

PROJECT-I REPORT
ON
DDoS Detection Using Deep Learning

*SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENT FOR THE
AWARD OF THE DEGREE OF*

**BACHELOR OF ENGINEERING
(INFORMATION TECHNOLOGY)**



Supervisor

Prof. Krishan Saluja
Information Technology

Submitted by:

Sanchi Bhalla (UE178084)
Utkarsh Tiwari (UE178109)
Vastvikta Sandhir (UE178110)

To
Department of Information Technology
University Institute of Engineering and Technology
Panjab University, Chandigarh
2020

Declaration

We declare that the work embodied in this project hereby, titled “**DDoS Detection Using Deep Learning**”, forms our own contribution to the research work carried out under the guidance of **Prof. Krishan Saluja** and is a result of our own research work and has not been previously submitted to any other University for any other Degree/ Diploma to this or any other University.

Wherever reference has been made to previous works of others, it has been clearly indicated as such and included in the bibliography.

We hereby further declare that all information of this document has been obtained and presented in accordance with academic rules and ethical conduct.

Sincerely,

SANCHI BHALLA

UTKARSH TIWARI

VASTVIKTA SANDHIR

Acknowledgment

We take this opportunity to express our profound gratitude and deep regards to **Prof. Krishan Saluja** for his exemplary guidance, monitoring and constant encouragement throughout the course of this project. The blessing, help and guidance given by him from time to time has served as a constant source of motivation.

We also take this opportunity to express a deep sense of gratitude to **Ms. Meenakshi** and **Ms. Jyoti Chandel** for their cordial support, valuable information and guidance, which helped us in completing this task through various stages.

Lastly, we thank our parents and friends for their constant encouragement without which this project would not be possible.

Table of Contents

S. No.	Content	Page No.
1.	Introduction	5
2.	Technology Used	6-7
3.	Deep Learning Model Design <ul style="list-style-type: none"> ● Requirement Gathering ● Feasibility Study ● Proposed Methodology ● Project Architecture ● Implementation ● Testing 	8-30
4.	Results	30-32
5.	Future Scope	33
6.	Bibliography	34

Introduction

A Distributed Denial of Service (DDoS) attack is an attempt to make an online service unavailable by overwhelming it with traffic from multiple sources. They target a wide variety of important resources, from banks to news websites, and present a major challenge to making sure people can publish and access important information.

The number of DDoS attacks is increasing day by day. More than 929,000 DDoS attacks occurred in May 2020, representing the single largest number of attacks ever seen in a month. 4.83 million DDoS attacks occurred in the first half of 2020, a 15% increase. The DDoS attack frequency jumped 25% during peak pandemic lockdown months (March through June, 2020).

There are many types of DDoS attack schemes that are used today and they are steadily becoming more sophisticated. However, their common goal is to overwhelm targeted network resources with traffic or requests for service from many different sources – potentially hundreds of thousands or more. This effectively makes it impossible to stop the attack simply by identifying and blocking a single IP address. The sheer distribution of attacking sources also makes it very difficult to distinguish legitimate user traffic from attack traffic when spread across so many points of origin.

The first step in avoiding or stopping a DDoS attack is knowing that an attack is taking place. To detect an attack, one has to gather sufficient network traffic information, then perform analysis to figure out if the traffic is a friend or foe.

DDoS detection is the key to quickly stopping or mitigating attacks and in order for this to happen, two success criteria need to be met: 1) speed of detection and 2) accuracy of detection. So detection methods are a key consideration in formulating a strong DDoS defense.

This project is a DDoS detection method based on deep learning. It is a Convolutional Neural Network (CNN), a deep learning based model which is capable of identifying if an incoming data packet is a benign or an attack packet. Not only this, it also detects the type of DDoS attack (Syn, Portmap, UDP and UDPlag). Hence, it is an efficient as well as accurate way of detecting DDoS attacks which can help prevent them by taking timely action to safeguard the target system or website.

Technology Used

Deep Learning is a subset of Machine Learning, which in itself is a subset of Artificial Intelligence, a broad Computer Science subject. Deep Learning is inspired by the structure and function of the brain, using algorithms that mimic the working of neural networks in our brain. Thus these algorithms are often called Artificial Neural Networks. Deep Learning can be implemented using various methods like Fully Connected Neural Networks, Convolutional Neural Networks, Recurrent Neural Network, Long Short-Term Memory Networks and many more. This project is centered around Deep Learning based Convolutional Neural Network Model (CNN).

Convolutional Neural Network (CNN)

Convolutional neural networks differ from other artificial neural networks as they treat data as spatial. Instead of neurons being connected to every neuron in the previous layer, they are instead only connected to neurons close to it and all have the same weight. This simplification in the connections means the network upholds the spatial aspect of the data set.

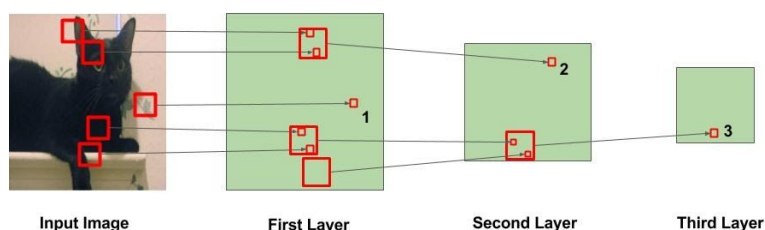


Diagram showing image classification using CNN

As clear from the above diagram CNN uses feature extraction for image classification. The different layers used to develop a CNN model are further explained as this report progresses. Although image classification is the most popular use case of CNN, this project uses it for data classification.

Libraries and Tools Used

To implement CNN we need different tools. Our entire project is built on Python Programming Language. We have used Google Colab (as our code editor) and Python libraries such as Pandas, Numpy, Matplotlib, Scikit Learn and Keras. A brief description of each is provided below.

Google Colab

Colaboratory, or “Colab” for short, is a product from Google Research. Colab allows anybody to write and execute arbitrary python code through the browser, and is especially well suited to machine learning, data analysis and education. More technically, Colab is a hosted Jupyter notebook service that requires no setup to use, while providing free access to computing resources including GPUs.

Pandas

Pandas is a software library written for the Python programming language for data manipulation and analysis. In particular, it offers data structures and operations for manipulating numerical tables and time series.

Numpy

Numpy is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.

Scikit Learn

Scikit-learn is a free software machine learning library for the Python programming language. It features various classification, regression and clustering algorithms including support vector machines, random forests, gradient boosting, k -means and DBSCAN, and is designed to interoperate with the Python numerical and scientific libraries NumPy and SciPy.

Matplotlib

Matplotlib is a plotting library for the Python programming language and its numerical mathematics extension NumPy. It provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits like Tkinter, wxPython, Qt, or GTK+.

Keras

Keras is a minimalist Python library for deep learning that can run on top of Theano or TensorFlow. It was developed to make implementing deep learning models as fast and easy as possible for research and development.

Requirement Gathering

To build a dependable Deep Learning Model we first need a reliable dataset. For that we were fortunate to access the CIC-DDoS 2019 dataset. CIC is the Canadian Institute for Cybersecurity, University of New Brunswick. This dataset has samples collected over a span of two days.

The CIC-DDoS 2019 Dataset

The CIC-DDoS 2019 dataset contains benign and the most up-to-date common DDoS attacks, which resembles the true real-world data. The dataset was created over a span of 2 days and contains different types of modern DDoS attacks such as UDP, UDP-Lag, Portmap, MSSQL, NetBIOS, LDAP, SYN, SNMP, NTP, TFTP and DNS.

