Assignment 7, RNN

November 25, 2018

1 Assignment 7 - RNN

Matthew Dunne

Recurrent Neural Network & Classification:

The objective is to detect the security breach by predicting suspicious access using an RNN model and the provided Logfile data. Logfile data includes login information like LogID, Timestamp, Method, Path, Status Code, Source, Remote Address, User Agent etc. The last indicator in each row denotes breach (1) and no breach (0) which is the target variable.

The expectation is that you will use the keras package to solve this problem.

1. Data Processing:

This data set is a bit messy, so the preprocessing portion is largely a tutorial to make sure students have data ready for keras.

a) Import the following libraries:

```
In [2]: import sys
        import os
        import json
        import pandas
        import numpy
        import optparse

        from sklearn.model_selection import train_test_split
        from keras.callbacks import TensorBoard
        from keras.models import Sequential, load_model
        from keras.layers import LSTM, Dense, Dropout
        from keras.layers.embeddings import Embedding
        from keras.preprocessing import sequence
        from keras.preprocessing.text import Tokenizer
        from collections import OrderedDict
```

```
Using TensorFlow backend.
```

b) We will read the code in slightly differently than before:

```
In [4]: dataframe = pandas.read_csv("dev-access.csv", engine='python', quotechar='|', header=N
```

c) We then need to convert to a numpy.ndarray type:

```
In [5]: dataset = dataframe.values
```

d) Check the shape of the data set - it should be (26773, 2). Spend some time looking at the data.

```
In [6]: dataset.shape

Out[6]: (26773, 2)
```

e) Store all rows and the 0th index as the feature data:

```
In [7]: X = dataset[:,0]
```

f) Store all rows and index 1 as the target variable:

```
In [8]: Y = dataset[:,1]
```

g) In the next step, we will clean up the predictors. This includes removing features that are not valuable, such as timestamp and source.

```
In [9]: for index, item in enumerate(X):
            # Quick hack to space out json elements
            reqJson = json.loads(item, object_pairs_hook=OrderedDict)
            del reqJson['timestamp']
            del reqJson['headers']
            del reqJson['source']
            del reqJson['route']
            del reqJson['responsePayload']
            X[index] = json.dumps(reqJson, separators=(',', ':'))
```

h) We next will tokenize our data, which just means vectorizing our text. Given the data we will tokenize every character (thus char_level = True)

```
In [10]: tokenizer = Tokenizer(filters='\t\n', char_level=True)
         tokenizer.fit_on_texts(X)

         # we will need this later
         num_words = len(tokenizer.word_index)+1
         X = tokenizer.texts_to_sequences(X)
```

i) Need to pad our data as each observation has a different length

```
In [11]: max_log_length = 1024
         X_processed = sequence.pad_sequences(X, maxlen=max_log_length)
```

j) Create your train set to be 75% of the data and your test set to be 25%

```
In [12]: X_train, X_test, y_train, y_test = train_test_split(X_processed, Y, test_size=0.25, ra
```

```
In [13]: X_train[20078]
```

```
Out[13]: array([ 0,  0,  0, ...,  1, 19, 19])
```

2. Model 1 - RNN:

The first model will be a pretty minimal RNN with only an embedding layer, LSTM layer, and Dense layer. The next model we will add a few more layers.

a) Start by creating an instance of a Sequential model: https://keras.io/getting-started/sequential-model-guide/

```
In [50]: model_1 = Sequential()
```

b) From there, add an Embedding layer: https://keras.io/layers/embeddings/

Params: - input_dim = num_words (the variable we created above) - output_dim = 32 - input_length = max_log_length (we also created this above) - Keep all other variables as the defaults (shown below)

```
In [51]: model_1.add(Embedding(input_dim=num_words, output_dim=32, input_length=max_log_length
```

c) Add an LSTM layer https://keras.io/layers/recurrent/#lstm

Params: - units = 64 - recurrent_dropout=0.5

```
In [52]: model_1.add(LSTM(units=64, recurrent_dropout=0.5))
```

d) Finally, we will add a Dense layer: https://keras.io/layers/core/#dense

Params: - units = 1 (this will be our output) - activation = relu

```
In [53]: model_1.add(Dense(units=1, activation='relu'))
```

e) Compile model using the .compile() method: https://keras.io/models/model/

Params: - loss = binary_crossentropy - optimizer = adam - metrics = accuracy

```
In [54]: model_1.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

f) Print the model summary

```
In [55]: model_1.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_4 (Embedding)      (None, 1024, 32)          2016
_________________________________________________________________
lstm_4 (LSTM)                (None, 64)                24832
_________________________________________________________________
dense_4 (Dense)              (None, 1)                 65
```

```
=================================================================
Total params: 26,913
Trainable params: 26,913
Non-trainable params: 0
_________________________________________________________________
```

g) Use the .fit() method to fit the model on the train data. Use a validation split of 0.25, epochs=3 and batch size = 128.

