Capstone Project
Machine Learning Engineer Nanodegree

Yan Kang

1. Definition

1.1. Project Overview

With the rise of online social media platforms such as Twitter, Facebook and Weibo, and the proliferation of customer reviews on e-commercial sites like Amazon and Yelp, it is increasing important that companies are able to accurately analyze customers' sentiment around reviews for products and feedback for advertising campaigns. This sort of analysis is called sentiment analysis and it becomes probably one of the most popular applications of Natural Language Processing these days.

However, most models in sentiment analysis use bag of words (BoW) representations [1] that ignore word orders and thus they cannot accurately address hard sentiment analysis issues, such as negation and sense ambiguity. Recurrent Neural Network (RNN)[2] is able to address these hard issues to some degree as they treat text as sequences. Recursive Neural Network [3] takes into account the syntactic structure of text and thus they also are capable of partially resolving these issues. In this project, we develop a RNN model and a Recursive Neural Network model for solving a sentiment analysis problem, and use BoW-based Naïve Bayes model as baseline model to evaluate the performance of the two models.

1.2. Problem Statement

In this project, we will use three different models to perform the sentiment analysis task on movie reviews. More specifically, we use Naïve Bayes model, Recurrent Neural Network (RNN) and Recursive Neural Network.

The Naïve Bayes model serves as the benchmark model, based on which the performance of RNN and Recursive Neural Network are analyzed. Since the Naïve Bayes does not take the positions or orders of words into account, it has difficulty in addressing issues such as negation and sense ambiguity. RNN has been proven to be very successful at addressing such issues because it treats text as a list sequences and incorporate information over time. However, RNN does not fully exploit the syntactic structure of text, while Recursive Neural Network computes compositional vector representations for phrases of variable length and syntactic type, and empirically, it is proven to perform well on sentiment analysis [3].

We will evaluate and compare the performance of the three models based on the accuracy of classifying customer movie reviews.

1.3. Evaluation Metrics

In this project, we mainly use accuracy to evaluate the overall performance of models. Accuracy takes into account both true positives and true negatives with equal weight:

$$accuracy = \frac{true\ positives + true\ negatives}{dataset\ size}$$

Because we have no imbalanced class issue (we have 3910 negative samples and 4090 are positive samples, using accuracy is sufficient.

2. Analysis

2.1 Data Exploration

We use dataset from Stanford Sentiment Treebank [4]. This dataset is designed for Recursive Neural Networks:

- Each sample in this dataset is a serialized version of a parse tree that is structured to identify dependencies between words. Following is an example of training data point:

  (4 (3 (2 A) (4 (4 (2 (2 deep) (2 and)) (3 meaningful)) (2 film))) (2 .))

- Each node in a parse tree is labeled using Amazon Mechanic Turk [5], and have rating: 0-4, in which 2 is neural, 0 is the most negative and 4 is the most positive. For example, the parse tree of aforementioned training data point is depicted as follow:

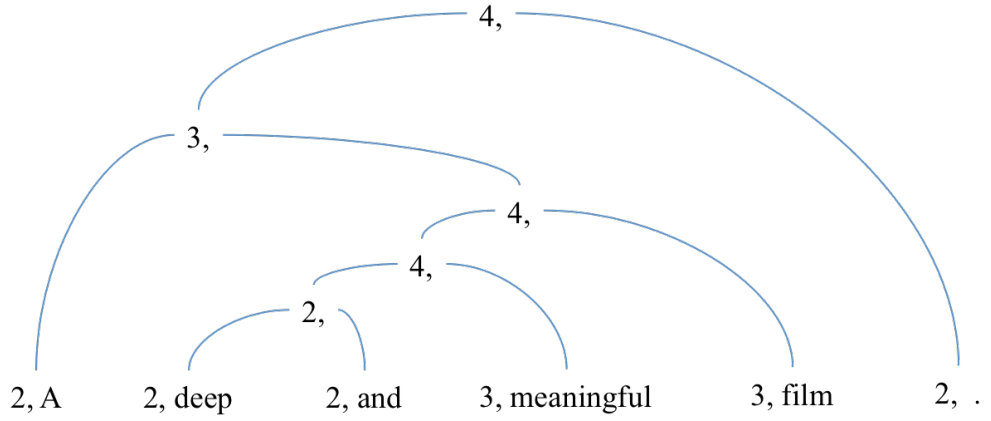

Figure 1: parse tree of sentence “A deep and meaningful film”

In this project, we only perform positive/negative classification. Therefore, we label all samples with rating 0-1 as negative, all samples with rating 3-4 as positive, and ignore samples with rating 2. After transforming, we have 8000 training samples and 739 test samples. 3910 of the 8000 training samples are negative samples while the rest 4090 are positive samples. Therefore, There is no imbalanced class issue.

