ANLY 580 Final Project
Sentimental Analysis for IMDB Movie Review: Neural Network Models

Yue Han
Georgetown University
2111 Jefferson Davis Highway
Arlington, Virginia 22202
yh584@georgetown.edu

Abstract

In this project, we explore the sentimental classification function performed by two Neural Network models on an IMDB movie review sentiment dataset. We develop and optimize one simple Neural Network model with only base unit, and one Recurrent Neural Network model with Long Short-Term Memory (LSTM) unit. The performance of those two models are evaluated with *loss* and *accuracy*.

1 Introduction

The purpose of sentiment analysis is to classify text content based on their overall positivity or negativity. IMDB, as the world's most popular and authoritative source for movie, whose review is a rich source of text examples expressing sentiment. In this project, we investigated the dual sentiment classification of IMDB movie review data where the labels are "positive" and "negative".

We initialize two Neural Network models with layers and explore methods to optimize their performance. The simple model is designed for improvement with dealing with overfitting by inserting Dropout layer, whereas the LSTM model is constructed and optimized with the bidirectional RNN. In addition, the pretrained embedding is applied afterwards to replace the randomly initialized word embedding for both of the models. The effect of each application is evaluated with *loss* and *accuracy*.

2 Data Collection and Preprocessing

The IMDB dataset is preprocessed into 25,000 reviews for training and 25,000 reviews for testing, and can be easily obtained using Keras. Both sets are balanced, which means there are equal number of items in both classes, positive and negative in this case. The review texts have been translated into sequences of integers and each integer represents a specific word in a dictionary. The dictionary is also pre-built.

Since the inputs must be of the same length for later use in embedding, we will use the helper function pad_sequences in Keras to unify the lengths. Cut the long review from the beginning and fill the short review from the beginning with <PAD>, both into the same length of 100.

3 Simple Model

A. Build the Model

To build the simple Neural Network model, we added four layers and the layers are linked sequentially, i.e., the output of the previous layer is sent to the next layer only.

The first layer is an Embedding layer. A word embedding represents words and documents using a dense vector representation. Compare with the traditional bag-of-word model encoding schemes where large sparse vectors were used to represent each word or score each word within a vector to represent an entire vocabulary, the word embedding makes a great improvement. In an embedding, words are represented by dense vectors where a vector represents the projection of the word into a continuous vector space. The embedding layer takes a sequence of word IDs (integer) and looks up an embedding matrix for a vector that represents that ID. This layer converts the 2D input of shape (batch, sequence_len) to (batch, sequence_len, embedding_size). Hence vectors are learned as the model trains.

The next layer, GlobalAveragePooling1D layer, calculates an average of the second dimension. So a batch of sequences of embeddings with shape (batch, sequence_len, embedding_size) will be averaged to a shape (batch, embedding_size).

The last two layers are fully-connected (Dense) layer with 16 and 1 hidden unit(s). First fully-connected layer can be thought of as a feature reduction. The final layer is applying the sigmoid activation function, which has an output value between 0 and 1, to act as a probability or confidence level.

Train the model and evaluate it on the testing set.

B. Deal with Overfitting

In general, when training the data, we will take consideration of the overfitting problem. Overfitting occurs when the model performs much better on the training data than it does on new data. After a point, the model over-optimizes and learns representations specific to the training data that do not generalize to test data. To deal with this problem, we apply a simple method that uses Dropout. Insert Dropout layer in between our previous layers. Dropout(rate) where rate indicates the fraction of the input units to drop.

Retrain the model and evaluate it on the testing set.

C. Pretrained Embedding

Instead of randomly initialize the word embeddings, we can train them using a huge amount of (unlabeled) data in an unsupervised fashion. The Keras Embedding layer can also use a word embedding learned elsewhere. For example, the researchers behind GloVe method provide a suite of pre-trained word embeddings. The smallest package of embeddings is 822Mb, called "*glove.6B.zip*". It was trained on a dataset of one billion tokens (words) with a vocabulary of 400 thousand words. After downloading and unzipping, we see a few files, one of which is "*glove.6B.100d.txt*", which contains a 100-dimensional version of the embedding. Inside the file, there is a token (word) followed by the weights (100 numbers) on each line, in the following format:

word1 0.xx -0.xx ...
word2 0.yy 0.yy ...

