Final Project

Fake News Detection Using Hybrid CNN-RNN Model

Pavankumar Pendela

ppendela@kent.edu

81174790

1.Introduction

Fake news is one of the greatest threats to commerce, journalism, and democracy all over the world, with huge collateral damages. The term ‘fake news’ is often described in related literature as ‘misinformation’, ‘disinformation’, ‘hoax’, and ‘rumor’, which are different variations of false information. *Misinformation* is used to refer to the spreading of false information disregarding the true intent.

The dissemination of fake news through different medium especially online platforms has not been stopped completely or scaled down to a degree to reduce the adverse effects fake news can lead to. The reason is that there is no system that exists that can control fake news with little or no human involvement.

Deep learning techniques have great prospect in fake news detection task. The model proposed is the hybrid neural network model which is a combination of convolutional neural networks and recurrent neural networks. As this model is required to classify between fake news and legitimate news, so this problem is cast as a binary classification problem.

2.Dataset Used

We are using Kaggle dataset.
Link: https://www.kaggle.com/c/fake-news/data?select=submit.csv

3.Word embeddings

When dealing with text classification and neural networks, the input text must take a vector or matrix numeric format so that it can be fed to the network. Words in the text can be represented as vectors, which are referred to as word vectors with each word having a unique word vector. These word vectors are referred to as word embeddings.

Word embeddings are trained from a large corpus, which is usually language specific, or domain specific. Instead of training word embeddings, it is more feasible to use publicly available pre-trained word embeddings. The most popular pre-trained word embeddings available are Word2Vec provided by Google, and GloVe.

In our model, we have used GloVe. The words from the text which are present in GloVe are kept and remaining are ruled out.

4.Convolution Neural Network

In the case of text classification or NLP, a one-dimensional CNN (Conv1D) is used generally. Conv1D deals with one dimensional array’s representing word vectors. In a CNN, a filter of fixed size window iterates through the training data, which at each step multiplies the input with the filter weights and gives an output that is stored in an output array. This output array is a feature map or output filter of the data.

5.Recurrent Neural Network

In the case of NLP, many news articles can be considered for learning relative to each other instead of separately learning each news article. RNN is composed of layers with memory cells. There are different types of memory cells to utilize in RNN. One such type is the Long Short-Term Memory (LSTM) unit or cell. LSTM consists of a cell state and a carry in addition to the current word vector in process as the sequence is processed at each time state. The carry is responsible to ensure that there is no information loss during the sequential process.

6.Hybrid CNN-RNN model

The proposed model makes use of the ability of the CNN to extract local features and of the LSTM to learn long-term dependencies. First, a CNN layer of Conv1D is used for processing the input vectors and extracting the local features that reside at the text-level. The output of the CNN layer (i.e. the feature maps) are the input for the RNN layer of LSTM units/cells that follows. The RNN layer uses the local features extracted by the CNN and learns the long-term dependencies of the local features of news articles that classify them as fake or real.

7.Implementation

The hybrid deep learning model is implemented on Google colab.

We have specifically used tensorflow, keras, numpy, pandas and sklearn packages.

7.1). Dataset splitting and pre-processing

The dataset is read as Pandas Data-Frame. The rows with missing values are dropped. The csv file has 5 attributes, we have taken only 2 of them.

X = text attribute and Y=label attribute.

Using, train_test_split() function of sklearn.model_selection; we split the dataset in 75-25 ratio in training and testing part.

7.2). Mapping text to vectors using word embeddings

“Tokenizer()” function from keras.preprocessing. text is used to tokenize the training set and created a usable vocabulary of 1,40,000 words.
All the texts are not of the same length, so post-padding is used to make all texts of same (size=400). Since, the texts are taken in article form, they comprise of 500-600 words. To keep it simple, we assumed text_size=400.

“pad_sequences()” function from keras.preprocessing.sequence is used to for padding.

“glove.6B.100d.txt” pre-trained word embedding file is used to convert each word into a dense vector of size=100.

Working: Tokenizer(num_words=150000).fit_on_texts(x_train) ->
First, it splits the entire x_train set into words and then creates the vocabulary. Second, it counts the frequency of each word and orders them in descending order. So, now the word with highest frequency sits at 1st index. Third, indexes are allotted to these words in dictionary format. The word with maximum frequency sits at index=1 and so on.

This vocabulary is created by the name “word_index”. Fourth, each text in the x_train is converted in a integer vector with size=length of text. Each word of these text is replaced by the index of that word in the “word_index” dictionary.

So, now we have the converted the texts into numbers but still show is on.

pad_sequences(x_train, padding= ‘post’, maxlen=400) -> all the texts are not of same size. We decide to set the maxlen of all texts as 400. The longer texts will be cut down to 400 and shorter texts will be post-padded with 0.

