MALICIOUS URL CLASSIFICATION AND DETECTION USING DEEP LEARNING

Abstract:

The purpose of the project is to detect malicious URLs using deep learning architectures. In this project I explored an implementation of a deep learning framework to learn a nonlinear URL embedding for Malicious URL Detection. I apply Convolutional Neural Networks and LSTM implementation of Recurrent Neural Networks to both characters of the URL String to learn the URL embedding in an optimized framework. I tested three architectures which implemented LSTM and Convolutional neural network models to identify the suitable implementation with highest accuracy. After careful evaluation, experimentation and testing of three different datasets with differing ranges, I was able achieve close to 90% accuracy when combining both Convolutional neural network and LSTM into one model. I achieved better accuracy and lower false positive rate

Introduction:

URL is the abbreviation of Uniform Resource Locator, which is the global address of documents and other resources on the World Wide Web. A URL has two main components: (i) protocol identifier, it indicates what protocol to use, (ii) resource name, it specifies the IP address or the domain name where the resource is located. The protocol identifier and the resource name are separated by a colon and two forward slashes. An example is shown in Figure 1.

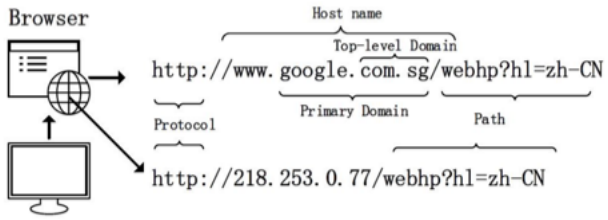


Fig. 1. Example of a URL - "Uniform Resource Locator"

A Malicious URL or a malicious web site hosts a variety of unsolicited content in the form of spam, phishing, or drive-by-exploits in order to launch attacks. Unsuspecting users visit such web sites and become victims of various types of scams, including monetary loss, theft of private information (identity, credit-cards, etc.), and malware installation. Effective systems to detect such malicious URLs in a timely manner can greatly help to counter large number of and a variety of cyber-security threats. The most common method to detect malicious URLs deployed by many antivirus groups is the black-list method. Black-lists are essentially a database of URLs that have been confirmed to be malicious in the past. This database is compiled over time (often through crowd-sourcing solutions, e.g. PhishTank, as and when it becomes known that a URL is malicious. However, it is almost impossible to maintain an exhaustive list of malicious URLs, especially since new URLs are generated every day. Attackers use creative techniques to evade blacklists and fool

users by modifying the URL to “appear” legitimate via obfuscation. Once the URLs appear legitimate, and user’s visit them, an attack can be launched. To overcome these issues, in the last decade, researchers have applied machine learning techniques for Malicious URL Detection.

Machine Learning approaches, use a set of URLs as training data, and based on the statistical properties, learn a prediction function to classify a URL as malicious or benign. One would need to extract suitable features based on some principles or heuristics to obtain a good feature representation of the URL. This may include lexical features (statistical properties of the URL string, bag of words, n-gram, etc.), host based features (WHOIS info, geo-location properties of the host, etc.). The quality of feature representation of the URLs is critical to the quality of the resulting malicious URL predictive model learned by machine learning. Finally, using the training data with the appropriate feature representation, the next step in building the prediction model is the actual training of the model. There are plenty of classification algorithms can be directly used over the training data (Naive Bayes, Support Vector Machine, Logistic Regression, etc.). While the above approaches have shown successful performance, they suffer from several limitations, particularly in the context of very large scale Malicious URL Detection: (i) Inability to effectively capture semantic or sequential patterns: They fail to effectively capture the sequence in which words (or characters) appear in the URL String; (ii) Require substantial manual feature engineering important features for the task (e.g. which statistical properties of the URL to use, what type of n-gram features would be better, etc.). (iii) Inability to handle unseen features.

About 80% of the work that goes into machine learning based cyber-attack detection is focused on feature engineering – identifying what values to extract from binaries, network packets, and other signals to use as inputs to our machine learning systems. In order to further improve detection capabilities and improve performance deep neural networks are used. Researchers are focusing on designing deep neural networks which automatically learn to extract features, identifying attributes of the data that lead to better detection performance than we could have achieved otherwise.

