Open Innovation in Data Science

Up2053685

Word count= 1100

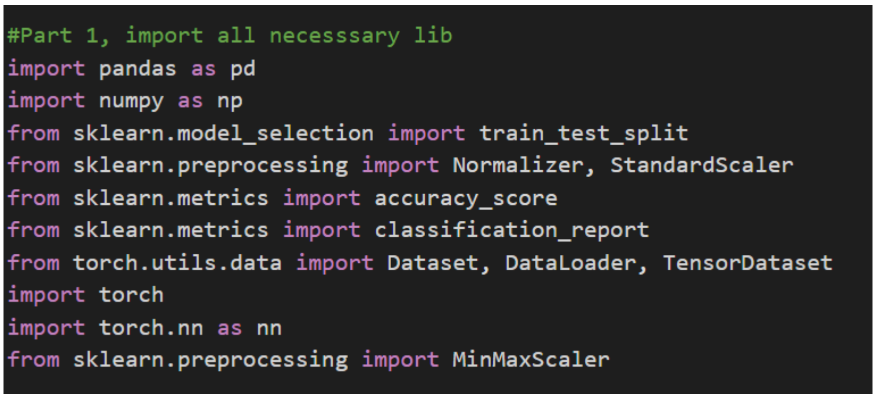
Part 1: Botnet/Benign Classification in IoT Devices

1.1 Introduction

This project consists of designing a deep neural network for classification in IoT devices. This report will highlight the architecture used to gain the best accuracy. We will then showcase training loss and accuracy plots of the model. Then, we will evaluate the model. Once the model has been completed and evaluated, samples will be provided that are not from the training dataset. Then, this report will show two ways to improve our classifier with evidence-based results.

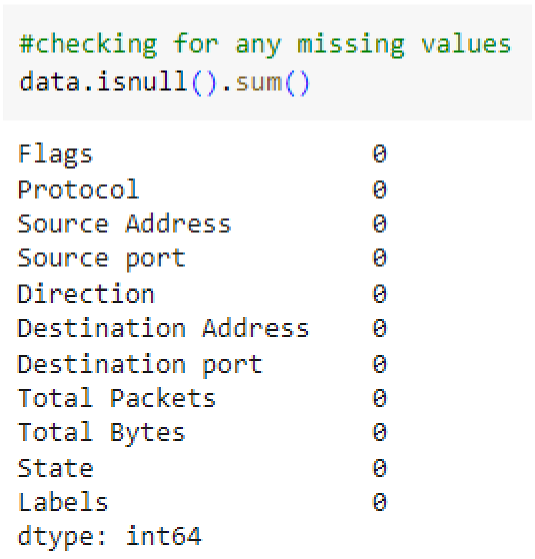
1.2 Preprocessing the dataset

Firstly, all the necessary libraries were imported shown in Figure 1. This will be used for preprocessing the dataset. We split the dataset into 30% testing and 70% training using 'train_test_function'. These packages will be used for model evaluation.
To read the CSV file, first, we had to mount Google Drive to have a file directory to the CSV file which was stored on Google Drive. Once mounted, using pandas, we read the CSV file from Google Drive.
The next step was to understand the dataset and preprocess it. Firstly, we checked for missing values and nothing was missing.



```
#Part 1, import all necesssary lib
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import Normalizer, StandardScaler
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report
from torch.utils.data import Dataset, DataLoader, TensorDataset
import torch
import torch.nn as nn
from sklearn.preprocessing import MinMaxScaler
```

Figure 1- Importing all necessary libraries such as torch

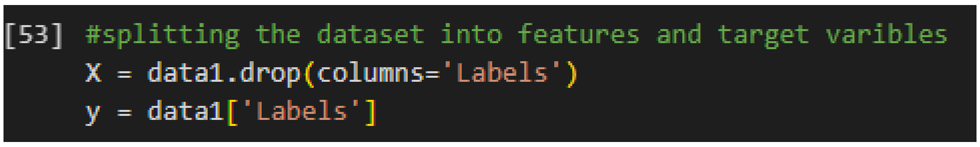


```
#checking for any missing values
data.isnull().sum()
```

```
Flags                  0
Protocol               0
Source Address         0
Source port            0
Direction              0
Destination Address    0
Destination port       0
Total Packets          0
Total Bytes            0
State                  0
Labels                 0
dtype: int64
```

Figure 2- Checking for any Missing Values

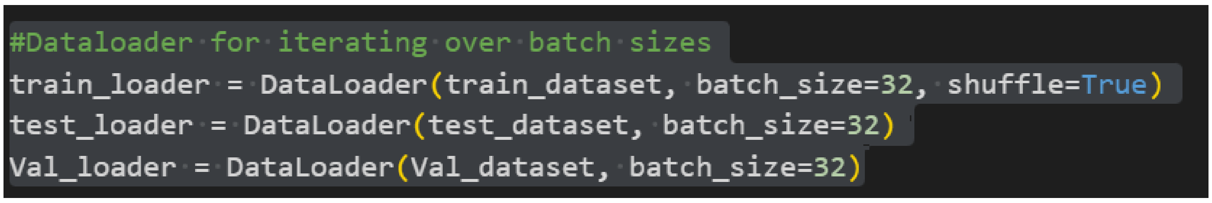
There are no missing values as shown in Figure 2, we have to assess the dataset and drop two columns.
The two columns dropped were the source port and destination port. This is because these attributes will lead to a bias in our results. We already have the IP address under the source address and destination address.
Then, the next step is to split the dataset into features and target variables.



```
[53] #splitting the dataset into features and target varibles
     X = data1.drop(columns='Labels')
     y = data1['Labels']
```

Figure 3- Splitting the dataset into features and target variables

Then we need to specify that the dataset will be split into 70% training, 15% testing and 15% valuation. The next step to preprocess the dataset is normalising the dataset. I've chosen this method because it helps avoid model instability and normalising the dataset helps our neural network model increase accuracy results.
The next stage of preprocessing the dataset is converting the dataset to pytorch sensors. This is essential because the data must be in pytorch tensors when executing the model. In addition, the next step is to use DataLoader for iterating over batch size shown in Figure 4.



