Task 2: IoTMal2020-CDMC: IOT Malware Detection

Malware Family classification using Convolutional Neural Networks on Byte Sequences

Submitted By:

Siddharth Garg

i@siddharthgarg.in

Vellore Institute of Technology, Vellore

Under guidance of

Prof. Aswani Kumar Cherukuri,

Vellore Institue of Technology, Vellore, India

Dr. Gang Li,

Deakin University, Australia

I. Problem Background

1.1 Problem Statement

Based on the byte sequences collected at the entry points of ELF files as discriminant features and the malware families of the programs as training labels, the participants are required to perform a classification task to predict the malware families of the test samples. The dataset consists of 72,638 samples generated following the procedure below: First, a collection of malicious and benign Linux programs in ELF format were collected from various sources. Then, from each of these programs, the first 2K bytes (0-padded if the file is not long enough) starting at the entry point of the file were extracted. These ASCII strings were then encoded by a simple encryption cipher to remove the sensitive information and fed to a base64 encoder to yield readable radix-64 representations. Label (family type of malware) of the binary files are determined by the state-of-art antivirus engines.

The participants are required to provide the prediction of labels of the test samples based on information provided in the task.

1.2 Data Set Used

The original analysis data of the IoT malware classification task was kindly contributed by Taiwan Information Security Center (TWISC). The data was processed by the CDMC 2020 committee with all sensitive information removed.

- "Family" column indicates the family type of the binary file, and is taken as a class label of this classification task.
- "CP" column indicates the CPU architecture on which the file is compiled. Participants can determine whether or not to use it as side information to improve the prediction performance.
- "ByteSequence" column is the encoded first 2K bytes of the ELF files following the aforementioned steps.

Malware Families (labels) to be classified:

Android, Bashlite, BenignWare, Dofloo,Hajime, Mirai,Pnscan, Tsunami, Xorddos

II. Proposed Solution

2.1 Approach Used

The provided encoded byte sequences are extracted from the column of spreadsheet provided using base64 decoder in order to obtain actual byte sequences. These binary files are then stored separately in a directory for further processing. The representation of binary sequences is translated into grayscale images which is further processed through a Convolutional Neural Network.

The solution is inspired from the paper *"Malware Classification using Deep Learning based Feature Extraction and Wrapper based Feature Selection Technique"*

2.2 Pipelines for training and inference

2.2.1 Input Pipeline

This input pipeline is common to both the process of training as well as inference, The byte sequences are first processed through a sequence of data processing procedures in order to make them compatible with a CNN. Original byte sequences are provided in radix 64 encoded format, hence to obtain the binary string the sequence is translated using a base64 decoder. The value of each byte is represented as an integer and processed in the form of an array. These integral values are used to determine the brightnesses of each pixel of a grayscale image. The images formed are resized to 32x32 in order to achieve uniformity. These 32x32 images pixel values are flattened and converted to a Comma Separated File (CSV) with each row representing a single image.

