University Jean Monnet

Year-end Project

Learning Word Embeddings

Author:
Tu-My Doan

Supervisor:
Prof. Emilie Morvant

June, 2018

Contents

Abstract

The main objective of this project is to learn about Word Embeddings and its algorithms which are used widely in the field of Natural Language Processing (NLP). This report will come in two parts, the first one will discuss mainly about Word Embeddings, some algorithms that help create word embeddings and the introduction of TensorFlow framework. In the second part we will discuss about the practical aspect where we learn and apply word embeddings for our sentiment analysis on a selected dataset.

1 Introduction

We might have seen several definitions of word embeddings before from different sources of information ranging from research papers to online articles, but in general, it is a way of representing words in the form of numerical values usually in a shape of a vector in $\mathbb{R}^d$. Word embeddings are feature engineering techniques that widely used in the the field of Natural Language Processing (NLP) such as sentiment analysis, language modeling etc. To put it another way, word embeddings are unsupervisedly learned word representation vectors whose relative similarities correlate with semantic similarity.

Although being laid the theoretical foundations as early as in 1950's, its popularity peaked in the the year 2010 because of the increase of Deep Learning methods for NLP and remains popular until today.

2 Word Embeddings

2.1 Motivation

Nowadays, we are being surrounded with different kinds of information from images to audio, text and the need to process them required us to transfer them into another form that can be read by the computers or any algorithms which accept numeric inputs. While images and audio are treated as the form of rich, high dimension vectors, words come in the form of atomic symbols.

2.2 Types of word embeddings

There are two categories of word embedding models:

- Count-based methods
- Predictive methods

Let us understand each of these categories in details.

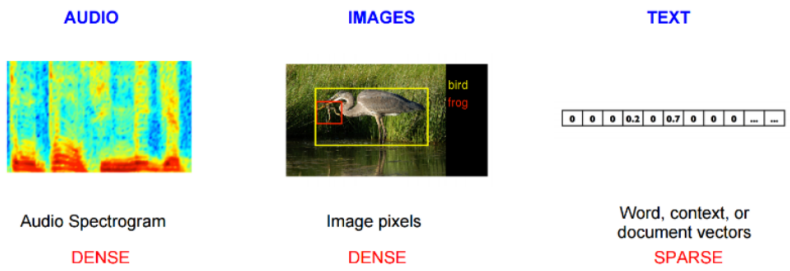

Figure 1: Density of different data sources. [Amit Mandelbaum et. al., 2016][5]

Table 1: Word counts for Document 1 and 2

	flower	weather	nice
D1	2	0	0
D2	0	1	1

2.2.1 Count-based methods

There are generally three types of vectors that we encounter under this category.

1. **Count Vector:**

   Consider a Corpus C of D documents $\{d1, d2, d3, ..., dD\}$ and N unique tokens (words) extracted out of the corpus C. The N tokens will form our dictionary and the size of the Count Vector matrix M will be given by $D \times N$. Each row in the matrix M contains the frequency of tokens in document $D(i)$. Let us understand this using a simple example:

   D1:
   `It is a flower.  It is another flower.`

   D2:
   `The weather is nice.`

   The dictionary created may be a list of unique tokens in the corpus =['flower', 'weather', 'nice']. Here, $D = 2$, $N = 3$. The count matrix M of size 2×4 will be represented as in Table 1.

   In practice, we usually have corpus containing millions or sometimes billions of words hence creating matrix like this will not be a computational efficiently way as they are very sparse.

2. **TF-IDF Vector (term frequency–inverse document frequency)**

   This is used to address the importance of a word in our corpus. In English language or any other languages, there will be cases that many words will have the tendency to appear more than the others. For example:

Document 1:
`This is his house which is really big.`

Document 2:
`This is the best time of the year.`

We can easily see that words like "the", "is", "this" etc. will have higher frequency than other words like "year", "house"...

The idea here is to decrease the weight of those frequent words and give more weight to other words in the subset of documents.

Table 2: Count for document 1

Document 1	This	is	his	house	which	really	big
Count	1	2	1	1	1	1	1

Table 3: Count for document 2

Document 2	This	is	the	best	time	of	year
Count	1	1	2	1	1	1	1

Term frequency

$$tf(t,d) = \frac{f(t,d)}{n}$$

$f(t,d)$: Number of times term t appears in a document d.

n: Total number of terms in document d.

Therefore, we have:

$$tf(This, d1) = \frac{1}{8}$$

$$tf(This, d2) = \frac{1}{8}$$

Inverse document frequency

If a word tends to appear more frequent in a document, term frequency will incorrectly identify the importance of that word (eg. "this", "is") and not giving enough weights for other words in the same document such as "house", "big" which are going to give us more information about the context. Therefore, we can say that terms like "this", "is" are not good keywords in distinguishing relevant and non-relevant documents and terms. As a result, we need a new way to reduce the weight of frequent words and give more weight to other terms in the document which are less frequent.

$$idf(t, D) = log(\frac{N}{|\{d \in D : t \in d\}|})$$

N: total number of documents in the corpus $N = |D|$
$|\{d \in D : t \in d\}|$: number of documents where the term t appears

Using the same example above we have:

$$idf(\text{"}this\text{"}, D) = log(\frac{2}{2}) = 0$$

The results tell us that the word "this" appears in both of the documents hence give us no interesting information. Now, let's try to calculate the idf of other word "house":

$$idf(\text{"}house\text{"}, D) = log(\frac{2}{1}) \approx 0.301$$

Because "house" exists in only document 1 which means it has more weight and more relevance to the document where it presents.

Term frequency–Inverse document frequency

Then tf–idf is calculated as:

$$tfidf(t, d, D) = tf(t, d) \times idf(t, D)$$

$$tfidf(\text{"}house\text{"}, d1, D) = tf(\text{"}house\text{"}, d1) \times idf(\text{"}house\text{"}, D) = \frac{1}{8} \times log(\frac{2}{1}) \approx 0.037$$

3. **Co-Occurrence Vector**

Similar words tend to stay together and have similar context. For example: Melon is a fruit. Durian is a fruit. Both Melon and Durian are also fruits - same context.

Co-occurrence – For a given corpus, the co-occurrence of a pair of words is the number of times they have appeared together in a Context Window.

```
penny wise and pound foolish.
a penny saved is a penny earned.
```

Letting $count(w(next)|w(current))$ represent how many times $wordw(next)$ follows the $wordw(current)$, we can summarize co-occurrence statistics for words "a" and "penny" as in Table 4.