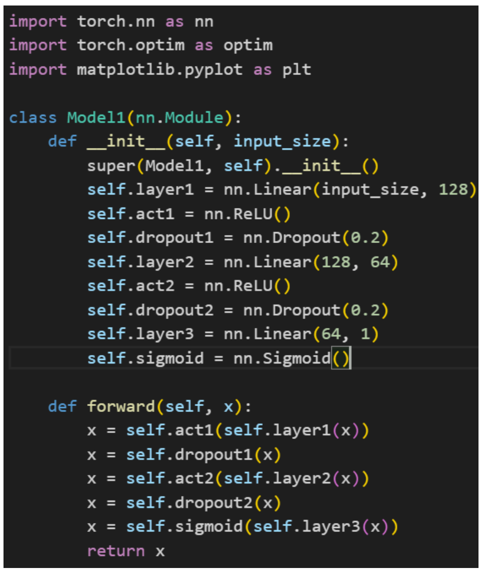
```python
#Dataloader for iterating over batch sizes
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=32)
Val_loader = DataLoader(Val_dataset, batch_size=32)
```

Figure 4: Using train, test and validation data loader into batch size.

1.3 Designing the deep neural network for classifying samples

The next main stage is to create a deep neural network model using PyTorch.

1.3.1 Structure of the model



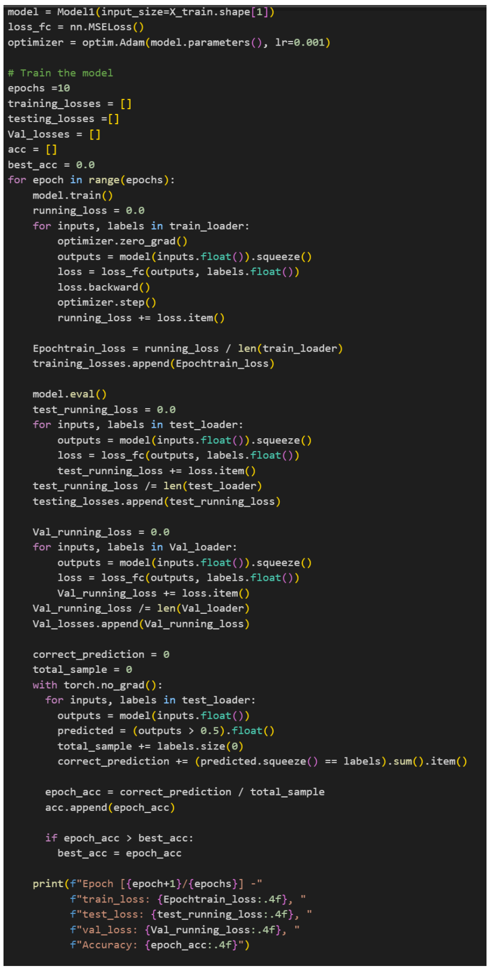
```python
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt

class Model1(nn.Module):
    def __init__(self, input_size):
        super(Model1, self).__init__()
        self.layer1 = nn.Linear(input_size, 128)
        self.act1 = nn.ReLU()
        self.dropout1 = nn.Dropout(0.2)
        self.layer2 = nn.Linear(128, 64)
        self.act2 = nn.ReLU()
        self.dropout2 = nn.Dropout(0.2)
        self.layer3 = nn.Linear(64, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.act1(self.layer1(x))
        x = self.dropout1(x)
        x = self.act2(self.layer2(x))
        x = self.dropout2(x)
        x = self.sigmoid(self.layer3(x))
        return x
```

Figure 5- Deep Neural network model using binary classification PyTorch

This model has 3 hidden layers, with uses of a Rectified Linear Unit (ReLU) activation function, dropouts and a Sigmoid activation function.
This deep neural network model using PyTorch is a binary classification model to classify Botnet devices with the highest accuracy. This model starts with an input size in the first layer with 128 output neurons. Then the ReLU activation function is added as 'self act1'. After a ReLU activation, a dropout with a dimension rate of 0.2 is added. This was chosen because the function will drop 20% to prevent overfitting for a model. The model gains more layers with layer 2 with 128 inputs and 64 outputs as neurons. Similar to layer 1, layer 2 is also followed by an activation function called ReLU called 'self.act2' and a dropout with a 0.2 dropout rate. Finally, we put our final layer, self.layer3, as a linear function with 64 input dimensions and ending with 1. In addition, we add a Sigmoid activation function because this is a binary classification model. This is to keep the output to be either 1 or 0.

After, we use the forward method to create a pass through this model architecture network.