```
In [56]: model_1.fit(X_train, y_train, batch_size=128, epochs=3, validation_split=0.25)
```

```
Train on 15059 samples, validate on 5020 samples
Epoch 1/3
15059/15059 [==============================] - 738s 49ms/step - loss: 0.9064 - acc: 0.5128 - va
Epoch 2/3
15059/15059 [==============================] - 341s 23ms/step - loss: 0.6467 - acc: 0.5918 - va
Epoch 3/3
15059/15059 [==============================] - 336s 22ms/step - loss: 0.5921 - acc: 0.6062 - va
```

```
Out[56]: <keras.callbacks.History at 0x170bd6777f0>
```

h) Use the .evaluate() method to get the loss value & the accuracy value on the test data. Use a batch size of 128 again.

```
In [57]: model_1.evaluate(X_test, y_test, batch_size=128)
```

```
6694/6694 [==============================] - 49s 7ms/step
```

```
Out[57]: [0.5736413450518187, 0.5911263817988619]
```

This gives us a loss value of 0.574. We tried to minimize the loss value of binary cross-entropy. This also gives us an accuracy of 0.59, which does not seem especially good.

3. Model 2 - RNN + Dropout Layers + New Activation Function:

Now we will add a few new layers to our RNN and switch the activation function. You will be creating a new model here, so make sure to call it something different than the model from Part 2.

a) This RNN needs to have the following layers (add in this order):

- Embedding Layer (use same params as before)
- Dropout Layer (https://keras.io/layers/core/#dropout - use a value of 0.5 for now
- LSTM Layer (use same params as before)
- Dropout Layer - use a value of 0.5
- Dense Layer - (switch to a sigmoid activation function)

```
In [14]: model_2 = Sequential()
         model_2.add(Embedding(input_dim=num_words, output_dim=32, input_length=max_log_length)
         model_2.add(Dropout(rate=0.5))
         model_2.add(LSTM(units=64, recurrent_dropout=0.5))
         model_2.add(Dropout(rate=0.5))
         model_2.add(Dense(units=1, activation='sigmoid'))
```

b) Compile model using the .compile() method:

Params: - loss = binary_crossentropy - optimizer = adam - metrics = accuracy

```
In [13]: model_2.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

c) Print the model summary

```
In [14]: model_2.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 1024, 32)          2016
_________________________________________________________________
dropout_1 (Dropout)          (None, 1024, 32)          0
_________________________________________________________________
lstm_1 (LSTM)                (None, 64)                24832
_________________________________________________________________
dropout_2 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 65
=================================================================
Total params: 26,913
Trainable params: 26,913
Non-trainable params: 0
_________________________________________________________________
```

d) Use the .fit() method to fit the model on the train data. Use a validation split of 0.25, epochs=3 and batch size = 128.

```
In [15]: model_2.fit(X_train, y_train, batch_size=128, epochs=3, validation_split=0.25)
```

```
Train on 15059 samples, validate on 5020 samples
Epoch 1/3
15059/15059 [==============================] - 347s 23ms/step - loss: 0.6033 - acc: 0.6500 - va
Epoch 2/3
15059/15059 [==============================] - 336s 22ms/step - loss: 0.3529 - acc: 0.8716 - va
Epoch 3/3
15059/15059 [==============================] - 339s 23ms/step - loss: 0.2367 - acc: 0.9343 - va
```

```
Out[15]: <keras.callbacks.History at 0x2855338f320>
```

e) Use the .evaluate() method to get the loss value & the accuracy value on the test data. Use a batch size of 128 again.

```
In [16]: model_2.evaluate(X_test, y_test, batch_size=128)
```

```
6694/6694 [==============================] - 36s 5ms/step
```

```
Out[16]: [0.11570187640708349, 0.9772930981901321]
```

With a loss value of 0.116 (also for binary cross-entropy) and accuracy of 0.98, this model performs decidedly better.

4. Recurrent Neural Net Model 3: Build Your Own

You wil now create your RNN based on what you have learned from Model 1 & Model 2:

a) RNN Requirements:

- Use 5 or more layers
- Add a layer that was not utilized in Model 1 or Model 2 (Note: This could be a new Dense layer or an additional LSTM)

```
In [15]: model_3 = Sequential()
         model_3.add(Embedding(input_dim=num_words, output_dim=32, input_length=max_log_length)
         model_3.add(Dropout(rate=0.4))
         model_3.add(LSTM(units=64, recurrent_dropout=0.5))
         model_3.add(Dropout(rate=0.4))
         model_3.add(Dense(units=1, activation='relu'))
         model_3.add(Dense(units=1, activation='sigmoid'))
```