Table 4: Words co-occurrence matrix.[12]

	a	and	earned	foolish	is	penny	pound	saved	wise
a	0	0	0	0	0	2	0	0	0
penny	0	0	1	0	0	0	0	1	1

A context window is specified by a number and the direction. Figure 2 is an example of context window 1. We can see that the word "penny" appears after the word "a" twice (line 1 and 6 in figure 2) whereas the word "earned", or "saved" follow the word "penny" only once. If we have a corpus of size N then we need a table of size $N \times N$ to represent a bigram frequency. Unfortunately, this kind of representation is very sparse as it contains a lot of zeros. But the advantage of this matrix is that it help preserve the semantic between words, eg. man and woman will be closer than man and durian. Once we computed this matrix, we can take use of it later but the memory required to store this matrix is quite expensive.

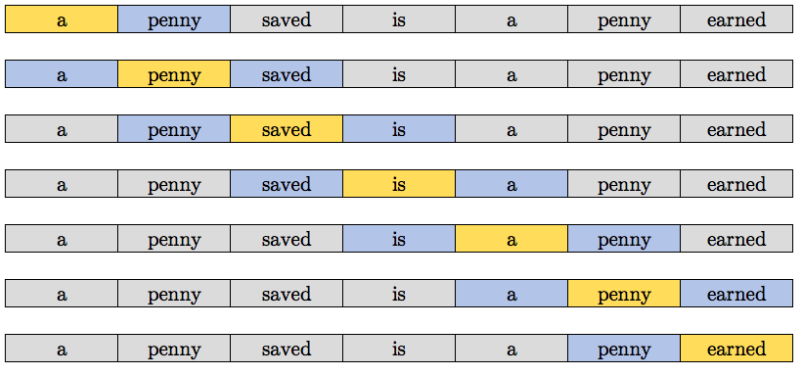

Figure 2: Context window size 1.

2.2.2 Predictive methods

In this part we will discuss about other methods called neural probabilistic language models. Its first proposals was introduced by Hinton back in 1986. Unlike the first methods, the predictive methods try to predict a word from its neighbors in terms of learned small, dense embedding vectors. To understand more about those methods, let us consider some of the popular ones namely Word2Vec [Mikolov et. al., 2013] [2][3][4], GloVe (Global Vectors for Word Representation) [Pennington et. al., 2014][6], and fasttext [Bojanowski et.al., 2016][7, 8].

Before going into details about those mentioned methods, let us have a quick look at the Neural Probabilistic Language Model [Bengio et. el., 2003][1] which was used as the basic for the Word2Vec model [Mikolov et. al., 2013][2][3][4]

1. **Neural Probabilistic Language Model**

   In this model, Bengio et. al. wanted to make improvements to other model [Goodman, 2001] by:

   - Taking into account contexts further than 1 or 2 words when using n-gram because there are much more information in the sequence immediately precedes the word to predict than just the identity of the previous couple of words.

- Taking into account the “similarity” between words. For example, by seeing a sentence like "The cat is walking in the bedroom" in the training corpus should help us generalize to make the sentence “A dog was running in a room” almost as likely, simply because “dog” and “cat” (resp. “the” and “a”, “room” and “bedroom”, etc...) have similar semantic and grammatical roles.

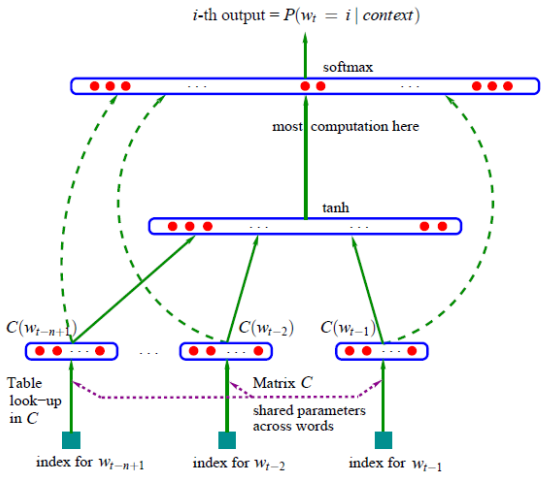

Figure 3: Neural architecture: $f(i, w_{t-1}, \cdots, w_{t-n+1}) = g(i, C(w_{t-1}), \cdots, C(w_{t-n+1}))$ where g is the neural network and C(i) is the i-th word feature vector. [Bengio et. al., 2003][1]

In the proposed model, Bengio et. al. used a continuous real-vector for each word to represent similarity between words which is helpful in representing compactly the probability distribution of word sequences from natural language text.

In Figure 3, we have a sequence w_1,...,w_T is the training set with $w_t \in V$ (large and finite Vocabulary). The objective is to learn a model $f(w_t, ..., w_{t-n+1}) = \hat{P}(w_t|w_1^{t-1})$ that maximize the out-of-sample likelihood (n is the value in n-gram). C is the mapping matrix ($|V| \times m$) of any element i of V to real vector $C(i) \in \mathbb{R}^m$ (m is the number of features associated with each word) which represents the distributed feature vector associated with each word in the vocabulary and is being shared across network.

The probability function g maps an input sequence of feature vectors for words in context to a conditional probability distribution over words in V for the next word w_t, this function can be implemented by a feed-forward or recurrent neural network or another parametrized function, with parameters ω. We want to find $\theta = (C, \omega)$ that help maximize the training corpus penalized log-likelihood:

$$L = \frac{1}{T} \sum_i log f(w_t, w_{t-1}, ..., w_{t-n+1}; \theta) + R(\theta)$$

where $R(\theta)$ is a regularization term applied to the weights of neural network only. The output of the model is created by the softmax activation function which guarantees positive probabilities summing to 1.

However, with this proposed model, there is a computational bottleneck with the output activation function.

2. **Word2Vec**

   In 2013, Mikolov et.el. proposed new model architectures that tackled the issue with computation time of model by Bengio et. al., 2003[1], not only allow user to compute continuous vector representations of words from very large data sets in less time but also improve the semantic regularities of learned words by taking advantage of word off-set technique. Before we go into details of the method, let us have a look at the word off-set technique first.

   (a) Word off-set technique:

   This technique is used for identifying linguistic regularities in continuous space word representations. The goal is to learn vector representation that can show us semantic as below:

$$x_{apple} - x_{apples} \approx x_{car} - x_{cars}, x_{family} - x_{families} \approx x_{car} - x_{cars}$$

   Mikolov et. al. have found that a simple vector offset method based on cosine distance is remarkably effective in solving the analogy question about syntactic and semantic tasks. They assumed that the relationships are presented as vector offsets, in the embedding space, all pairs of words sharing a particular relation are related by the same constant offset (Figure 4)

   In this model, to answer the analogy question $a : b :: c : d$ where d is unknown, we find the embedding vectors x_a ,x_b , x_c, and compute $y = x_b - x_a + x_c$.

