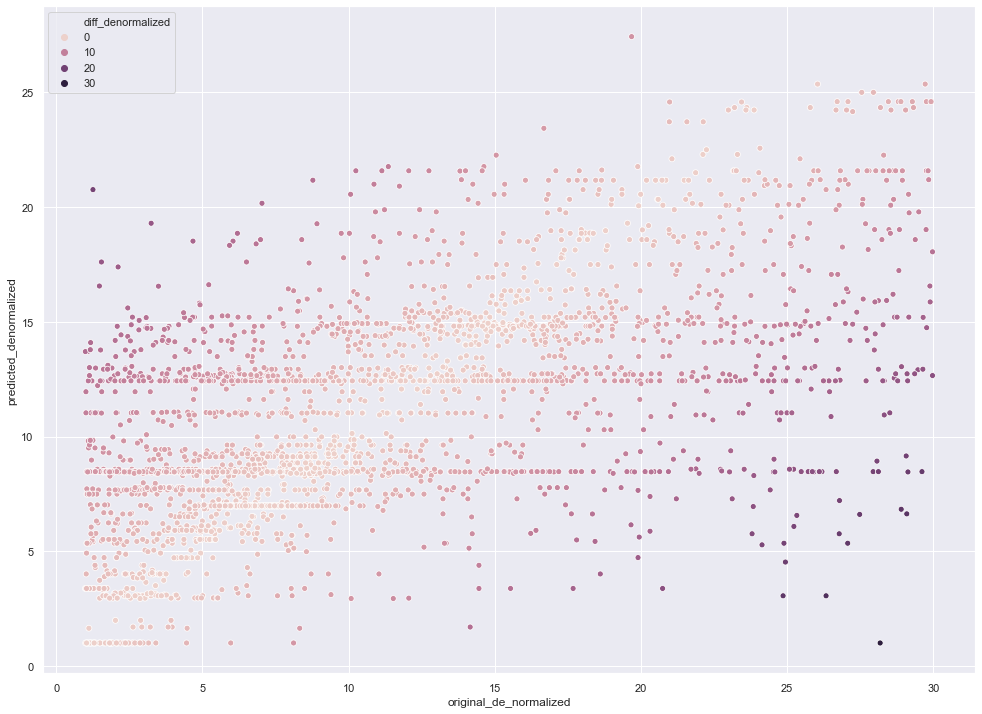
Description automatically generated

Evaluation and Results on test set:

Using all of the devset\_normalized.csv, model was created and saved to ../../models/model\_final.h5 location. Upon loading the saved file and test on held out test set “testset\_normalized.csv”( **can be found on git hub**) following results were seen

|  |  |
| --- | --- |
| Mean squared error | 0.04061568903523597 |

Following chart shows the actual v/s predicted chart on the final test set. Similar to the above chart, blurry points are the points where the predicted values are exactly equivalent to the actual values



Conclusion:

From the above experiments we can conclude that throughput prediction is a non-linear model and I was able to build a system that produces reasonable results.

`To run the code do the following

Cd src/test

Python predict.py ‘www.google.com’

This should print the output as

“throughput: 14.08505”

\*\*Also there is a jupyter notebook that will explain all of the above steps. It is available on git hub on the name: “sns-throughput-prediction.ipynb”

Appendix:

Data and source code for following can be found on git hub. Also there is a readme file to help with process execution

Git hub information is as follows:

Following source Code can be found here: [https://github.com/rashmi-patil-1492/sns-throughput-prediction](https://www.google.com/url?q=https://github.com/rashmi-patil-1492/sns-throughput-prediction&sa=D&source=hangouts&ust=1586731603730000&usg=AFQjCNGVh8VknR51rAvfNQjyJ2KlEx1NVw)

References:

[1] "January 2020 crawl archive now available – Common Crawl", *Commoncrawl.org*, 2020. [Online]. Available: https://commoncrawl.org/2020/02/january-2020-crawl-archive-now-available/. [Accessed: 11- Apr- 2020]

[2] A. Krizhevsky, I. Sutskever and G. Hinton, *ImageNet Classification with Deep Convolutional Neural Networks*. pdf [Online]. Available: http://www.cs.toronto.edu/~fritz/absps/imagenet.pdf. [Accessed: 11- Apr- 2020]