Neural Networks for Text Classification

Nidhi Mannadiar, Kemal Gürsoy
Rutgers Business School Newark and New Brunswick, New Jersey, USA
e-mail: nidhi.mannadiar@rutgers.edu, kgursoy@business.rutgers.edu

Abstract

With the instant growth of information, text classification has become a vital technique for handling and organizing text data. It plays an important role in information extraction, text summarization, text retrieval, medical diagnosis, news group filtering, spam filtering, and sentiment analysis. Text classification is a foundation task in many Natural Language Processing (NLP) applications. Traditional text classifiers often rely on many human-designed features, such as dictionaries, knowledge bases and special tree kernels. In contrast to traditional methods, a recurrent convolutional neural network can be used for text classification without human-designed features. In our model, we apply a recurrent structure to capture contextual information when learning word representations, which may introduce considerably less noise compared to traditional approach. We also employ a max-pooling layer that automatically judges which words play key roles in text classification to capture the key components in texts. We use Gensim which is a topic modelling library for Python that provides access to Word2Vec and other word embedding algorithms for training, and it also allows pre-trained word embeddings to be downloaded from the internet and to be loaded. The vector representations of words learned by word2vec models can carry semantic meanings and are useful in various NLP tasks. Word embedding is a type of mapping that allows words with similar meaning to have similar representation. This work will classify text using neural networks, word embedding methods and Word2Vec with their implementation in Gensim.

Keywords - Text classification, word embeddings, word2Vec, recurrent convolutional neural network, max-pooling, regression.

1. Introduction

Automatic text classification has always been an important application and research topic since the inception of digital documents. Today, text classification is a necessity due to the very large amount of text documents that we have to deal with daily. With the explosion of information driven by the growth of the World Wide Web it is no longer feasible for a human observer to understand all the data coming in or even classify it into categories. Text classification is an essential component in many applications, such as web searching, information filtering, and sentiment analysis (Aggarwal and Zhai 2012). A key problem in text classification is feature representation, which is commonly based on the bag-of-words (BoW) model, where unigrams, bigrams, n-grams or some exquisitely designed patterns are typically extracted as features. Furthermore, several feature selection methods, such as frequency, Linear Discriminant Analysis (LDA) (Hingmire et al. 2013), are applied to select more discriminative features. Nevertheless, traditional feature representation methods often ignore the contextual information or word order in texts and remain unsatisfactory for capturing the semantics of the words. For example, in the sentence “A sunset stroll along the South Bank affords an array of stunning vantage points.”, when we analyze the word “Bank” (unigram), we may not know whether it means a financial institution or the land beside a river. In addition, the phrase “South Bank” (bigram), particularly considering the two uppercase letters, may mislead people who are not particularly knowledgeable about London to take it as a financial institution. After we obtain the greater context “stroll along the South Bank” (5-gram), we can easily distinguish the meaning. Although high-order n-grams and more complex features (such as tree kernels (Post and Bergsma 2013)) are designed to capture more contextual information and word orders, they still have the data sparsity.
problem, which heavily affects the classification accuracy. Recently, the rapid development of pre-trained word embedding, and deep neural networks has brought new inspiration to various NLP tasks. Word embedding is a distributed representation of words and greatly alleviates the data sparsity problem (Bengio et al. 2003). Mikolov, Yih, and Zweig (2013) shows that pre-trained word embeddings can capture meaningful syntactic and semantic regularities. With the help of word embedding, some composition-based methods are proposed to capture the semantic representation of texts. Socher et al. (2011a; 2011b; 2013) proposed the Recursive Neural Network (RNN) that has been proven to be efficient in terms of constructing sentence representations. However, the RNN captures the semantics of a sentence via a tree structure. Its performance heavily depends on the performance of the textual tree construction. Moreover, constructing such a textual tree exhibits a time complexity of at least O(n²), where n is the length of the text. This would be too time-consuming when the model meets a long sentence or a document. Furthermore, the relationship between two sentences can hardly be represented by a tree structure. Therefore, RNN is unsuitable for modeling long sentences or documents. Another model, which only exhibits a time complexity O(n), is the Recurrent Neural Network (RNN). This model analyzes a text word by word and stores the semantics of all the previous text in a fixed-sized hidden layer (Elman 1990). The advantage of RNN is the ability to better capture the contextual information. This could be beneficial to capture semantics of long texts. However, the RNN is a biased model, where later words are more dominant than earlier words. Thus, it could reduce the effectiveness when it is used to capture the semantics of a whole document, because key components could appear anywhere in a document rather than at the end. To tackle the bias problem, the Convolutional Neural Network (CNN), an unbiased model is introduced to NLP tasks, which can fairly determine discriminative phrases in a text with a max-pooling layer. Thus, the CNN may better capture the semantic of texts compared to recursive or recurrent neural networks. The time complexity of the CNN is also O(n). However, previous studies on CNNs tend to use simple convolutional kernels such as a fixed window (Collobert et al. 2011; Kalchbrenner and Blunsom 2013). When using such kernels, it is difficult to determine the window size: small window sizes may result in the loss of some critical information, whereas large windows result in an enormous parameter space (which could be difficult to train). Therefore, it raises a question: can we learn more contextual information than conventional window-based neural networks and represent the semantic of texts more precisely for text classification. To address the limitation of the above models, we propose a Recurrent Convolutional Neural Network (RCNN) and apply it to the task of text classification. First, we apply a bi-directional recurrent structure, which may introduce considerably less noise compared to a traditional window based neural network, to capture the contextual information to the greatest extent possible when learning word representations. Moreover, the model can reserve a larger range of the word ordering when learning representations of texts. Second, we employ a max-pooling layer that automatically judges which features play key roles in text classification, to capture the key component in the texts. By combining the recurrent structure and max-pooling layer, our model utilizes the advantage of both recurrent neural models and convolutional neural models. Furthermore, our model exhibits a time complexity of O(n), which is linearly correlated with the length of the text length.