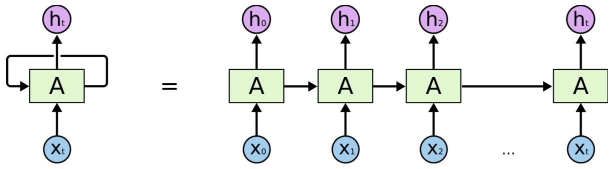
Therefore, we use the GloVe embeddings to initialize our embedding weights. Here, we read the entire file and see which of those words are in our dictionary. If we find one, we will set its embedding accordingly. Other words that are not found in the pretrained embeddings can be initialized as all zeros or very small numbers.

Test the effect of pretrained embeddings on the simple model.

4 LSTM Model

A. Build the Model

Now we start to build another model—the Recurrent Neural Network model with Long Short-Term Memory (LSTM). A LSTM network is a kind of recurrent neural network. A recurrent neural network is performed by feeding back the output of a neural network layer at time *t* to the input of the same network layer at time *t + 1*. It looks like this:



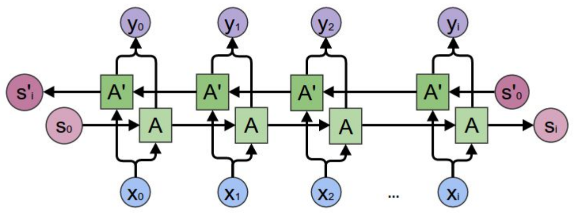
In the diagram above, we have a simple recurrent neural network with t input nodes. These input nodes are fed into a hidden layer, with sigmoid activations, as per any normal densely connected neural network. Then the output of the hidden layer is then *fed back* into the same hidden layer. The hidden layer outputs are passed through a conceptual delay block to allow the input of ht−1ht−1 into the hidden layer. The point is that we can now model time or sequence-dependent data.

To build the model, based on the previous one, I replaced the GlobalAveragePooling1D layer and the first Dense layer with two LSTM layers. By default, a recurrent cell only returns the last output. But for multiple layers of recurrent cells, we need the entire output. So, we have to set a layer with return_sequences=True for all recurrent layers except the last one, and another layer with return_sequences=False for the last output.

Train the model and evaluate on test data.

B. Combine with Bidirectional

A major issue with the above network is that it learns representations from previous time steps. Sometimes, we might have to learn representations from future time steps to better understand the context and eliminate ambiguity. To resolve the ambiguity, we need to look ahead. This is what Bidirectional RNNs accomplish.



The structure and the connections of a bidirectional RNN are represented in figure above. There are two type of connections, one going forward in time, which helps us learn from previous representations and another going backwards in time, which helps us learn from future representations.

Apply this method and retrain the model, evaluate with the test data.

C. Pre-trained Embedding

Same as for the simple model, we apply the pretrained embedding to the LSTM model. Train the model again and evaluate on the test set.

5 Result and Evaluation

We would like to show the effectiveness of each optimization application on the model with *loss* and *accuracy*.