   Where y is the continuous space representation of the word we expect to be the best answer. Obviously, we can't find a word exists at that exact position, so we then search for the word whose embedding vector has the greatest cosine similarity to y and output it:

$$w^* = argmax_w \frac{x_w y}{\|x_w\| \, \|y\|}$$

   When d is given, as in our semantic test set, we simply use $cos(x_b - x_a + x_c, x_d)$ for the words provided.

   (b) CBOW (Continuous bag-of-words)

   In this model, Mikolov et. al. used n words before and after the target word w_t to predict it as in Figure 5. They called it continuous bag-of-words (CBOW), as it uses continuous representations whose order doesn't matter.

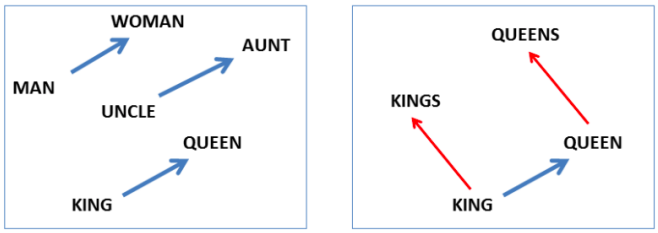

Figure 4: Left panel shows vector offsets for three word pairs illustrating the gender relation. Right panel shows a different projection, and the singular/plural relation for two words. In high-dimensional space, multiple relations can be embedded for a single word.[Mikolov et. al., 2013][2]

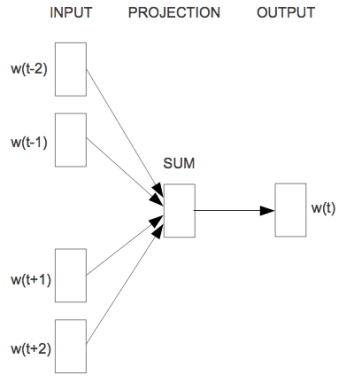

Figure 5: Continuous Bag of Words architecture.[Mikolov et. al., 2013][3]

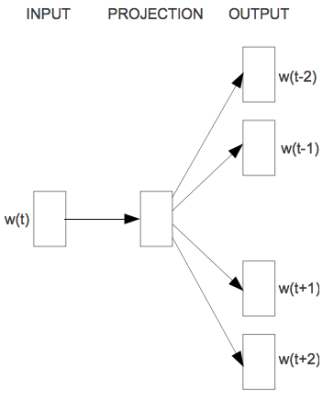

Figure 6: The Skip-gram model architecture. The training objective is to learn word vector representations that are good at predicting the nearby words.[Mikolov et. al., 2013][3]

(c) Skip-gram

We use Skip-gram model in order to find representation of word that

can help us predict the surrounding words given the context word. The objective is to maximize the average log probability given a sequence of training words $w_1, w_2, w_3, ..., w_T$

$$L = \frac{1}{T} \sum_{t=1}^{T} \sum_{-c \leq j \leq c, j \neq 0} logp(w_{t+j}|w_t)$$

where c is the size of the training context, the larger value of c, the more training examples we need and the higher accuracy we will get.

The basic Skip-gram formulation defines $p(w_{t+j}|w_t)$ using the softmax function:

$$p(w_O|w_I) = \frac{exp(v'^{\intercal}_{wO} v_{wI})}{\sum_{w=1}^{W} exp(v'^{\intercal}_{w} v_w)}$$

where v_w and v'_w are the "input" and "output" vector representations of w, and W is the number of words in the vocabulary. Unfortunately, the computation of this softmax function is very expensive when W is large. A solution to tackle this issue is to use hierarchical softmax.

i. *Hierarchical Softmax:*
Mikolov et. al. took advantages from the Hierarchical Softmax introduced by Morin et. al., 2005, the idea is to evaluate only about $log_2(W)$ output nodes instead of W nodes in neural network.
A binary tree representation is used for the output layer of W words as its leaves and each node represents the probabilities of its child nodes. The path from the root to word w is defined as $L(w)$, $n(w, j)$ is the j-th node on this path to w. Therefore, $n(w, 1) = root$ and $n(w, L(w)) = w$. Then the hierarchical softmax is as below:

$$p(w_O|w_I) = \prod_{j=1}^{L(w)-1} \sigma([n(w, j+1) = ch(n(w, j))] \, . v'^{\intercal}_{n(w,j)} v_{wI})$$

where $\sigma(x) = 1/(1 + exp(-x))$ the sigmoid function, $ch(n)$ is an arbitrary fixed child of n and $[x]$ is 1 if x is true and -1 otherwise. With this binary tree representation, the tree will have depth of $log_2(|W|)$ and at most $log_2(|W|)$ nodes needed to be evaluated to get the probabilities of a word.
Instead of calculating the output probabilities of the whole vocabulary as in the standard softmax function, now we only have to calculate the probabilities of taking left or right branch at the conjunction of the tree. Therefore, we need embeddings v'_n of every node in the tree and one representation v_w for each word w instead of representations v_w and v'_w for every word in standard softmax function.

ii. *Negative Subsampling*
The Negative sampling (NEG) objective below is used to replaced every term $p(w_O|w_I)$ in the skip-gram objective.

$$J_t(\theta) = log\sigma(v'^T_{w_O} v_{w_I}) + \sum_{i=1}^{k} \mathbb{E}_{wi \sim P_n(w)}[log\sigma(-v'^T_{wi} v_{w_I})]$$

where k are negative samples taken randomly from the noise distribution $P_n(w)$ for each data sample. For small dataset, k should be in range of 5-20 and 2-5 for large datasets. The overall objective function is:

$$J(\theta) = \frac{1}{T} \sum_{t=1}^{T} J_t(\theta)$$

Maximizing the probability of two words co-occurring in first log. To be clear, we want to maximize probability that real outside word appears while minimizing probability that random words appear around center word (second log).

iii. *Subsampling of Frequent Words*
In the first part of our Word embeddings methods we already saw that many frequent words won't give us more information than rare words. We also discussed how to reduce the weights of those frequents words by applying the *idf* formula to terms in the document. In 2013, Mikolov et. al. proposed an idea to treat the imbalance between the frequent and non-frequent words by using a simple subsampling approach:

$$p(w_i) = 1 - \sqrt{\frac{t}{f(w_i)}}$$

where each word w_i in the training set is discarded with above probability. And $f(w_i)$ is the frequency of word w_i and t is a chosen threshold of around 10^{-5}. According to Mikolov et. al. this is just a heuristic subsampling but it turns out to work well in practice by significantly improving the accuracy of learned vector of non-frequent words.

