Masters Project:

ADFA Intrusion Detection Prediction.

Duncan Enzmann

**Introduction**

For my Masters Project, I have been working with Christian Skalka, Joe Near, and John Ring. Initially, I was tasked with remodeling a paper <https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8540398>. The paper itself was a good base for creating and evaluating the data. Unfortunately, it seems some of the metrics the paper itself was following were flawed which lead to me reforming some of my analysis and evaluation of the data. The data itself is composed of 3 different folders. I will talk about my implementation and results in the following 8 sections. 1: The dataset itself and preprocessing, 2: What data I am using for training, validation, and testing, and what these mean, 3: The model I am using to evaluate the data, 4: How the model is trained 5: The probability score of if the sequence is deemed an attack or not, 6: I will show my evaluation models compared from my models against Johns simple GRU network, 7: Future implementations and related work.

**1a: Dataset:**

As mentioned in the Introduction the obtained data from <https://www.unsw.adfa.edu.au/unsw-canberra-cyber/cybersecurity/ADFA-IDS-Datasets/ADFA-LD.zip> contains the following.

Attack\_Data\_Master: 746 files containing 6 different attacks

1. 10 Folders of Trace Files to Add users

- 91 Trace Files

2. 10 Folders of Trace Files to use Hydra FTP attacks

- 162 Trace Files

3. 10 Folders of Trace Files to use Hydra SSH attacks

- 176 Trace Files

4. 10 Folders of Trace Files for Java Meterpreter attacks

- 124 Trace Files

5. 10 folders of Trace Files for Meterpreter attacks

- 75 Trace Files

6. 10 Folders of Trace Files for Web\_shell attacks

- 118 Trace Files

Train\_Data\_Master: 833 Trace Files

Validation\_Data\_Master: 4373 Trace Files

**1b: Preprocessing of data**

To use this dataset and preprocess it to be usable in my model I need to load the files into text files. Following the initial papers preprocessing of the data I have created a few variables to determine the model inputs and how the data should look for it (Once the data is preprocessed into files the Boolean flag can be changed to move into pulling data out of the files and moving onto generators):

input\_dims = [len(np.load("../data/encoder.npy", allow\_pickle=True).item()) + 1]  
seq\_len = 10  
epochs = 25  
batch\_size = 64  
vocab\_size = input\_dims[0]  
trials = 10

The data needs to contain a certain number of input dimensions which is 175. To create the encoder, I have a simple function that takes in each item in a list of all the sequence numbers possible and relabels them starting from 0 to max. In this case, 175 different sequence numbers need to be labeled. The reason for using a sequence length of 10, allows for fast training on the data while still giving a large enough training and testing trace length of 87,779 and 31,643 respectively. I split the given data into 70% training and 30% testing. I will talk about the creation and use of the data more in **section 2**. After combining the normal and attack data while keeping a 1:1 ratio, I use a function get\_adfa\_seqeunce, which takes in the data and returns out the data into X and Y traces of N-grams where N is the size of the sequence length.

test = get\_adfa\_sequence\_group(dataTest, seq\_len, skip=True, keep\_nested=False)  
attack = get\_adfa\_sequence\_group(dataAtt, seq\_len, skip=True, keep\_nested=False)  
normal = get\_adfa\_sequence\_group(dataNormalLeftOver, seq\_len, skip=True, keep\_nested=False)  
  
np.savetxt('data\_to\_use/normalDataX.out', normal[0], delimiter=',')  
np.savetxt('data\_to\_use/normalDataY.txt', normal[1], delimiter=',')  
  
  
xTrain, xval, yTrain, yval = train\_test\_split(normal[0], normal[1], test\_size=.3, random\_state=42)

To keep the normal data with its entire length I write the data now. After the split of the data, I save the data I have manipulated to text files in folders containing the data for sequences of the following: normalDataX/Y, trainingDataX/Y, validationDataX/Y, testingDataX/Y, and attackDataX/Y. This allows me to obtain my X and Y traces, and make sure the data stays the same for every model I will run and evaluate on. Once the data is split into these files the preprocessing is almost complete with X and Y sequences of length 10. Next, I use a generator to save memory on machines

normal\_data\_gen = Generator(normal[0], normal[1], batch\_size, vocab\_size)  
train\_data\_gen = Generator(xTrain, yTrain, batch\_size, vocab\_size)  
testing\_data\_gen = Generator(test[0], test[1], batch\_size, vocab\_size)  
# testingVal\_data\_gen = Generator(testingVal[0], testingVal[1], batch\_size, vocab\_size)  
val\_data\_gen = Generator(xval, yval, batch\_size, vocab\_size)  
atk\_data\_gen = Generator(attack[0], attack[1], batch\_size, vocab\_size)

These Generators maintain the data structure and sort the data into categorical information (One hot encoded).