2. Traditional Approach

Traditional text classification works mainly focus on: feature engineering, feature selection and using different types of machine learning algorithms. For feature engineering, the most widely used feature is the bag-of-words feature. Each document is a row in a count matrix, each word is a column. Feature selection aims at deleting noisy features and improving the classification performance. The most common feature selection method is removing the stop words (e.g., “the”). Advanced approaches use information gain, mutual information (Cover and Thomas 2012), or L1 (Lasso Regression) regularization (Ng
to select useful features. Machine learning algorithms often use classifiers such as logistic regression (LR), naïve Bayes (NB), and support vector machine (SVM). However, these methods have the data sparsity problem. The traditional text classification involves four phases:

1. Text pre-processing (Tokenize Text, Stemming, Delete Stop words) Figure 1. Shows the text classification process.
2. Feature extraction
3. Training classifier
4. Classification model using which text classification is done.

3. Today’s Approach – Neural Networks

Recently, neural networks have led to new ideas for solving the data sparsity problem, and many neural models for learning word representations have been proposed (Mnih and Hinton 2007; Mikolov 2012; Collobert et al. 2011; Mikolov et al. 2013). The neural representation of a word is called word embedding and is a real valued vector. The word embedding enables us to measure word relatedness by simply using the distance between two embedding vectors. With the pre-trained word embeddings, neural networks demonstrate their great performance in many NLP tasks.

4. Model

We propose a deep neural model to capture the semantics of the text. Figure 2 shows the network structure of our model. The input of the network is a document $D$, which is a sequence of words $w_1, w_2, ..., w_n$. The output of the network contains class elements. We use $p(k | D, \theta)$ to denote the probability of the document being class $k$, where $\theta$ is the parameters in the network.

Word Representation Learning

We combine a word and its context to present a word. The contexts help us to obtain a more precise word meaning. In our model, we use a recurrent structure, which is a bidirectional recurrent neural network, to capture the contexts.

We define $c_l(w_i)$ as the left context of word $w_i$ and $c_r(w_i)$ as the right context of word $w_i$. Both $c_l(w_i)$ and $c_r(w_i)$ are dense vectors with $|c|$ real value elements. The left-side context $c_l(w_i)$ of word $w_i$ is calculated using Equation (1), where $e(w_{i-1})$ is the word embedding of word $w_{i-1}$, which is a dense vector with $|e|$ real value elements. $c_l(w_{i-1})$ is the left-side context of the previous word $w_{i-1}$. The left-side context for the first word in any document uses the same shared parameters $c_l(w_1)$. $W^{l}$ is a matrix that transforms the hidden layer (context) into the next hidden layer. $W^{d}$ is a matrix that is used to combine the semantic of the current word with the next word’s left context. $f$ is a non-linear activation function. The right-side context $c_r(w_i)$ is calculated in a similar manner, as shown in Equation (2). The right-side contexts of the last word in a document share the parameters $c_r(w_n)$.