3. **GloVe**

The GloVe model was developed by Pennington et. el., 2014[6] in order to tackle the problems with the global matrix factorization methods, such as latent semantic analysis (LSA) [Deer- wester et. al., 1990] and local context window methods, such as the skip-gram model [Mikolov et. al., 2013][3][4] . It took the best advantages of both of these families models in order to not only work well with with statistical information (LSA) but also the analogy tasks (Skip-gram). The cost function of this model is as follow:

$$\hat{J} = \sum_{i,j} f(X_{ij})(w_i^T \tilde{w}_j - logX_{ij})^2$$

Where:

X is the word co-occurrence counts, whose entries X_{ij} tabulate the number of times word j occurs in the context of word i.

$f(X_{ij})$ is the weighting function which was used to fix the issue with all co-occurrences which are treated equally, even those that happen rarely or never. Such rare co-occurrences are noisy and carry less information than the more frequent ones. This weighting function has the following properties:

(a) $f(0) = 0$.

(b) $f(x)$ should be non-decreasing so that rare co-occurrences are not over-weighted.

(c) $f(x)$ should be relatively small for large values of x, so that frequent co-occurrences are not overweighted.

Both W and $\tilde{W}$ are the two vectors from all the vectors w and $\tilde{w}$. Vector w and $\tilde{w}$ are similar to output and input vectors of skip-gram model which are learned for both output and input words, but here w and $\tilde{w}$ are learned for all words in W.

When X is symmetric, W and $\tilde{W}$ are equivalent and differ only as a result of their random initializations; the two sets of vectors should perform equivalently and capture similar co-occurrence information. It turns out that summing them up gives a small boost in performance, with the biggest increase in the semantic analogy task.

$$X_{final} = W + \tilde{W}$$

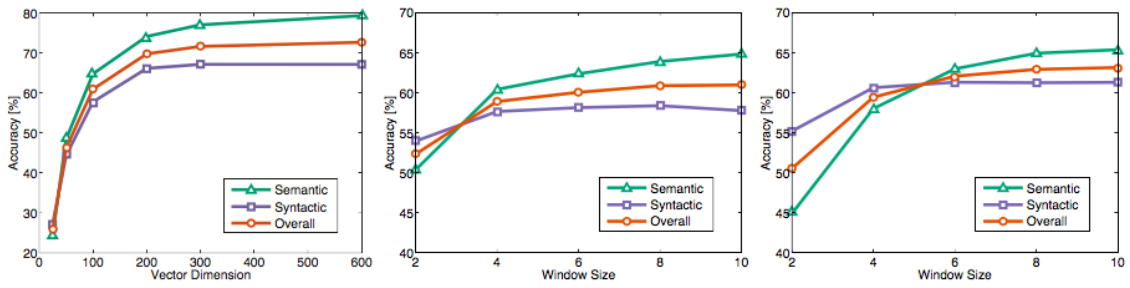

Figure 7: (a) Symmetric context; (b) Symmetric context; (c) Asymmetric context

Figure 7 shows the results of experiments of various vector dimension and context window. Symmetric is extending window size to the left and the right of target word whereas asymmetric is extending to the left only.

In (a), we can see that for vector dimension larger than 300, there is no improvement in accuracy. In (b) and (c) for small and asymmetric context windows, we have better performance as word order and intermediate context gives more information in syntactic task whereas the semantic information

depends on the more frequently non-local and can be captured more with larger window size.

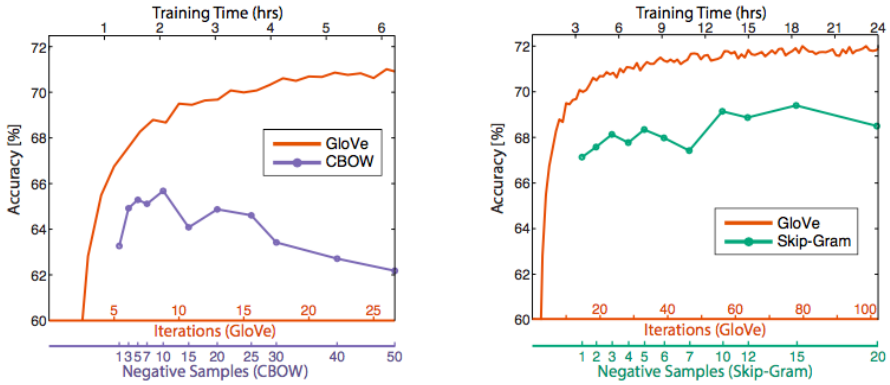

Figure 8: (a) GloVe vs CBOW; (b) GloVe vs Skip-Gram.
Overall accuracy on the word analogy task as a function of training time, which is governed by the number of iterations for GloVe and by the number of negative samples for CBOW (a) and skip-gram (b). In all cases, 300-dimensional vectors on the same 6B token corpus (Wikipedia 2014 + Gigaword 5) with the same 400,000 word vocabulary, and use a symmetric context window of size 10 are trained.

4. **`fastText`**

   `fastText` is an open-sourcing library developed by Facebook Research Lab which combines concepts from other successful models such as Bag-Of-Word, N-gram with the help of new extension called subword information to tackle problem with rich languages.

   It is being used for two different tasks: efficient text classification and learning word vector representations (Figure 9). It helps solve the issues of computation when working with large datasets. It uses hierarchical softmax function (similar to Mikolov et. al., 2013)[3]. It also exploits the fact that some classes appear more frequent than the others by using Huffman code when building the tree for categories representation. Therefore, lead to smaller depth for more frequent categories than the infrequent ones and less computation time.

   In `fastText`, bag-of-word model is used and every word in the vocabulary has a low dimensional vector associated to it, ignore word order. This is the hidden representation of the model which is shared across all classifiers for different categories, allowing information about words learned for one category to be used by other categories. Vectors learned by N-grams are also used to take into account local word order. `fastText` can be said to outperform other deep learning based methods in terms of training times by significantly decreasing it from several days to just a few seconds while increasing the accuracy on many standard problems, such as sentiment analysis (Table 5). This will be discussed in section (a) and (b).