To address the above issues, I would like to experiment Deep Learning based solution for Malicious URL Detection. Deep Learning uses layers of stacked nonlinear projections in order to learn representations of multiple levels of abstraction. It has demonstrated state of the art performance in many applications (computer vision, speech recognition, natural language processing, etc.). In particular, Convolutional Neural Networks (CNNs) have shown promising performance for text classification in recent years. Following their success, I am experimenting the use of CNNs to learn a URL embedding for Malicious URL Detection. A URL string is used as input and applies CNNs to both characters in the URL to learn patterns and distinguish malicious and benign. For Character-level CNNs we first identify unique characters in the training corpus, and represent each character as a vector. Using this, the entire URL (a sequence of characters) is converted to a matrix representation, on which convolution can be applied. Character CNNs identify important information from certain groups of characters appearing together which could be indicative of maliciousness. I first implement a simple LSTM architecture. Long Short-Term Memory (LSTM) networks are a type of recurrent neural network capable of learning order dependence in sequence prediction problems.

Methods (Code and Documentation):

Dataset:

The dataset I received is 100,000 URLs from Virus Total among which 375,786 benign urls (no hits in VT) and 124,214 malicious urls (7+ hits in VT) are definitely malicious.

	A	B	C
1	url	malicious	
2	http://www.facebook-log1in.com/	1	
3	http://getir.net/yg4t	1	
4	http://crosscitydental.com/	0	
5	http://www.tenttrails.com/jscripts/product.js	0	
6	http://103.224.193.105/	0	
7	http://www.utopiapalmsandcycads.com/images/caryotagigas_002	0	
8	http://www.nephilimmerch.com/	0	
9	http://santostefano.sextantio.it/wp-content/plugins/gk-tabs/gk-ta	0	
10	http://support.itum.no/	0	

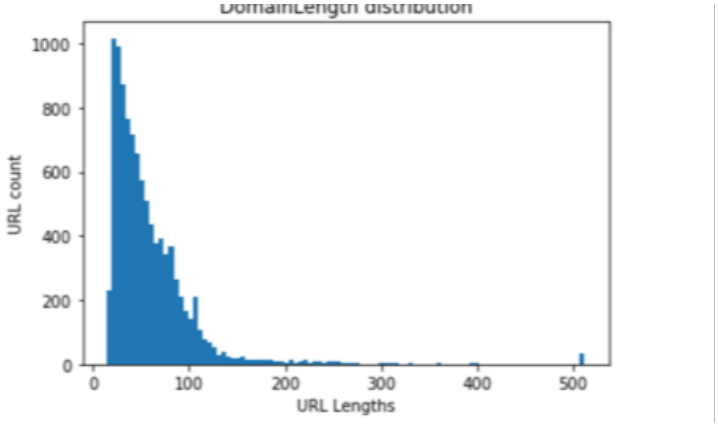
URL Pre-Processing:

For the Deep Learning approach, no hand-crafted features nor API queries are needed. However, limited pre-processing of the raw URLs is still necessary. The raw URL string needs to be split into "words". Very easily done, every single character can be considered a "word". In addition, each character has to be expressed as unique integer. This requires building a dictionary first. A short cut can be considering Python's 100 printable characters. (only relevant for English language).

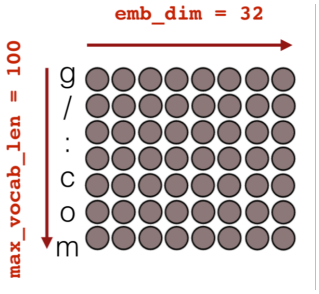
```
test_url = 'http://www.facebook-log1in.com/'
print([printable.index(x)+1 for x in test_url if x in printable])
```

```
[18, 30, 30, 26, 78, 77, 77, 33, 33, 33, 76, 16, 11, 13, 15, 12, 25, 25, 21, 75, 22, 25, 17, 2, 19, 24, 76, 13, 25, 2
3, 77]
```

The next step is to learn an embedding that captures the properties about the sequence in which the characters appear in a URL. We set the length of the sequence L1 = 100 characters. URLs longer than 100 characters would get truncated from the
100th character, and any URLs shorter than 100 would get padded till their lengths reached 100. A domain length distribution of my dataset shows majority if url have length under 100.



Each character is embedded into a k−dimensional vector. In our work, we choose k = 32 for characters. This embedding is randomly initialized and is learnt during training. For ease of implementation, these representations are stored in an embedding matrix as represented below.