Now, all our texts are of same length.

Using the ‘glove.6B.100d.txt’ word embedding, every word is going to be transformed into a dense vector.
glove.6B.100d.txt consists of lines. Each line has a word as index and a dense vector as its value. We loop through the glove file, if a word is present int the vocabulary (word_index) then pick its dense vector and put it at correct index in embedding matrix (correct index is taken from the word_index dictionary).

Now, we have our “x_train” set which consists of 2-D vectors for each input. Every input has 400 1-D vectors of length 100.

8.Model implementation in keras

The proposed hybrid deep learning model is implemented using the Sequential model of the Keras deep learning Python library. The Sequential model comprises several layers of neurons:

1. The first layer of the neural network is the Keras embedding layer. This is the input layer through which the pre-trained word embeddings are utilized by providing the prepared embedding matrix and the model is trained by feeding in the training data.

   Working: model.add(layers.Embedding(vocab_size, embedding_dim, weights=[embedding_matrix], input_length=maxlen, trainable=True))
   input_dim=vocab_size, number of words in the vocabulary.
   output_dim=embedding_dim, size of dense vectors.
   input_length=maxlen, size of each sequence of text
   weights=[embedding_matrix], to use it
   trainable=True, embedding layer will be trained.

2. The next layer is the one-dimensional CNN layer (Conv1D) for extraction of local features by using 128 filters of size 5. The default Rectified Linear Unit (ReLU) activation function is used.

   Working: model.add(layers.Conv1D(128,5,activation= ‘relu’))
   128=number of nodes/neurons in convolution layer
   5=size of filter/kernel matrix; (5,1)
   stride=1, by default
   activation= “relu”, rectified linear unit. It converts the negative values to 0 and don’t change positive values.

3. After that, the large feature vectors generated by CNN are pooled by feeding them in to a MaxPooling1D layer with a window size of 2, in order to down-sample the feature vectors, reduce the amount of parameters, and consequently the computations without affecting the network's efficiency.

   Working: model.add(layers.MaxPooling1D(2))
   2=size of window
   Since, we are dealing with 1D array, no need to adding Flatten() layer. Generally, Flatten() layer converts 2D array into 1D before passing it to Dense layer.

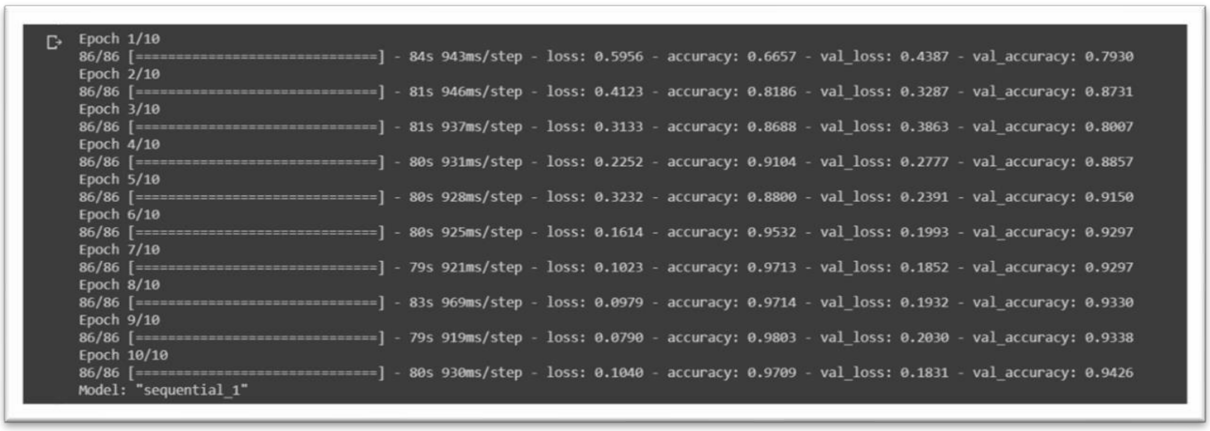
4. The pooled feature maps are fed into the RNN (LSTM) layer that follows. This input is used to train the LSTM, which outputs the long-term dependent features of the input feature maps, while retaining a memory. The dimension of the output is set to 32. The default linear activation function (i.e. f(x)=x) of Keras is used in this layer.

   Working: model.add(layers.LSTM(32))
   32= number of hidden units in LSTM cell or the number of memory units in cell

5. Finally, the trained feature vectors are classified using a Dense layer that shrinks the output space dimension to 1, which corresponds to the classification label (i.e. fake or not fake). This layer applies the Sigmoid activation function.