class Generator(Sequence):  
 def \_\_init\_\_(self, x\_set, y\_set, batch\_size=64, vocab\_size=176):  
 self.x, self.y = x\_set, y\_set  
 self.batch\_size = batch\_size  
 self.indices = np.arange(self.x.shape[0])  
 self.vocab\_size = vocab\_size  
  
 def \_\_len\_\_(self):  
 return np.int(np.ceil(self.x.shape[0] / self.batch\_size))  
  
 def \_\_getitem\_\_(self, idx):  
 inds = self.indices[idx \* self.batch\_size:(idx + 1) \* self.batch\_size]  
 batch\_x = self.x[inds]  
 batch\_y = self.y[inds]  
 return to\_categorical(batch\_x, num\_classes=self.vocab\_size), to\_categorical(batch\_y, num\_classes=self.vocab\_size), [None]  
  
 def on\_epoch\_end(self):  
 np.random.shuffle(self.indices)

Once the data is placed into generators for each type of data, I use this data in the model that will be talked about in **Sections 3 and 4.**

**2a: Obtaining the data from the folders for training testing and validation.**

As mentioned in **Section 1** the data itself is spread across a total of 3 Folders with Attack containing 6 more folders of different attack types, and 2 folders of normal data being Validation and Training.

The first thing to do is get out each of these trace files from these folders and strip the text of the sequences.

def load\_files(file\_group: str) -> List[List[int]]:  
 *"""Fetches integer sequences form specified AFDA group  
  
 Looks for ADFA-LD datset in ../data/ADFA-LD fetching it to that location if required.  
  
 Args:  
 file\_group: ("attack" | "train" | "val")  
  
 Returns:  
 List of lists of integer sequences  
  
 """* Path("../data").mkdir(exist\_ok=True)  
  
 root\_path = Path("../data/ADFA-LD")  
 if file\_group == 'train':  
 path = root\_path / "Training\_Data\_Master/"  
 elif file\_group == "val":  
 path = root\_path / "Validation\_Data\_Master/"  
 elif file\_group == "attack":  
 path = root\_path / "Attack\_Data\_Master/"  
 elif file\_group == "train\_att":  
 path = "C:/Users/djenz/OneDrive/Desktop/TS/uvm\_threat\_stack/data/ADFA-LD/train\_att/"  
 else:  
 raise ValueError('group must be one of "attack", "train", "val", "train\_att"')  
  
 if not root\_path.exists():  
 urllib.request.urlretrieve(  
 "https://www.unsw.adfa.edu.au/unsw-canberra-cyber/cybersecurity/ADFA-IDS-Datasets/ADFA-LD.zip", "data.zip")  
 with zipfile.ZipFile("data.zip", "r") as zip\_ref:  
 zip\_ref.extractall("../data/")  
 Path("data.zip").unlink()  
 shutil.rmtree('../data/\_\_MACOSX')  
 out = []  
 files = path.rglob('\*.txt')  
 for f in files:  
 with open(str(f), 'r') as myFile:  
 seq = [int(x) for x in myFile.read().strip().split(' ')]  
 out.append(seq)  
 return out

If the folders already exist on the hard drive of the computer which is going to move the contents of the file into variables then the code will continue, otherwise this gets the data from online.

dataTrain = load\_files('train')  
dataAtt = load\_files('attack')  
dataVal = load\_files('val')

After loading the data into variables, the data should be resized to avoid bias. To do this first the normal data of train and validation are resized to be the same size as the attack data.

def one\_one\_ratio(train, att):  
 numElemsDelete = len(train) - len(att)  
 numElemsKeep = len(train) - numElemsDelete  
 print("Size of train: " + str(len(train)))  
 print("Size of att: " + str(len(att)))  
 random.shuffle(train)  
 trainResized = train[:numElemsKeep]  
 trainLeftOver = train[numElemsKeep:]  
 print("Size of att: " + str(len(att)))  
 print("Size trainResized: " + str(len(trainResized)))  
  
 return trainResized, trainLeftOver

The training data for the program will be composed of training data used in the resize function and original attack data. Similarly, the validation data is composed of the validation data that is resized and the attack data. Both are to ensure that there is not a bias towards normal data. The test data will be composed of the normal data resized and the attack data, finally, the normal data will be the rest of the normal data left over. The data in each of these variables are run through the get\_adfa\_sequences described in **section 1** along with saved to the machine in text files.

**2b How these data subsets are used:**

I used two approaches for my project to determine the likelihood of attacks, where one was using training of only normal data, and no attack data is introduced (so any attack data would be considered an anomaly) and training where attack and normal data are evenly split.

Normal Data is composed of Training and validation files.

***For training and NO attack data included***

Training Set / Validation Set: Normal data only. 70%

Testing Set: Normal data only. 30%

***For training with attack data included***

Training Set: Normal files and attack files concatenated

Validation Set: Normal files and attack files concatenated

Testing Set: Testing files (Normal data and attack data)

The above sets contain the same attack data for each.

dataTrain = load\_files('train')  
dataAtt = load\_files('attack')  
dataVal = load\_files('val')  
  
dataTrainResized = one\_one\_ratio(dataTrain, dataAtt)  
dataValResized = one\_one\_ratio(dataVal, dataAtt)  
  
dataNormal = np.concatenate((dataTrain, dataVal), 0)  
dataNormalResized, dataNormalLeftOver = one\_one\_ratio(dataNormal, dataAtt)  
  
dataTest = np.concatenate((dataAtt,dataNormalResized), 0)

To create the data sets they are run through the one to one ratio, by using the first n attack files to match the n files in the data for training, validation, and testing sets respectively.