Using an Embedding layer as input to the neural network is computationally efficient when using very big datasets because the embedded vectors also get updated during the training process of the deep neural network. Also instead of ending up with huge one-hot encoded vectors we can use an embedding matrix to keep the size of each vector much smaller and achieve to goal identifying similar patterns for malicious URL.

Input and Output Layers of Neural Networks:

I would like to show what the input and output layers implemented in this project would look like. The very first initial layer is always an input layer where you define the initial input shape (here initial 100 characters of the URL).

```
# Input
main_input = Input(shape=(max_len,), dtype='int32', name='main_input')
```

Next is the Embedding layer that is built upon the Input layer.

```
# Embedding layer
emb = Embedding(input_dim=max_vocab_len, output_dim=emb_dim, input_length=max_len,
            dropout=0.2, W_regularizer=W_reg)(main_input)
```

This general process of "chaining" would continue...

```
first_layer = <LayerType>(<parameters>)(main_input)
<next_layer> = <LayerType>(<parameters>)(first_layer)
<penultimate_layer> = <LayerType>(<parameters>)(<next_layer>)
```

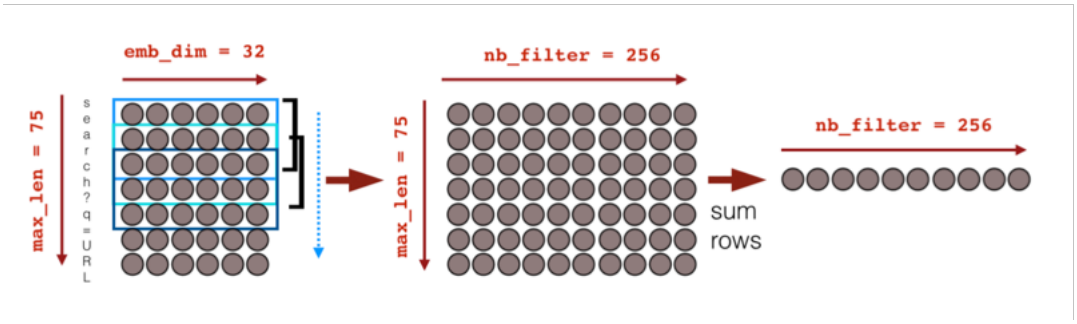
. until the output layer where the actual classification takes place. Since this project is about a binary classification task, a sigmoid activation function is used. Furthermore, since the target vector containing the labels is a binary 1D vector. In a multi-class scenario, the target vector would have needed to be one-hot encoded and the number of neurons would be n classes with a softmax activation function as an alternate option.

# Output layer (last fully connected layer)
output = Dense (1, activation='sigmoid', name='output')(<penultimate_layer>)

Convolution 1D with text data:

We now have a 2D array of "features" for each URL. One of the approaches I implemented is LSTM with a simple single later between embedded later and output layer. Long Short-Term Memory (LSTM) networks are a type of recurrent neural network capable of learning order dependence in sequence prediction problems.

A second approach I implemented is to run 1D convolutions, also called temporal convolutions as shown in the figure below. In contrast to 2D convolutions the width of the sliding window is of constant size of the emb_dim and the length (filter_length or kernel_size) here is 3 (see left panel). The window then slides along the embedded character sequence of the URL. Since I used strides=1 and "same" padding the resulting matrix is a 2D matrix with the same number of rows, but the dimension of the columns has changed to the number of "neurons" I specified (another parameter you can tune called nb_filter or filters). This convolutional process is indeed like a filter where mini-neural networks are fitted to the data as the window slides. In the process a more latent semantic structure is learned. To end up with a regular feature matrix simply take the sum of the rows. Again, thereafter the fully connected hidden layers can follow.