1.3.2 Training Loop



```
model = Model1(input_size=X_train.shape[1])
loss_fc = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Train the model
epochs =10
training_losses = []
testing_losses =[]
Val_losses = []
acc = []
best_acc = 0.0
for epoch in range(epochs):
    model.train()
    running_loss = 0.0
    for inputs, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(inputs.float()).squeeze()
        loss = loss_fc(outputs, labels.float())
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

    Epochtrain_loss = running_loss / len(train_loader)
    training_losses.append(Epochtrain_loss)

    model.eval()
    test_running_loss = 0.0
    for inputs, labels in test_loader:
        outputs = model(inputs.float()).squeeze()
        loss = loss_fc(outputs, labels.float())
        test_running_loss += loss.item()
    test_running_loss /= len(test_loader)
    testing_losses.append(test_running_loss)

    Val_running_loss = 0.0
    for inputs, labels in Val_loader:
        outputs = model(inputs.float()).squeeze()
        loss = loss_fc(outputs, labels.float())
        Val_running_loss += loss.item()
    Val_running_loss /= len(Val_loader)
    Val_losses.append(Val_running_loss)

    correct_prediction = 0
    total_sample = 0
    with torch.no_grad():
      for inputs, labels in test_loader:
        outputs = model(inputs.float())
        predicted = (outputs > 0.5).float()
        total_sample += labels.size(0)
        correct_prediction += (predicted.squeeze() == labels).sum().item()

      epoch_acc = correct_prediction / total_sample
      acc.append(epoch_acc)

      if epoch_acc > best_acc:
        best_acc = epoch_acc

    print(f"Epoch [{epoch+1}/{epochs}] -"
          f"train_loss: {Epochtrain_loss:.4f}, "
          f"test_loss: {test_running_loss:.4f}, "
          f"val_loss: {Val_running_loss:.4f}, "
          f"Accuracy: {epoch_acc:.4f}")
```

Figure 6- Training loop with training loss and accuracy plots

Firstly, we have added a loss function, as ‘loss_fc’, called Mean Squared Error (MSE) to help calculate training losses. We’ve used an optimiser from Torch. optim called Adam. I have set the parameters to 0.001. This number shows the learning rate.

Now to train the model, we have added 10 epochs for the loop to run 10 times separately for training, testing and valuation. The model trains within each epoch through its batch sizes from the data loader previously. Then the model clears the gradient with ‘optimizer.zero.grad()’. The squeeze() function helps the dimension sizes. Outputs contain the model prediction from the input data. Then, use loss to calculate the MSE loss from correct labels and predicted labels.

After the training loop has been completed, the model is now evaluating its accuracy to find losses. Then this would start generating its results.

1.3.3 Training loss and Accuracy plots report.

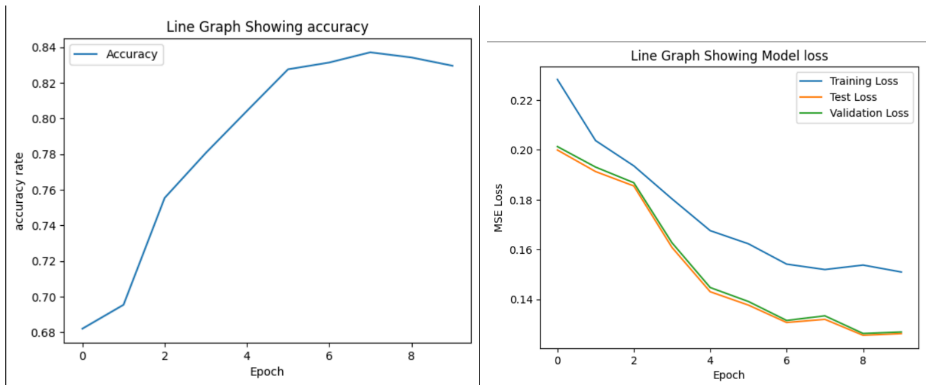
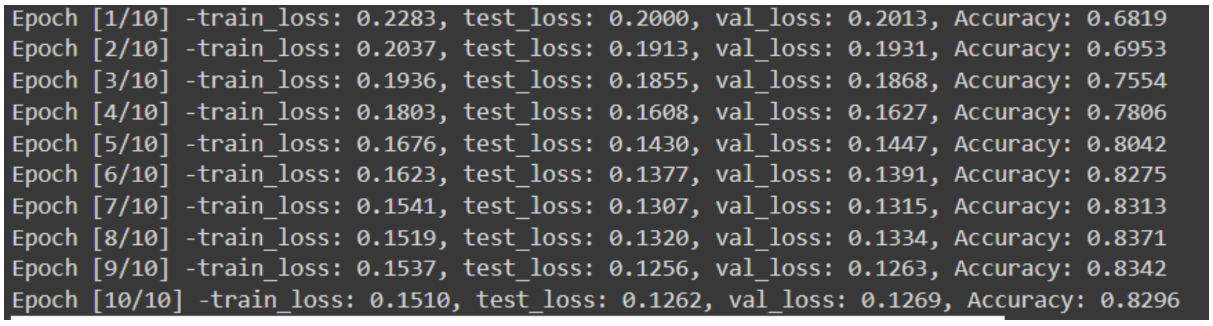


Figure 7: Line Graph showing Model accuracy and Model Loss

The first graph in Figure 7 shows a steady increase in accuracy. The model started at 68.2% and then increased to 83%. The results are shown in Figure 8.