$$c_l(w_i) = f(W^{l} c_l(w_{i-1}) + W^{d} e(w_{i-1})) \quad (1)$$
$$c_r(w_i) = f(W^{r} c_r(w_{i+1}) + W^{dr} e(w_{i+1})) \quad (2)$$
As shown in Equations (1) and (2), the context vector captures the semantics of all left-side and right-side contexts. For example, in Figure 2, \( c_l(w_i) \) encodes the semantics of the left-side context “stroll along the South” along with all previous texts in the sentence, and \( c_r(w_i) \) encodes the semantics of the right-side context “affords an ...”. Then, we define the representation of word \( w_i \) in Equation (3) which is the concatenation of the left-side context vector \( c_l(w_i) \), the word embedding \( e(w_i) \) and the right-side context vector \( c_r(w_i) \). In this manner, using this contextual information, our model may be better able to disambiguate the meaning of the word \( w_i \) compared to conventional neural models that only use a fixed window (i.e., they only use partial information about texts).

\[
x_i = [c_l(w_i); e(w_i); c_r(w_i)]
\]  

(3)

The recurrent structure can obtain all \( c_i \) in a forward scan of the text and \( c_i \) in a backward scan of the text. The time complexity is \( O(n) \). We obtain the representation \( x_i \) of the word \( w_i \), we apply a linear transformation together with the tanh activation function to \( x_i \) and send the result to the next layer.

\[
y^{(2)}_i = \tanh \left( W^{(2)} x_i + b^{(2)} \right)
\]  

(4)

\( y^{(2)}_i \) is a latent semantic vector, in which each semantic factor will be analyzed to determine the most useful factor for representing the text.

**Text Representation Learning**

The convolutional neural network in our model is designed to represent the text. From the perspective of convolutional neural networks, the recurrent structure we previously mentioned is the convolutional layer. When all of the representations of words are calculated, we apply a max-pooling layer.

\[
y^{(3)}_i = \max_{1 \leq i \leq n} y^{(2)}_i
\]  

(5)

The max function is an element-wise function. The \( k \)-th element of \( y^{(3)} \) is the maximum in the \( k \)-th elements of \( y^{(2)}_i \). The pooling layer converts texts with various lengths into a fixed-length vector. With the pooling layer, we can capture the information throughout the entire text. There are other types of pooling layers, such as average pooling layers (Collobert et al. 2011). We do not use average pooling here because only a few words and their combination are useful for capturing the meaning of the document. The max-pooling layer attempts to find the most important latent semantic factors in the document. The pooling layer utilizes the output of the recurrent structure as the input. The time complexity of the pooling layer is \( O(n) \).
The overall model is a cascade of the recurrent structure and a max-pooling layer, therefore, the time complexity of our model is still $O(n)$. The last part of our model is an output layer. Similar to traditional neural networks, it is defined as

$$y^{(4)} = W^{(4)}y^{(3)} + b^{(4)} \tag{6}$$

Finally, the softmax function is applied to $y^{(4)}$. It can convert the output numbers into probabilities.

$$p_k = \frac{\exp(y^{(4)}_k)}{\sum_{k=1}^{n} \exp(y^{(4)}_k)} \tag{7}$$

**Training**

**Training Network parameters** We define all of the parameters to be trained as $\theta$.

$$\theta = \{E, b^{(2)}, b^{(3)}, c_l(w_1), c_r(w_n), W^{(2)}, W^{(4)}, W^{(l)}, W^{(r)}, W^{(sl)}, W^{(sr)}\} \tag{8}$$

Specifically, the parameters are word embeddings $E \in \mathbb{R}^{|e| \times |V|}$, the bias vectors $b^{(2)} \in \mathbb{R}^{|c|}$, $b^{(3)} \in \mathbb{R}^{|c|}$, the initial contexts $c_l(w_1); c_r(w_n) \in \mathbb{R}^{|c|}$ and the transformation matrices $W^{(2)} \in \mathbb{R}^{|e| \times |c|}$, $W^{(4)} \in \mathbb{R}^{|c| \times |c|}$, $W^{(l)} \in \mathbb{R}^{|c| \times |c|}$, $W^{(r)} \in \mathbb{R}^{|c| \times |c|}$, $W^{(sl)} \in \mathbb{R}^{|c| \times |c|}$, $W^{(sr)} \in \mathbb{R}^{|c| \times |c|}$, where $|V|$ is the number of words in the vocabulary, $H$ is the hidden layer size, and $O$ is the number of document types.

The training target of the network is used to maximize the log-likelihood with respect to $\theta$:

$$\theta \leftarrow \sum_{D \in \mathcal{D}} \log p(\text{class}_D | D, \theta) \tag{9}$$

where $\mathcal{D}$ is the training document set and \text{class}_0 is the correct class of document $D$. We use stochastic gradient descent (Bottou 1991) to optimize the training target. In each step, we randomly select an example $(D; \text{class}_0)$ and make a gradient step.

$$\theta \leftarrow \theta + \alpha \frac{\partial \log p(\text{class}_D | D, \theta)}{\partial \theta} \tag{10}$$

where $\alpha$ is the learning rate.