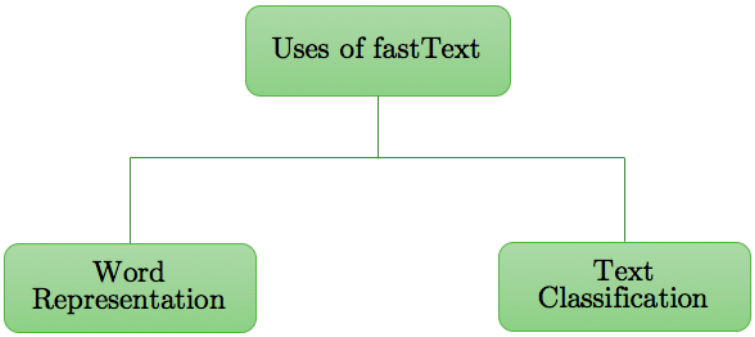

Figure 9: The use of `fastText`.[10]

Table 5: Comparison between `fastText` and deep learning-based methods.[9]

	Yahoo		Amazon Full		Amazon Polarity	
	Accuracy	Time	Accuracy	Time	Accuracy	Time
Char-CNN	71.2	1 day	59.5	5 days	94.5	5 days
CDCNN	73.4	2h	63	7h	95.7	7h
fastText	72.3	5s	60.2	9s	94.6	10s

Besides good performance in text classification, `fastText` can also be used to learn vector representations of words. It can work well with other languages than English such as German, Spanish, French, and Czech, by taking advantage of the languages morphological structure and achieve significantly better performance than the popular `word2vec` tool, or other word representations. (Section c)

(a) Simple Linear Model

This is used for sentence classification problems to represent sentences as bag of words (BoW) and train a linear classifier, e.g., a logistic regression. This model is shown in Figure 10, it is similar to CBOW of Mikolov et. al.,2013 [3]. Instead of using the middle word as the context, here it is replaced by a label and a softmax function is used for the output layer to calculate the probability distribution over the predefined classes. The objective function is to minimize the negative log likelihood over the classes as below:

$$-\frac{1}{N}\sum_{n=1}^{N} y_n log(f(BAx_n))$$

where A and B the weight matrices. N is the total number of documents, x_n is the normalized bag of features of the n-th document, y_n the label. The complexity of this model is $O(kh)$ where k is the number of classes and h the dimension of the text representation.

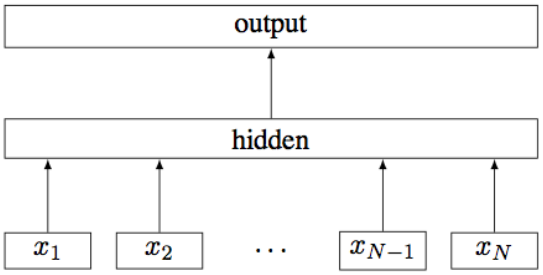

Figure 10: Model architecture of `fastText` for a sentence with N n-gram features x_1,...,x_N . The features are embedded and averaged to form the hidden variable [7]

(b) Hierarchical Softmax

The hierarchical softmax based on the Huffman coding tree [Mikolov et. al., 2013][3] was used in order to solve the issues with linear model as the former works better than the latter with large number of classes in terms of computing time. Moreover, linear classifiers do not share parameters among features and classes which limits their generalization in the context of large output space where some classes have very few examples. Complexity during training and test time is $O(hlog_2(k))$.

(c) Subword Information

Currently, most of the models used to learn continuous representations of words usually ignore the morphology of words and having no parameters sharing. This is the limitation in learning rich languages such as French, Turkish, Spanish etc. where there exist many forms of words (verbs, nouns...) and rare words also. As a result, there is a need to improve the current models to resolve those limitations. In 2017, Bojanowski et. al.[8] introduced an extension of the continuous skip-gram model [Mikolov et. al., 2013][4] that can learn subword information.

As in the skip-gram model, 2 vectors u_{w_t} and v_{w_t} in $\mathbb{R}^d$ are learned for word w_t and word w_c. Let call s the score of the dot product of 2 vectors above:

$$s(w_t, w_c) = \mathbf{u}_{w_t}^T \mathbf{v}_{w_c}$$

In Subword Model, each word w is represented as a bag of character n-gram where special symbols "<" and ">" were added at the beginning and the end of the word and the word w is also included in the set of n-grams to learn a representation for each word (in addition to character n-grams).

Example: word `where` and $n = 3$, its character n-grams is as below:

<`wh, whe, her, ere, re`> with special sequence <`where`>.

The new scoring function is as follow:

Table 6: Test accuracy [%] on sentiment datasets. `fastText` has been run with the same parameters for all the datasets. It has 10 hidden units and we evaluate it with and without bigrams.

Model	AG	Sogou	DBP	Yelp P.	Yelp F.	Yah. A.	Amz. F.	Amz. P.
BoW (Zhang et al., 2015)	88.8	92.9	96.6	92.2	58.0	68.9	54.6	90.4
ngrams (Zhang et al., 2015)	92.0	97.1	98.6	95.6	56.3	68.5	54.3	92.0
ngrams TFIDF (Zhang et al., 2015)	92.4	**97.2**	**98.7**	95.4	54.8	68.5	52.4	91.5
char-CNN (Zhang and LeCun, 2015)	87.2	95.1	98.3	94.7	62.0	71.2	59.5	94.5
char-CRNN (Xiao and Cho, 2016)	91.4	95.2	98.6	94.5	61.8	71.7	59.2	94.1
VDCNN (Conneau et al., 2016)	91.3	96.8	**98.7**	**95.7**	**64.7**	**73.4**	**63.0**	**95.7**
`fastText`, $h = 10$	91.5	93.9	98.1	93.8	60.4	72.0	55.8	91.2
`fastText`, $h = 10$, bigram	**92.5**	96.8	98.6	**95.7**	63.9	72.3	60.2	94.6

$$s(w, c) = \sum_{g \in \mathbf{G}_w} \mathbf{z}_g^T v_c$$

with a given dictionary of n-grams of size G, a word w has its set of n-grams as $\mathbf{G}_w \subset \{1, ..., G\}$. Each vector $\mathbf{z}_g$ is associated to each n-gram g.

A hashing function is used to map n-grams to integers in 1 to K with $K = 2.10^6$ to bound the memory requirements of our model.

2.3 TensorFlow Framework

In this part we will discuss some features of TensorFlow framework which is funded and supported by Google. This is just a short discussion about the features that we are going to use later in developing our Deep Learning application.