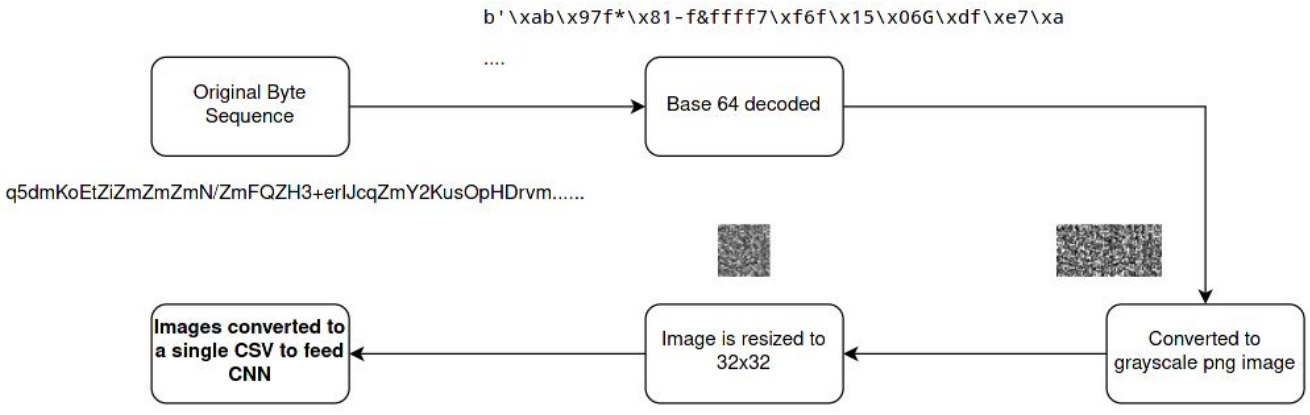

Figure 2.1 - Input Pipeline

2.2.2 Training Pipeline

The CSV obtained from the input pipeline is sampled into training, testing and validation. Training sample is used to fit the model. The validation set is used for early stopping and hyperparameter tuning.Testing sample is used to as a metric of model performance. The samples are then normalized using min max normalization for the feature columns. The weights for the CNN are randomly initialized. Model is fit according to training data and hyperparameters are tuned according to its performance on validation set. The weights of the best model are then saved and are used to infer the test data

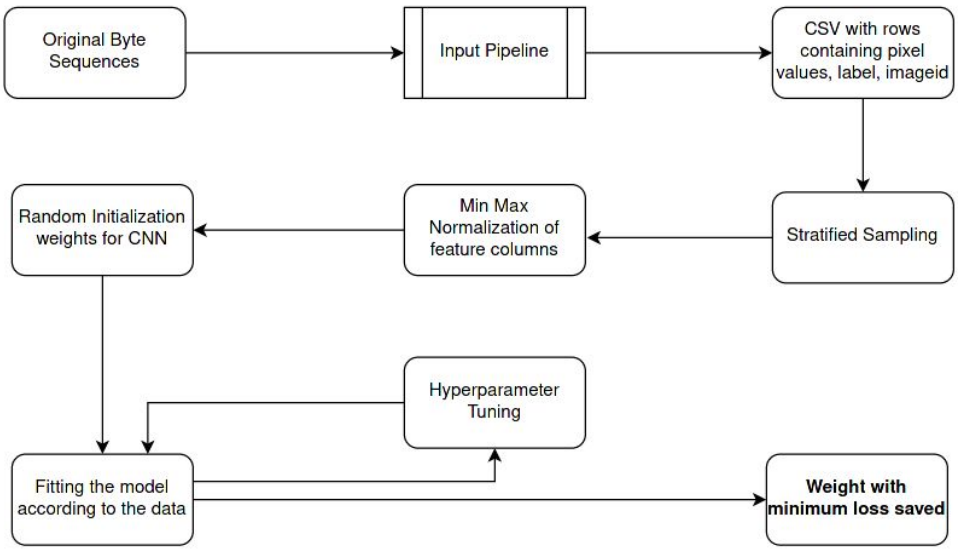

Figure 2.2 - Training Pipeline

2.3 CNN Architecture

A custom defined CNN is used in order to obtain inferences. This model uses a combination of convolutional and dense layers in order to generalize the data and perform the classification task.

```
ConvNet_single(
  (layer1): Sequential(
    (0): Conv2d(1, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): LeakyReLU(negative_slope=0.01)
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (l1_batchnorm): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (layer2): Sequential(
    (0): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): LeakyReLU(negative_slope=0.01)
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (l2_batchnorm): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (layer3): Sequential(
    (0): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): LeakyReLU(negative_slope=0.01)
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (l3_batchnorm): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (layer4): Sequential(
    (0): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): LeakyReLU(negative_slope=0.01)
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (l4_batchnorm): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (layer5): Sequential(
    (0): Conv2d(512, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): LeakyReLU(negative_slope=0.01)
  )
  (l5_batchnorm): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (leakyrelu): LeakyReLU(negative_slope=0.01)
  (fc): Linear(in_features=4096, out_features=1000, bias=True)
  (fc01): Linear(in_features=1000, out_features=500, bias=True)
  (fc1): Linear(in_features=500, out_features=9, bias=True)
)
```