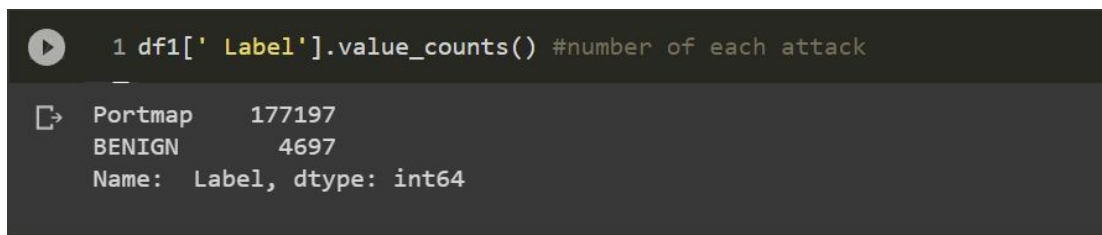
For creating this dataset, raw data including the network traffic and event logs per machine were recorded. For feature extraction from raw data, CICFlowMeter-V3 was used and more than 80 traffic features were extracted.

In our project, we used the **Portmap Dataset (Binary Classification)** and **UDPLag Dataset (Multi Classification)**.

- **The Portmap Dataset :**

Portmap Attack : In 2015, the Port mapper service was discovered to be used in DDoS attacks. By using a spoofed port mapper request, an attacker can amplify the effects on a target because a portmap query will return many times more data than in the original request. Portmapper DDoS amplification works in a similar way to other known amplified (or reflective) DDoS attacks that use standard UDP accessible internet services.

Dataset Composition :

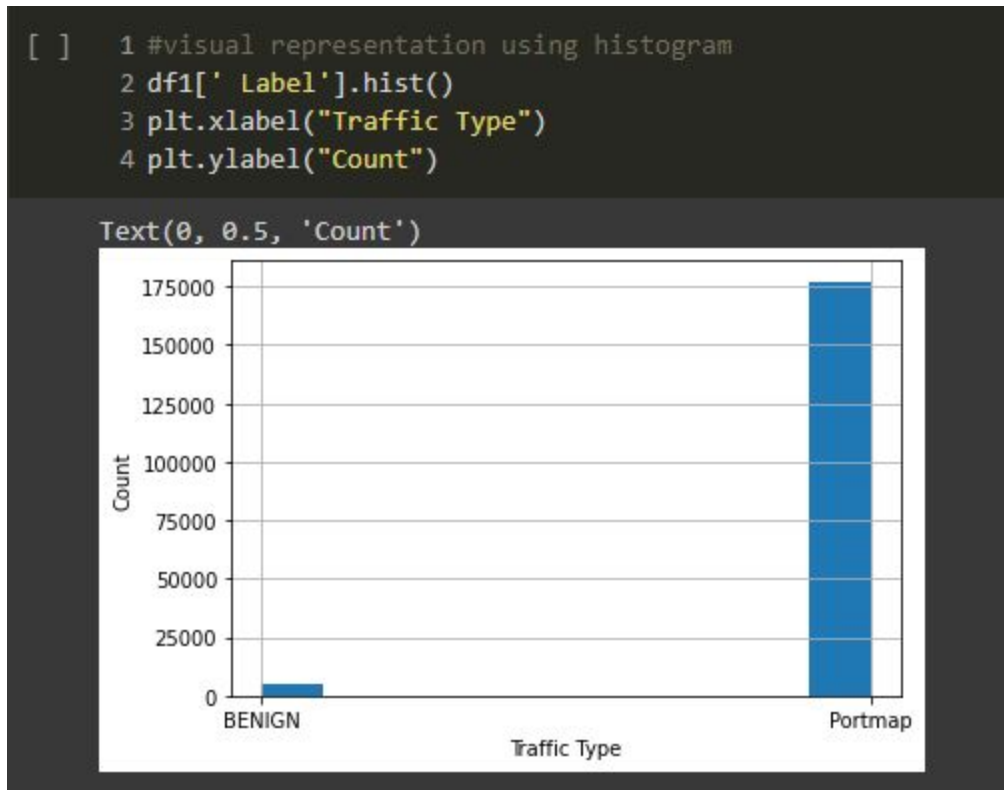


```
1 df1['Label'].value_counts() #number of each attack
```

Portmap	177197
BENIGN	4697

Name: Label, dtype: int64

The count of each type of traffic in the Portmap Dataset



Graph showing the count of each type of traffic in Portmap Dataset

- **The UDPlag Dataset :**

UDPLag Attack : The UDP-Lag attack is that kind of attack that disrupts the connection between the client and the server. This attack is mostly used in online gaming where the players want to slow down/interrupt the movement of other players to outmaneuver them. This attack can be carried in two ways, i.e. using a hardware switch known as a lag switch or by a software program that runs on the network and hogs the bandwidth of other users.

Dataset Composition :

```
[21] 1 df2['Label'].value_counts() #number of each attack
```

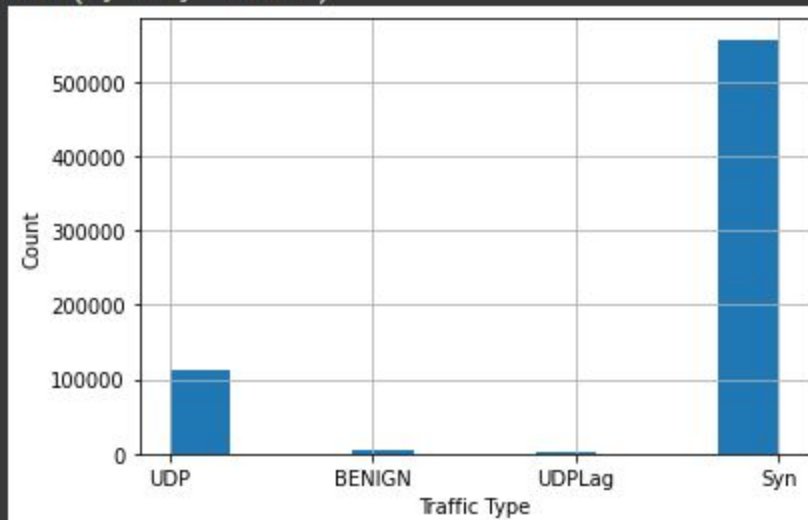
Syn	556755
UDP	111819
BENIGN	4016
UDPLag	1873

Name: Label, dtype: int64

The count of each type of traffic in the UDPLag Dataset

```
[ ] 1 #visual representation using histogram  
2 df2['Label'].hist()  
3 plt.xlabel("Traffic Type")  
4 plt.ylabel("Count")
```

Text(0, 0.5, 'Count')



Graph showing the count of each type of traffic in UDPLag Dataset

- **The Combined Dataset :** The Combined dataset was obtained by merging the Portmap dataset and the UDPLag dataset. Hence, the combined data has 5 types of packets: Benign, Portmap, UDP, UDPLag and Syn.

Dataset Composition :

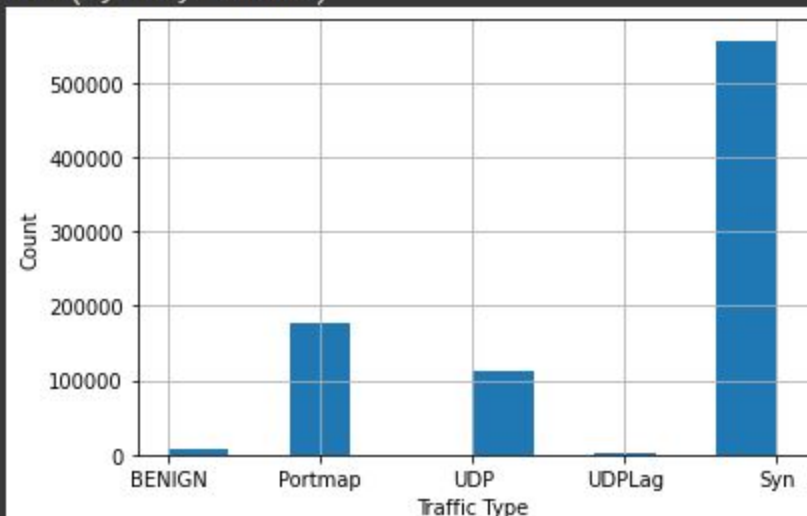
```
[44] 1 df['Label'].value_counts() #number of each attack
```

```
Syn          556755
Portmap      177197
UDP          111819
BENIGN        8713
UDPLag        1873
Name: Label, dtype: int64
```