2.3.1 Ranks and Tensors

Tensor is mathematical notation for multidimensional arrays holding data values and the number of the dimension of tensor is called *rank*.

```
3. #a rank 0 tensor; a scalar with shape [],
[1., 2., 3.] #a rank 1 tensor; a vector with shape [3]
[[1., 2., 3.],[4., 5., 6.]] #a rank 2 tensor; a matrix with shape [2, 3]
[[[1., 2., 3.]],[[7., 8., 9.]]] #a rank 3 tensor with shape [2, 1, 3]
```

Listing 1: Examples of tensor ranks. [14]

2.3.2 Placeholders

In TensorFlow, placeholders are used for feeding data. Placeholders are predefined tensors with specific types and shapes. Unlike tensor constant `tf.constant()` which

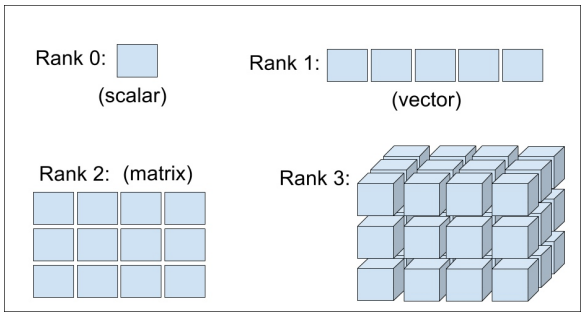

Figure 11: Tensor Ranks.[13]

has value assigned directly in it, placeholders don't have any data inside, and will be fed with data arrays during execution.

```
tf.placeholder(
    dtype,      # The type of elements in the tensor to be fed.
    shape=None, # The shape of the tensor to be fed (optional).
    name=None   # A name for the operation (optional).
)
```

Listing 2: TensorFlow Placeholder. [15]

In Listing 3, we define a placeholder x with data type of `tf.float32` and shape of `(1024, 1024)`. Next, we multiply `x` to itself `(x*x)` then we initiate a TensorFlow session. The line `print(sess.run(y))` will generate an error because for placeholder we have to feed it with data arrays (since it's not constant). Optional argument `feed_dict` is used to feed data to `Session.run()`.

```
x = tf.placeholder(tf.float32, shape=(1024, 1024))
y = tf.matmul(x, x) # matmul is the same as x * x

with tf.Session() as sess:
  print(sess.run(y))  # ERROR: will fail because x was not fed.

  rand_array = np.random.rand(1024, 1024) # Create random array to feed
    into placeholder.
  print(sess.run(y, feed_dict={x: rand_array}))  # Will succeed.
```

Listing 3: Example for Placeholder. [15]

2.3.3 Variables

Variables are a special type of tensor objects that allow us to store and update the parameters of our models in a TensorFlow session during training.

There are two ways to define variables:

- `tf.Variable(<initial-value>, name="variable-name")`
- `tf.get_variable(name, ...)`

The first method `tf.Variable()` is to create an object for a new variable and add it to the graph. Its `dtype` and `shape` will be the same as the initialized value.

The second one `tf.get_variable()` is to reuse an existing variable with a given name or create new one if it doesn't exist. We can set `dtype` and `shape` explicitly with this method. Therefore, it is a good practice to choose this one over the `tf.Variable()`. Listing 4 shows the creation of variables by the second option.

```
my_variable = tf.get_variable(
    "my_variable",
    [1, 2, 3])

my_int_variable = tf.get_variable(
    "my_int_variable",
    [1, 2, 3],
    dtype = tf.int32,
    initializer = tf.zeros_initializer)
```

Listing 4: Variables.[16, 13]

The `my_variable` is created by using the `tf.get_variable()` with default data type `tf.float32` and three-dimensional tensor with shape `[1, 2, 3]`. Its initial value will be randomized via `tf.glorot_uniform_initializer`.

For `my_int_variable`, same three-dimensional tensor is created but with different data type `tf.int32` which is initialized with zeros.

It is important to note that variables do not have memory and values assigned to them. Therefore, they must be initialized (assign memory and initial values) before executing any node in the computation graph. TensorFlow provides function `tf.global_variables_initializer()` that returns an operator to initialize all variables in computation graph. More information about graph and session will be discussed in the next section.

```
with tf.Session(graph=g1) as sess:
    sess.run(tf.global_variables_initializer())
)
```

Listing 5: Initialization of variables. [15]

2.3.4 Computation Graphs and Sessions

In TensorFlow, computation graph is used to derive relationships between tensors from the input all the way to the output. For example, Figure 12 shows the computation graph of `z = 2*(a-b)+c`.

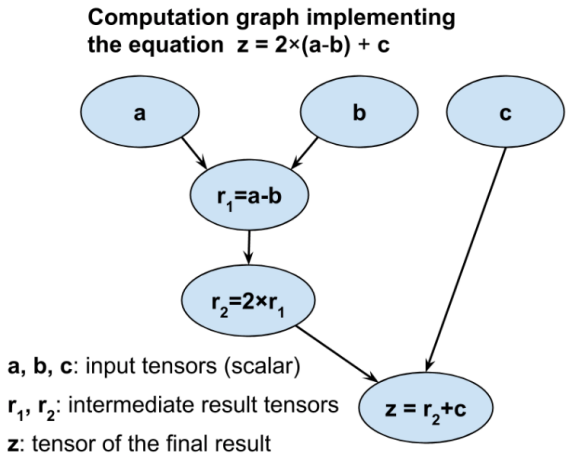

Figure 12: Computation graph.[13]

As we can see that the graph consists a collection of nodes and each node represent an operation that applies a function to its input tensor(s) and return none or more tensors to the output. The graph is built in order to support for the purpose of calculating the gradients.

Steps for building and compiling computation graph in TensorFlow:

1. Instantiate a new, empty computation graph.
2. Add nodes (tensors and operations) to the computation graph.
3. Execute the graph:
   (a) Start a new session
   (b) Initialize the variables in the graph
   (c) Run the computation graph in this session

```
# Instantiate new, empty computation graph
g = tf.Graph()
# Add nodes to the graph
with g.as_default():
    # Assign values 1, 2, 3 for a, b, c respectively
    a = tf.constant(1, name='a')
    b = tf.constant(2, name='b')
    c = tf.constant(3, name='c')

    z = 2*(a-b) + c
# Launch the graph
with tf.Session(graph = g) as sess:
    print('2*(a-b)+c => ', sess.run(z))
```

Listing 6: Graph example.[13]

Output: `2*(a-b)+c => 1`

A TensorFlow session is an environment where tensors and operations can be executed. This can be called using `tf.Session(graph = g)` where `g` is the graph that we created earlier. If we don't give any value for argument `graph` then the default graph will be used (which is empty as we didn't add any node to it). However, it is a good practice to always pass value for this argument, especially when we develop large neural network applications with many graphs. Next, we have to call function `run()` to execute operations.

2.3.5 Saving and restoring a model

When we build a neural network model, sometimes it will take hours, days, weeks or longer to finish and every time we need to use that trained model again, we have to repeat the process which is not practical and very time-consuming. But this is not gonna happen again with the help of TensorFlow `tf.train.Saver` class.

To save a model we can just simply add a new node into our graph as an instance of `Saver` class called `saver = tf.train.Saver()`, then call the method `save` as in `saver.save(sess, './trained-model')`, this statement will create three files with extensions `.data, .index, .meta`.

To restore the saved model, we have to follow below steps:

1. Rebuild the graph that has the same nodes and names as the saved model.
2. Restore the saved variables in a new `tf.Session` environment.

```
with tf.Session() as sess:
    new_saver = tf.train.import_meta_graph('./trained-model.meta')
    new_saver.restore(sess, './trained-model')
```

Listing 7: Restore the saved model.[13, 18]

2.3.6 Embeddings

In this part, we are going to look into details how to create word embeddings using TensorFlow.