We transformed all samples in the form of parse trees into bag-of-words vectors and sequences such that we can feed them into Naïve Bayes model and RNN.

There is one important decision to make when we are preprocessing training/test data for Recurrent Neural Network. That is we need to define the maximum sequence length to make all training/test sequences have the same length. I can either pad the training/test sequences that are shorter than the maximum length and truncate the training/test sequences that are longer than the maximum length.

However, we want the maximum sequence length to be just right. If maximum sequence length is too small, we would loss some information. On the other side, if maximum sequence length is too large, we may add too much useless information to the training/test data that may jeopardize the performance of the trained network. To better estimate the maximum sequence length, we drew sequence length distribution over both training and test data sets:

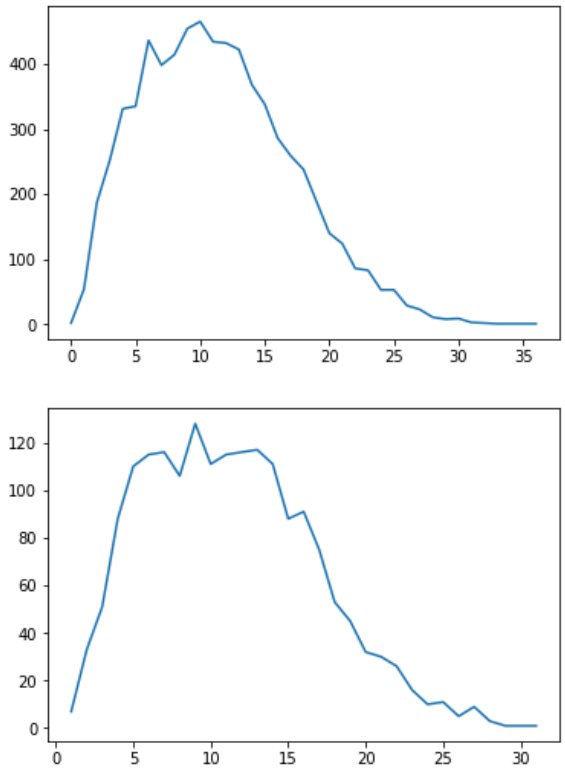

Figure 2: Sequence lengths distribution over the training and test sets. The picture on the top shows the sequence length distribution over training data, while picture on the bottom shows the sequence length distribution over test data.

Based these distributions, we can see that most of the training/test data have length shorter than 30. Therefore, we set the maximum sequence length to 30 .

2.2 Algorithm and Techniques

2.2.1. Recurrent Neural Network

I adopted Recurrent Neural Networks (RNN) to address the sentiment analysis problem. With the purpose of exploring various RNN architectures and finding the best possible one in terms of classification accuracy, we performed experiments on two variants of RNN cell, which are Long Short Term Memory (LSTM) and Gated Recurrent Unit (GRU), and two types of RNN layers, which are unidirectional and bidirectional layers.

The quality of training data and the choice of hyperparameters significantly and subtly affect the performance of the RNN model. Therefore, we feed the RNN model with various versions of training data and tried various combinations of hyperparameters.

More specifically, I tried three versions of stopwords when preprocessing the data. The rationale behind this process is that many words that we normally treat as stopwords may play an important part in sentiment analysis. For example, negation words such as “no”, “not”, “nor”, “hasn’t”, “n’t” and “shan’t” can help address negation issue that cannot be solved by Naïve Bayes classifier. Comparison words such “over”, “than” and “beyond” may also subtly help improve the performance of sentiment analysis classifier.

The RNN architecture involves many hyperparameters. We tuned following hypermeters to find the best combination in terms of classification accuracy:

- batch_size: The number of training examples to feed the network in one training pass. Typically this should be set as high as you can go without running out of memory.
- num_steps: (or time steps) Number of words in a sequence of. Larger is better typically; the network will learn more long range dependencies. But it takes longer to train.
- lstm_units: Number of units in the hidden layers in the LSTM cells. Usually larger is better performance wise. Common values are 128, 256, 512, etc.
- learning_rate: Learning rate
- keep_prob: The dropout keep probability when training. If you're network is overfitting, try decreasing this.

I used two state-of-the-art deep-learning frameworks, Keras [6] and Tensorflow [7], to implement RNN architectures.