The count of each type of traffic in the Combined Dataset

```
1 #visual representation using histogram
2 df['Label'].hist()
3 plt.xlabel("Traffic Type")
4 plt.ylabel("Count")
```

```
Text(0, 0.5, 'Count')
```



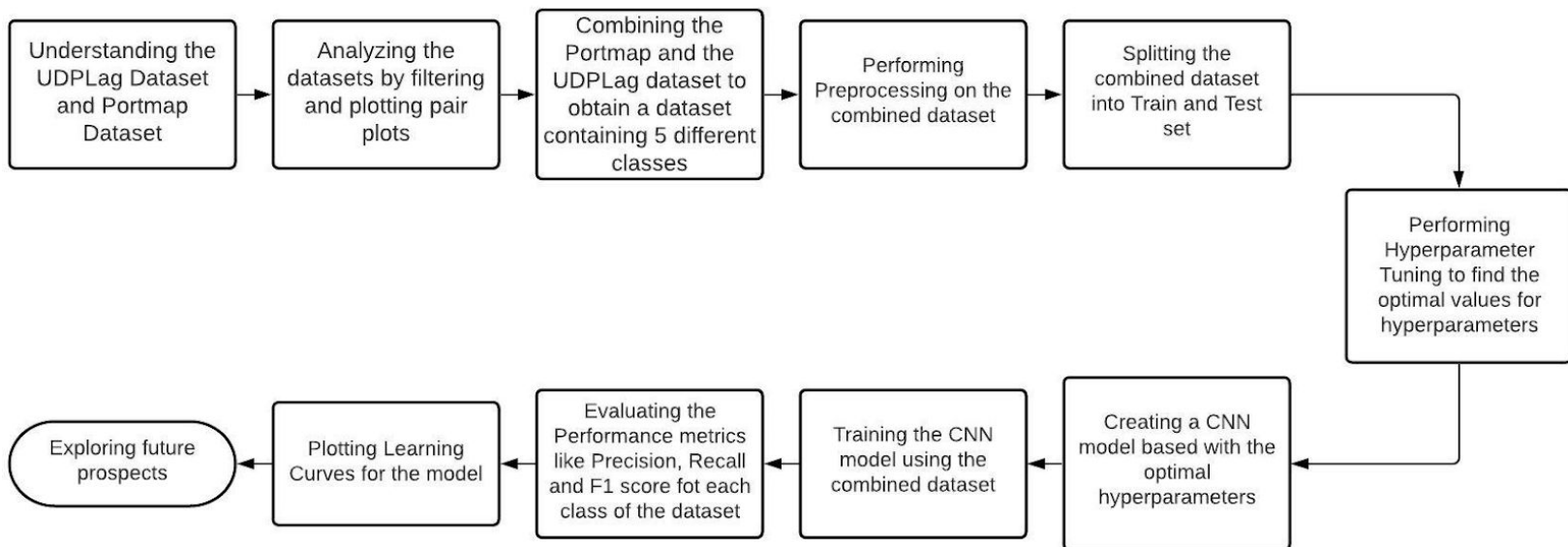
The graph showing the count of each type of traffic in the Combined Dataset

Feasibility Study

Feasibility studies are an important project planning tool that can help identify points of failure in a project before any money or time gets invested. They are particularly useful for machine learning projects because machine learning projects are generally experimental in nature. They can fail due to many reasons some of which can be clearly identified beforehand via a feasibility study. The feasibility study for our project included finding an appropriate labeled dataset which could be used for classification of network traffic.

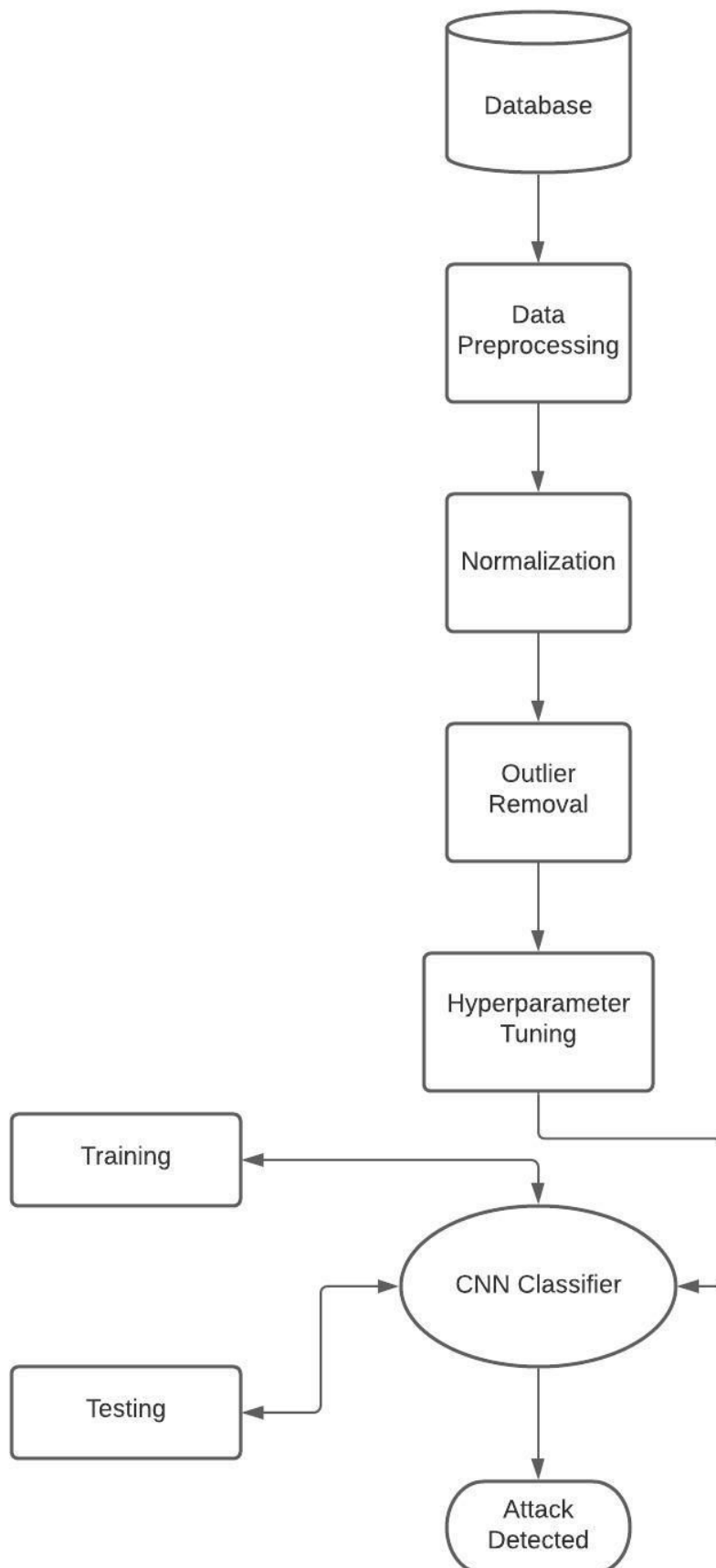
Detecting DDoS attacks using a CNN model requires a dataset which contains at least one feature that can be predicted using the other features. We were able to find the CICDDoS 2019 dataset, a labeled dataset. Since the dataset is labeled, we were able to use multi-classification on the dataset to predict the type of traffic.