   Working: model.add(layers.Dense(1, activation= 'sigmoid'))
   This is fully connected layer/dense layer/most common layer in neural networks.
   1= number of node/neurons in this dense layer, each node of dense layer takes input from all the nodes of the previous layer.
   F(W*X+B), W=weight matrix, X=input vector, B=bias and F is the activation function. Here, we have binary classification so we use sigmoid as AF.
   sigmoid= this is the activation function.
   The number of nodes in the last dense layer = number of labels in the label attribute.
   If for binary classification, sigmoid is used. In other case, we use 'softmax' AF.
   >> Activation function decides, whether a neuron should be activated or not by calculating weighted sum and further adding bias with it. The purpose of the activation function is to introduce non-linearity into the output of a neuron.

(from sub-dir: screenshot_1)



```
Epoch 1/10
86/86 [==============================] - 84s 943ms/step - loss: 0.5956 - accuracy: 0.6657 - val_loss: 0.4387 - val_accuracy: 0.7930
Epoch 2/10
86/86 [==============================] - 81s 946ms/step - loss: 0.4123 - accuracy: 0.8186 - val_loss: 0.3287 - val_accuracy: 0.8731
Epoch 3/10
86/86 [==============================] - 81s 937ms/step - loss: 0.3133 - accuracy: 0.8688 - val_loss: 0.3863 - val_accuracy: 0.8007
Epoch 4/10
86/86 [==============================] - 80s 931ms/step - loss: 0.2252 - accuracy: 0.9104 - val_loss: 0.2777 - val_accuracy: 0.8857
Epoch 5/10
86/86 [==============================] - 80s 928ms/step - loss: 0.3232 - accuracy: 0.8800 - val_loss: 0.2391 - val_accuracy: 0.9150
Epoch 6/10
86/86 [==============================] - 80s 925ms/step - loss: 0.1614 - accuracy: 0.9532 - val_loss: 0.1993 - val_accuracy: 0.9297
Epoch 7/10
86/86 [==============================] - 79s 921ms/step - loss: 0.1023 - accuracy: 0.9713 - val_loss: 0.1852 - val_accuracy: 0.9297
Epoch 8/10
86/86 [==============================] - 83s 969ms/step - loss: 0.0979 - accuracy: 0.9714 - val_loss: 0.1932 - val_accuracy: 0.9330
Epoch 9/10
86/86 [==============================] - 79s 919ms/step - loss: 0.0790 - accuracy: 0.9803 - val_loss: 0.2030 - val_accuracy: 0.9338
Epoch 10/10
86/86 [==============================] - 80s 930ms/step - loss: 0.1040 - accuracy: 0.9709 - val_loss: 0.1831 - val_accuracy: 0.9426
Model: "sequential_1"
```

The model is trained using the adaptive moment estimation (Adam) optimizer to define the learning rate in each iteration, the binary cross-entropy as the loss function, and the accuracy for the evaluation of results. The training is performed for 10 epochs using a batch size of 64.

Working: model.compile(optimizer= ‘adam’, loss= ‘binary_crossentropy’)
optimizer= ‘adam’
In the dense layer, weights and bias values are found. This is done by back propagation. Optimizer is used to reduce the loss in much less effort.
loss= ‘binary_crossentropy’
Loss function calculates that how wrong the prediction of the model is when is trains the model. Optimizer tries to reduce this loss by selecting more accurate hyperparameters.

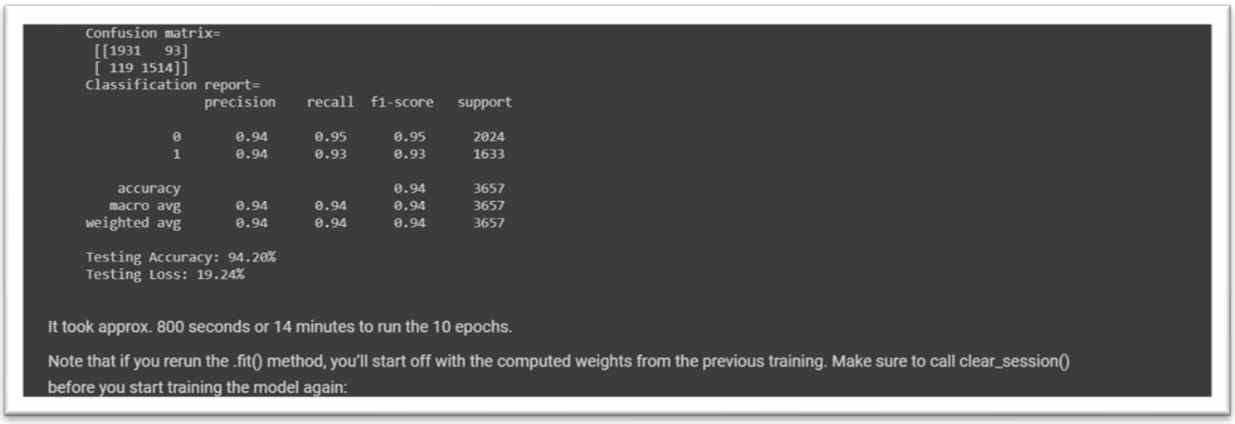
history=model.fit(x_train, y_train, epochs=10, validation_data=(x_valid, y_valid), batch_size= 64)