Keras is a high-level deep-learning framework that has taken care of most of the implementation work. The only things we need to do by using Keras are to design and build up the architecture. The reason I adopt Keras in this project is because I want to build up a running model in a quick and efficient way to test my designed architectures.

Tensorflow requires more detailed implementation from us but allows us to fine-tune our models. The reason I adopt Tensorflow in this project is two-fold: I want to learn more about Tensorflow and master the theories behind RNN while working on this project.

2.2.2. Recursive Neural Network

I used Theano framework [8] to implement Recursive Neural Network. Theano is a python library that allows us to define, optimize, and evaluate mathematical expression involving multi-dimensional arrays efficiently. It requires us to implement almost all the parts of a Recursive Neural Network models including, weights defining, embedding layer, recurrent unit, time steps, and optimization approaches. The only burden that Theano takes off from us is computing gradients. The major reason I adopt Theano to implement Recursive Neural Network is to learn the Recursive Neural Network theories thoroughly. The experimental result of my Recursive Neural Network implementation is not as good as RNN implemented by using Keras and Tensorflow. There are many factors that can contribute to the less-than-satisfactory result. I will leave this for further research in the future.

2.3 Benchmark model

We used built-in Naïve Bayes model with default parameters from scikit-learn as the benchmark model. The Naïve Bayes model achieved decent performance (around 81% accuracy) on the training and test data. However it has difficulty in addressing issues such as negation and sense ambiguity. Using Naïve Bayes model as benchmark is an appropriate choice since it can really verify whether RNN and Recursive Neural Network have the edge on addressing those difficult issues.

3. Methodology

3.1 Data Preprocessing:

3.1.1. Training/testing data for Recursive Neural Network

I explored three models in this project. They are Naïve Bayes, Recurrent NN and Recursive NN respectively. Therefore, the original data needs to be transformed into the format that is able to feed into each of three models.

The Recursive NN requires data in the form of sequences that encodes the tree structure of the original data. In other words, I need to transform original data in the form of serialized tree structure into sequences encoding that tree structure information. With careful examination on the original data set, we can see that each example is arranged in a list such that children always come before parent. For example, following shows one example from the training set:

(4 (3 (2 A) (4 (4 (2 (2 deep) (2 and)) (3 meaningful)) (2 film))) (2 .))

Its pictorial presentation is shown in Figure 2. Each leaf represents a word/token in the example and each intermediate node is the combination of its left and right subtrees. The number in each node represents the sentimental score. In this project, we only consider the sentimental score for the whole review. That is, we only use the sentimental score of the root node (in this example, it is 4). Also, we only take into account positive and negative sentiment (we will convert the sentimental score into either 0 or 1).

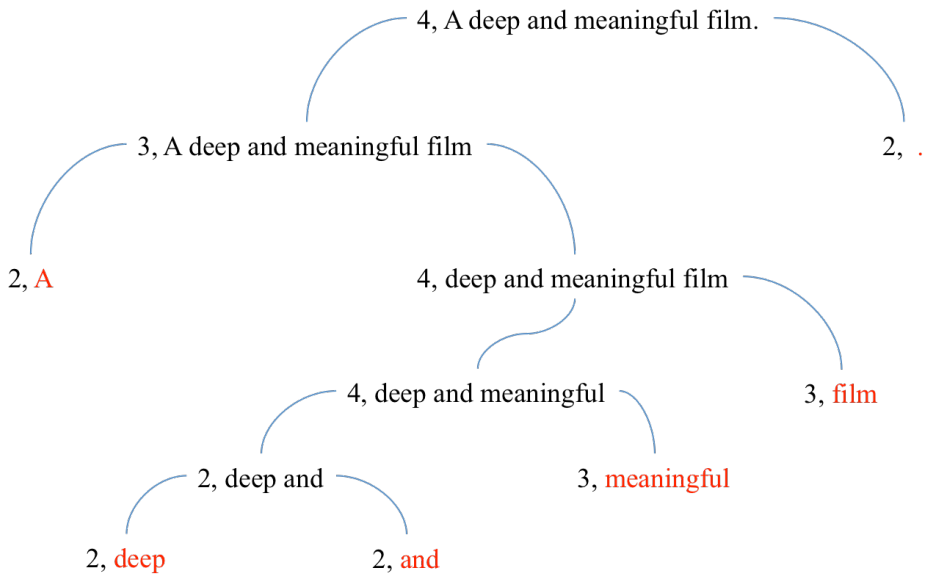

Figure 3: parse tree of training example
"(4 (3 (2 A) (4 (4 (2 (2 deep) (2 and)) (3 meaningful)) (2 film))) (2 .))"