The second graph in Figure 7 shows the training, test and validation loss. All lines have a consistent curve and a negative slope. The training loop does not cross the training line which is a good sign as the model has generalised well. Both testing and validation lines are very similar.

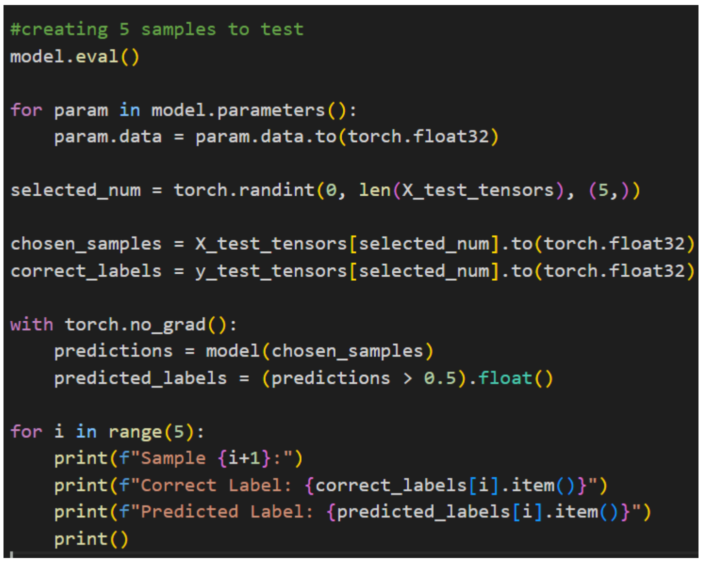
However, there's a big gap between training and validation. This suggests the dataset might be sensitive.



```
Epoch [1/10] -train_loss: 0.2283, test_loss: 0.2000, val_loss: 0.2013, Accuracy: 0.6819
Epoch [2/10] -train_loss: 0.2037, test_loss: 0.1913, val_loss: 0.1931, Accuracy: 0.6953
Epoch [3/10] -train_loss: 0.1936, test_loss: 0.1855, val_loss: 0.1868, Accuracy: 0.7554
Epoch [4/10] -train_loss: 0.1803, test_loss: 0.1608, val_loss: 0.1627, Accuracy: 0.7806
Epoch [5/10] -train_loss: 0.1676, test_loss: 0.1430, val_loss: 0.1447, Accuracy: 0.8042
Epoch [6/10] -train_loss: 0.1623, test_loss: 0.1377, val_loss: 0.1391, Accuracy: 0.8275
Epoch [7/10] -train_loss: 0.1541, test_loss: 0.1307, val_loss: 0.1315, Accuracy: 0.8313
Epoch [8/10] -train_loss: 0.1519, test_loss: 0.1320, val_loss: 0.1334, Accuracy: 0.8371
Epoch [9/10] -train_loss: 0.1537, test_loss: 0.1256, val_loss: 0.1263, Accuracy: 0.8342
Epoch [10/10] -train_loss: 0.1510, test_loss: 0.1262, val_loss: 0.1269, Accuracy: 0.8296
```

Figure 8: Test Accuracy score from Model

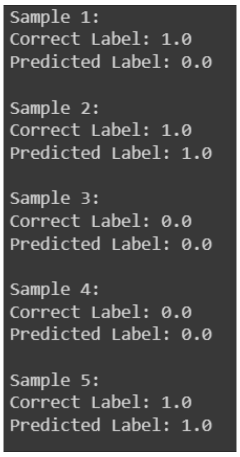
1.4 Five test samples



```
#creating 5 samples to test
model.eval()

for param in model.parameters():
    param.data = param.data.to(torch.float32)

selected_num = torch.randint(0, len(X_test_tensors), (5,))

chosen_samples = X_test_tensors[selected_num].to(torch.float32)
correct_labels = y_test_tensors[selected_num].to(torch.float32)

with torch.no_grad():
    predictions = model(chosen_samples)
    predicted_labels = (predictions > 0.5).float()

for i in range(5):
    print(f"Sample {i+1}:")
    print(f"Correct Label: {correct_labels[i].item()}")
    print(f"Predicted Label: {predicted_labels[i].item()}")
    print()
```

Figure 9- Methodology to select 5 samples and report 5 predicted labels

We use Model. eval() to disable dropout features. We convert all data types to 'torch.float32'. This will generate 5 random numbers from testing data and extract them. Then, we perform the predictions. Once that's complete, we present the results as shown below in Figure 10.



```
Sample 1:
Correct Label: 1.0
Predicted Label: 0.0

Sample 2:
Correct Label: 1.0
Predicted Label: 1.0

Sample 3:
Correct Label: 0.0
Predicted Label: 0.0

Sample 4:
Correct Label: 0.0
Predicted Label: 0.0

Sample 5:
Correct Label: 1.0
Predicted Label: 1.0
```

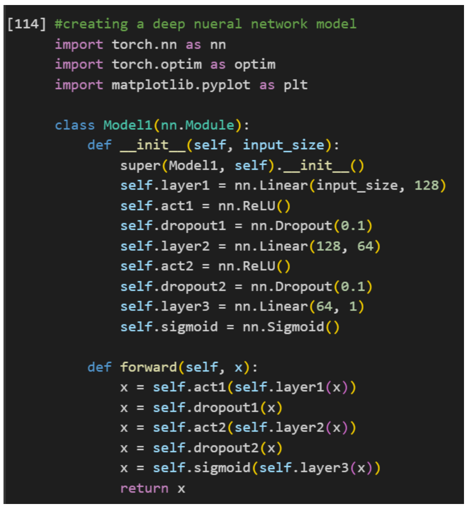
Figure 10- Five test samples

All samples have been predicted correctly, this correlates to the test accuracy score and evaluates this model's performance.