Listing 8 is an example of word embeddings.

```
blue:
(0.01359, 0.00075997, 0.24608, ..., -0.2524, 1.0048, 0.06259)

blues:
(0.01396, 0.11887, -0.48963, ..., 0.033483, -0.10007, 0.1158)

```

```
orange:
(-0.24776, -0.12359, 0.20986, ..., 0.079717, 0.23865, -0.014213)

oranges:
(-0.35609, 0.21854, 0.080944, ..., -0.35413, 0.38511, -0.070976)
```

Listing 8: A 300-dimensional embeddings for some English words.[19]

Steps to learn word embeddings are as follow:

1. Splitting the text into words.
2. Assigning an integer (`word_ids`) to every word in the vocabulary.

   For example: the sentence `"I have a cat."` can be split up into 5 words [`I, have, a, cat, .`] and the corresponding `word_ids` tensor would have shape [5] and consist of 5 integers.
3. Mapping those `word_ids` to vectors by using the `tf.nn.embedding_lookup` function.

```
word_embeddings = tf.get_variable(
    "word_embeddings",
    [vocabulary_size, embedding_size]
    )

embedded_word_ids = tf.nn.embedding_lookup(
    word_embeddings,
    word_ids
    )
```

Listing 9: Create `word_embeddings` variable and mapping `word_ids` to vectors.[19]

The shape of `embedded_word_ids` is [`5, embedding_size`] and contains vectors for each of the 5 words. We will repeat steps above for the whole vocabulary so that at the end of the training, `word_embeddings` will contain all embeddings of all words in vocabulary.

3 Deep Learning Application

In this part we will discuss about the practical aspect where we first learn different embeddings (`word2vec & fastText`), and use them (for `GloVe` we will use the pre-trained embedding) with TensorFlow framework to build our Recurrent Neural Network (RNN) model for sentiment analysis problem on the selected dataset.

There will be 4 different Jupyter Notebooks to be built in this part. They are uploaded under github folder: `https://github.com/doantumy/Word-Embeddings`. For more information about the running files please check the above repository.

- Learning word embeddings for `word2vec` and `fastText` with `Gensim`
- Building RNN models for:
  - pre-trained `GloVe` word embeddings
  - `word2vec` word embeddings
  - and `fastText` word embeddings

Later we will compare the accuracy of those three RNN models.

3.1 Dataset

We are going to take the dataset from Kaggle challenge "*Bag of Words Meets Bags of Popcorn*"[21]. This is a movie review dataset that consists of 50,000 IMDB movie reviews, specially selected for sentiment analysis. A single-layer RNN model for sentiment analysis using a many-to-one architecture will be implemented in this problem.

3.2 Learning word embedding with `Gensim`

3.2.1 Data preparation

Details information about the codes can be found in the notebook `Word Embeddings with Gensim`. In this part, I will discuss about the key points only.

As mentioned earlier, the datasets from Kaggle come in three different parts: train data, unlabeled data and test data. For learning word embeddings, we can take use the train and unlabeled datasets.

There are 25,000 reviews in the train set and 50,000 reviews in the unlabeled set. We will use punkt tokenizer from NLTK to split reviews into sentences. First, convert reviews into lowercase, remove HTML tags, replace `'-'` with space, `'.'` by adding space after the dot. Two sentences will be considerd as one sentence if there is no space after the end of first sentence, tokenizer won't understand this well so we have to manually fix it first.

Then we will break down sentences into words, using the `special_characters` filter. The output of this is a list of sentences with a list of words for each sentence. For more information, please check the function `convert_to_sentences` in Jupyter Notebook.

Below is an emxample of first two sentences.

```
['with', 'all', 'this', 'stuff', 'going', 'down', 'at', 'the', 'moment',
    'with', 'mj', 'ive', 'started', 'listening', 'to', 'his', 'music',
   'watching', 'the', 'odd', 'documentary', 'here', 'and', 'there', '
   watched', 'the', 'wiz', 'and', 'watched', 'moonwalker', 'again']

```

```
['maybe', 'i', 'just', 'want', 'to', 'get', 'a', 'certain', 'insight', '
    into', 'this', 'guy', 'who', 'i', 'thought', 'was', 'really', 'cool'
    , 'in', 'the', 'eighties', 'just', 'to', 'maybe', 'make', 'up', 'my'
    , 'mind', 'whether', 'he', 'is', 'guilty', 'or', 'innocent']
```

Listing 10: First two sentences

When our data is ready, let's start to train embedding with `Gensim`. First, we will define values for the model which will be used for `word2vec` and `fastText`.

- `num_feature`: The dimension of word vector. The more dimension the better representation but this is going to take more time to learn and more data. However, since we don't have that much data, let's set this to 50 only (default is 100).
- `min_word_count`: Any words appears less than this number will not be considered in the learning (default is 5).
- `window_size`: For any given word, window defines how many words to consider to it's left and right (default is 5). This is the maximum distance between the current and predicted word within a sentence.
- `down_sampling`: The threshold for configuring which higher-frequency words are randomly downsampled, useful range is (0, 1e-5).
- `num_thread`: Number of parallel processes to run.
- `iteration`: Number of iterations (epochs) over the corpus (default is 5). However, in practice, it's advised that more iteration will improve the representations.
- Training algorithm: we will select between CBOW and Skip-gram model.
  - CBOW: works well with small dataset, well representation with rare words/phrases.
  - Skip-gram: faster training time, slightly better accuracy for frequent words.

  In this case, we will pick CBOW which is the default value.

3.2.2 `word2vec` embedding

Training time for this embedding is around 851.8s (roughly 14 mins).

```
num_feature = 50
min_word_count = 20
num_thread = 5
window_size = 10
```

```
down_sampling = 0.001
iteration = 20

logging.basicConfig(format='%(asctime)s : %(levelname)s : %(message)s',
    level=logging.INFO)
model = word2vec.Word2Vec(sentences,
                          iter = iteration,
                          size=num_feature,
                          min_count = min_word_count,
                          window = window_size,
                          sample = down_sampling,
                          workers=num_thread)
```

Listing 11: `word2vec` embedding

3.2.3 `fastText` embedding

Training time for this embedding is around 470.8s (roughly 7.8 mins) which is nearly 50% faster compared to `word2vec` embedding. This certifies that fact mentioned in the research paper that it can learn really fast hence having the name `fastText`.

```
num_feature = 50
min_word_count = 20
num_thread = 5
window_size = 10
down_sampling = 0.001
iteration = 20

logging.basicConfig(format='%(asctime)s : %(levelname)s : %(message)s',
    level=logging.INFO)
model_fastText = FastText(sentences,
                          size=num_feature,
                          window=window_size,
                          min_count=min_word_count,
                          workers=num_thread)
```

Listing 12: `fastText` embedding

3.2.4 Discussion

For checking with each models, they are available in the Jupyter notebook `Word Embeddings with Gensim`. In this part, I will only point out my own opinions about those two models.