We use one trick that is widely used when training neural networks with stochastic gradient descent in the training phase. We initialize all of the parameters in the neural network from a uniform distribution. The magnitude of the maximum or minimum equals the square root of the “fan-in” (Plaut and Hinton 1987). The number is the network node of the previous layer in our model. The learning rate for that layer is divided by “fan-in”.

**Pre-training Word Embedding** Word embedding is a distributed representation of a word. Distributed representation is suitable for the input of neural networks. Traditional representations, such as one-hot representation, will lead to the curse of dimensionality (Bengio et al. 2003). Recent research (Hinton and Salakhutdinov 2006; Erhan et al. 2010) shows that neural networks can converge to a better local minima with a suitable unsupervised pre-training procedure.

In this work, we use the Skip-gram model to pre-train the word embedding. this model is the state-of-the-art in many NLP tasks (Baroni, Dinu, and Kruszewski 2014). The Skipgram model trains the embeddings of words $w_1, w_2, ..., w_T$ by maximizing the average log probability

$$\frac{1}{T} \sum_{t=1}^{T} \sum_{1 \leq i < j \leq T, i \neq j} \log p(w_{t+j} | w_t) \tag{11}$$

$$p(w_t | w_a) = \frac{\exp(e'(w_t)^T e(w_a))}{\sum_{k=1}^{|V|} \exp(e'(w_k)^T e(w_a))} \tag{12}$$

where $|V|$ is the vocabulary of the unlabelled text. $e'(w_i)$ is another embedding for $w_i$. We use the embedding $e'\cdot$ because some speed-up approaches (e.g., hierarchical softmax (Morin and Bengio 2005)) will be used here, and $e'$ is not calculated in practice.
5. Experiments

Datasets
To demonstrate the effectiveness of the proposed method, we perform the experiments using the 20Newsgroups dataset. 20Newsgroups: This dataset contains messages from twenty newsgroups. We use the by date version and select four major categories (computers, politics, recreation, religion) followed by Hingmire et al. (2013).

We pre-process the dataset as follows. For English documents, we use the Stanford Tokenizer to obtain the tokens. The evaluation metric of the 20News- groups is the Macro-F1 measure followed by the state-of-the-art work. The hyper-parameter settings of the neural networks may depend on the dataset being used. We choose one set of commonly used hyper-parameters following previous studies (Collobert et al. 2011; Turian, Ratinov, and Bengio 2010). Moreover, we set the learning rate of the stochastic gradient descent as 0.01, the hidden layer size as $H = 100$, the vector size of the word embedding as $|e| = 50$ and the size of the context vector as $|c| = 50$. We train word embeddings using the default parameter in word2vec\(^\text{2}\) with the Skip-gram algorithm. We use Wikipedia dumps in English to train the word embedding.

6. Comparison of Methods

We compare our method with widely used text classification methods and the state-of-the-art approaches for each dataset.

**Bag of Words/Bigrams + LR/SVM** Wang and Manning (2012) proposed several strong baselines for text classification. These baselines mainly use machine learning algorithms with unigram and bigrams as features. We use logistic regression (LR) and (Support vector Machine)SVM\(^\text{3}\), respectively. The weight of each feature is the term frequency.

**Average Embedding + LR** This baseline uses the weighted average of the word embeddings and subsequently applies a softmax layer. The weight for each word is its tfidf value. Huang et al. (2012) also used this strategy as the global context in their task. Klementiev, Titov, and Bhattacharai (2012) used this in cross lingual document classification.

**LDA** LDA-based approaches achieve good performance in terms of capturing the semantics of texts in several classification tasks. We select two methods as the methods for comparison: Classify LDA-EM(Emperical Bayes) (Hingmire et al. 2013) and Labeled-LDA (Li, Sun, and Zhang 2008).

**Tree Kernels** Post and Bergsma (2013) used various tree kernels as features. It is the state-of-the-art work in the native language classification task. We list two major methods for comparison: the context-free grammar (CFG) produced by the Berkeley parser (Petrov et al. 2006) and the reranking feature set of Charniak and Johnson (2005) (C&J).

**RecursiveNN** We select two recursive-based methods for comparison with the proposed approach: the Recursive Neural Network (RecursiveNN) (Socher et al. 2011a) and its improved version, the Recursive Neural Tensor Networks (RNTNs) (Socher et al. 2013).