The testing set is split up so that the neural network does not see any of the normal data used in the training set.

The data in the sequences is described in an example below:

To generate the sequence\_pairs, to train the model, I am using a function that takes each file in and will return two lists out. One list will be our trainable data and one list will be our target data.

Let’s say we have Trace File 1: [1,2,3,4], and Trace File 2: [5,6,7,8] to create engrams that will relate to each other, I have created NGrams, which will represent these lists.

We will get two sequences from the being: [(1,2,3,4)], [(5,6,7,8)]

If we have 2 sequences and want a seq\_len of 2 this function does the following:

Since we are using a seq\_len of 2, in this scenario, we will take the first 2 X values, and the next 2 values to be the Y value giving us X[0] = [[1,2]] Y[0] = [[3,4]]. Next, the window slides by 1 to start at 2 and attempt to take [2,3] for X, but since there are not enough values to go to Y in the first trace file, we move to the next trace file, and we get. X[1] = [[5,6]] Y[1] = [[7,8]]

Show X and Y and we see our sequence pairs: X = [[1,2], [5,6]] Y = [[3,4], [7,8]]. Giving 2Grams as a result.

This same idea can be scaled upwards to sequence length of n.

In the following example code, there is a Boolean flag: skip passed in which instead of sliding the window up by 1, it will slide the window by the seq\_len: ex: seq = [[1,2,3,4,5], [6,7,8,9,10]] will return: X = [[1, 2], [6,7]] Y = [3,4], [8,9]]

def get\_adfa\_sequence\_group(data, seq\_len, skip, keep\_nested):  
 encoder = Encoder("../data/encoder.npy")  
 vec\_encode = np.vectorize(encoder.encode)  
 # vec = np.vectorize(data)  
 x\_val = []  
 y\_val= []  
 for row in data:  
 x, y = generate\_sequence\_pairs(vec\_encode(row), seq\_len, skip=skip)  
 x\_val.append(x)  
 y\_val.append(y)  
  
 if not keep\_nested:  
 x\_val = np.concatenate(x\_val)  
 y\_val = np.concatenate(y\_val)  
  
 return x\_val, y\_val

**3a Model Construction:**

After preprocessing is completed and all of the data is saved into files and run through the generator, the program then is set with an integer flag which determines which Neural network model to train under. Following the initial paper, I have created a model that uses different learning rates and a different number of layers.

# Change this number to change the neural net test  
# 0 -> nothing  
# 1 -> 1 layer lr = 0.1  
# 2 -> 2 layers lr = 0.1  
# 3 -> 3 layers lr = 0.1  
# 4 -> 4 layers lr = 0.001

Once the user determines which type of model to train on we pass this into the run function. This function takes in the test to run, sequence length, vocab size, training data, validation data, number of epochs, batch size, and trials. The paper kept the batch size at 64 constant, and epochs I have locked into being 10, along with the number of trials being 10.

if test\_to\_run == 1:  
 # model1  
 '''  
 :Hidden layers: 1  
 :nodes per layer: 256  
 :training batch: 64  
 :dropout: 0.5  
 :lr: 0.1  
 :maximum graident clipping threshold: 5  
 :return: model to train  
 '''  
 for idx in range(trials):  
 model = create\_gru\_test1(seq\_len, vocab\_size, layers=1, lr=0.1)  
 train\_gru\_test(train\_data\_gen, val\_data\_gen, test\_to\_run, model=model,

epochs=epochs, batch\_size=batch\_size, patience=0)

The code above shows an example of choosing model 1. This will pass in the sequence length, vocab size, number of layers, and the learning rate to create the model.

The model that will be trained on will be a Gated Recurrent Unit (GRU) neural network for sequential learning. This decision was to follow the paper, and after researching it seems the best networks to train sequential data on is any type of recursive neural network (RNN). The two best are GRUs and Long Short-Term Memory (LSTM) (artificial RNN). I have decided to follow the paper of using GRUs instead of an LSTM for training. The main difference is that GRU networks are easier to implement with each layer only containing two gates being the reset and update gate vs LSTM using an input gate, output gate, forget gate. A GRU can control the flow of information without using a memory unit while LSTM requires it.

This model is created using Keras and is going to be sequential. After saying the model is Sequential I add in a base layer of a GRU. Next to follow the paper I implement a Self-attention layer to relate different positions of a single sequence to compute a representation of the same sequence. Depending on the size of the GRU network there is a for loop to determine the number of layers to train on, the for loop adds 1 less than the size of the neural network we want to construct. Since the paper I am recreating has 4 models, with the only difference being the number of GRU layers this for loop lets me create n-1 total layers. For the regularization of the data, a dropout layer is added after each of these layers. Lastly, to get the last layer for a total of n layers I add one last GRU layer. Each layer uses ‘relu’ as an activation since it is considered the best learning type for neural networks.