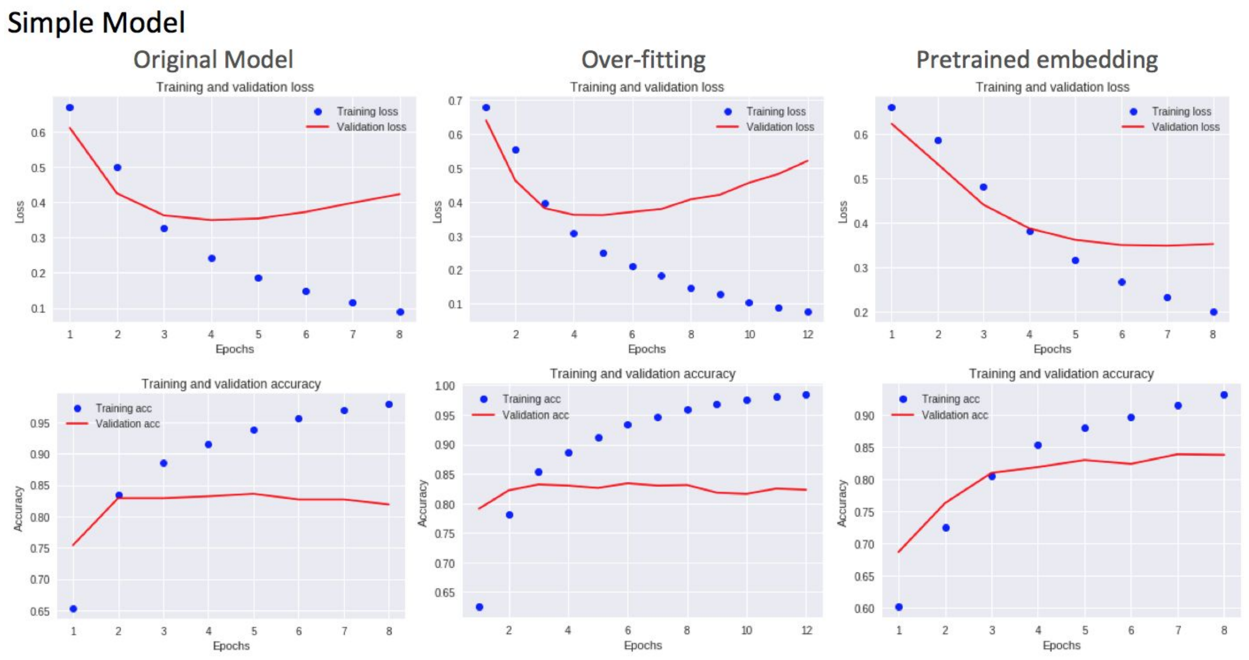
The *loss* is calculated on training and testing and its interpretation is how well the model is doing for these two sets. It is a summation of the errors made for each example in training or testing sets. In the case of neural networks, the *loss* is usually *negative log-likelihood* and *residual sum of squares* for classification and regression respectively. Then naturally, the main objective in a learning model is to reduce (minimize) the loss function's value with respect to the model's parameters by changing the weight vector values through different optimization methods. Loss value implies how well or poorly a certain model behaves after each iteration of optimization. Ideally, one would expect the reduction of *loss* after each, or several, iteration(s).

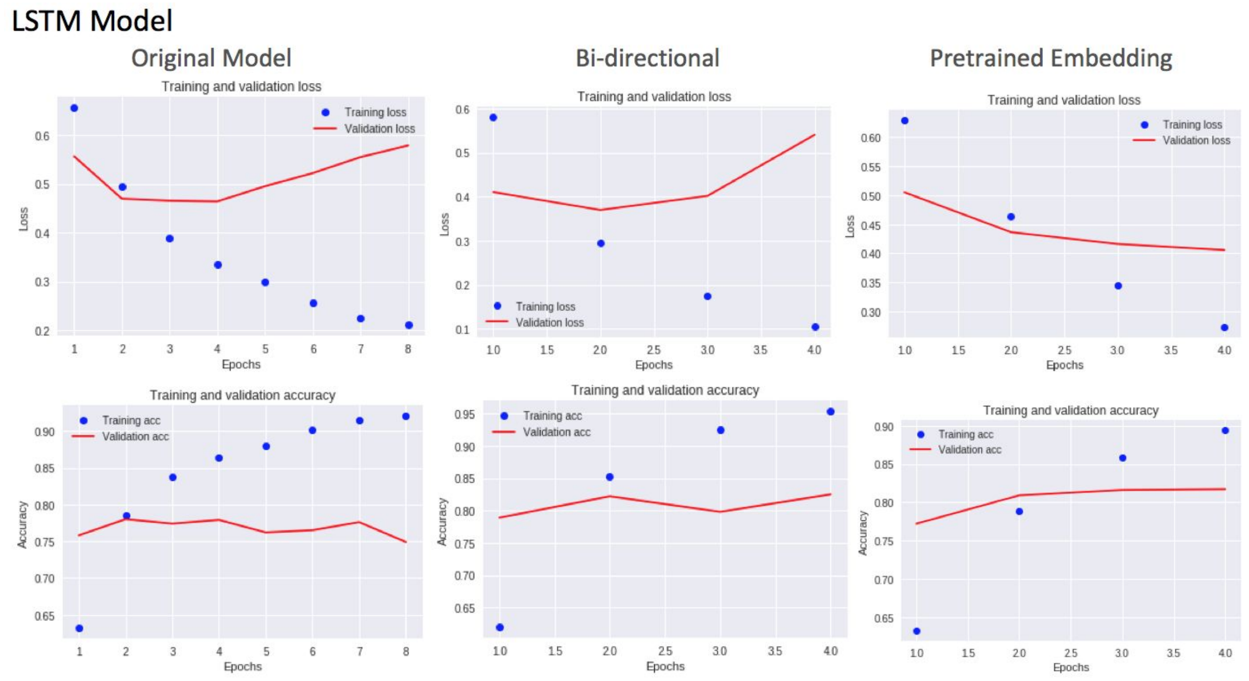
The *accuracy* of a model is usually determined after the model parameters are learned and fixed and no learning is taking place. Then the test samples are fed to the model and the number of mistakes the model makes are recorded, after comparison to the true targets. Then the percentage of misclassification is calculated.