Proposed Methodology



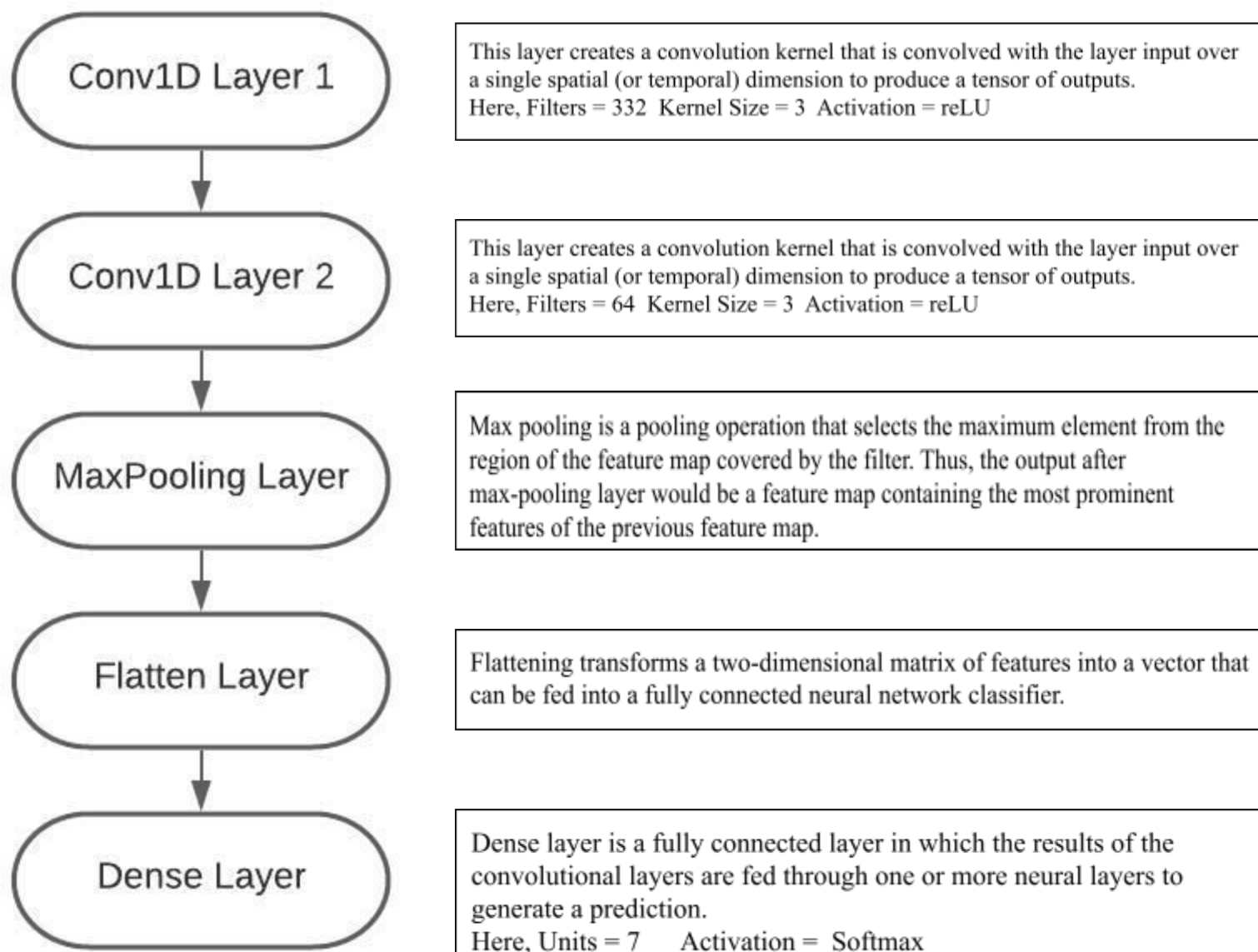
Flowchart displaying the proposed methodology for our project

Project Architecture



The architecture for DDoS Attack detection using CNN model

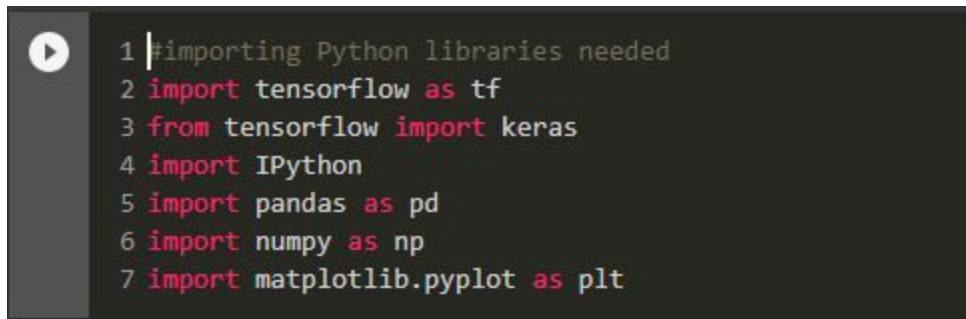
Model Architecture



The CNN model was obtained after hyperparameter tuning search for optimal hyperparameters. The CNN model consists of 5 layers as mentioned above.

Implementation

1. First we imported the necessary Python and few open-source libraries. These include: Pandas, Numpy, Matplotlib, Scipy, Scikit-Learn, Seaborn, Keras and TensorFlow.



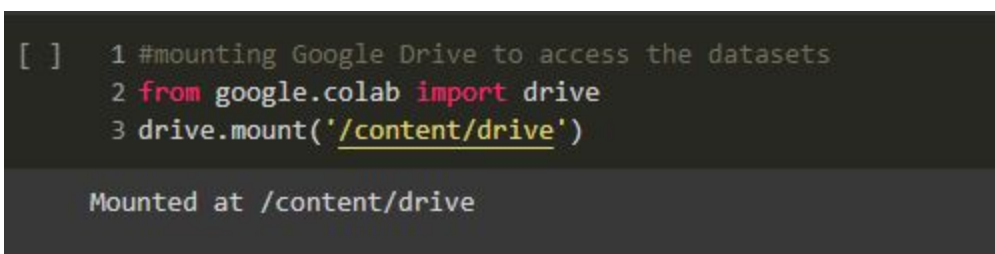
```

1 #importing Python libraries needed
2 import tensorflow as tf
3 from tensorflow import keras
4 import IPython
5 import pandas as pd
6 import numpy as np
7 import matplotlib.pyplot as plt

```

Some of the basic Python and Open-source libraries imported

2. We mounted the Google drive containing the UDPLag and Portmap datasets. Then we stored these two datasets in separate dataframes. Later, to make a cumulative dataset we combined these two datasets and obtained the resultant **Combined Dataset**. We used this Combined Dataset throughout the project.



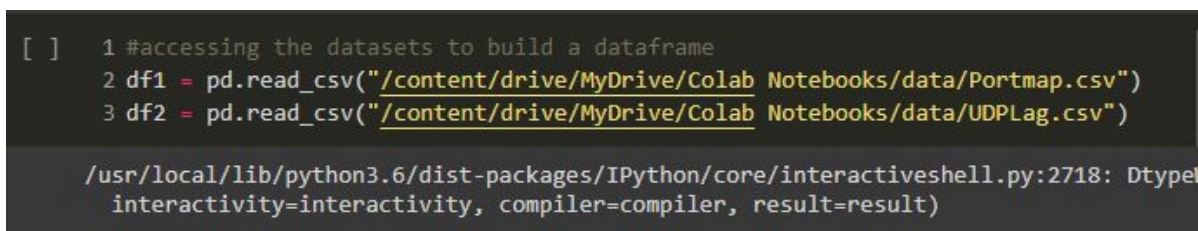
```

[ ] 1 #mounting Google Drive to access the datasets
    2 from google.colab import drive
    3 drive.mount('/content/drive')

```

Mounted at /content/drive

Mounted the Google Drive containing required datasets



```

[ ] 1 #accessing the datasets to build a dataframe
    2 df1 = pd.read_csv("/content/drive/MyDrive/Colab Notebooks/data/Portmap.csv")
    3 df2 = pd.read_csv("/content/drive/MyDrive/Colab Notebooks/data/UDPLag.csv")

```

/usr/local/lib/python3.6/dist-packages/IPython/core/interactiveshell.py:2718: DtypeError: Could not convert the 'Dtype' object to a NumPy array. (interactivity=interactivity, compiler=compiler, result=result)

Loading the datasets into Colab and then storing them in two separate dataframes

Using Combined Dataset containing both Portmap and UDPLag Dataset

```
[ ] 1 df = pd.concat([df1, df2], axis=0)
```