Figure 2.3 - CNN Architecture

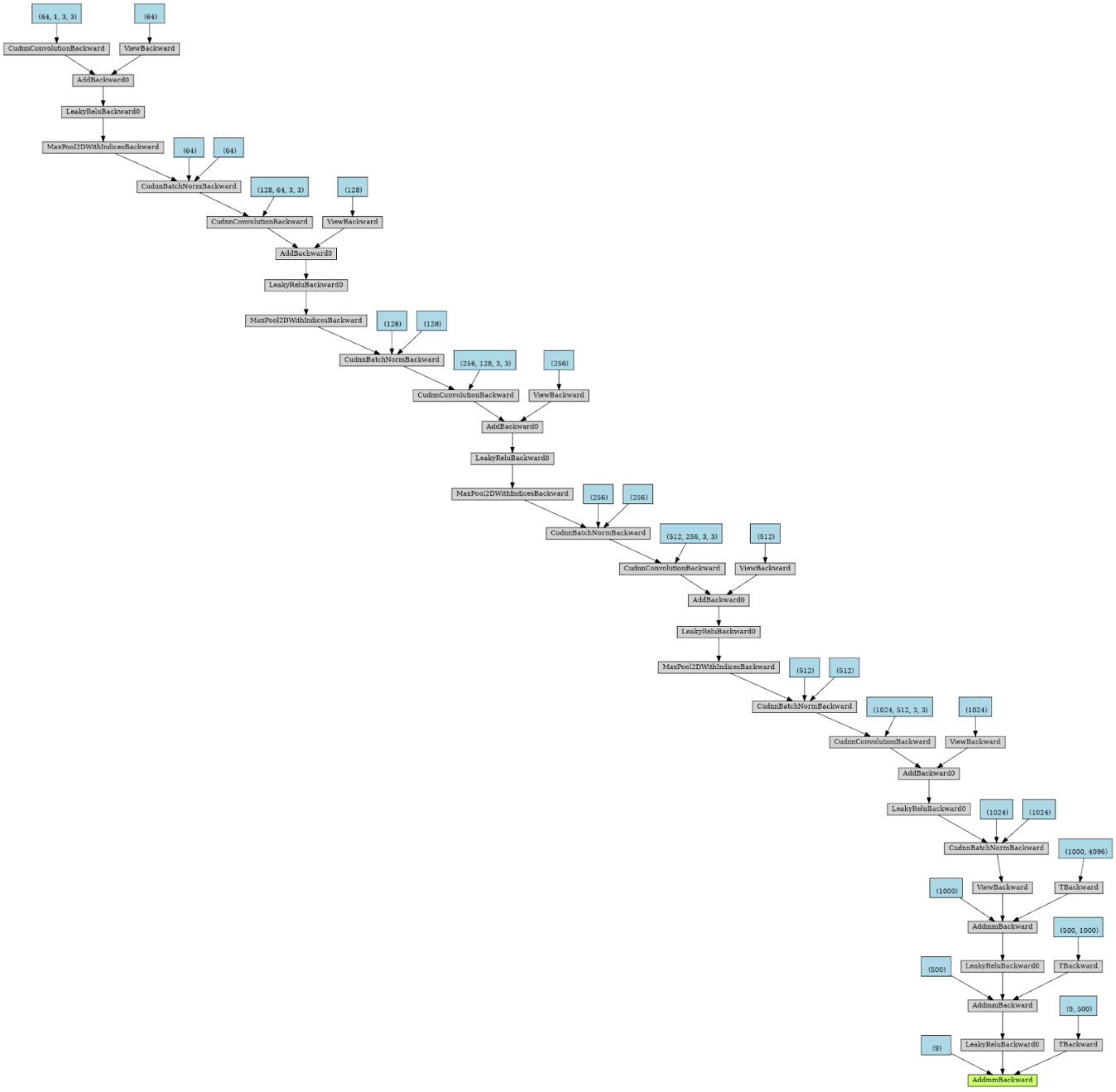

Figure 2.4 - CNN calculation dot graph

III. Experiment

3.1 Code Screenshots

```
In [6]: import base64

In [9]: for i in range(len(df)):
            row = df.iloc[i,:]
            with open(f'ByteFiles/{row.id}.bytes','wb') as f:
                f.write(base64.b64decode(row.ByteSequence))
```

Figure 3.1 - Conversion of ByteSequence to decoded byte files

```
In [18]: destination='ByteImages/'
         folder='ByteFiles/'

In [19]: width = 64

In [21]:
for name in malwarelist:
    b = ".bytes"
    nam=name[0].strip('"')
    loc = folder+nam + b
    hexar = []
    with open(loc, 'rb') as f:
        byte_str = f.read()
        hexar = [i for i in byte_str]

    print(len(hexar))
    if len(hexar)!=0:
        rn = len(hexar) // width
        fh = np.reshape(hexar[:rn * width], (-1, width))
        fh=  np.uint8(fh)
        print(nam,'  ',fh.shape)
        img = Image.fromarray(fh)
        img.save(destination+nam+".png")
    if len(hexar)==0:
        corrupted_files.append(nam)
```