I used post-order traversal algorithm to parse the original data and generate sequences for Recursive NN. While traversing the parse tree in post order, I create three lists: word index array, parent index array and relation indicator array.

- word index array: each value in the array represents an index of a word. If a node is not a word, the value would be -1
- parent index array: each value in the array represents index of parent. If a node has no parent, the value would be -1
- relation indicator array: each value in the array indicates the relation to parent. If a node is the left node of its parent, the value is 0; if a node is the right node its parent, the value is 1; if a node has no parent, the value would be -1.

Take the training data point in Figure 2 as an example:

Given the word-to-index map: {'.': 5, 'film': 4, 'a': 0, 'and': 2, 'meaningful': 3, 'deep': 1}

The three arrays are:

Word index array: [0, 1, 2, -1, 3, -1, 4, -1, -1, 5, -1]

Parent index array: [8, 3, 3, 5, 5, 7, 7, 8, 10, 10, -1]
Relation indicator array: [0, 0, 1, 0, 1, 0, 1, 1, 0, 1, -1]

If we convert the word index array to word array, the word array would be:
['a', 'deep', 'and', Null, 'meaningful, Null, 'film', Null, Null, '.', Null]

For word 'a', it has value 0 (the first element) in the relation indicatory array. This means that the word 'a' is the left child of its parent. Similarly, word 'meaningful' is the right child of its parent .

After these data processing steps, each example in original data set is transformed into three integer sequences that together encode the three structure of that example. In the Implementation section, I will explain how the three sequences are feed into Recursive NN.

3.1.2. Training/testing data for Recurrent NN and Naïve Bayes

The word index array is simply the integer representation of movie review (e.g., [0, 1, 2, -1, 3, -1, 4, -1, -1, 5, -1] represents "a deep and meaningful film."). Therefore, I reconstructed movie review comments from these word index arrays and then preprocessed these review comments to generate appropriate data for both Recurrent NN and Naïve Bayes model.

More specifically, to generate data for Recurrent NN, I performed following preprocessing steps for each example:

1. Remove punctuation
2. Tokenization
3. Remove stopwords
4. Encode examples into integer sequences
5. Padding example to the same length

It is worth mentioning that we tried three versions of stopwords when preprocessing the data. This is because that many words that we normally treat as stopwords may play an important part in sentiment analysis. For example, negation words such as "no", "not", "nor", "hasn't", "n't" and "shan't" can help address negation issue that cannot be solved by Naïve Bayes classifier. Comparison words such "over", "than" and "beyond" may also subtly help improve the performance of sentiment analysis classifier.

When it comes to padding, we considered both left padding and right padding. It turns out that left padding works better for Tensorflow. we chose the maximum sequence length based on the sequence length distribution over both training and test data sets:

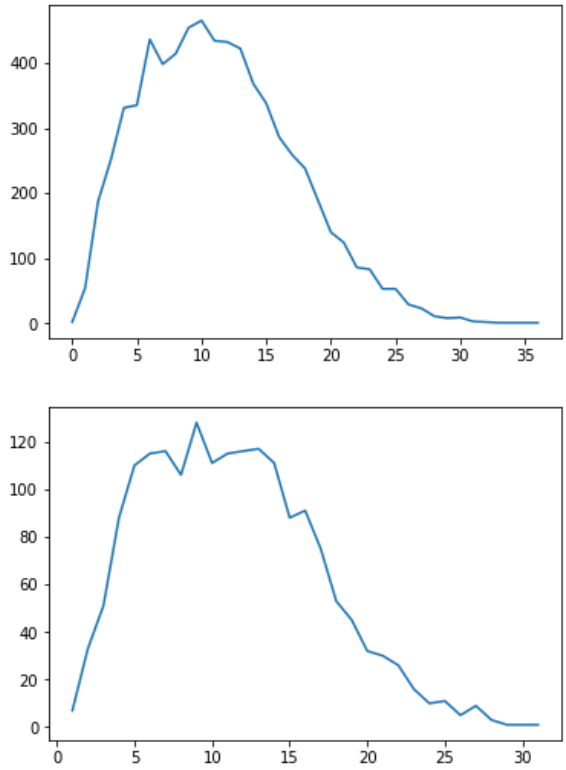

Figure 4: Sequence lengths distribution over the training and test sets. The picture on the top shows the sequence length distribution over training data, while picture on the bottom shows the sequence length distribution over test data.

Based these distributions, we set the maximum sequence length to 31.

Naïve Bayes model requires data in the form of bag-of-words (BoW). I exploited sklearn CounterVector and TfdfVector to perform preprocessing tasks.