*Dataframe **df** storing the combined dataset*

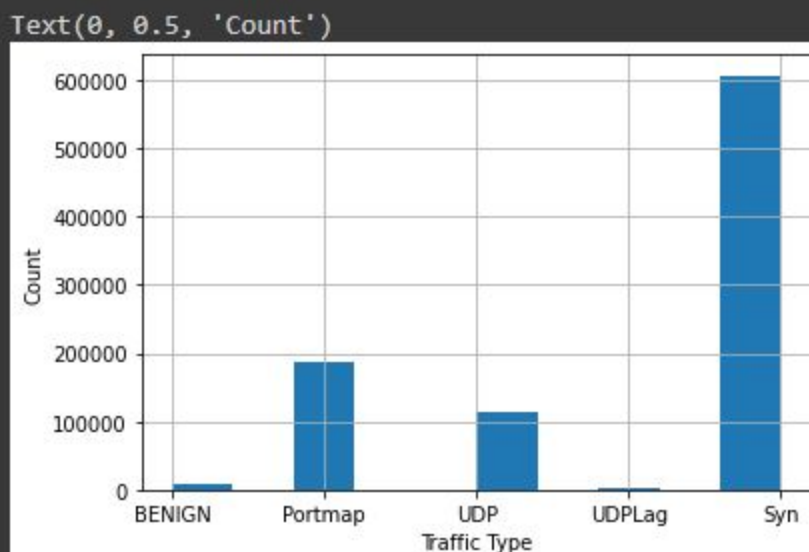
3. We found the value counts of the different attack types present in the combined dataset and plotted histograms to visualize the dataset composition.

```
[ ] 1 df['Label'].value_counts() #number of each attack
```

Syn	606749
Portmap	186960
UDP	112475
BENIGN	8802
UDPLag	1873
Name: Label, dtype: int64	

*The combined dataset contains **5 different classes** : Syn, Portmap, UDP, Benign and UDPLag*

```
[ ] 1 df['Label'].hist() #visual representation using histogram
2 plt.xlabel("Traffic Type")
3 plt.ylabel("Count")
```



A histogram visualizing the combined dataset

4. We performed **Data Preprocessing**, in various steps as follows:

- **Dropping values** : Removed abnormally large and null values and dropped all the columns with duplicate and non numerical values. After removing these columns and rows we were left with **82 columns** and **856357 rows**.

```
[ ] 1 df = df.replace([np.inf, -np.inf], np.nan) #replacing abnormally large values with nan
     2 df.dropna(inplace=True) #dropping all nan values
     3 df=df.drop_duplicates() #dropping all duplicates
     4 df.shape

(856357, 88)
```

Dropping the abnormally large, null and duplicate values

```
[ ] 1 #dropping all non-numerical and repetitive features
     2 df=df.drop_duplicates()
     3 df=df.drop(' Destination IP', axis=1)
     4 df=df.drop(' Source IP', axis=1)
     5 df=df.drop('Flow ID', axis=1)
     6 df=df.drop(' Timestamp', axis=1)
     7 df=df.drop('Unnamed: 0', axis=1)
     8 df=df.drop('SimillarHTTP', axis=1)
     9 df.shape

(856357, 82)
```

Dropping all non-numerical and repetitive features

- **Label encoding** : The names of the types of traffic are the labels of the dataset. These names were character strings and hence the model wouldn't accept them as input. Therefore we assigned them numeric labels as follows:

Label	Traffic Type
0	Benign
1	Portmap
2	Syn
3	UDP
4	UDPLag

```

1 print(df['Label'].unique())

['BENIGN' 'Portmap' 'UDP' 'UDPLag' 'Syn']

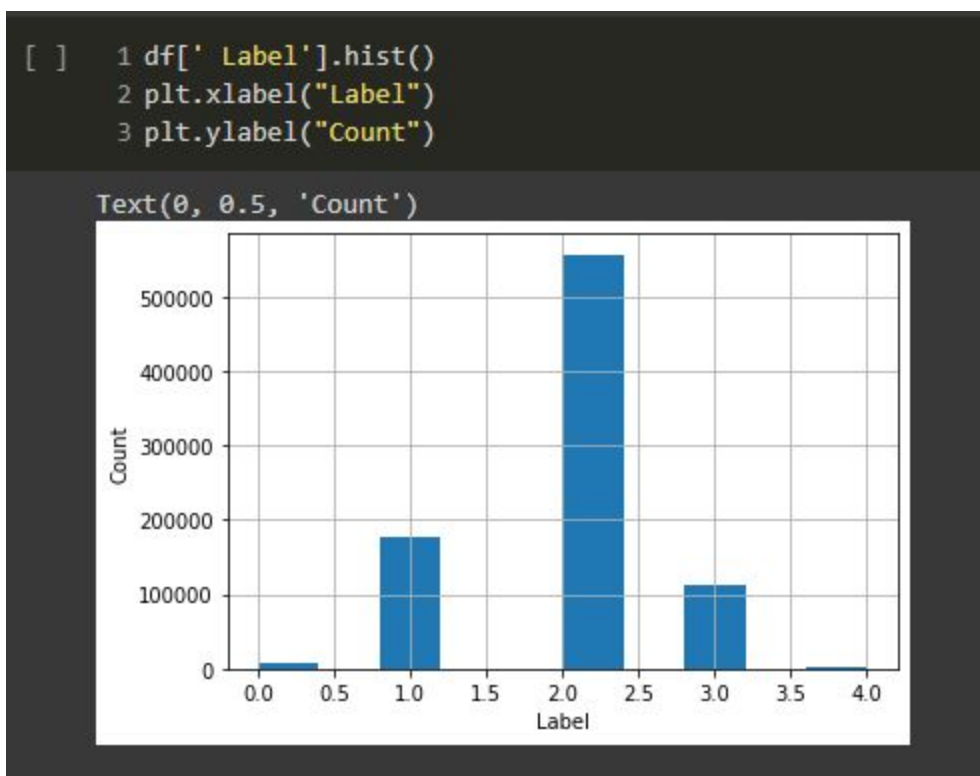
[ ] 1 from sklearn.preprocessing import LabelEncoder
     2 le = LabelEncoder()
     3 df['Label'] = le.fit_transform(df['Label'])

[ ] 1 print(df['Label'].unique())

[0 1 3 4 2]

```

Converting Labels from character strings into numeric values



Histogram showing the composition of the combined dataset in terms of new numeric labels

- **Normalization** : We used **MinMax Scaler** to normalize input features/variables. Variables that are measured at different scales do not contribute equally to the model fitting & model learned function and might end up creating a bias. MinMax Scaler transforms all features into the **range [0,1]** meaning that the minimum and maximum value of a feature/variable is going to be 0 and 1, respectively. MinMax scaling was carried out using **MinMaxScaler function of scikit-learn library**.

```
[ ] 1 from sklearn.preprocessing import MinMaxScaler
     2 scaler = MinMaxScaler()
     3 scaled = scaler.fit_transform(df)
```

Performing normalization of dataset using MinMax Scaler

- **Removing Outliers** : We removed the outliers from the dataset using z-score. A **z-score** (also called a standard score) gives an idea of how far from the mean a data point is. If the z score of a data point is more than a threshold value, it indicates that the data point is quite different from the other data points. Such a data point can be an outlier. We first detected the outliers and then removed them from the dataset.

```
[ ] 1 from scipy import stats
     2 z = np.abs(stats.zscore(df))
     3 print(z)
```

```
[[ 1.39681215  1.71367728  1.87036322 ...  2.55290522  9.98501346
   3.10112665]
 [ 1.39681215  1.71367728  1.87036322 ...  2.48226662  9.98501346
   3.10112665]
 [ 1.08917726  1.69040808  0.72107033 ... 11.026001   9.98501346
   3.10112665]
 ...
 [ 0.33320493  1.17149263  0.72107033 ...  0.25278582  0.10015009
   0.15007374]
 [ 0.3332503   0.93817038  0.72107033 ...  0.25278582  0.10015009
   0.15007374]
 [ 0.33329566  0.38932206  0.72107033 ...  0.25278582  0.10015009
   0.15007374]]
```