Figure 3.2 - Conversion of byte files to images

```
In [3]: destination='resized/'
        folder='ByteImages/'
```

Below code reads actual images and resize them to 32x32 using BICUBIC interpolation, whereas corrupted file names stored in the c_files list variable

```
In [4]: for name in malwarelist:
            b = ".bytes"
            nam=name[0].strip('"')
            loc = folder+nam + b
            img=Image.open(folder+nam+".png")
            img = img.resize((32, 32), Image.BICUBIC)
            img.save(destination+nam+".png")
```

Figure 3.3 -Resizing images

In [10]:
```
folder='resized/'
```

Below code read the images and then flatten their values(i.e. 32x32=1024). These values stored in the data_to_write list against their IDs and labels

In [4]:
```
data_to_write=[]
for name in malwarelist:
    b = ".bytes"
    nam=name[0].strip('"')
    loc = folder+nam + b
    img = np.asarray(Image.open(folder+nam+".png"))
    img = img.flatten()
    img= img.tolist()
    img.append(nam)
    img.append(name[1])
    data_to_write.append(img)
```

Following commands generate the CSV from the values store in data_to_write list

In [5]:
```
myFile = open('data_new.csv', 'w', newline='')
with myFile:
    writer = csv.writer(myFile)
    writer.writerows(data_to_write)
```

Figure 3.4 - Writing images to a CSV file

In [1]:
```
from PIL import Image
import numpy as np
import csv
train_full='data_to_traFull.csv'
test='data_to_test.csv'
train='data_to_tra.csv'
valid='data_to_val.csv'
```

In this section of code, train data is normalized with respect to its max and min values against each feature.

In [2]:
```
new_train_data = np.genfromtxt(train, delimiter=",", dtype=str)
tra_data= new_train_data[:,0:-2]
tra_lab=new_train_data[:,-1]
tra_name=new_train_data[:,-2]
tra_lab=tra_lab.astype(np.int)
tra_data=tra_data.astype(np.float)
tra_min=np.min(tra_data,axis=0)
tra_max=np.max(tra_data,axis=0)
tra_dom=np.subtract(tra_max,tra_min)
tra_dom=np.where(tra_dom == 0, 1, tra_dom)
img = np.subtract(tra_data,tra_min)
img = np.divide(img, tra_dom)
img= img.tolist()

for i in range(len(tra_lab)):
    img[i].append(tra_name[i])
    img[i].append(tra_lab[i])

myFile = open('data_to_tra.csv', 'w', newline='')

with myFile:
    writer = csv.writer(myFile)
    writer.writerows(img)
```

Figure 3.5 - Min Max Normalization

```
import torch.nn as nn

class ConvNet_single(nn.Module):
    def __init__(self):
        super(ConvNet_single, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(1, 64, kernel_size=3, stride=1, padding=1),
            nn.LeakyReLU(),
            nn.MaxPool2d(kernel_size=2))

        self.l1_batchnorm=nn.BatchNorm2d(64)

        self.layer2 = nn.Sequential(
            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
            nn.LeakyReLU(),
            nn.MaxPool2d(kernel_size=2))

        self.l2_batchnorm=nn.BatchNorm2d(128)

        self.layer3 = nn.Sequential(
            nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1),
            nn.LeakyReLU(),
            nn.MaxPool2d(kernel_size=2))

        self.l3_batchnorm=nn.BatchNorm2d(256)


        self.layer4 = nn.Sequential(
            nn.Conv2d(256, 512, kernel_size=3, stride=1, padding=1),
            nn.LeakyReLU(),
            nn.MaxPool2d(kernel_size=2))

        self.l4_batchnorm=nn.BatchNorm2d(512)

        self.layer5 = nn.Sequential(
            nn.Conv2d(512, 1024, kernel_size=3, stride=1, padding=1),
            nn.LeakyReLU())
```

Figure 3.6 - CNN Architecture Definition (Pytorch)

```
        self.l5_batchnorm=nn.BatchNorm2d(1024)

        self.leakyrelu=nn.LeakyReLU()

        self.fc = nn.Linear(2*2*1024,1000)
        self.fc01= nn.Linear(1000,500)
        self.fc1= nn.Linear(500,9)

    def forward(self, x):
        out = self.layer1(x)
        out = self.l1_batchnorm(out)
        out = self.layer2(out)
        out = self.l2_batchnorm(out)
        out=  self.layer3(out)
        out = self.l3_batchnorm(out)
        out=  self.layer4(out)
        out = self.l4_batchnorm(out)
        out=  self.layer5(out)
        out = self.l5_batchnorm(out)
        out = out.reshape(out.size(0), -1)
        out = self.fc(out)
        out=self.leakyrelu(out)
        out= self.fc01(out)
        out=self.leakyrelu(out)
        out= self.fc1(out)
        return out
```