Implementing Deep Learning Architectures in Keras:

LSTM or 1D convolutions can be used separately or together for the URL classification task. In any case before arriving to the output layer, it is very common to add a few fully connected hidden layers. Finally, in every architecture the model has to be compiled while defining the optimizer and parameters such as learning rate to train the model in an iterative fashion as well as the loss function that quantifies how well the model is performing at each iteration.

```
# Compile model and define optimizer
    model = Model(input=[main_input], output=[output])
    adam = Adam(lr=1e-4, beta_1=0.9, beta_2=0.999, epsilon=1e-08, decay=0.0)
    model.compile(optimizer=adam, loss='binary_crossentropy', metrics=['accuracy'])
```

Simple LSTM

```
def  simple_lstm(max_len=75,  emb_dim=32,  max_vocab_len=100,  lstm_output_size=32,
W_reg=regularizers.l2(1e-4)):
    # Input
    main_input = Input(shape=(max_len,), dtype='int32', name='main_input')
    # Embedding layer
    emb = Embedding(input_dim=max_vocab_len, output_dim=emb_dim, input_length=max_len,
                dropout=0.2, W_regularizer=W_reg)(main_input)

    # LSTM layer
    lstm = LSTM(lstm_output_size)(emb)
    lstm = Dropout(0.5)(lstm)

    # Output layer (last fully connected layer)
    output = Dense(1, activation='sigmoid', name='output')(lstm)

    # Compile model and define optimizer
    model = Model(input=[main_input], output=[output])
    adam = Adam(lr=1e-4, beta_1=0.9, beta_2=0.999, epsilon=1e-08, decay=0.0)
    model.compile(optimizer=adam, loss='binary_crossentropy', metrics=['accuracy'])
    return model
```

This is a simple one LSTM layer architecture which uses sigmoid as the activation function and Binary cross entropy for loss function. Dropout refers to ignoring units (i.e. neurons) during the training phase of certain set of neurons which is chosen at random. By “ignoring”, I mean these units are not considered during a particular forward or backward pass. This is done to prevent over-fitting.

Model features

1. Activation function: Sigmoid.
2. Loss Function binary_crossentropy
3. Number of Epochs 10

4. Optimizer: Adam: is an optimization algorithm that can used instead of the classical stochastic gradient descent procedure to update network weights iterative based in training data.

Here is the connectivity in the layer I implemented.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
main_input (InputLayer)      (None, 100)               0
_________________________________________________________________
embedding_1 (Embedding)      (None, 100, 32)           3200
_________________________________________________________________
lstm_1 (LSTM)                (None, 32)                8320
_________________________________________________________________
dropout_1 (Dropout)          (None, 32)                0
_________________________________________________________________
output (Dense)               (None, 1)                 33
=================================================================
Total params: 11,553
Trainable params: 11,553
Non-trainable params: 0
_________________________________________________________________
```

Final Cross-Validation Accuracy 83.6% for 10 epoch and batch_size of 32. Total input is 10,000 URL and sample training set is 7498.

1D Convolutions and Convoluted Neural Networks:

```
 def conv_fully(max_len=75, emb_dim=32, max_vocab_len=100, W_reg=regularizers.l2(1e-
4)):
    # Input
    main_input = Input(shape=(max_len,), dtype='int32', name='main_input')
    # Embedding layer
    emb = Embedding(input_dim=max_vocab_len, output_dim=emb_dim, input_length=max_len,
                W_regularizer=W_reg)(main_input)
    emb = Dropout(0.25)(emb)


    def sum_1d(X):
        return K.sum(X, axis=1)

    def get_conv_layer(emb, kernel_size=5, filters=256):
        # Conv layer
        conv = Convolution1D(kernel_size=kernel_size, filters=filters, \
                     border_mode='same')(emb)
        conv = ELU()(conv)

        conv = Lambda(sum_1d, output_shape=(filters,))(conv)
        #conv = BatchNormalization(mode=0)(conv)
        conv = Dropout(0.5)(conv)
        return conv

    # Multiple Conv Layers

    # calling custom conv function from above
    conv1 = get_conv_layer(emb, kernel_size=2, filters=256)
```

```
    conv2 = get_conv_layer(emb, kernel_size=3, filters=256)
    conv3 = get_conv_layer(emb, kernel_size=4, filters=256)
    conv4 = get_conv_layer(emb, kernel_size=5, filters=256)

    # Fully Connected Layers
    merged = concatenate([conv1,conv2,conv3,conv4], axis=1)

    hidden1 = Dense(1024)(merged)
    hidden1 = ELU()(hidden1)
    hidden1 = BatchNormalization(mode=0)(hidden1)
    hidden1 = Dropout(0.5)(hidden1)

    hidden2 = Dense(1024)(hidden1)
    hidden2 = ELU()(hidden2)
    hidden2 = BatchNormalization(mode=0)(hidden2)
    hidden2 = Dropout(0.5)(hidden2)