In z variable we stored the z-score of every data point of the combined dataset

```
[ ] 1 arr=np.unique(np.where(z>10)[0])
      2 arr

array([ 2, 3, 4, ..., 853032, 853879, 856275])

[ ] 1 len(arr)

18620
```

*We created an array of all the data points having z-score greater than the threshold value, we set the threshold value equal to 10
There were total **18620 data points** in the combined dataset that were detected as outliers and removed*

```
[ ] 1 df_out = df.loc[~df.index.isin(l1)]
```

*We created a new data frame **df_out** which contains the original combined dataset minus the outliers*

```
[ ] 1 df.shape

(856357, 82)

[ ] 1 df_out.shape

(833625, 82)
```

*The new data frame created after removing the outliers contains **833625 rows and 82 columns***

```

1 df_out['Label'].value_counts() #new number of each attack with labels

2 545095
3 175056
4 106509
5 5210
6 1755
Name: Label, dtype: int64

```

The new count for each label

- 5. Splitting into Training and Testing data :** X is the input dataset/variables/features we are using. y is the output variable/label. 82 columns are under x. **80% of the dataset is for training and 20% for testing.**

```

[ ] 1 X = df_out.iloc[:, :81]
     2 y = df_out.iloc[:, -1]
     3 X=X.values
     4 X = X.reshape(X.shape[0], X.shape[1], 1)
     5 y=y.values
     6 from sklearn.model_selection import train_test_split
     7 X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.2,random_state=101)

1 X_train.shape
(666900, 81, 1)

[ ] 1 y_train.shape
(666900,)

```

This snippet shows the code for splitting the dataset into test and train set and also depicts the shape of the training set

- 6. Balancing the dataset :** The combined dataset has very few training examples for the UDPLag attack i.e. label 4 as compared to the other labels. As a result of this, the dataset becomes imbalanced and hence the predictions made by the model trained on this imbalance dataset will not be accurate. To resolve this issue, we balanced the dataset using the **SMOTE algorithm**.

SMOTE (synthetic minority oversampling technique) is one of the most commonly used oversampling methods to solve the imbalance problem. It aims to balance class distribution by randomly increasing minority class examples by replicating them. SMOTE synthesises new minority instances between existing minority instances. It generates the **virtual training records by linear interpolation** for the minority class. These synthetic training records are generated by randomly selecting one or more of the k-nearest neighbors for each example in the minority class. After the oversampling process, the data is reconstructed and several classification models can be applied for the processed data.

```
[ ] 1 from imblearn.over_sampling import SMOTE
     2 smote=SMOTE('minority')
     3 X_sm, y_sm=smote.fit_sample(X_train,y_train)
     4 print(X_sm.shape,y_sm.shape)

/usr/local/lib/python3.6/dist-packages/sklearn/externa
"(https://pypi.org/project/six/).", FutureWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/utils/d
warnings.warn(message, FutureWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/utils/d
warnings.warn(msg, category=FutureWarning)
(1101456, 81) (1101456,)
```

X_sm and y_sm are the new balanced training sets

```
[ ] 1 X_train = np.reshape(X_train, (X_train.shape[0],X_train.shape[1],1))
     2 print(X_train.shape)

(666900, 81, 1)

[ ] 1 X_sm = np.reshape(X_sm, (X_sm.shape[0],X_sm.shape[1],1))
     2 print(X_sm.shape)

(1101456, 81, 1)
```

*The earlier training set contained 666900 rows but after using SMOTE algorithm on the training set, the new training set has **1101456** rows*


```

1 [unique, counts] = np.unique(y_sm, return_counts=True)
2 frequencies = np.asarray((unique, counts)).T
3 frequencies

array([[ 0, 4181],
       [ 1, 139909],
       [ 2, 435991],
       [ 3, 85384],
       [ 4, 435991]])

```

The new label wise count after implementing SMOTE algorithm to create balanced training sets

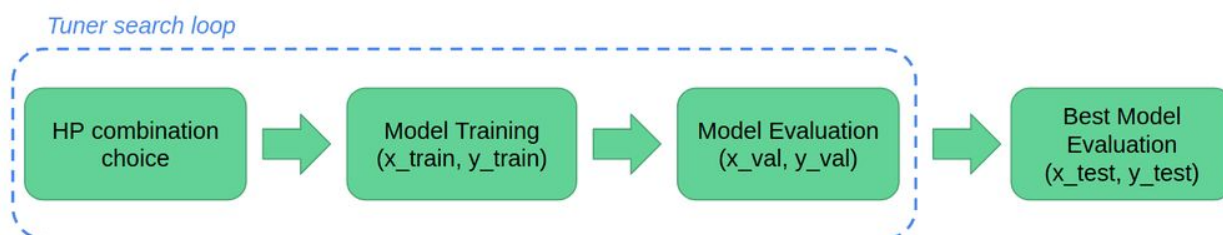
7. Hyperparameter Tuning :

A Machine Learning model is defined as a mathematical model with a number of parameters that need to be learned from the data. By training a model with existing data, we are able to fit the model parameters.

However, there are other kinds of parameters, known as **Hyperparameters**, that cannot be directly learned from the regular training process. They are usually fixed before the actual training process begins. These parameters express important properties of the model such as its complexity or how fast it should learn.

We performed hyperparameter tuning using **Keras Tuner** which is a library to easily perform hypertuning using Tensorflow 2.0.

How Keras Tuner works :



Hyperparameter tuning process with Keras Tuner

First, a tuner is defined. Its role is to determine which hyperparameter combinations should be tested. The library search function performs the iteration loop, which evaluates a certain number of hyperparameter combinations. Evaluation is performed by computing the trained model's accuracy on a held-out validation set. Finally, the best hyperparameter combination in terms of validation accuracy can be tested on a held-out test set.

We created a **sequential** model consisting of **5 layers** which are as follows:

- 1D Convolution Layer 1 : Consists of 322 filters and **reLU** activation function
- 1D Convolution Layer 2 Consists of 64 filters and **reLU** activation function
- Pooling Layer : We used **MaxPooling 1D** layer
- Flatten Layer : It flattens the input without affecting the batch size
- Dense Layer : We used dense layer with **Softmax** as activation function because we have multiple classes

```
[ ] 1 def model_builder(hp):
2     model = keras.Sequential()
3
4     # Tuning the number of units in the first Convolution layer
5     hp_units = hp.Int('units', min_value = 300, max_value = 400, step = 32)
6     model.add(keras.layers.Conv1D(hp_units, 3, activation = 'relu', input_shape=(81,1)))
7     model.add(keras.layers.Conv1D(64, 3, activation = 'relu'))
8     model.add(keras.layers.MaxPooling1D())
9     model.add(keras.layers.Flatten())
10    model.add(keras.layers.Dense(7, activation='softmax'))
11
12    # Tuning the learning rate for the optimizer
13    # Choosing an optimal value from 0.01, 0.001, or 0.0001
14    hp_learning_rate = hp.Choice('learning_rate', values = [1e-2, 1e-3, 1e-4])
15
16    model.compile(optimizer = keras.optimizers.Adam(learning_rate = hp_learning_rate),
17                  loss = keras.losses.SparseCategoricalCrossentropy(from_logits = True),
18                  metrics = ['accuracy'])
19
20    return model
```

*Defined the hypermodel using **model_builder** function*


```
[ ] 1 tuner = kt.Hyperband(model_builder,
2                           objective = 'val_accuracy',
3                           max_epochs = 10,
4                           factor = 3,
5                           directory = 'my_dir',
6                           project_name = 'intro_to_kt')
```

*Instantiated the tuner to perform hypertuning. We used the **Hyperband tuner**. The Hyperband tuning algorithm uses adaptive resource allocation and early-stopping to quickly converge on a high-performing model.*

```
[ ] 1 class ClearTrainingOutput(tf.keras.callbacks.Callback):
2     def on_train_end(*args, **kwargs):
3         IPython.display.clear_output(wait = True)
```