Figure 3.7 - CNN Architecture Definition pt 2(Pytorch)

In [5]:
```
train_path='data_to_tra.csv'
val_path= 'data_to_val.csv'

train_data = CustomDatasetFromImages(train_path)
val_data = CustomDatasetFromImages(val_path)

val_size = 500

train_data_loader = data.DataLoader(train_data, batch_size=batch_size,shuffle=True)
val_data_loader  = data.DataLoader(val_data, batch_size=val_size, shuffle=True)
```

Creating the model, defining loss function and optimizer

In [6]:
```
model= ConvNet_single().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

Function to save the state of the model

In [7]:
```
path_to_checkpoint='check_point/'
check_name='checkpoint.pth.tar'

def save_checkpoint(state, is_best, filename,loss):
    if is_best:
        print ("=> Saving a new lowest loss : "+str(loss))
        torch.save(state, filename)  # save checkpoint
```

This is the section where training takes place

Figure 3.8 - Data Loading

In [7]:
```
total_step = len(train_data_loader)
lowest_loss=0.800

for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_data_loader):
        images = images.to(device)
        labels = labels.to(device)
# Forward pass
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        loss = criterion(outputs, labels)
        x=np.array((predicted==labels).cpu())
# Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        val_loss_list=[]
        val_acc_list =[]
# Doing validation
        for y,(images,labels) in enumerate(val_data_loader):
            images=images.to(device)
            labels=labels.to(device)
            outputs=model(images)
            _, predicted = torch.max(outputs.data, 1)
            val_loss = criterion(outputs, labels)
            val_loss_list.append(val_loss.item())
            val_x=np.array((predicted==labels).cpu())
            val_acc=(sum(val_x))*100/len(val_x)
            val_acc_list.append(val_acc)
        mean_loss= np.mean(np.array(val_loss_list))
        mean_acc = np.mean(np.array(val_acc_list))
        is_best= bool(mean_loss<lowest_loss)
# If validation results are good then previous loss new state of the model will be save
        if(is_best):
            lowest_loss= min(mean_loss,lowest_loss)
            pathe= path_to_checkpoint+str(lowest_loss)+" "+str(epoch+1)+" "+check_name
            save_checkpoint({'epoch':epoch + 1,'state_dict': model.state_dict(),'lowest_loss': lowest_loss }, is_bes
            is_best= False

        print("Epoch ["+str(epoch+1)+"/"+str(num_epochs)+"],Batch_no["+str(i+1)+"/"+str(total_step)+"] "+"Loss:"+str
```

Figure 3.9 - Fitting the data

In [29]:
```
PATH = 'check_point/0.11215294844337872 5 checkpoint.pth.tar'
checkpoint = torch.load(PATH)
model.load_state_dict(checkpoint['state_dict'])
epoch = checkpoint['epoch']
loss = checkpoint['lowest_loss']
```

In [30]:
```
import seaborn as sn
import pandas as pd
import matplotlib.pyplot as plt

def plt_cm(array):
    df_cm = pd.DataFrame(array, index = [i for i in "ABCDEFGHI"],
                  columns = [i for i in "ABCDEFGHI"])
    plt.figure(figsize = (10,7))
    sn.heatmap(df_cm, annot=True)
```

In [10]:
```
nb_classes = 9
v_data_loader  = data.DataLoader(val_data, batch_size=val_size, shuffle=True)
confusion_matrix = torch.zeros(nb_classes, nb_classes)
with torch.no_grad():
    for i, (inputs, classes) in enumerate(v_data_loader):
        inputs = inputs.to(device)
        classes = classes.to(device)
        outputs = model(inputs)
        make_dot(outputs)
        _, preds = torch.max(outputs, 1)
        for t, p in zip(classes.view(-1), preds.view(-1)):
                confusion_matrix[t.long(), p.long()] += 1