The model uses Adam as its optimizer with our learning rate which is passed in, with a clipping value set to 5, to avoid exploding gradients. Compiling the model for its loss I use categorical cross-entropy because the data is one hot encoded from the generators. Categorical cross-entropy with one hot encoding uses the true class as the one-hot encoding and the closer the model's output is to the vector the lower the loss will be. For accuracy scoring on the data, I use categorical accuracy which evaluates the index of the maximum true value to the index of the maximum prediction value. I also keep a perplexity score which is defined as how well a probability model predicts a sample. The scoring is done between 1 and ‘n’ perplexity is considered perfect when it is 1 meaning the data is not complex at all, and the higher the n value means the data is that complex. Since perplexity is defined as 2^(entropy) this means that the higher the entropy is the worse the model is at predicting, and if we have an entropy score of 0 which will result in a perplexity score of 1.

def create\_gru\_test1(seq\_len=8, vocab\_size=175, layers=2, lr=0.1):  
 *'''  
  
 :Hidden layers: 1  
 :nodes per layer: 256  
 :training batch: 64  
 :dropout: 0.5  
 :lr: 0.1  
 :maximum graident clipping threshold: 5  
 :return: model to train  
 '''* model = Sequential()  
  
 model.add(keras.layers.GRU(vocab\_size, input\_shape=(seq\_len, vocab\_size),

activation='relu', return\_sequences=True))  
 model.add(SeqSelfAttention(attention\_activation='sigmoid'))  
 for i in range(layers-1):  
 model.add(keras.layers.GRU(vocab\_size, activation='relu',

return\_sequences=True))  
 model.add(keras.layers.Dropout(0.5))  
  
  
 model.add(keras.layers.GRU(vocab\_size, activation='softmax',

return\_sequences=True))  
  
 opt = optimizers.adam(lr=lr, clipvalue=5.0)  
  
 model.compile(loss='categorical\_crossentropy',  
 optimizer=opt,  
 metrics=['categorical\_accuracy', perplexity])  
 model.summary()

**3b Running the model:**

train\_gru\_test(train\_data\_gen, val\_data\_gen, test\_to\_run, model=model, epochs=epochs,

batch\_size=batch\_size, patience=0)

Now that the model is created running the model just requires passing in the train data, validation data, model itself, number of epochs, and the patience.

Since we have used a generator, we do not need to tell the model to fit using a batch size. The patience field is to say that we stop training as soon as the model’s loss does not reduce anymore after an epoch.

def train\_gru\_test(train\_data, val\_data, test\_to\_run, model: Model, epochs=int, batch\_size=int, patience=int) -> None:  
 *"""Trains model of ADFA-LD data  
 Model: First GRU test  
 Args:  
 model: Keras model to train  
 skip: see generate\_sequence\_pairs()  
  
 Returns:  
  
 """*  callbacks = [EarlyStopping(monitor='val\_loss', patience=patience),  
 ModelCheckpoint(filepath='tests/'+str(test\_to\_run) +

'/best\_model\_acc{categorical\_accuracy:02f}.hdf5',

monitor='val\_loss', save\_best\_only=True)]  
  
 history = model.fit(  
 x=train\_data, # train features  
 epochs=epochs, # num epochs  
 callbacks=callbacks, # Early stopping  
 verbose=1, # Verbose on  
 validation\_data=val\_data, # Data for eval  
 shuffle=True # Shuffle the data  
 )  
  
 figure, axes = pyplot.subplots(nrows=2, ncols=2)  
 axes[0,0].plot(history.history['categorical\_accuracy'], label='train')  
 axes[0,0].plot(history.history['val\_categorical\_accuracy'], label='test')  
 axes[0,0].legend()  
 # axes[0,0].show()  
  
 axes[0,1].plot(history.history['perplexity'], label='perplex\_train')  
 axes[0,1].plot(history.history['val\_perplexity'], label='perplex\_test')  
 axes[0,1].legend()  
 # axes[0,1].show()  
  
 figure.tight\_layout()  
 figure.show()

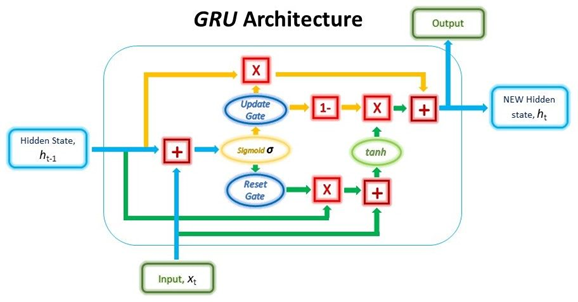
The model will use the test number we are running and save the model out with early stopping to a folder with the number of the test, and a file name showing the models accuracy each time the program is stopped early.

The actual running of the model takes in the training features, the number of epochs, the validation data, and will shuffle the data to ensure there is no biasing in the training. The generator also will shuffle data after each epoch to help the biasing issue as well.