3.2. Implementation

In this section, we will explain my implementation for Recurrent NN and Recursive NN.

3.2.1. Recurrent Neural Network Implementation

We adopt a standard way of implementing Recurrent NN. That is, the architecture of the Recurrent NN mainly contains three layers:

1. Embedding layer
2. LSTM layer
3. Fully connected hidden layer

To avoid overfitting, we also added a dropout layer to regularize the Recurrent NN. The architecture is shown in Figure 5 (the dropout layer does not shown in the figure):

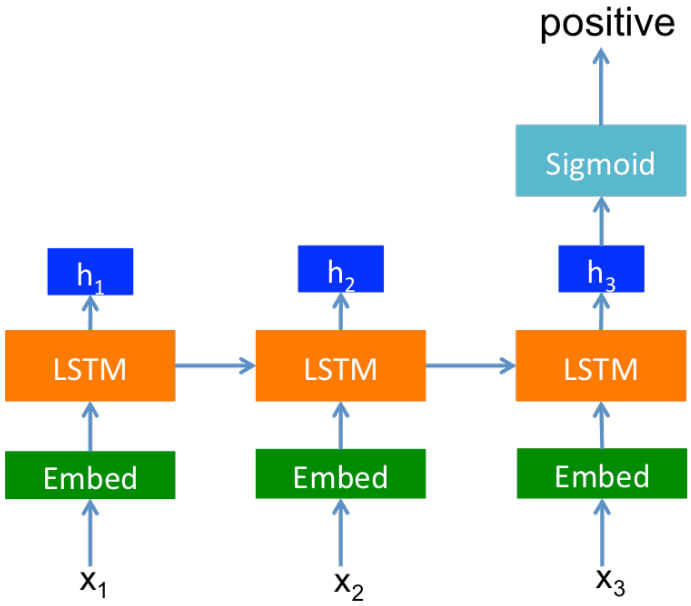

Figure 5: Recurrent Neural Network Architecture

We also tried bidirectional LSTM. The architecture is shown in Figure 6.

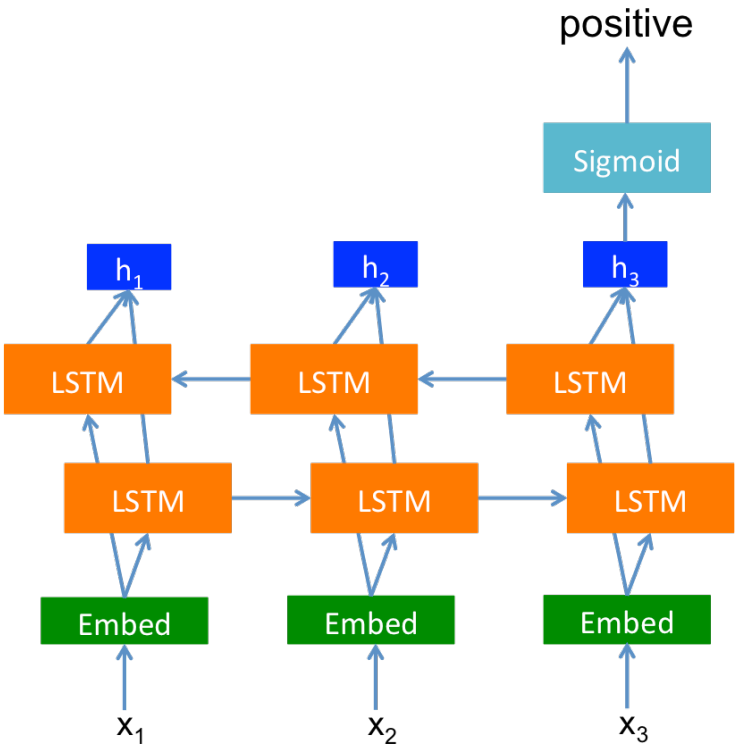

Figure 6: Bidirectional Recurrent Neural Network Architecture

Because we only use the last output of the bidirectional LSTM, the capability of the bidirectional LSTM is not exploited into its full potential. The experimental result also shows that bidirectional LSTM does not perform better than unidirectional LSTM.

We implemented this Recurrent Neural Network architecture by using both Keras and Tensorflow.

3.2.2. Recursive Neural Network Implementation

To my knowledge, there is no built-in Recursive Neural Network implemented in neither Keras nor Tensorflow. Therefore, I implemented Recursive Neural Network from scratch. In this section, I will explain the implementation detail for Recursive Neural Network.

In Recursive Neural Network, a node's value is defined by its children, and in turn children's values are defined by their children respectively, and so on so forth. This relationship is depicted in Figure 7.