print(confusion_matrix)
print(confusion_matrix.diag()/confusion_matrix.sum(1))
```

Figure 3.10 - Testing results and plotting Confusion Matrix

3.2 Confusion Matrices

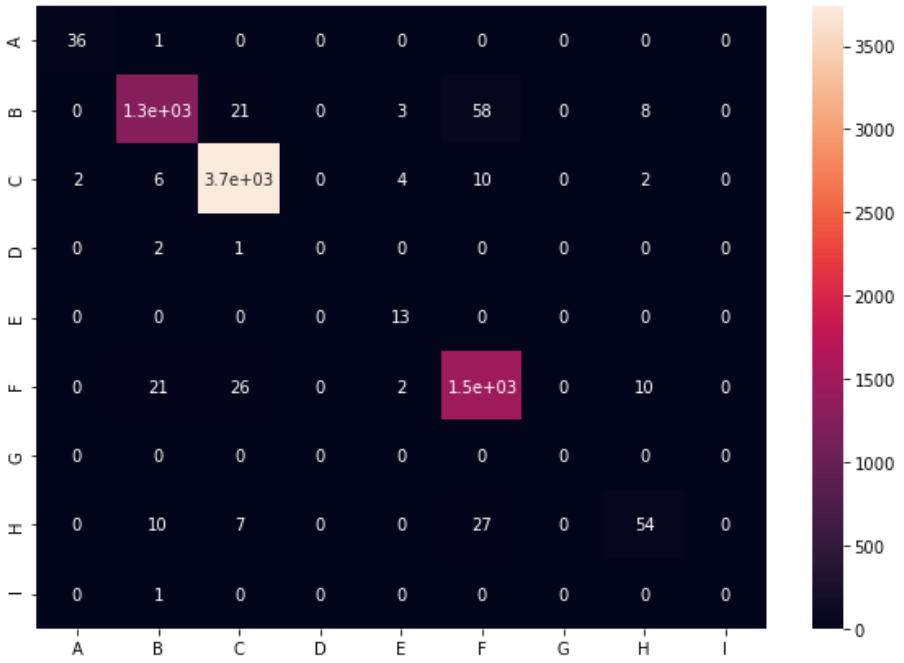

Figure 3.11 - Validation Confusion Matrix

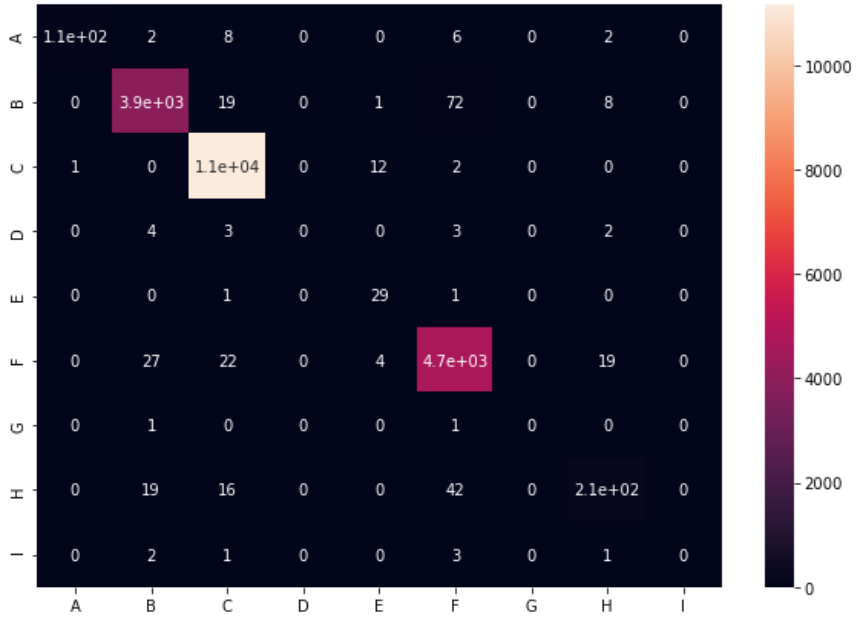

Figure 3.12 - Training sample Confusion Matrix

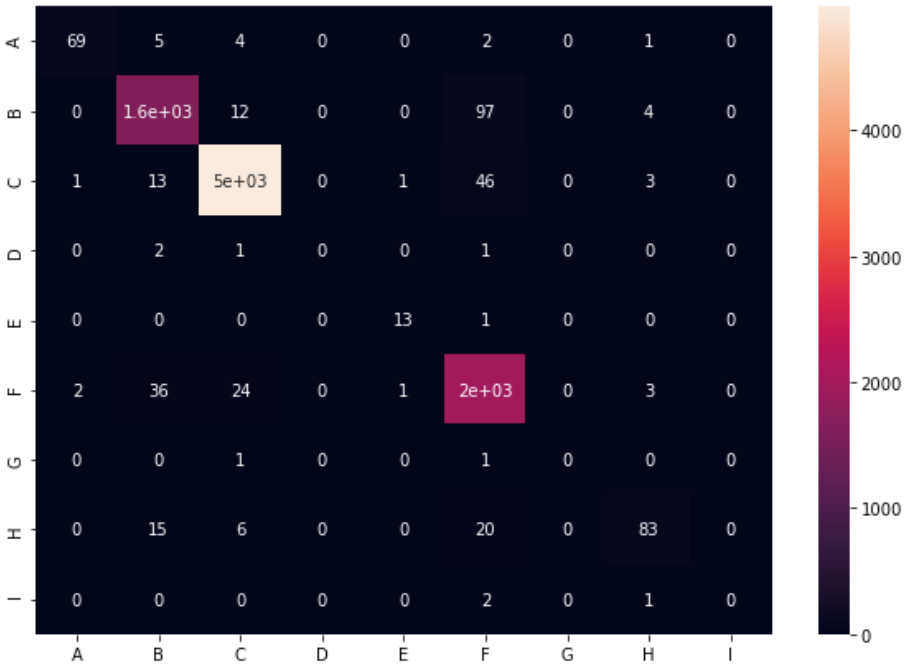

Figure 3.13 - Testing Data Confusion Matrix