Defined a callback to clear the training outputs at the end of every training step

```
1 tuner.search(X_sm, y_sm, epochs = 10, validation_data = (X_test, y_test), callbacks = [ClearTrainingOutput()])
2
3 # Getting the optimal hyperparameters
4 best_hps = tuner.get_best_hyperparameters(num_trials = 1)[0]
5
6 print(f"""
7 The hyperparameter search is complete. The optimal number of units in the first Convolution
8 layer is {best_hps.get('units')} and the optimal learning rate for the optimizer
9 is {best_hps.get('learning_rate')}.
10 """)
```

Running the hyperparameter search

```
[ ] Trial 11 Complete [00h 45m 41s]
    val_accuracy: 0.9357745051383972

    Best val_accuracy So Far: 0.9824801087379456
    Total elapsed time: 07h 27m 19s
    INFO:tensorflow:Oracle triggered exit

    The hyperparameter search is complete. The optimal number of units in the first Convolution
    layer is 332 and the optimal learning rate for the optimizer
    is 0.0001.
```

*Results of the hyperparameter tuning. The **best val_accuracy** comes out to be **0.982480** obtained after **11 trials**.*

*The optimal number of **filters for the first conv1D layer** is **332** and the optimal **learning rate for the optimizer** is **0.0001***

8. Early Stopping : Too many epochs can lead to overfitting of the training dataset, whereas too few may result in an underfit model. Early stopping is a method that allows you to specify an arbitrary large number of training epochs and stop training once the model performance stops improving on a hold out validation dataset. This training will automatically stop once the chosen metric stops improving.

We used **val_loss** as the chosen metric for early stopping.

```
1 from tensorflow.keras.callbacks import EarlyStopping
2 early_stop = EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience=2)
```

*Created a callback **early_stop***

9. Training the model : We trained the model with the hyperparameters obtained by the hypertuning search with 80% of our dataset. We used **X_test** and **y_test** as the **validation_data** to check the effectiveness of the training.

```
1 # Building the model with the optimal hyperparameters and training it on the dataset
2 model = tuner.hypermodel.build(best_hps)
3 history=model.fit(X_sm, y_sm, epochs = 10, validation_data = (X_test, y_test), callbacks = early_stop)
```

Trained the model obtained after the hyperparameter tuning search

10. Prediction : We obtained a confusion matrix for actual and predicted values of the labels.

```
[ ] 1 pred = model.predict(X_test)
    2 pred_y = pred.argmax(axis=-1)
```

Making prediction using the trained model

11. Finding Precision, Recall and f1-score for each class : The quantities necessary for calculating Precision and Recall are true positives, false negatives and false positives.

True Positives : A true positive is an outcome where the model *correctly* predicts the *positive* class.

False Positives : A false positive is an outcome where the model *incorrectly* predicts the *positive* class.

False Negatives : A false negative is an outcome where the model *incorrectly* predicts the *negative* class.

- **Precision :** Precision is defined as the number of true positives divided by the number of true positives plus the number of false positives. Precision expresses the proportion of the data points our model says was relevant actually were relevant.

$$\text{precision} = \frac{\text{true positives}}{\text{true positives} + \text{false positives}}$$

- **Recall :** The precise definition of recall is the number of true positives divided by the number of true positives plus the number of false negatives. Recall expresses the ability to find all relevant instances in a dataset.

$$\text{recall} = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}}$$

- **F1 score :** The F1 score is the harmonic mean of precision and recall taking both metrics into account in the following equation:

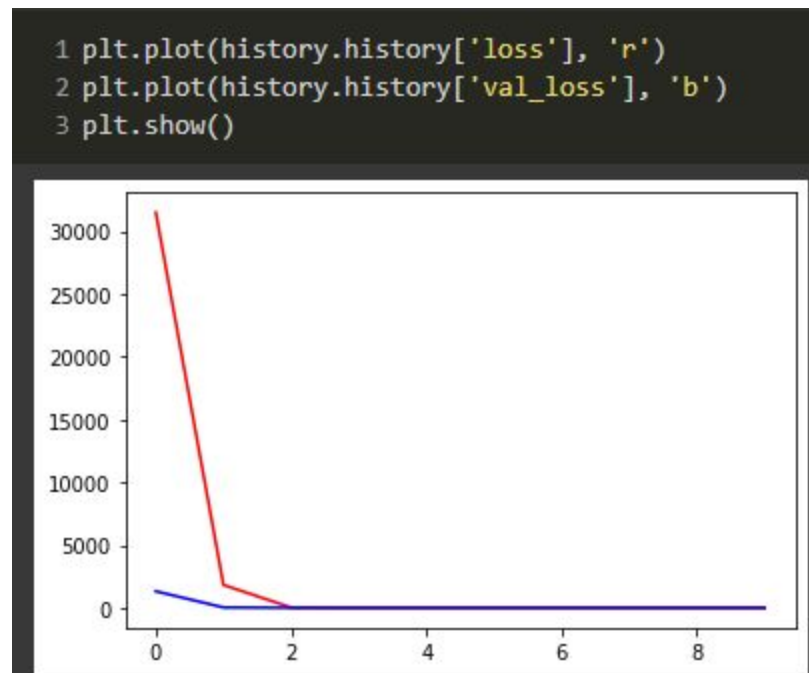
$$F_1 = 2 * \frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}}$$

```
[ ] 1 from sklearn import metrics
     2 print(metrics.classification_report(y_test, pred_y, digits=3))
```

	precision	recall	f1-score	support
0	0.994	0.957	0.975	1029
1	1.000	0.994	0.997	35147
2	1.000	1.000	1.000	109104
3	0.997	0.910	0.952	21125
4	0.132	0.988	0.232	320
accuracy			0.987	166725
macro avg	0.824	0.970	0.831	166725
weighted avg	0.998	0.987	0.991	166725

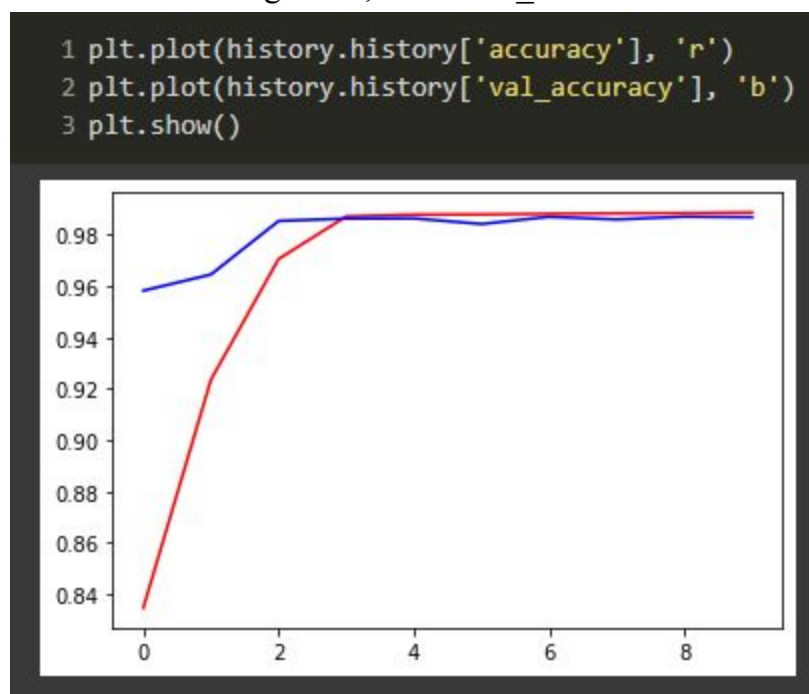
The precision, recall and corresponding F1 score for each class

12. Plotting learning curve : We plotted 2 curves. One between Training loss and validation loss and another between Training accuracy and validation accuracy.