    # Output layer (last fully connected layer)
    output = Dense(1, activation='sigmoid', name='output')(hidden2)

    # Compile model and define optimizer
    model = Model(input=[main_input], output=[output])
    adam = Adam(lr=1e-4, beta_1=0.9, beta_2=0.999, epsilon=1e-08, decay=0.0)
    model.compile(optimizer=adam, loss='binary_crossentropy', metrics=['accuracy'])
    return model
```

in this architecture I have 4 fully connected layers' custom convolution layers.

Layer (type)	Output Shape	Param #	Connected to
main_input (InputLayer)	(None, 100)	0	
embedding_2 (Embedding)	(None, 100, 32)	3200	main_input[0][0]
dropout_2 (Dropout)	(None, 100, 32)	0	embedding_2[0][0]
conv1d_1 (Conv1D)	(None, 100, 256)	16640	dropout_2[0][0]
conv1d_2 (Conv1D)	(None, 100, 256)	24832	dropout_2[0][0]
conv1d_3 (Conv1D)	(None, 100, 256)	33024	dropout_2[0][0]
conv1d_4 (Conv1D)	(None, 100, 256)	41216	dropout_2[0][0]
elu_1 (ELU)	(None, 100, 256)	0	conv1d_1[0][0]
elu_2 (ELU)	(None, 100, 256)	0	conv1d_2[0][0]
elu_3 (ELU)	(None, 100, 256)	0	conv1d_3[0][0]
elu_4 (ELU)	(None, 100, 256)	0	conv1d_4[0][0]
lambda_1 (Lambda)	(None, 256)	0	elu_1[0][0]
lambda_2 (Lambda)	(None, 256)	0	elu_2[0][0]
lambda_3 (Lambda)	(None, 256)	0	elu_3[0][0]
lambda_4 (Lambda)	(None, 256)	0	elu_4[0][0]
dropout_3 (Dropout)	(None, 256)	0	lambda_1[0][0]
dropout_4 (Dropout)	(None, 256)	0	lambda_2[0][0]
dropout_5 (Dropout)	(None, 256)	0	lambda_3[0][0]
dropout_6 (Dropout)	(None, 256)	0	lambda_4[0][0]
concatenate_1 (Concatenate)	(None, 1024)	0	dropout_3[0][0] dropout_4[0][0] dropout_5[0][0] dropout_6[0][0]
dense_1 (Dense)	(None, 1024)	1049600	concatenate_1[0][0]
elu_5 (ELU)	(None, 1024)	0	dense_1[0][0]
batch_normalization_1 (BatchNor	(None, 1024)	4096	elu_5[0][0]
dropout_7 (Dropout)	(None, 1024)	0	batch_normalization_1[0][0]
dense_2 (Dense)	(None, 1024)	1049600	dropout_7[0][0]
elu_6 (ELU)	(None, 1024)	0	dense_2[0][0]
batch_normalization_2 (BatchNor	(None, 1024)	4096	elu_6[0][0]
dropout_8 (Dropout)	(None, 1024)	0	batch_normalization_2[0][0]
output (Dense)	(None, 1)	1025	dropout_8[0][0]

Model features

1. 4 Fully connected layers with 2 dense later
2. Activation function used is sigmoid on the final dense later.
3. Optimizer in used is Adam.
4. Loss function in use Binary Cross entropy.
5. With 5 Epochs and a batch size of 32, Final Cross-Validation Accuracy 80.6

Each convolution layer is individually called and ELU is used as an activation function
On the final dense later the activation function is sigmoid.

1D Convolutions with Bi-directional LSTM and CNN.

This architecture uses a combination of Bi-directional LSTM and convolutional neural networks.