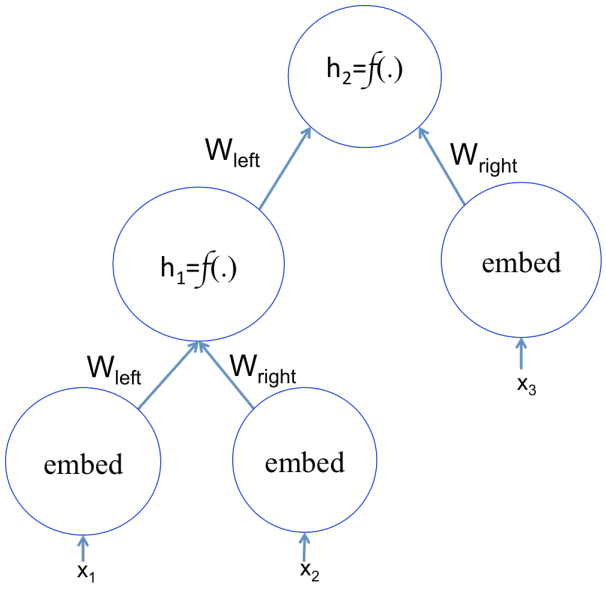

Figure 7:

We need weights (i.e., weight matrix) on each edge between a parent node and one of its child to tell how this child connects to the parent. Since we use binary tree, we only need two weights for each node (i.e., inner node): one for left child and one for right child. Note that although we may have multiple nodes that each has its own children, we use the same left weight matrix for all nodes to connect their left child, and the same right matrix for all nodes to connect their right child.

A neural network node (aka neuron) typically computes $f(Wx + b)$. Nodes in Recursive Neural Network conduct similar computation. Specifically, for leaf node the input of which is word index, we lookup the embedding vector for that word index via embedding lookup table *We*. For inner node, we conduct following computation (take the binary tree in Figure 6 as example):

$$h_1 = f(W_{left}W_e[x_1] + W_{right}W_e[x_2] + b)$$

$$h_2 = f(W_{left}h_1 + W_{right}W_e[x_3] + b)$$

It is not hard to notice that the computation is conducted through post-order traverse. However, instead of using recursive algorithm to perform this post-order traversal computation, we are using recurrent way to solve this problem. There are many advantages of using recurrent way to solve this recursive problem, two of which are firstly recurrent method is more computational efficiency since we do not need to create a binary tree for each training/test example; secondly the code is more straightforward to implement.

The recurrent algorithm for solving the recursive computation is described as follow:

Step 1: Define word embedding matrix *We*, weight matrix *Wh* and hidden state matrix *H*

- *We* is the embedding lookup table, with the shape of (word_number, hidden_state_dimension)
- *Wh* has the shape of (relation_index, hidden_state_dimension, hidden_state_dimension). The first dimension is indexes of relation. For binary three, 0 represents left child while 1 represents right child.
- *H* has the shape of (input_sequence_length, hidden_state_dimension). Each row of *H* is the informational representation for a node in the recursive network (aka binary tree).

Step 2: we recurrently traverse the input sequences and at each time step *t*, we do the following computation:

```
if words[t] >= 0:     // if words[t] is a word
    H[t] = We[words[t]]
else:
    H[t] = f(H[t] + b))

p = parents[t]
r = relation[t]
if p > 0:     // if has parent
    H[p] = H[p] + H[t].dot(Wh[r]))
```

To illustrate the algorithm described above, I take the binary tree depicted in Figure 6 as a training input to perform the computation. As described in section 3.1.1, there actually are three sequences/arrays for each training input: word index sequence, parent index sequence and relation indicator sequence. Table 1 shows values of the three sequences and others derived from the binary tree given the word-to-index map: {x_1:0, x_2:1, x_3:2}.

Table 1:

Time step/seq index	0	1	2	3	4
serialized tree node seq	x1	x2	h1	x3	h2
Word index seq	0	1	-1	2	-1
Parent index seq	2	2	4	4	-1
Relation indicator seq	0	1	0	1	-1

At time step 0, the input is word x_1. Its parent has index of 2 and it is the left child.

It is a word and we lookup the word embedding for x_1 based on:

$$H[0] = We[words[0]]$$

and update hidden state of x_1's parent based on:

$$H[2] = H[2] + H[0].dot(Wh[0]))$$

At time step 1, the input is word x_2. Its parent has index of 2 and it is the right child.