1.5 Methods to Improve Accuracy

1.5.1 Method 1: Changing the Dropout Rate

Figure 11 shows the dropout rate has been changed to 0.1 instead of 0.2. This decreases the rate to improve accuracy and prevents overfitting because it reduces the number of regularisation in the model and makes it less likely to overfitting. The test accuracy has increased by 2.8%. Figure 12 shows the improved accuracy rate of 86.7%.



```
[114] #creating a deep nueral network model
      import torch.nn as nn
      import torch.optim as optim
      import matplotlib.pyplot as plt

      class Model1(nn.Module):
          def __init__(self, input_size):
              super(Model1, self).__init__()
              self.layer1 = nn.Linear(input_size, 128)
              self.act1 = nn.ReLU()
              self.dropout1 = nn.Dropout(0.1)
              self.layer2 = nn.Linear(128, 64)
              self.act2 = nn.ReLU()
              self.dropout2 = nn.Dropout(0.1)
              self.layer3 = nn.Linear(64, 1)
              self.sigmoid = nn.Sigmoid()

          def forward(self, x):
              x = self.act1(self.layer1(x))
              x = self.dropout1(x)
              x = self.act2(self.layer2(x))
              x = self.dropout2(x)
              x = self.sigmoid(self.layer3(x))
              return x
```

Figure 11- Improvement model with a lower dropout rate

The first graph in Figure 12 shown below suggests a lower sensitivity to the dataset. The training loss curve remains the same as previously in the initial model. The gap being the training loop and validation loss shows an improvement in the model.
The second graph shows an improved accuracy score of 87.8% which is a 4.8% increase. Therefore, changing the dropout rate can improve the model. However, the accuracy rate fluctuates which can indicate class imbalance.

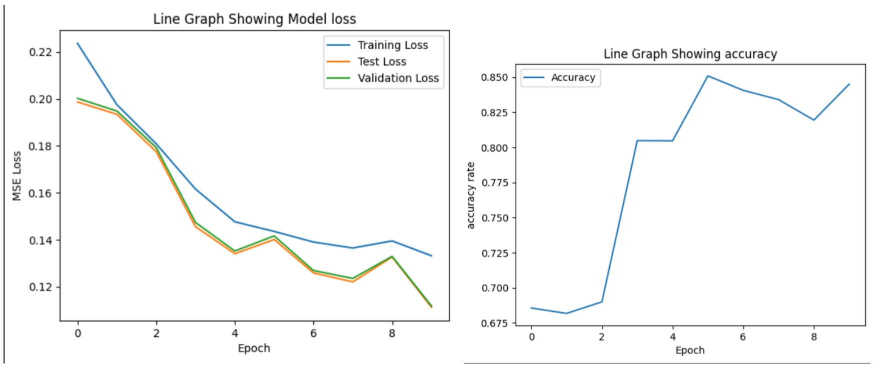
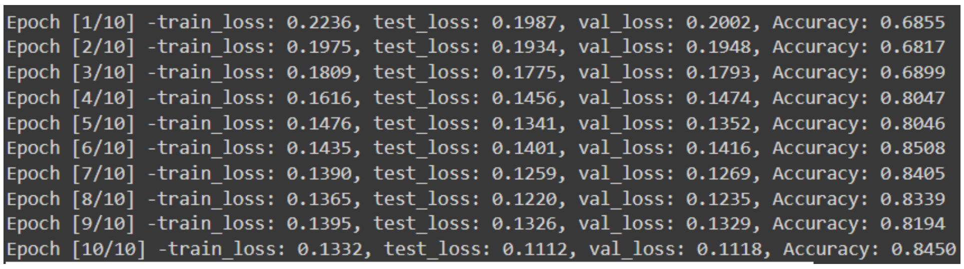
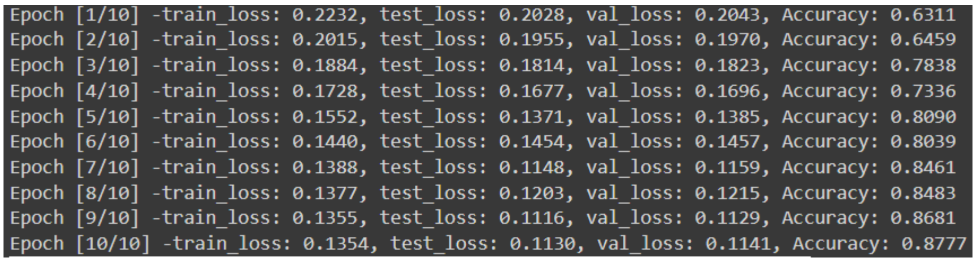