X-axis = Number of epochs, Y-axis = Loss

Red : loss i.e. Training Loss , **Blue** : val_loss i.e. Validation Loss



X-axis = Number of epochs, Y-axis = Accuracy

Red : accuracy i.e. Training Accuracy, **Blue** : val_accuracy i.e. Validation Accuracy

Testing

The testing for the 1D CNN model was done on a validation set used when model.fit function is called. The validation set is a subset of the training set.

```
history=model.fit(X_sm, y_sm, epochs = 10, validation_data = (X_test, y_test), callbacks = early_stop)
```

We used X_test and y_test as validation data

```
Epoch 1/10  
34421/34421 [=====] - 1177s 34ms/step - loss: 64808.2491 - accuracy: 0.7769 - val_loss: 1318.6693 - val_accuracy: 0.9582
```

Here the val_loss is the validation loss and val_accuracy is the validation accuracy

Results

1. Confusion Matrix : A Confusion matrix is an N x N matrix used for evaluating the performance of a classification model, where N is the number of target classes. The matrix compares the actual target values with those predicted by the machine learning model.

The rows represent the predicted values of the target variable.

The columns represent the actual values of the target variable.

```
[[ 985      8      5      1     30]
 [   1 34943     32     55    116]
 [   0      2 109050      4     48]
 [   4      0      0 19228  1893]
 [   1      1      1      1    316]]
```

The Confusion Matrix

As evident from the confusion matrix, all the results are predicted correctly as in the majority of the predictions, predicted value matches the actual value.

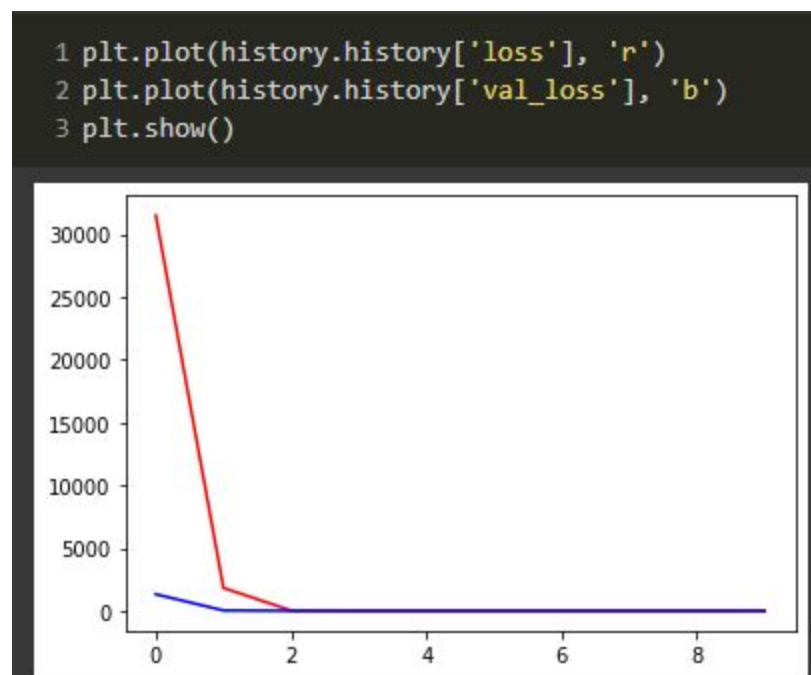
2. Performance Metrics :

We calculated the Precision, Recall and the F1 score for the CNN model.

	precision	recall	f1-score	support
0	0.994	0.957	0.975	1029
1	1.000	0.994	0.997	35147
2	1.000	1.000	1.000	109104
3	0.997	0.910	0.952	21125
4	0.132	0.988	0.232	320
accuracy			0.987	166725
macro avg	0.824	0.970	0.831	166725
weighted avg	0.998	0.987	0.991	166725

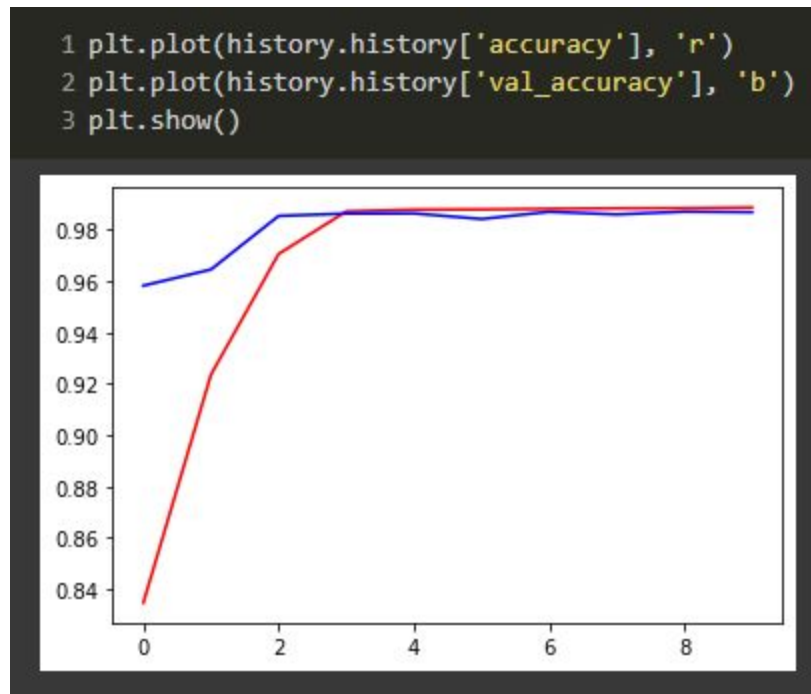
The Performance Metrics for the Model

3. Learning Curves : We plotted 2 curves, one for Loss and the other for Accuracy.



Graph 1 : X-axis = Number of epochs, Y-axis = Loss

Red : loss i.e. Training Loss , **Blue** : val_loss i.e. Validation Loss



Graph 2 : X-axis = Number of epochs, Y-axis = Accuracy

Red : accuracy i.e.Training Accuracy, **Blue** : val_accuracy i.e.Validation Accuracy

FUTURE SCOPE

The model we designed for the detection and classification of DDoS attacks can be used to protect websites from harmful DDoS attacks by prior detection of malicious traffic coming to the website.

We can consider the following example for the functioning of the model after deployment: Say a request contains UDP, UDPLag, Syn and Portmap Attacks.

We will create a web page that will contain a text box (users will be able to search for any text), for any searched query, we will scrape attacks and for all those scraped attacks, we will use the CNN detection model to classify the types of attacks.

In a typical machine learning and deep learning project, we usually start by defining the problem statement followed by data collection and preparation, understanding of the data, and model building. Model Deployment is one of the last stages of any machine learning project.

Model Deployment :

After building a predictive model we have to deploy it in production. We give a record of features on which the model is trained and then it gives prediction as an output. The model is placed on a server where it can receive multiple requests for predictions. Once deployed, the model is capable of taking inputs and giving output to as many different requests made to it for predictions. For real-time predictions, we have to make use of a framework called **Flask**.

Flask is a web application framework written in Python. It has multiple modules that make it easier for a web developer to write applications without having to worry about the details like protocol management, thread management, etc. Flask gives a variety of choices for developing web applications and it gives us the necessary tools and libraries that allow us to build a web application.

We will run the model on the server as a rest API that will capture all the incoming requests for prediction and will post the output for every request.

Bibliography

- <https://www.unb.ca/cic/datasets/ddos-2019.html>
- <https://colab.research.google.com/notebooks/intro.ipynb>
- <https://colab.research.google.com/github/AviatorMoser/keras-mnist-tutorial/blob/master/MNIST%20in%20Keras.ipynb>
- <http://neuralnetworksanddeeplearning.com/chap1.html>
- <https://digitalattackmap.com/understanding-ddos/>
- <https://www.f5.com/labs/articles/education/what-is-a-distributed-denial-of-service-attack->
- <https://machinelearningmastery.com/>
- <https://www.lucidchart.com/pages/>
- <https://www.sicara.ai/blog/hyperparameter-tuning-keras-tuner>

Research papers :

https://drive.google.com/drive/folders/1_PSt5I5ogiBC-XTuke6XNlx-WeHEbwCi?usp=sharing

Code :

Dataset Analysis:

https://colab.research.google.com/drive/1yak9-2xfy_iU6cC8GOCR4m0wH7yQn2cc?authuser=3#scrollTo=a-ChfJPhDBG

Preprocessing, hypertuning and model :

<https://colab.research.google.com/drive/1Y4OvNFFIXGRBKHSQYkIL9Exm4GQX8vSX?authuser=3>