Deep Learning for Sentiment Analysis
Term Project Report

Chunxu Tang
ctang02@syr.edu

Zhi Xing
zxing01@syr.edu

1 Introduction

Due to the "world of mouth" phenomenon, mining the social media has become one of the most important tasks in Data Mining. Particularly, Sentiment Analysis on social media is useful for various practical purposes such as brand monitoring, stock prediction, etc. Sentiment Analysis is inherently difficult because of things like negation, sarcasm, etc. in texts, but Machine Learning techniques are able to produce accuracy above 90% for multi-class classification in regular texts such as movie reviews, arguably better than human. Unfortunately, the irregularities of social-media texts, such as misspelling, informal acronyms, emoticons, etc., make social-media-oriented Sentiment Analysis, or Text Mining in general, extremely difficult.

The buzzing Deep Learning is dominating pattern recognition in computer vision and voice recognition. As it turned out, it may be good at text classification as well. Various deep neural nets achieve state-of-art sentiment-polarity classification on Twitter data (about 87%) [2, 3, 7]. One of the advantages of Deep Learning is its ability to automatically learn features from data, and this ability leads to lots of interesting designs [1, 2, 3, 6, 7]. In this term project, we studied Recurrent Neural Network and Convolutional Neural Network and their related techniques, implemented and experimented on the networks for Twitter sentiment analysis.

2 Background

2.1 Word Embedding

Image and audio data are rich and high-dimensional, which gives their learning systems lots of things to work with. However, one of the difficulties in Natural Language Processing (NLP) is due to the sparsity of data, because normal word encodings are arbitrary to learning systems and no useful information exists between encodings of any two words. For example, in the certain encoding scheme, the word "good" may be encoded as `Id123` while the word "fantastic" is encoded as `Id456`. These two IDs don't mean anything special to any system even though the two words share similar meanings in lots of contexts. Word embeddings can resolve this issue to certain degree.

A word embedding is a parameterized function mapping words in certain language to high-dimensional vectors. It is grounded on the assumption that words that appear in the same contexts share semantic meaning. In the high dimensional embedding space, semantically similar words are mapped to points that are nearby each other. For example, the embeddings for "dog" and "cat" are close. In fact, words for animals are in general close to each other. In addition, words' semantic differences are captured by their distances in the embedding space. For instance, the distance vector of "Beijing" to "China" is similar to that of "Paris" to "France", and the distance vector of "woman" to "man" is similar to that of "queen" to "king". Because word embeddings are able to capture semantic relationships between words, it can provide NLP systems richer data.

We use the `word2vec` model in [5] in this term project. It comes in two flavors, the Continuous Bag-of-Words model (CBOW) and the Skip-Gram model. These two models are algorithmically similar. The difference is that, CBOW model is trained to predict a target word from its context while Skip-Gram model is trained to predict context words from target word. As an example, consider the phrase "the cat sits on the mat". We parse the data one word at a time, so each word gets to be the target word once. If "mat" is the target word, CBOW predicts "mat" from "the cat sits on the", while Skip-Gram predicts "the", "cat", "sits", "on" or "the" from "mat". CBOW model is better for smaller datasets while Skip-Gram model is better for larger ones.

2.2 Convolutional Neural Network (CNN)

Deep Learning is pushing the cutting-edge of computer vision, and one of the essential reasons is Convolutional Neural Network (CNN). The key characteristic of CNN that makes it so successful is its ability to automatically select features from inputs. The convolutional layer of a CNN acts like a sliding window over an input matrix. At each step of the sliding, normally referred to as a *stride*, the convolutional layer reduces the submatrix within the window to a single output value. This transformation is done at every stride, and the output values' relative positions are kept. Therefore, at the end, the input matrix is converted to a smaller output matrix. Since the same convolutional layer is applied at every stride, the number of parameters to learn is relatively small, which is why the network can be "deep". As an example, consider a training set of 100×100 pictures, a convolutional layer with window size 10×10 slides over each picture from left to right, top to bottom, and converts every 10×10 submatrix of pixels to a single number. After this layer, the 100×100 picture essentially becomes a smaller one. The actual resulting size depends on the *stride size*, which is the number of pixels the window slides for the next stride. In the example, if the stride size is 1, the output matrix is 91×91.