This function will plot the accuracy and perplexity graphs. (I will talk about these results in **section 7**)

**4a How the model is trained:**

**Figure 4.1**

 Image: <https://blog.floydhub.com/gru-with-pytorch/>

Briefly mentioned in **section 3a** the model is trained, by using a loss metric of categorical cross-entropy. The main way the model will consider what is ‘good’ or ‘bad’ in a macro sense is based on if the model’s loss is decreasing. The lower the loss is the lower the error is between the output and the desired target values. The final scoring of accuracy used for the graphs is the training accuracy, and validation accuracy which will show the training over time and show if the model is over or underfitting. In a smaller view, each layer of the GRU has its way of ‘remembering’ information. Each layer will take in a sequence and learn what the important values are by weighting them towards a value of 1, and less frequent information will be ‘forgotten’. The main issue with this is the vanishing gradient and GRUs are better than LSTMs at solving this issue, by removing the cell state and using its hidden state to transfer information. Since there are only 2 gates, the GRU uses its update gate to function in a similar way to that of the LSTMs forget and input gate, where it decides what information to get rid of by implementing a sigmoid function on the data which will further rank the data between 0 and 1, where the closer to 0 is thrown away respectively to 1 is kept. The reset gate works on deciding how much of its previous information to forget. The output for the RNNs is the same by using a tanh function which regulates the information between -1 and 1 to keep the values from becoming highly significant or insignificant. For reference to a GRU, see **Figure 4.1.**

**5a Scoring sequences as an Attack or not for classification.**

Once training is completed, we grab our favorite model from one of the folders. We pass this model and our initial validation data and testing data into a function that will calculate the auc of the model compared against true data from the original generator data.

model = load\_model(model)  
  
# Get the predicted model scores.  
predX = val\_sequences[0][0]  
predY = test\_sequences[0][1]  
predv = [predX, predY]  
prediction = np.concatenate(predv)  
pred = model.predict(prediction)  
# This will contain a shape of (64, 10, 177) # Batch size, seq len, vocab size  
  
  
# Next we need to get our target which we will use as truth values... we can choose valseq[1][0] and testseq[1][1]  
# This is the target  
trgtx = val\_sequences[1][0]  
trgty = test\_sequences[1][1]  
trgtv = [trgtx, trgty]  
trgt = np.concatenate(trgtv)  
  
if len(trgt) == len(pred):  
 print("Matching lengths of length: " + str(len(pred)))  
  
# Since they are in sequence format we need to turn them into binary.  
# Score 1 for Not attack and Score 0 for attack  
# First we need to get some attack data to compare against.  
atk = atk\_sequences[0][0]  
normal = val\_sequences[0][0]  
  
# Create the truth comparative: This is a mix of the normal and attack values.  
truth = np.concatenate(  
 [np.ones(len(normal)), np.zeros(len(atk))]  
)

This function will first load our model using Keras.load\_model and set this to a variable. Once we have our model loaded, we need to give it data to process, to become our prediction sequence. Since the preprocessing of the data was split into X and Y values, I need to create a predictionX, and predictionY. These variables will be set to the validation data of X at the zero index and the testing data of Y at the zero index. Remember back that the testing data has never been seen by the model before, and that the validation data is a mix of normal and attack data. Creating the prediction array will be [predictionX, predictionY]. For a ROC graph with the area under the curve (auc) I need to also provide target values that will not be run through the model. The True values are calculated as targetX, and target using the validation data of X at the first index, and the testing data of Y at the first index. Again, these will create the target array of [targetX, targetY]. Now that both of these arrays are made, I call NumPy to concatenate to get the sequences into a shape of (128, 10, 177), while previously they were two arrays of shape (64, 10, 177). Once the prediction and target values are calculated, I create an array of values where the attacks are set to a value of 0, and the normal data is set to a value of 1. This is done by getting the length of the attack sequences (it could be the test sequences too they are the same length), and the length of the validation sequences. Then using np.zeros and np.ones respectively for attack and not attack it creates an array of 0’s at the length of the attacks and an array of 1’s at the length of validation data. The truth vector is then concatenated to be a sequence of all 0’s or 1’s.

Going back to the target and prediction sequences, for each sequence in the prediction sequence I get the length of each subsequence and append them into an array list. The final list should be similar to [10, 10, …, 10, 10] of length 128. Using this I create another list which will be a list of each of the subsequences.

file\_to\_idx = []  
offset = 0  
for num in num\_subsequences:  
 file\_to\_idx.append(np.arange(num) + offset)  
 offset += num  
# x\_data = np.concatenate(pred)

From here I concatenate the target data, to become a 2D tuple of shape (1280, 177). (The tuple will be 1280 sequences of length 177)

To calculate the probability of an attack or normal data I create a new variable called probs which will run a function get\_real\_probs, and I pass in the prediction and target data.

def get\_real\_probs(pred, trgt):  
 seq\_len = len(trgt[0])  
 probs = []  
 y\_pred = pred  
 y = trgt  
 for y\_elm, y\_pred\_elm in zip(y, y\_pred):  
 elm\_probs = []  
 for seq\_idx in range(seq\_len):  
 a = y\_elm[seq\_idx]  
 a = int(a)  
 elm\_probs.append(y\_pred\_elm[a][seq\_idx])  
 prob = max(elm\_probs)  
 a = -np.log(prob)  
 probs.append(-np.log(prob))  
 print(probs)  
 return np.array(probs)

