Project 3: Sentiment Analysis

Victoria Anugrah Lestari
Department of Computer Science
The University of Texas at Austin
`vlestari@cs.utexas.edu`

Abstract

(I discussed with Vivek Pradhan and Marko Vasic.)
This project instructs us to perform sentiment analysis using two methods: feedforward neural network (FFNN) for the first part and recurrent neural network (RNN) or convolutional neural network (CNN) for the second part, using TensorFlow. We are expected to reach at least 70% accuracy for the FFNN model and 75% accuracy for the RNN or CNN model. For the first part, I achieve 75.79% accuracy on the development set using two hidden layers and gradient descent optimizer. For the second part, I achieve 79.92% accuracy on the development set using bidirectional LSTM and Adam optimizer.

1 Introduction

Sentiment analysis has remained widely popular among NLP researchers because it is useful to learn what the market says about a particular product. There are several categories for sentiment classification, namely, binary classification or ordinal classification. Given a bunch of reviews for any kind of products (books, movies, hotels, restaurants, etc.), a good model should be able to predict whether a sentence has a positive or negative sentiment.

This project, which is a kind of introductory assignment to learn using TensorFlow[1], is divided into two parts. The first part is to build a model using feedforward neural network. The second part is to implement a more advanced model using either recurrent neural network (RNN) or convolutional neural network (CNN).

[1] https://www.tensorflow.org

2 Data

For this project, we are given data from Rotten Tomatoes, which consist of 8530 training sentences, 1066 development sentences, and 1066 test sentences. The task is binary classification; each sentence has either positive or negative sentiment. We are also provided with GloVe word embedding vectors, 50-dimension vectors and 300-dimension vectors.

3 Methods and Experiments

This section discusses about methods and experiments that I explored while working on this project.

3.1 FeedForward Neural Network

I implemented the feedforward neural network (FFNN) model following the example provided in `feedforward_example.py`. Basically, the FFNN model computes the probability of each class according to the following equation:

$$P(\mathbf{y}|\mathbf{x}) = softmax(Wg(Vf(\mathbf{x}))$$

and determines the label using the `argmax` function. The model uses sigmoid non-linearity.

The method used for this part is the Deep Averaging Networks (DAN) introduced in (Iyyer et al., 2015). Given a sentence **x** = $\{x_1, x_2, ..., x_n\}$ with n words, DAN extracts the word embeddings for each $x_i, 1 \leq i \leq n$, and averages the word embedding vectors. Thus, each sentence is represented with a 50- or 300-length vectors of averaged word embeddings. This input is then provided into the FFNN model.

I maintained the learning rate decay factor of 0.99 and the initial learning rate of 0.1 as in

Table 1: Experiments for the FFNN model. Training accuracy and dev accuracy are in percentages. GD stands for gradient descent. The values in the dropout column are the keep probabilities.

# hidden layers	**# epochs**	**optimizer**	**dropout (keep prob)**	**train accuracy**	**dev accuracy**
2	100	GD	None	78.53	75.60
2	100	GD	0.5	77.52	74.20
2	100	GD	0.75	77.79	75.04
2	200	GD	None	79.13	**75.79**
2	200	Adam	None	77.49	74.95
3	100	GD	None	**79.26**	75.32
3	100	GD	0.75	78.5	75.32

`feedforward_example.py`. However, 10 decay steps were too small for training the model, so I increased it to 1000. Using 300 word embeddings always worked better than using 50 word embeddings, as expected.

Additionally, some parameters I played around with this model included:

- **Number of hidden layers.** I experimented with 1 to 5 hidden layers and discovered that 2 and 3 hidden layers gave the best performance, beating the 75% accuracy threshold for the development set, as specified in the instruction.

- **Dropout.** I tried using dropout with keep probabilities of 0.5 and 0.75. Dropout did not improve the model's performance, yet it did not significantly make it worse.

- **Optimizer.** I tried both AdamOptimizer and GradientDescentOptimizer and found that gradient descent slightly worked better for this problem.