As model.fit returns a history object that contains a dictionary with everything that happened during training, for each model with each application, we plotted the *loss* and *accuracy* of each epoch. In each plot, the dots represent the training loss and accuracy, and the solid lines are the test *loss* and *accuracy*. Notice the training *loss* decreases with each epoch and the training *accuracy* increases with each epoch. This is expected because we designed the model to optimize for this goal. However, the *loss* and *accuracy* for test data is usually different from the training data. They usually peak after some epochs.

By putting the plot of *loss* and *accuracy* of each application of each model together, we can make a compare and see the improvement of the model after each application that, in general, the red line is moving closer to the blue dots. In other words, in most case, the loss and accuracy are moving closer to the expected value.

For the simple model, when comparing the loss, we observe that the final loss at epoch 8, decreases from 0.42 to 0.40 and then to 0.36 for the stage from the original model to dealing with the over-fitting and then adding the pretrained





embedding. We can also observe an obvious adjustment that the red line of loss is moving closer to the blue dot of expected value. In terms of the accuracy, we can see an improvement for each stage from 83% to 83.5% to 84%. However, this improvement is not that significant.

For the LSTM model, when comparing the loss at the final epoch 4, out of our expectation, it increases from 0.47 to 0.53 after applying the Bi-directional RNN function. Then after applying the pretrained embedding, it reduced to 0.40. In terms of accuracy, it starts very low at 78%. With the Bi-directional RNN, it increases significantly to 83%, while stay the same with the pre-trained embedding.

6 Discussion

From the result above, we have the following findings of our two models:

a. Although the dropout of dealing with over-fitting problem is not that significant on the simple model, the red lines of loss and accuracy history on test do have a trend matching better to the expected blue dots.

b. The bi-directional LSTM model outperforms the unidirectional LSTM model in the aspect of accuracy, whereas underperforms the unidirectional LSTM model in the aspect of loss.

c. For both models, the pre-trained embedding function have a significant effect on loss reduction, but a small effect on accuracy improvement. However, on the loss and accuracy history plot, the red line of both loss and accuracy on test moving evidently to the expected blue dots.

d. There is no guarantee that the deeper or more complex models perform better than simple models. In our two models, the LSTM model perform a deeper and more complex algorithm than the simple model. However, at both the original or the final stage, the LSTM model either not yield a lower loss nor a higher accuracy score than that of the simple model.

7 Conclusion and Further Work

We find that the LSTM model underperforms the simple model, but achieves superior improvement over the simple model after several optimizations. Although discrepancy exists, the two Neural Network models are both applicable models to classify sentimentals for the IMDB movie reviews. A clear path for further work is to investigate deeper to each optimization application to the models that why each methods can cause the loss and accuracy increase or decrease, and move in the way the result showed.

References

Amir Arsalan. (2015, December 29). *How to interpret "loss" and "accuracy" for a machine learning model.* Retrieved from https://stackoverflow.com/questions/34518656/how-to-interpret-loss-and-accuracy-for-a-machine-learning-model

Andy. (2018, February 3). *Keras LSTM tutorial – How to easily build powerful deep learning language model.* Retrieved from http://adventuresinmachinelearning.com/keras-lstm-tutorial/

Jason Brownlee. (2017, October 4). *How to Use Word Embedding Layers for Deep Learning with Keras.* Retrieved from https://machinelearningmastery.com/use-word-embedding-layers-deep-learning-keras/

Rohith Gandhi. (2018, June 26). *Introduction to Sequence Models — RNN, Bidirectional RNN, LSTM, GRU.* Retrieved from https://towardsdatascience.com/introduction-to-sequence-models-rnn-bidirectional-rnn-lstm-gru-73927ec9df15