This function sets the sequence length to the length of our target at index 0. (All the sequences lengths will match so this is just an arbitrary pick) I run a for loop on each subsequence in a zipped entity of the target, and prediction values. A new empty list of element probabilities is created. For each index of a sequence in the entire sequence, a new value is set to be the value of the sequence index at a subsequence. To ensure that we can run comparisons, I cast the target values to be integers from their original float values. (0.0 -> 0, 1.0 -> 1) Then append the value of the prediction values at the index of the new value, with the index of the sequence index. (The prediction values are a tuple of shape (10, 177) 10 sequences of length 177) Doing this lets us get a list of 177 values, which is the sequence at the first index. After getting the sequence list I take the max value of the sequence value. To get realistic scores I use a negative log function on the max value, this is to get the raw probabilities for the chosen class. The reason we choose the max value is to use the strongest class in the sequence.

This function’s final array will be a list of values. These values are the highest values in each sequence and have negative log applied. By choosing the predicted and the target sequences the function measures the probability of a specific sequence based on the sequences which were seen during the training. Get\_real\_probs does the following:







After the Elements list has been created for example:

maxValue([0.014231242, 0.0041650306, 0.011948042,…, 9.745137e-07] = 0.10971083

-np.log(0.10971083) = 

This function is repeated for the entire sequence of 176 values.

Giving an example following array: [2.2099073, 1.942878, 1.5952864,…, 1.5727472, 1.5112425, 1.6177171]

This list is a prediction of positive or negative attack classes, where the higher the number the more likely of an attack, since the values land between 1, and 2.5, a 1 would be the high probability of there not being an attack, and a 3 is a high probability of an attack occurring. Since these values are obtained from the highest values in the sequence, then they should be the most relevant for an attack occurring or not. (The lower the highest number means the lower the likelihood an attack occurred)

Once all of this is completed there will be a new list of prediction values to run against the true values.

scores =[]  
  
for n in range(len(file\_to\_idx)):  
 a = np.max(probs[n])  
 scores.append(a)  
plot\_roc(truth, scores)



Since the file to index is a list of lists each of length ten and counting from 0 to 177, we will set a value of a to be the map value of our probabilities sequence at index n. Our truth values are a sorted list of 64 0’s and 64 1’s and the prediction scores are a list of 128 float values.

To determine how these probabilities are turned into an attack vs baseline using the sklearn.roc\_curve we need the scores from the previous function to be tested against a true value. Typically the true values will be a 50% split between attack values and 50% to the normal data, which become a 0, and 1 respectively. Since we are classifying the attack values as 0 we set the positive label in this function as a 0. The prediction values will be the score array made above.

Y\_True: The true binary labels. Y\_Score: The target scores which are probability estimates of the positive class. Our scores as generated above are the negative natural log of the max value in each of the sequences. The max value helps determine how close to a score of 1 we obtain.

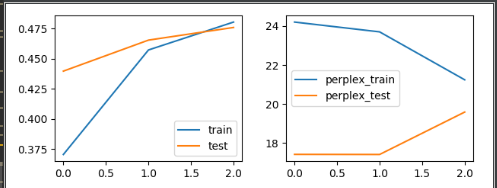
**5b Setting up the ROC for analysis**

Setting up the ROC is simple, and just requires calling sklearns metrics.roc\_curve, while passing in the true, and prediction values while setting the positive label to be 0, since we chose our attacks to be valued at 0. From here we calculate the roc auc, by calling metrics.auc on the false positives, and true positives returned from the roc\_curve.

def plot\_roc(y\_true, y\_pred):  
 from collections import Counter  
 print(Counter(y\_true)) # Expects 50% 0's and 50% 1's  
 fpr, tpr, thresholds = metrics.roc\_curve(y\_true, y\_pred, pos\_label=0)  
 print("fpr: " + str(fpr))  
 print("tpr: " + str(tpr))  
 print("thresh: " + str(thresholds))  
 roc\_auc = metrics.auc(fpr, tpr)  
  
 plt.figure()  
 plt.plot(fpr, tpr, label='ROC curve (area = %0.2f)' % roc\_auc)  
 plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')  
 # plt.xlim([0.0, 1.0])  
 # plt.ylim([0.0, 1.05])  
 plt.xlabel("False Positive Rate")  
 plt.ylabel('True Positive Rate')  
 plt.title("Receiver Operating Characteristic (ROC)")  
 plt.legend(loc="lower right")  
 plt.show()

**6a Evaluation of my Models compared to Johns base GRU network.**

Comparing my models Perplexity score to the paper.