In essence what I have done is added a Dense layer with a dfferent activation function and lowered the Dropout rate in the Dropout layers.

b) Compiler Requirements:

- Try a new optimizer for the compile step
- Keep accuracy as a metric (feel free to add more metrics if desired)

```
In [16]: model_3.compile(loss='binary_crossentropy', optimizer='Nadam', metrics=['accuracy'])
```

I have used the Nadam Optimizer which is very similar to Adam.

c) Print the model summary

```
In [17]: model_3.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_2 (Embedding)      (None, 1024, 32)          2016
_________________________________________________________________
dropout_3 (Dropout)          (None, 1024, 32)          0
_________________________________________________________________
lstm_2 (LSTM)                (None, 64)                24832
_________________________________________________________________
dropout_4 (Dropout)          (None, 64)                0
```

```
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 65
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 2
=================================================================
Total params: 26,915
Trainable params: 26,915
Non-trainable params: 0
_________________________________________________________________
```

d) Use the .fit() method to fit the model on the train data. Use a validation split of 0.25, epochs=3 and batch size = 128.

```
In [18]: model_3.fit(X_train, y_train, batch_size=128, epochs=3, validation_split=0.25)

Train on 15059 samples, validate on 5020 samples
Epoch 1/3
15059/15059 [==============================] - 255s 17ms/step - loss: 0.6291 - acc: 0.7119 - va
Epoch 2/3
15059/15059 [==============================] - 245s 16ms/step - loss: 0.5350 - acc: 0.8202 - va
Epoch 3/3
15059/15059 [==============================] - 249s 17ms/step - loss: 0.4760 - acc: 0.8694 - va


Out[18]: <keras.callbacks.History at 0x1f45c398908>
```

e) Use the .evaluate() method to get the loss value & the accuracy value on the test data. Use a batch size of 128 again.

```
In [19]: model_3.evaluate(X_test, y_test, batch_size=128)

6694/6694 [==============================] - 36s 5ms/step


Out[19]: [0.33202416606721574, 0.9645951599692957]
```

This does not perform quite as well as Model 2, with a slightly lower Accuracy and a higher loss value. The extra dense layer probably does not add much.

Conceptual Questions:

5. Explain the difference between the relu activation function and the sigmoid activation function.

The sigmoid is a shape we are familiar with from logistic regression. For a given input into an activation function you have an output. For the sigmoid, all values of the function along the y-axis (the output of the activation function) are between 0 and 1 with the shape of the function itself having an S-like curve (sigmoid). Its gives you a probability of a binary outcome. If the probability is above a certain level the nueron will fire, meaning the node's output will pass to the next layer, or it will not.

The relu activation function has a different shape: 0 for all input values that are negative or 0, and then a linear function with a positive slope after that. Unlike the sigmoid, the output of the reLU activation function can be large, anywhere from 0 to infinity.

6. In regards to question 5, which of these activation functions performed the best (they were used in Model 1 & Model 2)? Why do you think that is?

As between Model 1 and Model 2 the Sigmoid activation function worked better than ReLU. This is because of the nature of the output type we want. Here we want an output of either 0 or 1, which is what a Sigmoid gives us, rather than a continuous value, which is what ReLU gives us (if it is above 0). The ReLU activation function can give us 0 or 1, but also other numbers and so it will tend to have less accuracy, at least in this context.

7. Explain how dropout works (you can look at the keras code) for (a) training, and (b) test data sets.

Dropout creates many networks deriving from the original one but, based on a probability you choose (p), drops nodes and their connections from the derivative neural networks during *training*. During training, Dropout will sample from a large number of these thinned derivative networks.

For the *test* data, you use the complete, original network, but you reduce each activation function by a factor of p (the probability of dropout you used on the training data).

This is a regularization technique meaning it will shrink the weights of inputs and prevent overfitting.

8. Explain why problems such as this are better modeled with RNNs than CNNs.

RNNs are better when the data is sequential. When you want to recongize patterns across time rather than across space, you would use a RNN.

Here each row of the data can be thought of to represent a sequence of steps for logging in. Sequence matters, and so a RNN is preferrable to a CNN.

9. Explain what RNN problem is solved using LSTM and briefly describe how.

LSTMs solve the Vanishing Gradient Problem. The gradient can be thought of as the measure of the change in all weights with regard to the change in error. The Vanishing Gradient Problem occurs when, during back propogation, the gradient becomes smaller and smaller as you move further back into the network. This means that the weights are updated less and less the farther back you go in the network. This is a problem because (1) those earlier layers are not as trained as the later layers and (2) those earlier layers still influence the later layers. Therefore the whole network is not properly trained.

The LSTM (Long Short-Term Memory) solves this by taking an input and storing it for some length of time. The result of this is that when the LSTM is trained through back propogation *the gradient does not vanish*.