The set of parameters, once learnt, make the convolutional layer specialize at a certain aspect of the input. In a typical CNN, there can be multiple parallel *filters* in a convolutional layer, each of which specializes at a different aspect [4]. These aspects are the "features" learnt by the CNN. In the field of computer vision, one filter may specialize in detecting horizontal edges, while another may specialize in detecting vertical edges; one filter may specialize in colors, while another may specialize in contrasts. In the context of text mining, the 2-D picture becomes 2-D representation of sentence, which is normally obtained by converting a sentence to a sequence of word embeddings. Since the values in the vector can be combined to obtain different aspects of the word's meanings, the hope is that the convolutional layers can specialize so that CNN is able to automatically detect useful features from the word embeddings.

In a typical CNN, a convolutional layer is normally followed by a pooling layer, e.g. max pooling, which selects the most important the features. There can be multiple repetitions of convolution-pooling pairs and the final pooling layer is normally connected to a fully connected layer, which generates outputs for classification.

3 CNN for Twitter sentiment

3.1 Network structure

The structure of the CNN used in our term project is shown in Figure 1. It is a simplification of the model used in [3]. Although this model has a minimalist design, it has most of the typical

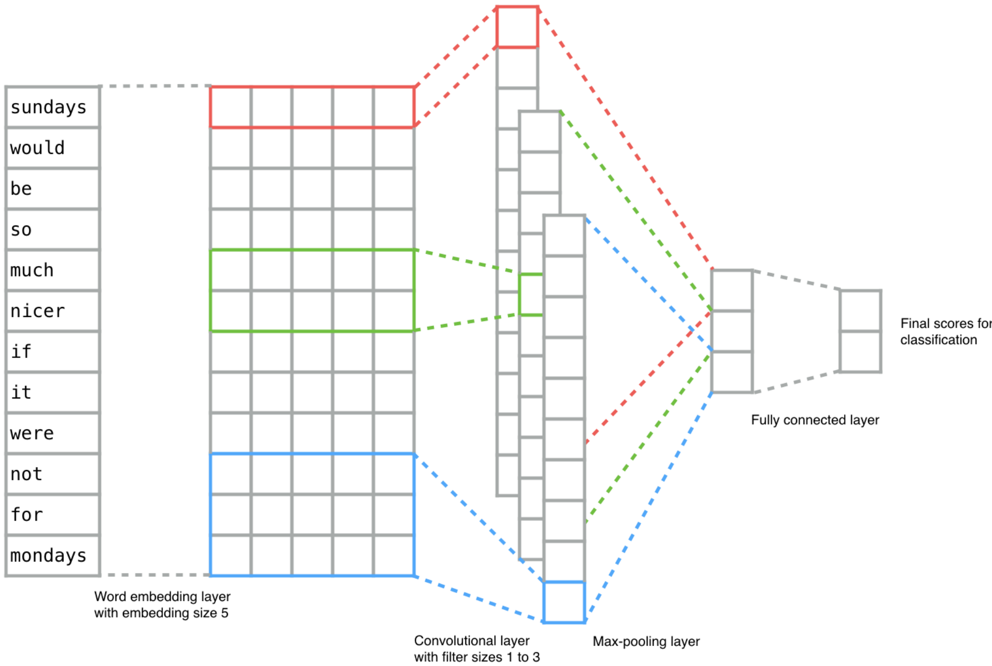

Figure 1: Structure of CNN

layers of a text-mining CNN: word embedding layer, convolutional layer, max-pooling layer, and fully-connected layer.

The first layer is the word embeddings, which is encoded as a weight matrix that is used as a lookup table. Each row of the weight matrix is a vector of size k represents a unique word in the vocabulary. Given a sentence of length n, padded if necessary, each word is replaced by its corresponding vector and the sentence is converted to an output matrix that is a concatenation of all the embeddings. This output matrix is represented as

$$\mathbf{x}_{1:n} = \mathbf{x}_1 \oplus \mathbf{x}_2 \oplus \ldots \oplus \mathbf{x}_n$$

where $\mathbf{x}_i \in \mathbb{R}^k$ is the embedding of the i-th word in the sentence, and $\mathbf{x}_{i:ij} \in \mathbb{R}^{kj}$ is used to denote the sub-matrix from the i-th word to the j-th word.

The second layer is the convolutional layer. Given a window size h, this layer is encoded by a *filter* $\mathbf{w} \in \mathbb{R}^{hk}$. When this filter is applied to the h-gram $\mathbf{x}_{i:i+h-1}$ of the input sentence, a feature c_i is generated by

$$c_i = f(\mathbf{w} \cdot \mathbf{x}_{i:i+h-1} + b)$$

where $b \in \mathbb{R}$ is bias and f is the activation function such as rectifier. The filter slides over all the possible h-grams of the input sentence to produce a feature map

$$\mathbf{c} = (c_1, c_2, \ldots, c_{n-h+1})$$

for $\mathbf{c} \in \mathbb{R}^{n-h+1}$.