It is a word and we lookup the word embedding for x_2 based on:

$$H[1] = We[words[1]]$$

and update hidden state of x_2's parent based on:

$$H[2] = H[2] + H[1].dot(Wh[1]))$$

At time step 2, the input is h_1. Its parent has index of 4 and it is the left child.

It is not a word, we update its hidden state based on:

$$H[2] = f(H[2] + b))$$

We also update hidden state of h_1's parent based on:

$$H[4] = H[4] + H[2].dot(Wh[0]))$$

At time step 3, the input is word x_3. Its parent has index of 4 and it is the right child.

It is a word and we lookup the word embedding for x_3 based on:

$$H[3] = We[words[3]]$$

and update hidden state of x_3's parent based on:

$$H[4] = H[4] + H[3].dot(Wh[1]))$$

At time step 4, the input is h_2. It has no parent.

It is not a word, we update its hidden state based on:

$$H[4] = f(H[4] + b))$$

3.4. Refinement

For Recurrent Neural Network, to find the best combination of hyperparameters, I tried different combinations of hyperparameters using grid search. Specifically, I considered hyperparameters: embedding dimension, RNN hidden units, dropout probability and learning rate. I adopted mini-batch gradient descent approach and Adam optimizer across the whole training process for Recurrent Neural Network.

For Recursive Neural Network, I adopted stochastic gradient descent simply because it is easy to implement. Because the instability of stochastic gradient descent, I tried different optimization approach such as momentum, AdaGrad and RMSprop. Also, I tried different hidden units and learning rate. I did not use dropout to regularize the network because it is relatively hard to implement from scratch for Recursive Neural Network. Alternatively, I used L2 to regularize the network.

4. Results

4.1. Model Evaluation and Validation

4.1.1. Experiment result for Keras implementation of RNN

IWe used to grid search to search the best combination of hyperparameters. Specifically, We considered following hyperparameters and their corresponding values:

- Embedding dim: [256, 300]
- RNN hidden units: [128, 256]
- Dropout probability: [0.7, 0.8]

We run the grid search on both LSTM and bidirectional LSTM (BiLSTM)., and each for 5 times. Table 2 and Table 3 show the results of the experiments.

Table 2: Experimental result for LSTM

LSTM			
Embedding dim	RNN units	Dropout probability	Accuracy
256	128	0.7	**0.825**
256	128	0.8	**0.826**
256	256	0.7	0.812
256	256	0.8	0.804
300	128	0.7	0.814
300	128	0.8	0.793
300	256	0.7	0.805
300	256	0.8	0.812

Table 3: Experimental result for Bidirectional LSTM

Bidirectional LSTM

Embedding dim	RNN units	Dropout probability	Accuracy
256	128	0.7	**0.821**
256	128	0.8	**0.820**
256	256	0.7	0.814
256	256	0.8	0.818
300	128	0.7	0.811
300	128	0.8	0.801
300	256	0.7	0.796
300	256	0.8	0.791

The result shows that the network has the best accuracy when the embedding dimension is 256 and hidden units is 128 for both LSTM and BiLSTM. This demonstrates that the network prefers simpler model to more complex one. The best accuracies (0.826 for LSTM and 0.821 for BiLSTM respectively) are better than the benchmark accuracy that is 0.812. This demonstrates that RNN can work better than Naïve Bayes for this particular sentimental analysis problem.

4.1.2. Experiment result for Tensorflow implementation of RNN

Similar to Keras implementation, we used to grid search to search the best combination of hyperparameters. The difference is that we added learning rate to the grid search and excluded embedding dimension, which was set to 256 across the whole experiment in this section, since we have already known that embedding dimension of 256 performed best. Specifically, I considered following hyperparameters and their corresponding values:

- Learning rate: [0.001, 0.005]
- RNN hidden units: [128, 256]
- Keep probability: [0.7, 0.8]

I run the grid search on both LSTM and bidirectional LSTM (BiLSTM)., and each for 5 times. Table 4 and Table 5 show the results of the experiments.

Table 4: Experimental result for LSTM

LSTM			
Learning rate	RNN units	Keep probability	Accuracy
0.001	128	0.7	0.790
0.001	128	0.8	0.787
0.001	256	0.7	0.788
0.001	256	0.8	0.770
0.005	128	0.7	0.800
0.005	128	0.8	**0.825**
0.005	256	0.7	**0.822**
0.005	256	0.8	0.795

Table 5: Experimental result for Bidirectional LSTM

Bidirectional LSTM			
Learning rate	RNN units	Keep probability	Accuracy

0.001	128	0.7	0.785
0.001	128	0.8	0.775
0.001	256	0.7	0.785
0.001	256	0.8	0.793
0.005	128	0.7	0.785
0.005	128	0.8	**0.818**
0.005	256	0.7	**0.821**
0.005	256	0.8	0.808

The result demonstrates that the network works best when the model is relatively simple and learning rate is relatively high. The best accuracies (0.825 for LSTM and 0.821 for BiLSTM respectively) are better than the benchmark accuracy that is 0.812.