Figure 12- Line graph of Model's loss with Improved model



```
Epoch [1/10] -train_loss: 0.2236, test_loss: 0.1987, val_loss: 0.2002, Accuracy: 0.6855
Epoch [2/10] -train_loss: 0.1975, test_loss: 0.1934, val_loss: 0.1948, Accuracy: 0.6817
Epoch [3/10] -train_loss: 0.1809, test_loss: 0.1775, val_loss: 0.1793, Accuracy: 0.6899
Epoch [4/10] -train_loss: 0.1616, test_loss: 0.1456, val_loss: 0.1474, Accuracy: 0.8047
Epoch [5/10] -train_loss: 0.1476, test_loss: 0.1341, val_loss: 0.1352, Accuracy: 0.8046
Epoch [6/10] -train_loss: 0.1435, test_loss: 0.1401, val_loss: 0.1416, Accuracy: 0.8508
Epoch [7/10] -train_loss: 0.1390, test_loss: 0.1259, val_loss: 0.1269, Accuracy: 0.8405
Epoch [8/10] -train_loss: 0.1365, test_loss: 0.1220, val_loss: 0.1235, Accuracy: 0.8339
Epoch [9/10] -train_loss: 0.1395, test_loss: 0.1326, val_loss: 0.1329, Accuracy: 0.8194
Epoch [10/10] -train_loss: 0.1332, test_loss: 0.1112, val_loss: 0.1118, Accuracy: 0.8450
```

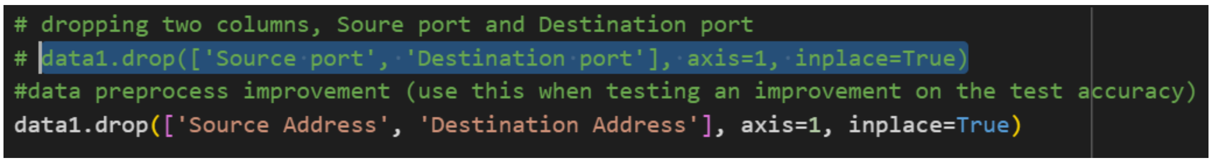


```
Epoch [1/10] -train_loss: 0.2232, test_loss: 0.2028, val_loss: 0.2043, Accuracy: 0.6311
Epoch [2/10] -train_loss: 0.2015, test_loss: 0.1955, val_loss: 0.1970, Accuracy: 0.6459
Epoch [3/10] -train_loss: 0.1884, test_loss: 0.1814, val_loss: 0.1823, Accuracy: 0.7838
Epoch [4/10] -train_loss: 0.1728, test_loss: 0.1677, val_loss: 0.1696, Accuracy: 0.7336
Epoch [5/10] -train_loss: 0.1552, test_loss: 0.1371, val_loss: 0.1385, Accuracy: 0.8090
Epoch [6/10] -train_loss: 0.1440, test_loss: 0.1454, val_loss: 0.1457, Accuracy: 0.8039
Epoch [7/10] -train_loss: 0.1388, test_loss: 0.1148, val_loss: 0.1159, Accuracy: 0.8461
Epoch [8/10] -train_loss: 0.1377, test_loss: 0.1203, val_loss: 0.1215, Accuracy: 0.8483
Epoch [9/10] -train_loss: 0.1355, test_loss: 0.1116, val_loss: 0.1129, Accuracy: 0.8681
Epoch [10/10] -train_loss: 0.1354, test_loss: 0.1130, val_loss: 0.1141, Accuracy: 0.8777
```

Figure 13- Test accuracy score

1.5.2 Method 2: Changing the column drop

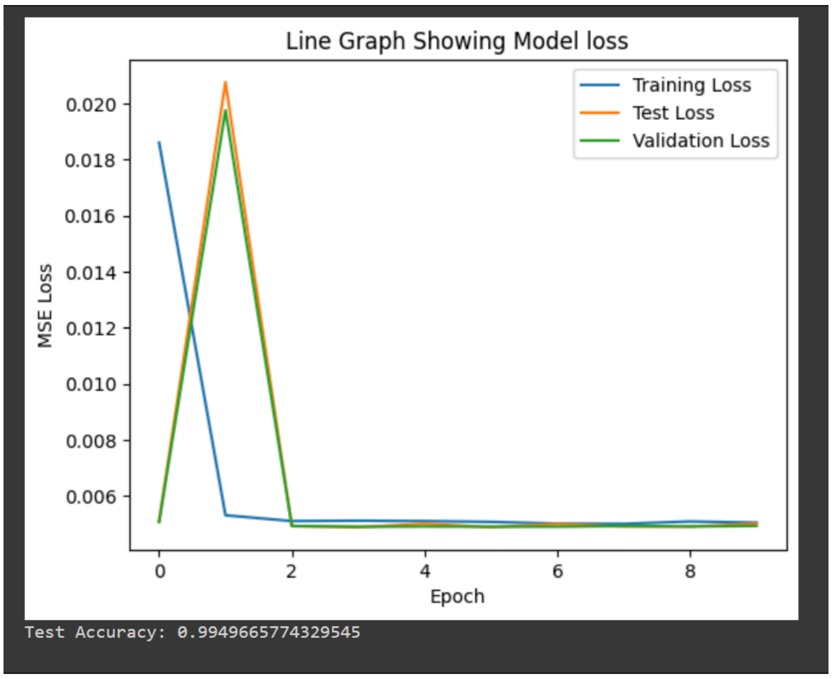
Lastly, changing our preprocessing methods has increased the accuracy.
Figure 13 shows we are dropping the source address and destination address instead of the source port and destination port.



```
# dropping two columns, Soure port and Destination port
# data1.drop(['Source port', 'Destination port'], axis=1, inplace=True)
#data preprocess improvement (use this when testing an improvement on the test accuracy)
data1.drop(['Source Address', 'Destination Address'], axis=1, inplace=True)
```

Figure 14- Showing data preprocessing change for improving the model.