```
# sentence input
in_sentence = Input(shape=(max_len,), dtype='int64')
# char indices to one hot matrix, 1D sequence to 2D
embedded = Lambda(binarize, output_shape=binarize_outshape)(in_sentence)
# embedded: encodes sentence
for i in range(len(nb_filter)):
    embedded = Conv1D(filters=nb_filter[i],
                      kernel_size=filter_length[i],
                      padding='valid',
                      activation='relu',
                      kernel_initializer='glorot_normal',
                      strides=1)(embedded)
    embedded = Dropout(0.2)(embedded)
    embedded = MaxPooling1D(pool_size=pool_length)(embedded)
bi_lstm_sent = \
    Bidirectional(LSTM(100, return_sequences=True, dropout=0.15, recurrent_dropout=0.15, implementation=0))(embedded)



# compute importance for each step
attention = TimeDistributed(Dense(1, activation='tanh'))(bi_lstm_sent)
attention = Flatten()(attention)
attention = Activation('softmax')(attention)
attention = RepeatVector(200)(attention)
attention = Permute([2, 1])(attention)


# apply the attention
sent_representation = Multiply()([bi_lstm_sent, attention])
sent_representation = Lambda(lambda xin: K.sum(xin, axis=1))(sent_representation)


# sent_encode = merge([forward_sent, backward_sent], mode='concat', concat_axis=-1)
sent_encode = Dropout(0.4)(sent_representation)
sent_encode = Activation("relu")(sent_encode)
sent_encode = Dense(units=2)(sent_encode)
sent_encode = Activation("softmax")(sent_encode)