A perplexity score is defined as a measurement of how well a probability distribution model accurately predicts a sample. To obtain the perplexity score I used the formula of taking the value of the categorical cross entropy of the model. From this value we use the formula of , which 2 to the power of entropy in bits. This score allows us to determine how easy a probability distribution is to learn. The lower the perplexity with 1 being the lowest, is the best possible outcome. A large perplexity score means that the probability distribution is wide, and therefore more difficult to learn.

def perplexity(y\_true, y\_pred):  
 cross\_entropy = K.categorical\_crossentropy(y\_true, y\_pred)  
 perplexity = K.pow(2.0, cross\_entropy)  
 return perplexity

The paper claimed to have obtained a score of .4 perplexity after 48 hours of training. From my research and my implementation of perplexity this is impossible and the lowest perplexity as mentioned in **section 3b** this was not possible. I did notice that my accuracy which is not a great metric did learn over time and may have benefited from increasing the patience value so that it may learn for longer. My perplexity for the training also decreased from a value of 2000 down to approximately 21. The graph only shows 2 epochs of time. In general, terms the higher the accuracy the better and the lower the perplexity is better. Since these scores only mean so much in the performance of a neural network I decided to implement a ROC curve which would be a better representation of how accurate a sequential model is.

To start I need to describe the model I decided to change my evaluations to compare against. Since the paper I was initially modeling seemed to have gone in strange directions I needed something more concrete to go against. I used some base code that John Ring provided for a simple GRU neural network which was able to train data. The model was simply one GRU layer and used categorical cross-entropy for its loss value, and categorical accuracy for its accuracy.

Before I moved into testing attack data vs training data, I wanted to make sure I could get ROC plots that were indicating the neural network was indeed learning despite its accuracy being only 40-50% with the best model.

**Figure 6.1 Figure 6.2**

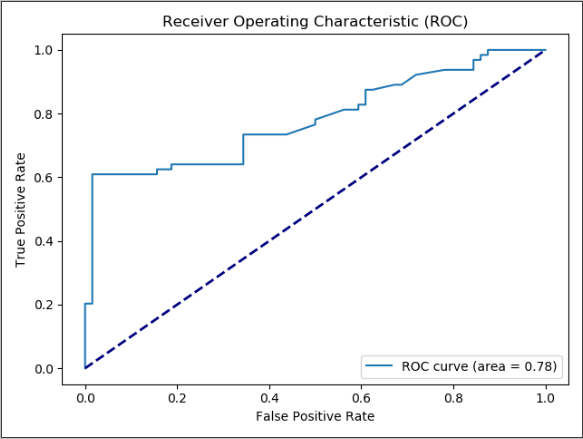
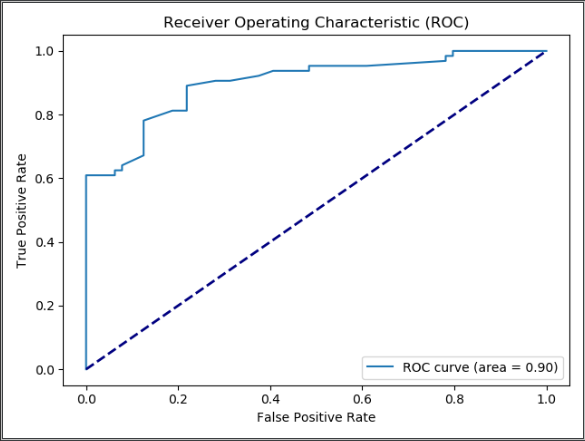


Figure 6.1 is a representation of a 4 layer GRU network using a learning rate of 0.01, 10 epochs, and a sequence length of 10. Figure 6.2 is John’s model of one GRU layer. (UVM IDS)

From just initial analysis before using attack data and normal data these graphs were plotted, where the model was only trained on normal data. The data was tested against another set of test data and attack data. This showed me that training a neural network even small on only normal data would present a high score of 78% auc and a larger deep neural network would be even better of 90%. See **figure 6.1** and **figure 6.2**.

I decided to retrain the model using both attack and normal data to see if this would improve the scores. This may be more representative of an actual network, where attack and training data exist. See **figure 6.3 and figure 6.4.**

**Figure 6.3 Figure 6.4**

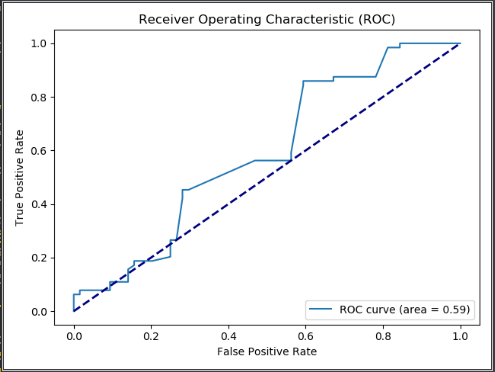
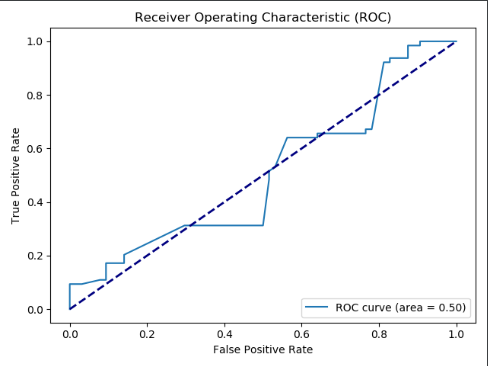
 

Figure 6.3 is a representation of a 4-layer GRU network using a learning rate of 0.01, 10 epochs, and a sequence length of 10.

Figure 6.4 is John’s model of one GRU layer.