4.1.3. Experiment result for implementation of Recursive Neural Network

We built Recursive Neural Network from scratch. To make implementation simpler, we only implemented stochastic gradient descent algorithm and L2 regularization (instead of dropout). The downside of this simple implementation is that the training time would be relatively long and the performance is not stable. Therefore, we did not use grid search to find the best combination of hyperparameters. Instead, we run a "smoke test" on the network for each combination of hyperparameters. More specifically, we examined each combination of hyperparameters for 10 epochs. If the performance of the network had a clear tendency of increasing accuracy and decreasing loss during the 10 epochs' training, we would let the network continue to train until we reached the predefined epoch threshold. Otherwise, we would stop current run and switch to examine the next combination of hyperparameters.

In the end, we obtained the best combination of tuned hyperparameters:

- learning rate: 0.008
- regularization factor: 0.01
- epochs: 30
- hidden state dimension: 10

Figure 8 depicts the cost and accuracy for one of the best training runs.

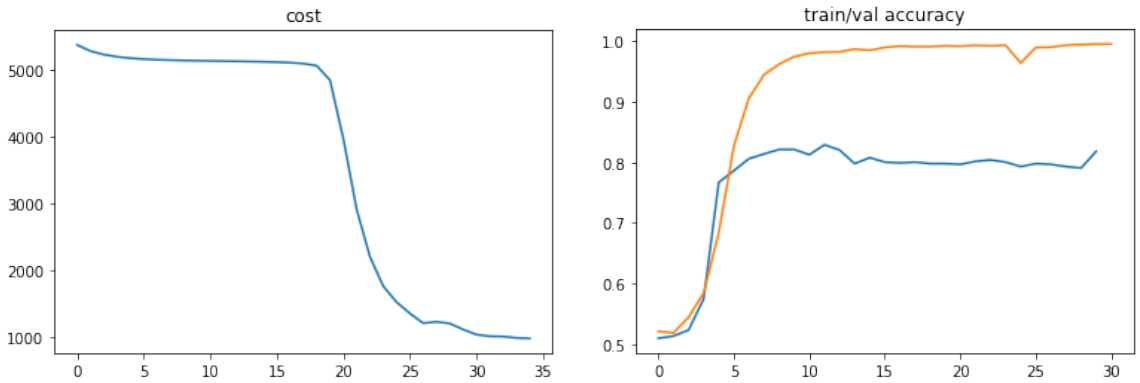

Figure 8:

The overall test accuracy we obtained for this trained model is 0.822, which is better than the benchmark by 0.01. However, the train/val graph in Figure 8 shows that the model is overfitting since there is a big gap between the training accuracy and validation accuracy. This gap can be narrowed down through many ways, such as adding more training data or regularization, or using mini-batching gradient descent. However, the model is good enough to demonstrate that the Recursive Neural Network performs better than Naïve Bayes for addressing sentimental analysis problem.

5. Conclusion

In this project, we have trained three models, Recursive Neural Network, Recurrent Neural Network and a benchmark model using Stanford Sentiment Treebank data to investigate whether the former two models can perform better than the benchmark in addressing sentimental analysis problem. The experimental results show that both Recursive Neural Network and Recurrent Neural Network achieved better accuracy than that of the benchmark.

Although the results are promising, both Recursive Neural Network and Recurrent Neural Network still have potential to achieve even better performance. One improvement we will be doing in the future project is adding attention mechanism to the two models. Moreover, we built a quite simple Recursive Neural Network in this project. We will add more features and functionalities to Recursive Neural Network in the future.

6. Reference

[1] Pang and L. Lee. 2008. Opinion mining and sentiment analysis. Foundations and Trends in Information Retrieval, 2(1-2):1–135.
[2] Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. Neural computation, 9(8), 1735-1780.
[3] Socher, R & Perelygin, A & Wu, J.Y. & Chuang, J & Manning, C.D. & Ng, A.Y. & Potts, C. (2013). Recursive deep models for semantic compositionality over a sentiment treebank. EMNLP. 1631. 1631-1642.
[4] https://nlp.stanford.edu/sentiment/treebank.html
[5] https://www.mturk.com/
[6] https://keras.io/
[7] https://www.tensorflow.org/
[8] http://deeplearning.net/software/theano/