3.3 Data Visualization

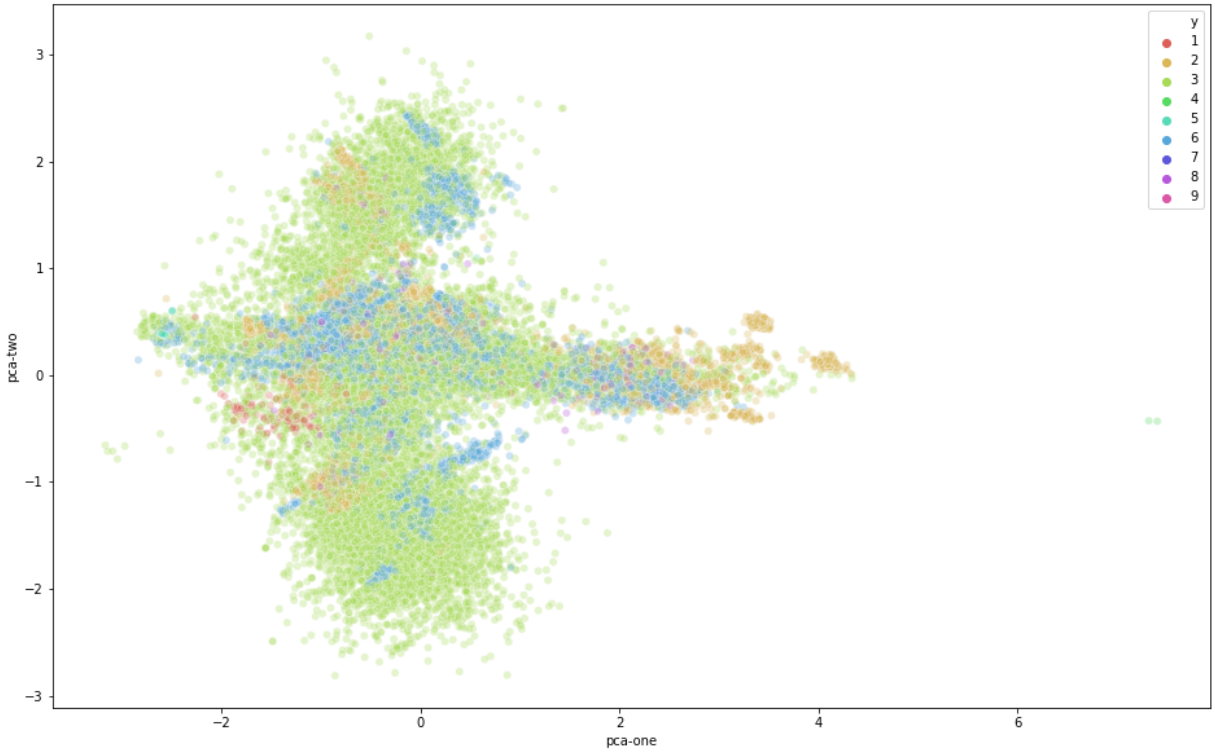

Figure 3.13 - PCA plot

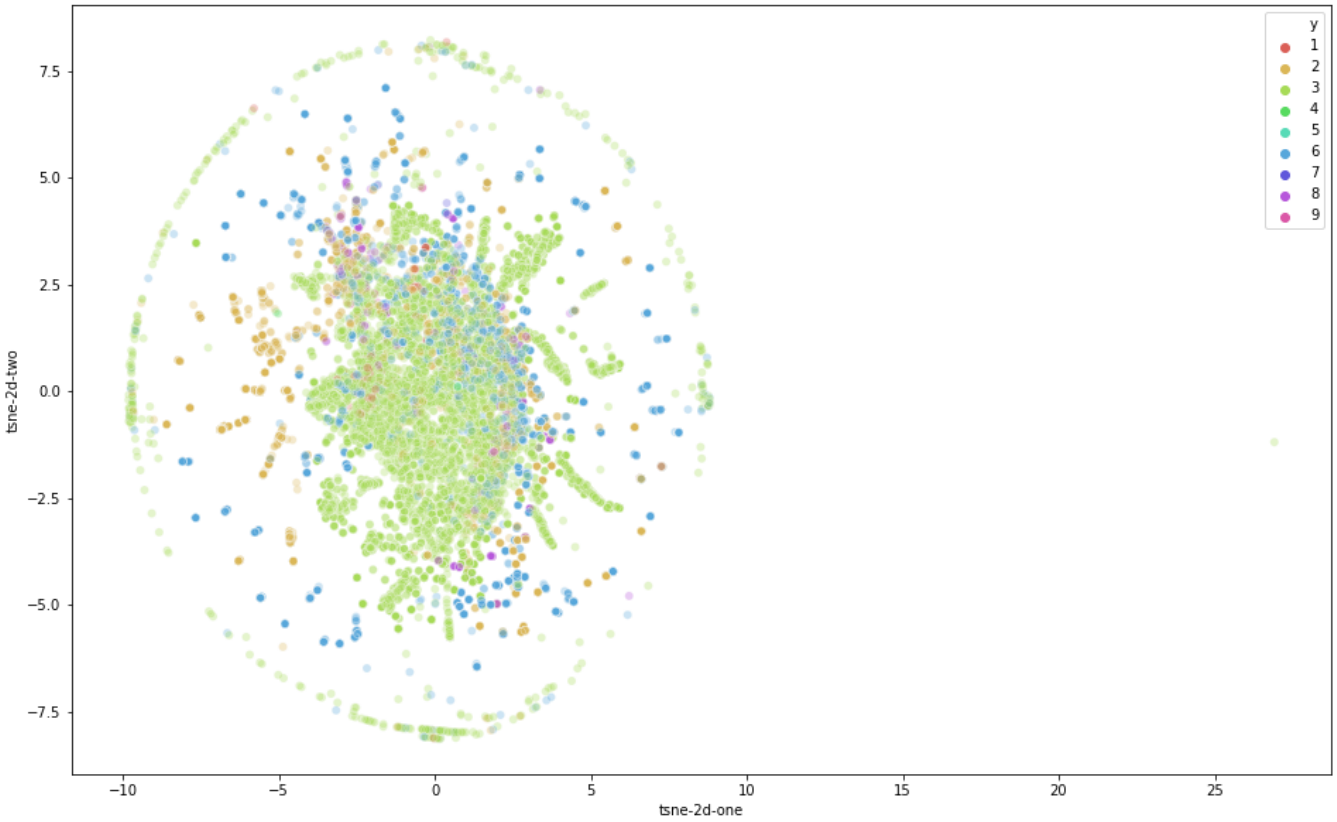

Figure 3.14 - TNSE plot

IV. Conclusion

Since the dataset provided had given sequential data and training a sequential model for byte sequences does not yield very accurate results, a CNN was used to generalize the data by converting the byte sequence to an intermediate representation of images. The trained model achieved an accuracy of 98% on the training sample and 96% on validation sample.

V. References

[1]Rafique, Muhammad Furqan, et al. "Malware Classification using Deep Learning based Feature Extraction and Wrapper based Feature Selection Technique." *arXiv preprint arXiv:1910.10958* (2019).

[2] “Malware Detection Using Machine Learning”
https://github.com/cyberhunters/Malware-Detection-Using-Machine-Learning