Unfortunately, it seems that mixing the attack data in with the training data originally led to essentially a coin toss on evaluation. I did notice that as I increased my models with more complexity that the auc increased by 9-10%. If a model were to be more complex it may prove vital and important for becoming a better intrusion detection system, however, the cost of computation would drastically increase as well. Training each complex model on its own already takes apx 1-2 hours depending on the model I was training with the skip value initiated. If I were to remove the skip Boolean which slides the window by the sequence length every time to slide up by 1 instead, which would result in longer sequences, I expect the accuracy would also increase.

**7a Future Implementations**

The code for running a BLEU score is mostly created in my python file. Since the classification scores are already created in the ROC section of my code moving the sequences over and comparing against a reference value would not be difficult. The reference value is already coded into the program being a validation sequence at index 0, so the candidate sequence needs to just be a test sequence at index 0 which runs through the model. BLEU score is a bilingual evaluation understudy. This score is determined by comparing sequences of strings and scoring how similar they are to a reference sequence. For example if we have a sequence [the, is, a, test] or [0 ,1, 3, 2] as the candidate (which are the list of tokens we can use) with the reference being a list of sequences being [[this, is , a, test], [this, is, test]], this will receive a score of 1.0 meaning one of the references matches the candidate exactly. A BLEU score is a way to determine if a sequence perfectly matches the candidate. There can be a score between 0.0 (100% mismatching sequences) and 1.0 (100% matching sequences). More information can be found at “<https://machinelearningmastery.com/calculate-bleu-score-for-text-python/>”. I used this for research and initial creation of BLEU testing.

Changing some of the model architectures could lead to higher rates of accuracy and better ROC plots in the future. Also running the Keras.GRUCUDNN layers can improve the learning rate of the model, but the parameters are less tunable. This base code shows that using a 50/50 split on the attack and normal data is not great to learn on, and training on only normal data and evaluating against normal and attack data will yield high results. Future work could implement closer to an actual company or workforces attack to normal data for training, which may increase the accuracy scores and the ROC auc graphs.

Expanding on the written code could lead to cost-effective and accurate neural network intrusion prediction on system calls. Right now, running a model that only takes an hour to train (on an Nvidia GTX 2080ti) is not that cost expensive.

Since LSTMs are a good way to model sequential data, converting the neural network over to an LSTM may also lead to a positive outcome. An LSTM is generally better at training on a dataset with longer sequences. Since a GRU uses less training parameters it is faster than using an LSTM, however by using more training parameters, and a longer time to train using an LSTM is preferred where accuracy is critical or a larger sequence is used. Since the sequences in this paper are generally small, being only 175, using a GRU seemed to be the better path especially since they require less memory and train significantly faster. In general, GRU’s are being used more over LSTMs due to the increase in training time and similar results. Implementing an LSTM also is used to solve the vanishing gradient like the GRU, since the LSTM includes an extra gate setting up an LSTM requires more care and attention than a GRU does. By implementing an LSTM there may be an increase in accuracy over the GRU even if it takes longer to train if the developer puts care into the design.

**7b Related Work.**

<https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8540398>

This paper is how I created most of my code by following their process on the evaluation of the ADFA-LD dataset.

<https://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf>

To step outside of pure intrusion detection simulations, I am looking to standard sequence to sequence models using neural networks. The standard approach is the same using a Recurrent Neural Network. Using a 4-layer LSTM model with 1000 cells on each layer and 1000-dimensional word embeddings using an input vocabulary of 160,000, and an output vocabulary of 80,000. The research they have found is that deep LSTMs are significantly better in performance for sequential models over a shallow LSTM. In addition to this, each layer reduced the perplexity of a sequence by ~10% for every layer added which they presume was from their larger hidden state. My results show a similar property that the deeper my network becomes the more accurate results I obtain and the lower the perplexity is per each layer. Since my model is just a sequential dataset, I can treat the normal data sequences as “English” and the attack data sequences as “French” in the case of their paper. My main take away is that switching my model from a GRU to an LSTM may yield better results. Typically, when working with sequential data, both an LSTM and a GRU are trained on and the developer chooses the one with better results. The main difference from implementation is the amount of data trained on, and that not every normal data sequence will be represented as an attack sequence in the way that near every English sentence can be translated into a French sentence. The goal for my sequential model is to detect only when a normal sequence matches an attack sequence.

Any papers that delve into a sequence to sequence models can be followed for setting up a DNN (Deep Neural Network). From my research, it shows that the deeper the neural network is better at predicting the output sequence will be, and the lower the perplexity score will be.

**7c Citations.**

Intrusion Prediction With System-Call Sequence-to-Sequence Model: <https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8540398>

Sequence to Sequence Learning with Neural Networks: <https://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf>

GRU Diagram (Section 4.1): <https://blog.floydhub.com/gru-with-pytorch/>

BLEU References: <https://machinelearningmastery.com/calculate-bleu-score-for-text-python/>

GRU and LSTM comparisons: <https://medium.com/mindboard/lstm-vs-gru-experimental-comparison-955820c21e8b>  
UVM IDS Project Source Code (John’s Code):