- **Number of epochs.** The standard number of epochs is 100, but I also trained the model with 200 epochs. Adding more epochs slightly increased the accuracy.

Table 1 shows the comparison of the best settings for this problem. Using 3 hidden layers seemed to yield the highest training accuracy at

3.2 LSTM

For the second part, I decided to implement basic LSTM (long short-term memory) and extended it to bidirectional LSTM.

3.2.1 Basic LSTM

I implemented the basic LSTM model following the reference by Adit Deshpande[2], who performed sentiment analysis using LSTM and TensorFlow as well. However, the code I submitted was written by myself, consulting the `feedforward_example.py` as a guide for parameter initialization, training, and evaluation.

[2]https://www.oreilly.com/learning/perform-sentiment-analysis-with-lstms-using-tensorflow

I used the Dynamic RNN from the TensorFlow library for the basic LSTM implementation. I experimented with the size of the LSTM cell, trying 10 at first, and then changing it to 64 (as in Deshpande's guide) and 128. 64 cell size yielded the best performance, reaching almost 100% accuracy on the training set and above 70% accuracy on the development set, whereas other sizes only produced about 50% accuracy on both training and development sets.

However, the more important aspect is the learning rate initialization. Setting the learning rate to the incorrect value (too small or too large) caused the model not to learn anything, yielding training and development accuracy to be 50% with no decrease in the loss per epoch. I found TensorFlow's default initial learning rate of 0.001 to give the best performance.

The prediction is simply calculated as follows:

```
prediction = tf.matmul(value[-1],
      weight) + bias
```

where `value` is the output of the last layer of the RNN.

I tried initializing the weight with Xavier initializer and normal distribution and the bias with Xavier initializer and constant values of 0.1, but they did not give significantly different performance. I finally decided on initializing the weights with Xavier initializer and the bias with 0.1 constant values on the reason that it makes sense to treat the bias equally at first, updating it while training. I used Adam Optimizer because it performed better than gradient descent. I set the training to stop at 100 epochs or after it reached 0.1 loss, whichever the model reached first. Mostly it

Table 2: Summary of experiments with the LSTM models. The values in the column for dropout are the keep probabilities.

LSTM model	**dropout**	**best dev accuracy**
Basic	None	79.36
Basic	0.75	77.48
Bidirectional	None	**79.92**
Bidirectional	0.75	77.86

stopped after 70-80 epochs since its loss was already below 0.1.

Due to randomization in the initialization, the basic LSTM model produced different results each time I ran the program. The best result I got was 100% accuracy on the training set and 77.48% accuracy on the development set. The worst result I got was 100% accuracy on the training set and 75.79% accuracy on the development set.

3.2.2 Bidirectional LSTM

I also implemented bidirectional LSTM as a simple extension to the basic LSTM model, using the static bidirectional RNN library from TensorFlow. The parameters for this model are similar to the basic one, with 64 LSTM cell size, 0.001 learning rate initialization, 100 epochs or 0.1 loss stopping criteria, and the Adam Optimizer.

The bidirectional LSTM model did not yield much improvement over the basic one. However, the loss in the bidirectional LSTM dropped more quickly than in the basic LSTM. As a comparison, the loss after the 12th epoch for the bidirectional model was already 477.41, while the loss after the 15th epoch for the basic LSTM model was still 541.45.

Table 2 shows the summary of my experiments with the LSTM model. I did not have too many parameter settings, only exploring the performance when I used dropout for the LSTM cells and when I did not. It seemed that not using dropout yielded the better performance for both the basic LSTM model and the bidirectional LSTM model.

In conclusion, the bidirectional LSTM model increases the performance by 0.56%. Keep in mind that my implementations of both models are very simple; further exploration can be made to improve the model.

References

Mohit Iyyer, Varun Manjunatha, Jordan L. Boyd-Graber, and Hal Daumé III. 2015. Deep unordered composition rivals syntactic methods for text classification. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, ACL 2015, July 26-31, 2015, Beijing, China, Volume 1: Long Papers*. pages 1681–1691. http://aclweb.org/anthology/P/P15/P15-1162.pdf.