The next layer is the max-pooling layer. It is applied to the feature map produced by the convolutional layer to produce a single feature $\hat{c} = \max(\mathbf{c})$. Taking the maximum essentially picks the most important feature in the feature map, which is an effective way to deal with variable sentence length, because the special word for padding has 0s in all the dimensions in its word embedding.

The last layer is a fully-connected layer. The $\hat{c}$ is the selected feature by a *single* filter, there're a number of filters with different window sizes. All the selected features from these filters are the input for this fully-connected layer, which uses softmax function to calculate the score for each class.

3.2 Regularization

The purpose of regularization is to prevent overfitting or co-adaptation. Unlike [3], only dropout is used to prevent co-adaptation, because according to [8] the L2-norm constraint used in [3] generally has little effect on the end result, and we want to keep the network as simple as possible. The dropout is applied to the fully-connected layer of the CNN. It works by randomly setting a proportion p of hidden units to 0 during learning. During testing, the dropout is disabled and the learnt weights are scaled down by p.

4 Experiment

4.1 CNN

Hyperparameter	**Value**
sentence length	200
word embedding size	200
filter window size	1, 2
number of filters per window size	128
dropout probability	0.5

Table 1: Hyperparameters of CNN model

The dataset used in the experiments is from the Stanford Twitter Sentiment corpus [1], which consists of 1.6 million two-class machine-labeled tweets for training, and 498 three-class hand-labeled tweets for test. We composed a smaller training set of 25,000 positive and 25,000 negative examples from the original training set, and a smaller test set consists of all the 359 positive and negative examples from the original test set. The data is preprocessed to remove punctuations and other symbols like "#" and "@", and to separate word contractions, e.g. "don't" to "do not". Each sentence is then tokenized by the `TweetTokenizer` provided in Python NLTK [2] library and padded to 200 tokens as necessary.

The CNN is implemented in TensorFlow, Google's deep learning library [3]. The network structure is defined in Python, but the backend is implemented in C++, so the training and testing procedures run as C++ programs. The hyperparameters of the model are listed in Table 1.

The skip-gram `word2vec` model proposed in [5] is used for the word embedding layer. The size of the embeddings is 200. The embedding model is pretrained using a subset of the Google News data used in [5] that consists of 17 million words, with a vocabulary of 71,291 words. After plugged into the CNN, the parameters of the `word2vec` model are set untrainable so it becomes a static lookup table. There're two reasons for fixing the parameters:

1. To reduce the number of parameters need to be learnt.

[1] `http://help.sentiment140.com/for-students/`
[2] `http://www.nltk.org`
[3] `https://www.tensorflow.org`

2. Tweets contain lots of noise, making the layer trainable exposes it to the noises, which may be counterproductive. [4]

Because of this layer, the vocabulary of the CNN is determined by the `word2vec` model, saved as a word-to-index dictionary. During data preprocessing, each word is converted to an index according to the dictionary.

For the convolutional layer, there can be a number of a filters with different window sizes. In order to keep our model simple and small, only two windows sizes, 1 and 2, are used, and each window size has 128 filters.

The model is trained with data batches of 128 tweets for 10 epochs. On a computer with 2.8GHz quad-core CPU and 16GB of memory, given pretrained `word2vec` model, the CNN model can be trained and tested well under half an hour. We run the training and testing 20 times, and obtained an average accuracy of 77.72%. Although this is not a very impressive performance, since our model has a very simple design and it is not optimized in any way, it still shows that there're lots of potentials in CNN.

5 Conclusion

References

[1] Cícero Nogueira dos Santos and Maira Gatti. Deep convolutional neural networks for sentiment analysis of short texts. In *COLING*, pages 69–78, 2014.

[2] Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. A convolutional neural network for modelling sentences. *arXiv preprint arXiv:1404.2188*, 2014.

[3] Yoon Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

[4] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[5] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[6] Richard Socher, Alex Perelygin, Jean Y Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the conference on empirical methods in natural language processing (EMNLP)*, volume 1631, page 1642. Citeseer, 2013.

[7] Xin Wang, Yuanchao Liu, Chengjie Sun, Baoxun Wang, and Xiaolong Wang. Predicting polarities of tweets by composing word embeddings with long short-term memory. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing*, volume 1, pages 1343–1353, 2015.

[8] Ye Zhang and Byron Wallace. A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification. *arXiv preprint arXiv:1510.03820*, 2015.

[4] We did test the model with the embedding layer set trainable. The performance is slightly worse.