# sentence encoder
```

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	(None, 100)	0	
lambda_5 (Lambda)	(None, 100, 71)	0	input_1[0][0]
conv1d_5 (Conv1D)	(None, 96, 128)	45568	lambda_5[0][0]
dropout_9 (Dropout)	(None, 96, 128)	0	conv1d_5[0][0]
max_pooling1d_1 (MaxPooling1D)	(None, 48, 128)	0	dropout_9[0][0]
conv1d_6 (Conv1D)	(None, 46, 196)	75460	max_pooling1d_1[0][0]
dropout_10 (Dropout)	(None, 46, 196)	0	conv1d_6[0][0]
max_pooling1d_2 (MaxPooling1D)	(None, 23, 196)	0	dropout_10[0][0]
conv1d_7 (Conv1D)	(None, 21, 256)	150784	max_pooling1d_2[0][0]
dropout_11 (Dropout)	(None, 21, 256)	0	conv1d_7[0][0]
max_pooling1d_3 (MaxPooling1D)	(None, 10, 256)	0	dropout_11[0][0]
bidirectional_1 (Bidirectional)	(None, 10, 200)	285600	max_pooling1d_3[0][0]
time_distributed_1 (TimeDistrib	(None, 10, 1)	201	bidirectional_1[0][0]
flatten_1 (Flatten)	(None, 10)	0	time_distributed_1[0][0]
activation_1 (Activation)	(None, 10)	0	flatten_1[0][0]
repeat_vector_1 (RepeatVector)	(None, 200, 10)	0	activation_1[0][0]
permute_1 (Permute)	(None, 10, 200)	0	repeat_vector_1[0][0]
multiply_1 (Multiply)	(None, 10, 200)	0	bidirectional_1[0][0] permute_1[0][0]
lambda_6 (Lambda)	(None, 200)	0	multiply_1[0][0]
dropout_12 (Dropout)	(None, 200)	0	lambda_6[0][0]
activation_2 (Activation)	(None, 200)	0	dropout_12[0][0]
dense_4 (Dense)	(None, 2)	402	activation_2[0][0]
activation_3 (Activation)	(None, 2)	0	dense_4[0][0]

Total params: 558,015
Trainable params: 558,015
Non-trainable params: 0

The above diagram is summary of the model implemented and shows the connectivity between the layers.

Model features

1. 3 convolution layers and one bi directional LSTM later
2. Each layer uses a combination of softmax and relu activation function
3. Optimizer in used is Adam.
4. Loss function in use sparse categorical crossentropy.
5. With 10 Epochs and a batch size of 32, Final Cross-Validation Accuracy 85.46

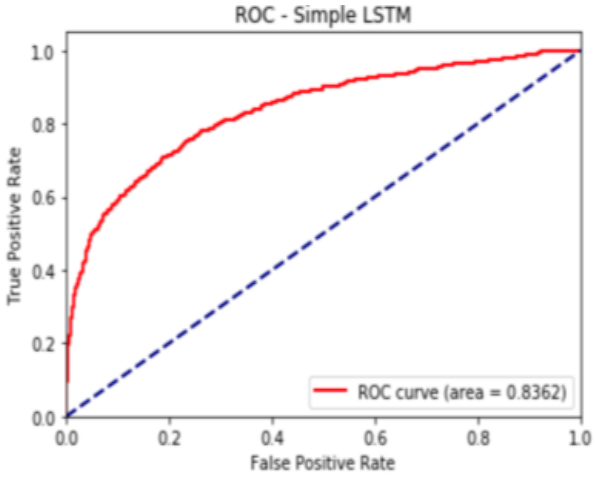
Results and Prediction

Metrics

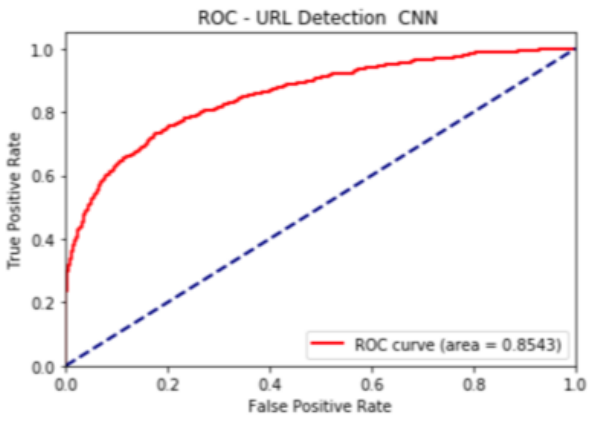
The ROC curve is created by plotting the true positive rate (TPR) against the false positive rate (FPR) at various threshold settings. The true-positive rate is also known as sensitivity, recall or probability of detection in machine learning. The false-positive rate is also known as the fall-out

or probability of false alarm and can be calculated as (1 − specificity) ROC analysis provides tools to select possibly optimal models and to discard suboptimal ones independently from (and prior to specifying) the cost context or the class distribution. ROC analysis is related in a direct and natural way to cost/benefit analysis of diagnostic decision making.

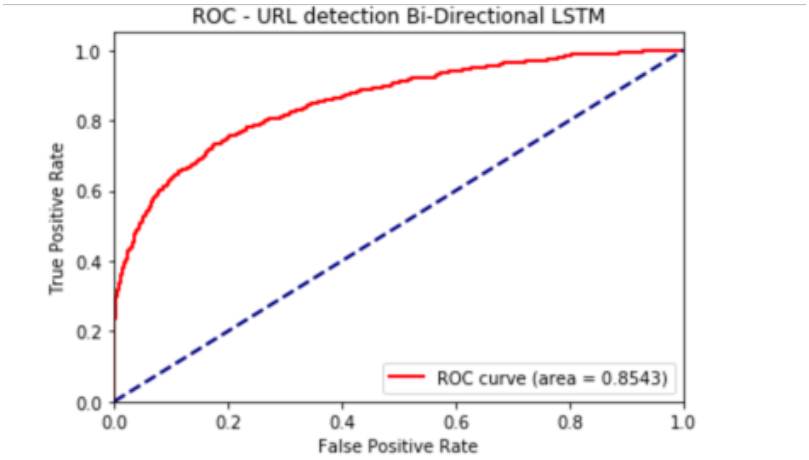
The below verification is done on the **Simple LSTM** model and the ROC curve indicates a higher false positive rate. Area for the ROC curve is 0.8634



The below verification is done on the **Convolutional Neural** network model and the ROC curve indicates a higher false positive rate. Area for the ROC curve is 0.8543



The below verification is done on the **Bi-Directional LSTM model** and the ROC curve indicates a false positive rate lower than the above two models. Area for the ROC curve is 0.8634



F1 score
In statistical analysis of binary classification, the F1 score (also F-score or F-measure) is a measure of a test's accuracy. It considers both the **precision p** and the **recall r** of the test to compute the score:

Precision is the number of correct positive results divided by the number of all positive results returned by the classifier,

Recall is the number of correct positive results divided by the number of all relevant samples (all samples that should have been identified as positive).

The **F1 score** is the harmonic average of the precision and recall, where an F1 score reaches its best value at 1 (perfect precision and recall) and worst at 0.

$$F_1 = \frac{2}{\frac{1}{\text{recall}} + \frac{1}{\text{precision}}} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

Precision is a description of random errors, a measure of statistical variability.
Accuracy has two definitions:

- More commonly, it is a description of systematic errors, a measure of statistical bias; as these cause a difference between a result and a "true" value, ISO calls this trueness.
- Alternatively, ISO defines accuracy as describing a combination of both types of observational error above (random and systematic), so high accuracy requires both high precision and high trueness.