**CNN** We also select a convolutional neural network (Collobert et al. 2011) for comparison. Its convolution kernel simply concatenates the word embeddings in a pre-defined window. Formally,

$$x_i = \{e(w_i-\lfloor\text{win}/2\rfloor); \ldots; e(w_i); \ldots; e(w_i+\lfloor\text{win}/2\rfloor)\}.$$

7. Results and Discussion

- Neural network approaches (RecursiveNN, CNN, and RCNN) when compared to the widely used traditional methods (e.g., BoW+LR), show that the neural network approaches outperform the traditional methods. It proves that neural network-based approach can effectively compose the semantic representation of texts. Neural networks can capture more contextual information of features compared with traditional methods based on BoW model and may suffer from the data sparsity problem less.
When comparing CNNs and RCNNs to RecursiveNNs, the convolution-based approaches achieve better results. This illustrates that the convolution-based framework is more suitable for constructing the semantic representation of texts compared with previous neural networks. We believe the main reason is that CNN can select more discriminative features through the max-pooling layer and capture contextual information through convolutional layer. By contrast, RecursiveNN can only capture contextual information using semantic composition under the constructed textual tree, which heavily depends on the performance of tree construction. Moreover, compared to the recursive-based approaches, which require O(n^2) time to construct the representations of sentences, our model exhibits a lower time complexity of O(n). In practice, the training time of the RNTN as reported in Socher et al. (2013) is approximately 3-5 hours.

In the 20News dataset, RCNN outperforms the state-of-the-art methods. We reduce the error rate by 33% for the 20News dataset with the best baselines. The results prove the effectiveness of the proposed method.

We compare our RCNN to well-designed feature sets in the 20News dataset. We believe that the RCNN can capture long-distance patterns, which are also introduced by tree kernels. Despite the competitive results, the RCNN does not require handcrafted feature sets, which means that it might be useful in low-resource languages.

We also compare the RCNN to the CNN and find that the RCNN outperforms the CNN in all cases. We believe that the reason is the recurrent structure in the RCNN captures contextual information better than window-based structure in CNNs. This results demonstrate the effectiveness of the proposed method. To illustrate this point more clearly, we propose a detailed analysis in the next subsection.

Contextual Information

In this subsection, we investigate the ability of the recurrent structure in our model for capturing contextual information in further detail. The difference between CNNs and RCNNs is that they use different structure for capturing contextual information. CNNs use a fixed window of words as contextual information, whereas RCNNs use the recurrent structure to capture a wide range of contextual information. The performance of a CNN is influenced by the window size. A small window may result in a loss of some long-distance patterns, whereas large windows will lead to data sparsity. Furthermore, a large number of parameters are more difficult to train. We consider all odd window sizes from 1 to 19 to train and test the CNN model. For example, when the window size is one, the CNN only uses the word embedding [e(wi)] to represent the word. When the window size is three, the CNN uses [e(wi-1); e(wi); e(wi+1)] to represent word wi. The test scores for these various window sizes are shown in Figure 3. Because of space limitations, we only show the classification results for the 20Newsgroups dataset. In this figure, we can observe that the RCNN outperforms the CNN for all window sizes. It illustrates that the RCNN could capture contextual information with a recurrent structure that does not rely on the window size. The RCNN outperforms window based CNNs because the recurrent structure can preserve longer contextual information and introduces less noise.

Learned Keywords

To investigate how our model constructs the representations of texts, we list the most important words in the test set. The most important words are the information most frequently selected in the max-pooling layer. Because the word representation in our model is a word together with its context, the context may contain the entire text. We only present the center word and its neighboring trigram. For comparison, we also list the most positive/negative trigram phrases extracted by the RNN (Socher et al. 2013).
In contrast to the most positive and most negative phrases in RNTN, our model does not rely on a syntactic parser, therefore, the presented n-grams are not typically “phrases”. The results demonstrate that the most important words for positive sentiment are words such as “worth”, “sweetest”, and “wonderful”, and those for negative sentiment are words such as “awfully”, “bad”, and “boring”.

Algorithm

In our model, we apply a recurrent structure to capture contextual information when learning word representations, which may introduce considerably less noise compared to traditional approach. We also employ a max-pooling layer that automatically judges which words play key roles in text classification to capture the key components in texts. We use Gensim which is a topic modelling library for Python that provides access to Word2Vec and other word embedding algorithms for training, and it also allows pre-trained word embeddings to be downloaded from the internet and to be loaded. The vector representations of words learned by word2vec models can carry semantic meanings and are useful in various NLP tasks. Word embedding is a type of mapping that allows words with similar meaning to have similar representation. This model will classify text using neural networks, word embedding methods and Word2Vec with their implementation in Gensim. See Appendix A for the algorithm for text classification using neural networks.