The accuracy score increases significantly to 99.5%. This shows an improvement in our model shown in Figure 14.



```
Test Accuracy: 0.9949665774329545
```

Figure 15- Line Graph Model loss showing test accuracy and training, test and validation loss

However, this might increase the accuracy but it might be misleading due to this graph. As training loss goes down, test and validation go up and then down after two epochs. This shows dropping the two columns can show a bias in the classification model.

Appendix 1

```
#Part 1, import all necessary lib
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import Normalizer, StandardScaler
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report
from torch.utils.data import Dataset, DataLoader, TensorDataset
import torch
import torch.nn as nn
from sklearn.preprocessing import MinMaxScaler
#connecting to google drive
from google.colab import drive
drive.mount('/content/gdrive/')
#read the CSV dataset file from my drive
data1 = pd.read_csv('/content/gdrive/MyDrive/Colab
Notebooks/TDataset.csv')
#display first few rows of dataset
data1.head()
#checking for any missing values
data1.isnull().sum()
#checking all columns and data types
data1.info()
#printing all data columns
print(data1.columns)
#dataset stats
data1.describe()
# dropping two columns, Soure port and Destination port
data1.drop(['Source port', 'Destination port'], axis=1, inplace=True)
#data preprocess improvement (use the code when testing an improvement
on the test accuracy and comment out the code above)
# data1.drop(['Source Address', 'Destination Address'], axis=1,
inplace=True)
```

```python
#dataset shape
data1.shape
#splitting the dataset into features and target varibles
X = data1.drop(columns='Labels')
y = data1['Labels']
# #Normalizing the dataset (dont run this when testing for improvements
section)
normalizer = Normalizer()
X_normalizer = normalizer.fit_transform(X)
X = pd.DataFrame(X_normalizer, columns=X.columns)
#splitting dataset into training (70%) and testing (30%)
X_train, X_test, y_train, y_test = train_test_split(X, y,
train_size=0.80, random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,
train_size=0.85, random_state=42)

#converting the data to Pytorch Tensors
X_train_tensors = torch.tensor(X_train.values.astype(float))
y_train_tensors = torch.tensor(y_train.values.astype(float))
X_val_tensors = torch.tensor(X_val.values.astype(float))
y_val_tensors = torch.tensor(y_val.values.astype(float) )
X_test_tensors = torch.tensor(X_test.values.astype(float))
y_test_tensors = torch.tensor(y_test.values.astype(float))

#creating the TensorDatasets for Pytorch Dataloader
train_dataset = TensorDataset(X_train_tensors, y_train_tensors)
test_dataset = TensorDataset(X_test_tensors, y_test_tensors)
Val_dataset = TensorDataset(X_val_tensors, y_val_tensors)
#Dataloader for iterating over batch sizes
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=32)
Val_loader = DataLoader(Val_dataset, batch_size=32)
#creating a deep nueral network model with 3 layers, ReLU functions and
2 dropouts
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt

class Model1(nn.Module):
    def __init__(self, input_size):
        super(Model1, self).__init__()
        self.layer1 = nn.Linear(input_size, 128)
        self.act1 = nn.ReLU()
        self.dropout1 = nn.Dropout(0.2)
        self.layer2 = nn.Linear(128, 64)
        self.act2 = nn.ReLU()
        self.dropout2 = nn.Dropout(0.2)
        self.layer3 = nn.Linear(64, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.act1(self.layer1(x))
        x = self.dropout1(x)
        x = self.act2(self.layer2(x))
        x = self.dropout2(x)
        x = self.sigmoid(self.layer3(x))
        return x
# define model, adding MSE loss and optimizer
model = Model1(input_size=X_train.shape[1])
loss_fc = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Train the model
epochs =10
training_losses = []
testing_losses =[]
Val_losses = []
acc = []
best_acc = 0.0
for epoch in range(epochs):
    model.train()
    running_loss = 0.0
    for inputs, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(inputs.float()).squeeze()
        loss = loss_fc(outputs, labels.float())
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

    Epochtrain_loss = running_loss / len(train_loader)
    training_losses.append(Epochtrain_loss)

    model.eval()
    test_running_loss = 0.0
    for inputs, labels in test_loader:
```

```python
        outputs = model(inputs.float()).squeeze()
        loss = loss_fc(outputs, labels.float())
        test_running_loss += loss.item()
    test_running_loss /= len(test_loader)
    testing_losses.append(test_running_loss)

    Val_running_loss = 0.0
    for inputs, labels in Val_loader:
        outputs = model(inputs.float()).squeeze()
        loss = loss_fc(outputs, labels.float())
        Val_running_loss += loss.item()
    Val_running_loss /= len(Val_loader)
    Val_losses.append(Val_running_loss)

    correct_prediction = 0
    total_sample = 0
    with torch.no_grad():
      for inputs, labels in test_loader:
        outputs = model(inputs.float())
        predicted = (outputs > 0.5).float()