$$\text{ACC} = \frac{\text{TP} + \text{TN}}{P + N} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

In simplest terms, given a set of data points from repeated measurements of the same quantity, the set can be said to be precise if the values are close to each other, while the set can be said to be accurate if their average is close to the true value of the quantity being measured. The two concepts are independent of each other, so a particular set of data can be said to be either accurate, or precise, or both, or neither

Metrics for the three models

Model	Accuracy	F1-score	AUC
Simple LSTM	83.6	0.57	0.83
Implementing CNN	81.08	0.41	0.8543
Bi-Directional LSTM with CNN	84.56	0.65	0.854

Prediction

In contrast to more traditional Machine Learning, where preprocessing and/or future engineering can be quite tedious to transform a new URL into feature vector space, it's quite straight forward in the approach proposed here.

1. Convert raw URL string in list of lists where characters that are contained in "printable" are stored encoded as integer.
2. Cut URL string at max length or pad with zeros if shorter
3. Make a new prediction with trained model

	site	score	prediction
0	wikipedia.com	0.007914	False
1	google.com	0.007872	False
2	pakistanifacebookforever.com	0.064834	False
3	www.radsport-voggel.de	0.008443	False
4	ahrenhei.without-transfer.ru	0.042111	False
5	www.itidea.it	0.008915	False
6	stackoverflow.com	0.009805	False
7	dzone.com	0.007869	False
8	keras.io	0.007861	False
9	www.tiktik.co.il	0.009828	False

Conclusion:

Using embedding's with convolutions, as input layer; neural network coupled with supervised training, allowed me to train the model to learn semantic features and patterns. Based on the implemented results, we can correctly infer that the deep learning architectures can be very effective in field of cybersecurity, in general, and malicious URL detection, in particular.

In the three models I experimented with its quite clear the LSTM and Recurrent neural networks are more efficient in learning the semantic model in comparison to the Convolutional Neural Networks. One of the major issues during my experimentation was the computational cost of training on longer strings, that prevented me from trying more complex architectures within the score of this project. With current advances in hardware and distributed training modules added to modern frameworks, the results can be improved with more computationally expensive architecture that I was unable to try.

Discussion and Future Scope:

- What I like about the Deep Learning approach for URL classification is that it doesn't require hand-crafted features and domain knowledge. Training time is definitely longer, however it may generalize better to tomorrow's malicious URLs.
- The model with the convolutional neural network alone does not achieve the expected accuracy. It also has high False positive and false negative rates.
- A combination of the LSTM with CNN or simple LSTM to learn the sequence patterns of the URL, better improves the models accuracy even with smaller data
- Further tuning can be achieved with optimizing each layer and also implementing an encoding that can better learn the URL patterns.
- Implementing distributed training modules with frequent adaptive learning can increase accuracy and improve effectiveness of detection.

Project GitHub Link:

https://github.com/VineelaPeri/Malicious-URL-Detection-DL

References:

Malicious URL Detection using Machine Learning
https://arxiv.org/pdf/1701.07179.pdf

Learning a URL Representation with Deep Learning for Malicious URL Detection
https://arxiv.org/pdf/1802.03162.pdf

Expose A Character-Level Convolutional Neural Network with Embedding for Detecting Malicious URLs
https://arxiv.org/pdf/1702.08568.pdf

Word2Vec Embedding
https://www.tensorflow.org/tutorials/word2vec

Activation Function
https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6

Convolution Neural Networks
https://wiki.tum.de/display/lfdv/Recurrent+Neural+Networks+Combination+of+RNN+and+CN
https://github.com/nikbearbrown/NEU_COE/tree/master/Deep_Learning/CNNs

RNN
https://github.com/nikbearbrown/NEU_COE/tree/master/Deep_Learning/RNNs/

LSTM
https://machinelearningmastery.com/develop-bidirectional-lstm-sequence-classification-python-keras/

Embedding Layers
https://towardsdatascience.com/deep-learning-4-embedding-layers-f9a02d55ac12/

Metrics
https://en.wikipedia.org/wiki/F1_score
https://en.wikipedia.org/wiki/Accuracy_and_precision