Conclusion

We introduced recurrent convolutional neural networks to text classification. Our model captures contextual information with the recurrent structure and constructs the representation of text using a convolutional neural network. The experiment demonstrates that our model outperforms CNN and RNN using a text classification dataset.

Acknowledgments

I would like to thank Prof. Kemal Gursoy from the bottom of my heart for having mentored me and for the excellent guidance and thought-provoking insights throughout my project.

References


Siwei Lai, Liheng Xu, Kang Liu, Jun Zhao Recurrent Convolutional Neural Networks for Text Classification National Laboratory of Pattern Recognition (NLPR) Institute of Automation, Chinese Academy of Sciences, China


Ng, A. Y. 2004. Feature selection, l1 vs. l2 regularization, and rotational invariance. In ICML, 78.


APPENDIX A

Algorithm for Text Classification using RCNN:

Creating word embeddings:
1. Read the data into a list of strings using the `read_data()` Function. (Extracts the first file enclosed in a zip file as a list of words.)
2. Process raw inputs into a dataset using the `build_dataset()` function.
3. Build the vocabulary and replace rare words with UNK token. We define a vocabulary to optimize memory usage.
4. Define a function to generate a training batch for the skip-gram model.
   4.1. Generate batch data using `input_word` `skip_window()`
   4.2. The input word is at the center of the buffer and the context words are generated
   4.3. We backtrack a little bit to avoid skipping words in the end of a batch.
5. Dimension of the embedding vector.
   5.1. Find how many words are to considered to the left and right.
   5.2. Find how many times to reuse an input to generate a label.
6. Build and train a skip-gram model. We pick a random validation set to sample nearest neighbors.
   6.1. Here we limit the validation samples to the words that have a low numeric ID, which by construction are also the most frequent.
   6.2. Random set of words to evaluate similarity on.
   6.3. Only pick dev samples in the head of the distribution.
   6.4. Number of negative examples to sample.
7. Look up embeddings for inputs.
8. Construct the variables for the softmax.
9. Convert `train_context` to a one-hot format.
10. Construct the SGD optimizer using a learning rate of 1.0.
11. Compute the cosine similarity between minibatch examples and all embeddings.
12. We perform one update step by evaluating the optimizer `op` (including it in the list of returned values for `session.run()`) The average loss is an estimate of the loss over the last 2000 batches.
   Note that this is expensive (~20% slowdown if computed every 500 steps)
13. Construct the variables for the NCE loss and find all the word embeddings.

Implementing Recurrent Convolutional Neural Network:
1. Import `tensorflow` and define `TextRCNN()` Function includes following parameters: `sequence_length`, `num_classes`, `vocab_size`, `word_embedding_size`, `cell_type`, `hidden_size`, `l2_reg_lambda`.
2. Define the placeholders for input, output, max-pooling and dropout Layers.
3. Read the Embeddings file.
5. Calculate mean cross-entropy loss.
6. Calculate Accuracy of the model.
7. Use the 20NewsGroup dataset to test your model.
8. Validate your result.

APPENDIX B

Python Code: (Used Jupyter Notebook to implement the project)


```python
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
import collections
import math
```
import os
import random
import zipfile
import numpy as np
from six.moves import urllib
from six.moves import xrange
# pylint: disable=redefined-builtin
import tensorflow as tf

def maybe_download(filename, expected_bytes):
    """Download a file if not present, and make sure it's the right size.""
    if not os.path.exists(filename):
        filename, _ = urllib.request.urlretrieve(url + filename, filename)
    statinfo = os.stat(filename)
    if statinfo.st_size == expected_bytes:
        print('Found and verified', filename)
    else:
        print(statinfo.st_size)
        raise Exception('Failed to verify ' + filename + '.
Can you get to it with a browser?')
    return filename

filename = maybe_download('text8.zip', 31344016)
# Step 2: Build the dictionary and replace rare words with UNK token.

vocabulary_size = 50000

def build_dataset(words, n_words):
    """Process raw inputs into a dataset.""
    count = [[ UNK, -1]]
    count.extend(collections.Counter(words).most_common(n_words - 1))
    dictionary = dict()
    for word, _ in count:
        dictionary[word] = len(dictionary)
    data = list()
    unk_count = 0
    for word in words:
        index = dictionary[word]
        if index:
            data.append(index)
        else:
            unk_count += 1
            data.append(0)  # dictionary[UNK]
    count[0][1] = unk_count
    reversed_dictionary = dict(zip(dictionary.values(),
                                 dictionary.keys()))
    return data, count, dictionary, reversed_dictionary

data, count, dictionary, reverse_dictionary = build_dataset(vocabulary, vocabulary_size)
del vocabulary  # Hint to reduce memory.
print('Most common words (+UNK)', count[:5])
print('Sample data', data[:10], [reverse_dictionary[i] for i in data[:10]])
data_index = 0

def collect_data(vocabulary_size=10000):
    url = 'http://mattmahoney.net/dc/
    filename = maybe_download('text8.zip', url, 31344016)

    vocabulary = read_data(filename)
    print('Data size', len(vocabulary))

    data, count, dictionary, reverse_dictionary = build_dataset(vocabulary, vocabulary_size)
    del vocabulary  # Hint to reduce memory.
    return data, count, dictionary, reverse_dictionary