        total_sample += labels.size(0)
        correct_prediction += (predicted.squeeze() ==
labels).sum().item()

      epoch_acc = correct_prediction / total_sample
      acc.append(epoch_acc)

      if epoch_acc > best_acc:
        best_acc = epoch_acc

    print(f"Epoch [{epoch+1}/{epochs}] -"
          f"train_loss: {Epochtrain_loss:.4f}, "
          f"test_loss: {test_running_loss:.4f}, "
          f"val_loss: {Val_running_loss:.4f}, "
          f"Accuracy: {epoch_acc:.4f}")




plt.plot(training_losses, label='Training Loss')
plt.plot(testing_losses, label='Test Loss')
plt.plot(Val_losses, label='Validation Loss')
plt.title('Line Graph Showing Model loss')
plt.ylabel('MSE Loss')
plt.xlabel('Epoch')
plt.legend()
plt.show()

plt.plot(acc, label='Accuracy')
plt.title('Line Graph Showing accuracy')
plt.ylabel('accuracy rate')
plt.xlabel('Epoch')
plt.legend()
plt.show()

#creating 5 samples to test
model.eval()

for param in model.parameters():
    param.data = param.data.to(torch.float32)

selected_num = torch.randint(0, len(X_test_tensors), (5,))

chosen_samples = X_test_tensors[selected_num].to(torch.float32)
correct_labels = y_test_tensors[selected_num].to(torch.float32)

with torch.no_grad():
    predictions = model(chosen_samples)
    predicted_labels = (predictions > 0.5).float()

for i in range(5):
    print(f"Sample {i+1}:")
    print(f"Correct Label: {correct_labels[i].item()}")
    print(f"Predicted Label: {predicted_labels[i].item()}")
    print()
# improved model- changing dropout rate to 0.1
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt

#improvement made
class Model1(nn.Module):
    def __init__(self, input_size):
        super(Model1, self).__init__()
        self.layer1 = nn.Linear(input_size, 128)
        self.act1 = nn.ReLU()
        self.dropout1 = nn.Dropout(0.1)
        self.layer2 = nn.Linear(128, 64)
        self.act2 = nn.ReLU()
        self.dropout2 = nn.Dropout(0.1)
        self.layer3 = nn.Linear(64, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.act1(self.layer1(x))
        x = self.dropout1(x)
        x = self.act2(self.layer2(x))
        x = self.dropout2(x)
        x = self.sigmoid(self.layer3(x))
        return x
#training loop with MSe loss function and optimizer
model = Model1(input_size=X_train.shape[1])
loss_fc = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Train the model
epochs =10
training_losses = []
testing_losses =[]
Val_losses = []
acc = []
best_acc = 0.0
for epoch in range(epochs):
    model.train()
    running_loss = 0.0
    for inputs, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(inputs.float()).squeeze()
        loss = loss_fc(outputs, labels.float())
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

    Epochtrain_loss = running_loss / len(train_loader)
    training_losses.append(Epochtrain_loss)

    model.eval()
    test_running_loss = 0.0
    for inputs, labels in test_loader:
        outputs = model(inputs.float()).squeeze()
        loss = loss_fc(outputs, labels.float())
        test_running_loss += loss.item()
    test_running_loss /= len(test_loader)
    testing_losses.append(test_running_loss)

    Val_running_loss = 0.0
    for inputs, labels in Val_loader:
        outputs = model(inputs.float()).squeeze()
        loss = loss_fc(outputs, labels.float())
        Val_running_loss += loss.item()
    Val_running_loss /= len(Val_loader)
    Val_losses.append(Val_running_loss)

    correct_prediction = 0
    total_sample = 0
    with torch.no_grad():
      for inputs, labels in test_loader:
        outputs = model(inputs.float())
        predicted = (outputs > 0.5).float()
        total_sample += labels.size(0)
        correct_prediction += (predicted.squeeze() ==
labels).sum().item()

      epoch_acc = correct_prediction / total_sample
      acc.append(epoch_acc)

      if epoch_acc > best_acc:
        best_acc = epoch_acc

    print(f"Epoch [{epoch+1}/{epochs}] -"
          f"train_loss: {Epochtrain_loss:.4f}, "
          f"test_loss: {test_running_loss:.4f}, "
          f"val_loss: {Val_running_loss:.4f}, "
          f"Accuracy: {epoch_acc:.4f}")




plt.plot(training_losses, label='Training Loss')
plt.plot(testing_losses, label='Test Loss')
plt.plot(Val_losses, label='Validation Loss')
plt.title('Line Graph Showing Model loss')
plt.ylabel('MSE Loss')
plt.xlabel('Epoch')
plt.legend()
plt.show()


plt.plot(acc, label='Accuracy')
plt.title('Line Graph Showing accuracy')
plt.ylabel('accuracy rate')
plt.xlabel('Epoch')
plt.legend()
plt.show()
```

References

Chilamkurthy, S. (2023). *Writing custom datasets, DataLoaders and transforms*. Writing Custom Datasets, DataLoaders and Transforms - PyTorch Tutorials 2.1.1+cu121 documentation.

https://pytorch.org/tutorials/beginner/data_loading_tutorial.html

Howard, J., fast.ai, Thomas, R., & Ingham, F. (2023). *What is torch.nn really?*. What is torch.nn really? - PyTorch Tutorials 2.1.1+cu121 documentation.

https://pytorch.org/tutorials/beginner/nn_tutorial.html

Torch.tensor. torch.Tensor - PyTorch 2.1 documentation. (2023).

https://pytorch.org/docs/stable/tensors.html#tensor-class-reference