.fit() is the function to train the model.
Number of epochs=10, means the model will be trained 10 times on the given training data. In each epoch(iteration) the model will learn more and more. The loss will decrease and accuracy will improve. Usually, we can use epochs from 1 to infinity. But, in practice we use 10. Validation_data(); The idea is that you train on your training data and tune your model with the results of metrics (accuracy, loss etc) that you get from your validation set.

Our model doesn't "see" your validation set and isn´t in any way trained on it, but you as the architect and master of the hyperparameters tune the model according to this data.

9.Evaluation of model

“accuracy”, “confusion_matrix()” and “classification_report()” metrics are used for measuring the performance of the model.



```
Confusion matrix=
 [[1931   93]
 [ 119 1514]]
Classification report=
               precision    recall  f1-score   support

           0       0.94      0.95      0.95      2024
           1       0.94      0.93      0.93      1633

    accuracy                           0.94      3657
   macro avg       0.94      0.94      0.94      3657
weighted avg       0.94      0.94      0.94      3657

Testing Accuracy: 94.20%
Testing Loss: 19.24%
```

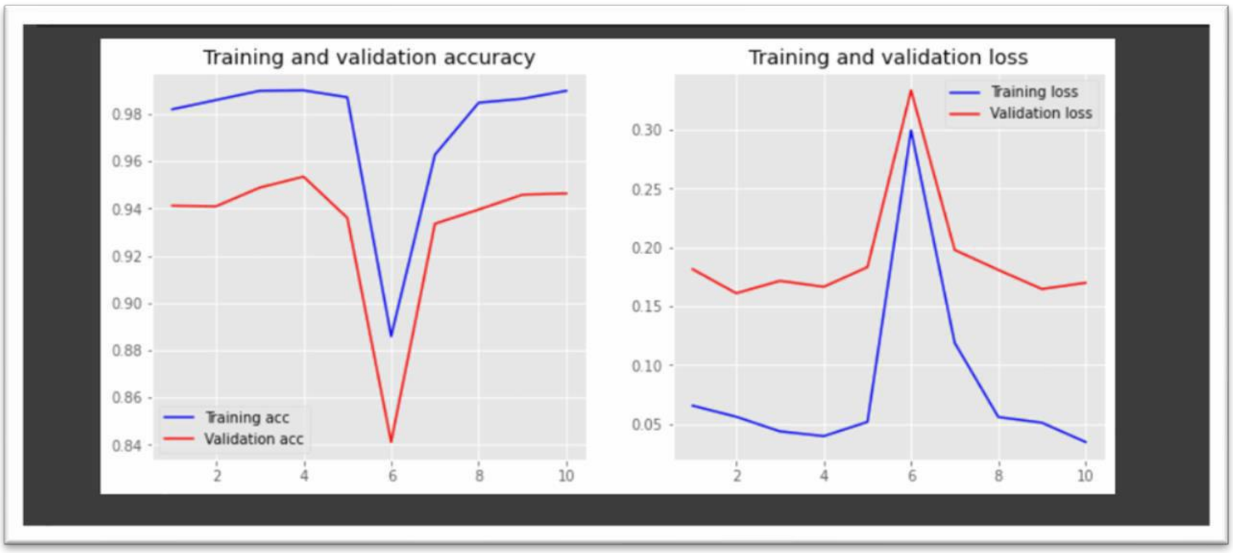
It took approx. 800 seconds or 14 minutes to run the 10 epochs.

Note that if you rerun the .fit() method, you’ll start off with the computed weights from the previous training. Make sure to call clear_session() before you start training the model again:

10. Plotting the graph

a) Training and validation accuracy

b) Training and validation loss



10. Conclusion

Despite the relative abundance of extant works addressing fake news detection, there is still plenty of space for experimentation, and the discovery of new insights on the nature of fake news may lead to more efficient and accurate models.

11.References

www.kaggle.com

Thank you