    # Step 3: Function to generate a training batch for the skip-gram model.
    data_index = 0
    # generate batch data
def generate_batch(batch_size, num_skips, skip_window):
    global data_index
    assert batch_size % num_skips == 0
    assert num_skips <= 2 * skip_window
    batch = np.ndarray(shape=(batch_size),
                        dtype=np.int32)
labels = np.ndarray(shape=(batch_size, 1), dtype=np.int32)
span = 2 * skip_window + 1  # [skip_window target skip_window]
buffer = collections.deque(maxlen=span)
for _ in range(span):
    buffer.append(data[data_index])
data_index = (data_index + 1) % len(data)
for i in range(batch_size // num_skips):
    target = skip_window # target label at the center of the buffer
targets_to_avoid = [skip_window]
    for j in range(num_skips):
        while target in targets_to_avoid:
            target = random.randint(0, span - 1)
        targets_to_avoid.append(target)
        batch[i * num_skips + j] = buffer[skip_window]
        labels[i * num_skips + j, 0] = buffer[target]
    data_index = (data_index + 1) % len(data)
# Backtrack a little bit to avoid skipping words in the end of a batch
data_index = (data_index + len(data) - span) % len(data)
return batch, labels

generate_batch(batch_size=8, num_skips=2, skip_window=1)
for i in range(8):
    print(batch[i], reverse_dictionary[batch[i]], '->', labels[i, 0], reverse_dictionary[labels[i, 0]])

# Step 4: Build and train a skip-gram model.
batch_size = 128
embedding_size = 128  # Dimension of the embedding vector.
skip_window = 1  # How many words to consider left and right.
um_skips = 2  # How many times to reuse an input to generate a label.
# We pick a random validation set to sample nearest neighbors. Here we limit the
# validation samples to the words that have a low numeric ID, which by
# construction are also the most frequent.
valid_size = 16  # Random set of words to evaluate similarity on.
valid_window = 100  # Only pick dev samples in the head of the distribution.
valid_examples = np.random.choice(valid_window, valid_size, replace=False)
um_sampled = 64  # Number of negative examples to sample.
graph = tf.Graph()
with graph.as_default():
    # Input data.
    train_inputs = tf.placeholder(tf.int32, shape=[batch_size])
    train_labels = tf.placeholder(tf.int32, shape=[batch_size, 1])
    valid_dataset = tf.constant(valid_examples, dtype=tf.int32)
    # Ops and variables pinned to the CPU because of missing GPU implementation
    with tf.device('/cpu:0'):
        # Look up embeddings for inputs.
        embeddings = tf.Variable(
            tf.random_uniform([vocabulary_size, embedding_size], -1.0, 1.0))
        embed = tf.nn.embedding_lookup(embeddings, train_inputs)
        # Compute the average NCE loss for the batch.
        loss = tf.reduce_mean(
            tf.nn.nce_loss(weights=nce_weights, biases=nce_biases, labels=train_labels, inputs=embed, num_sampled=num_sampled, num_classes=vocabulary_size))
        # Construct the variables for the NCE loss
        nce_weights = tf.Variable(
            tf.truncated_normal([vocabulary_size, embedding_size], stddev=1.0 / math.sqrt(embedding_size)))
        nce_biases = tf.Variable(tf.zeros([vocabulary_size]))
        # Compute the average NCE loss for the batch.
        # tf.nce_loss automatically draws a new sample of the negative labels each
        # time we evaluate the loss.
        loss = tf.reduce_mean(
            tf.nn.nce_loss(weights=nce_weights, biases=nce_biases, labels=train_labels, inputs=embed, num_sampled=num_sampled, num_classes=vocabulary_size))
        # Construct the SGD optimizer using a learning rate of 1.0.
        optimizer = tf.train.GradientDescentOptimizer(1.0).minimize(loss)
# Compute the cosine similarity between minibatch examples and all embeddings.

norm =
    tf.sqrt(tf.reduce_sum(tf.square(embeddings), 1, keep_dims=True))

normalized_embeddings = embeddings / norm

valid_embeddings =
    tf.nn.embedding_lookup(
        normalized_embeddings, valid_dataset)

similarity =
    tf.matmul(valid_embeddings, normalized_embeddings, transpose_b=True)

# Add variable initializer.

init =
    tf.global_variables_initializer()"
average_loss = 0
# Note that this is expensive (~20%
slowdown if computed every 500
steps)
if step % 10000 == 0:
sim = similarity.eval()
for i in xrange(valid_size):
    valid_word =
reverse_dictionary[valid_examples[i]]
    top_k = 8 # number of nearest
    neighbors
    nearest = (-sim[i, :]).argsort()[1:top_k + 1]
    log_str = 'Nearest to %s: %'
    valid_word
    for k in xrange(top_k):
        close_word = reverse_dictionary[nearest[k]]
        log_str = '%s %s,' % (log_str, close_word)
    print(log_str)
final_embeddings = normalized_embeddings.eval()

# Step 6: Visualize the embeddings.
def plot_with_labels(low_dim_embs, labels, filename='tsne.png'):
    assert low_dim_embs.shape[0] >= len(labels), 'More labels than
    embeddings'
    plt.figure(figsize=(18, 18))
    for i, label in enumerate(labels):
        x, y = low_dim_embs[i, :]
        plt.scatter(x, y)
        plt.annotate(label,
                        xy=(x, y),
                        xytext=(5, 2),
                        textcoords='offset points',
                        ha='right',
                        va='bottom')
    plt.savefig(filename)
try:
    # pylint: disable=g-import-not-at-top
    from sklearn.manifold import TSNE
    import matplotlib.pyplot as plt
    tsne = TSNE(perplexity=30,
                n_components=2, init='pca',
                n_iter=5000)
    plot_only = 500
    low_dim_embs =
    tsne.fit_transform(final_embeddings[:
plot_only, :])
    labels = [reverse_dictionary[i] for
              i in xrange(plot_only)]
    plot_with_labels(low_dim_embs, labels)
except ImportError:
    print('Please install sklearn,
    matplotlib, and scipy to show
    embeddings.')

import tensorflow as tf
class TextRCNN:
    def __init__(self, sequence_length,
                 num_classes, vocab_size,
                 word_embedding_size,
                 cell_type, hidden_size,
                 l2_reg_lambda=0.0):
        # Placeholders for input, output and
        dropout
        self.input_text =
        tf.placeholder(tf.int32,
                       shape=[None, sequence_length], name='inp
        self.input_y =
        tf.placeholder(tf.float32,
                       shape=[None, num_classes], name='inp
        self.dropout_keep_prob =
        tf.placeholder(tf.float32,
                       name='dropout_keep_prob')
        12_loss = tf.constant(0.0)
        text_length =
        self._length(self.input_text)
        # Embeddings
        with tf.device('/cpu:0'),
        tf.name_scope("bi-rnn"):
            self.output_fw, self.output_bw),
            states =
            tf.nn.bidirectional_dynamic_rnn
    with tf.name_scope("context"):
        shape =
        [tf.shape(self.output_fw)[8], 1,
         tf.shape(self.output_fw)[2]]
if cell_type == "vanilla":
    return
    tf.nn.rnn_cell.BasicRNNCell(hidden_size)
elif cell_type == "lstm":
    return
    tf.nn.rnn_cell.BasicLSTMCell(hidden_size)
elif cell_type == "gru":
    return
    tf.nn.rnn_cell.GRUCell(hidden_size)
else:
    print("ERROR: '" + cell_type + "' is a wrong cell type !!!")
    return None
# Length of the sequence data
@staticmethod
def _length(seq):
    relevant = tf.sign(tf.abs(seq))
    length = tf.reduce_sum(relevant, reduction_indices=1)
    length = tf.cast(length, tf.int32)
    return length
# Extract the output of last cell of each sequence
# Ex) The movie is good -> length = 4
# output = [ [1.314, -3.32, ..., 0.98]
#            [0.287, -0.50, ..., 1.55]
#            [2.194, -2.12, ..., 0.63]
#            [1.938, -1.88, ..., 1.31]
#            [0.0, 0.0, ..., 0.0]
#            ...
#            [0.0, 0.0, ..., 0.0] ]
# The output we need is 4th output of cell, so extract it.
@staticmethod
def last_relevant(seq, length):
    batch_size = tf.shape(seq)[0]
    max_length = int(seq.get_shape()[1])
    input_size = int(batch_size[1])
    index = tf.range(0, batch_size) *
    max_length + (length - 1)
    flat = tf.reshape(seq, [-1, input_size])
    return tf.gather(flat, index)