In general, `word2vec` works well in finding similar words in terms of meaning. For example, if we want to find similar words for `character`. Here, the word `protagonist` (the leading character or one of the major characters in a drama, movie, novel, or other fictional text) is the one closest to the word `character` - Listing 14.

```
[('protagonist', 0.7609272003173828),
 ('role',        0.7386566996574402),
 ('personality', 0.7352930307388306),
 ('villain',     0.6820997595787048),
 ('persona',     0.6581060886383057),
 ('antagonist',  0.6566777229309082),
 ('attraction',  0.6436684131622314),
 ('rapport',     0.6315550208091736),
 ('relationship',0.6272909641265869),
 ('presence',    0.6150318384170532)]
```

Listing 13: `word2vec` - similar words for `character`

However, the case is diferent with `fastText`, it breaks a word into subwords, making the learning more efficient. As a result, it can find better similiar words in terms of family of the word not really exactly the synonyms. By saying the family of the word I mean other forms of a word such as verb, noun, adjective, adverb, etc. We can see this clearly in Listing 15 that verb `characterize` is closest to its noun. This is said to be the advantage of `fastText` as it can give us other forms of a word which happens very regular in rich languages such as French, Spanish, etc. where we have different forms of word in different contexts (eg. genders for nouns, adjectives, etc.) or different verb tenses (eg. present, past, perfect, future tense, etc.).

```
[('characterize',          0.8779246211051941),
 ('characterisation',      0.8767824172973633),
 ('characteristically',    0.8579357862472534),
 ('characteristic',        0.8445369005203247),
 ('characterization',      0.8444110751152039),
 ('protagonist',           0.8273643255233765),
 ('uncharacteristically', 0.8241872787475586),
 ('characteristics',       0.8222066164016724),
 ('characterized',         0.8184616565704346),
 ('characterisations',     0.8086223602294922)]
```

Listing 14: `fastText` - similar words for `character`

When it comes to analogy tasks, `word2vec` performs better than its counterpart. For example, if we have `germany:berlin :: france:?` and we want to find the missing word. It's obvious that our expectation will be `paris` which is the capital city of France. We get the correct result with `word2vec` but with `fastText` it's not the case (Listing 16). Instead of giving us `paris`, its output is some words that is similar to `berlin` in terms of writing (or form) - `merlin`.

```
[('merlin',     0.8392232656478882),
 ('palmer',     0.8322012424468994),
 ('darlene',    0.827530026435852),
 ('della',      0.822186291217804),
```

```
 ('daphne',     0.821194052696228),
 ('erin',       0.8182156682014465),
 ('lynne',      0.8165528774261475),
 ('kavner',     0.8089902400970459),
 ('gabriella', 0.8058374524116516),
 ('brynner',    0.8057600259780884)]
```

Listing 15: `fastText` - Analogy task

For finding the most different word among the group of words, the results of both model are similar. The syntax to manipulate with the two models are the same in `Gensim`.

After we are done with the model, we can save the model for later use. However, due to the fact that we are going to use them for RNN models training later, and to make things easier, I use two lists to save the information: one for vocabularies `(word_list)` and the other for vectors `(word_vector)` instead of using the original model itself. Those 2 lists are corresponding to each other in terms of indexing.

3.3 Building RNN models with word embeddings

3.3.1 Recurrent Neural Network

For sentiment analysis problem, the order of words in the sentences is very important to us. Therefore, we need a model that takes into account the sequence of words and RNN is a good model for this problem. Let's take a deeper look into RNN.

1. **Structure**

   In the Standard feed-forward network, information from the input layer will be passed into hidden layer then from this layer to the output layer. However, in Recurrent neural network, it gets information from both input layer and from hidden layer of previous time step. (Figure 13)

   There are two types of RNN architecture: Single layer and Multiple layer RNN. (Figure 14)

   In RNN, we consider data as a time-series data where x and y follow the order according to their time axis, hence creating a sequence. In our sentiment analysis problem, each of training review is a sequence of words and the time here running from 0 to T where T is the maximal length of the sequence. (Figure 15)

   At the beginning, $t = 0$, we can choose to initialize hidden units with 0 or very small random values. Later on, hidden units will get input from $x^{(t)}$ and also from its previous time step $h^{(t-1)}$.

   In the case of multilayer RNN, for example 2-layers RNN:

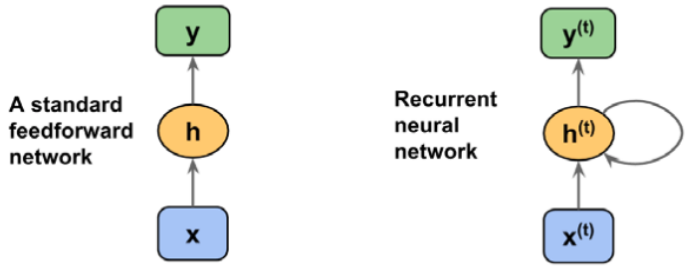

Figure 13: Standard feedforward and RNN structure [13]

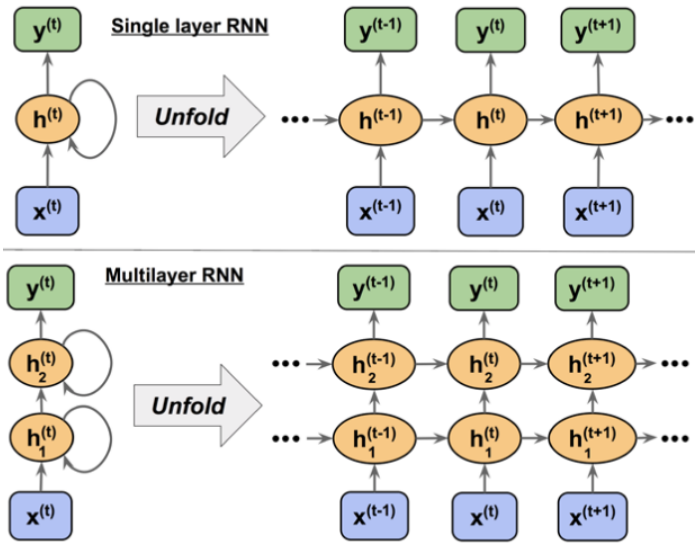

Figure 14: Different architectures of RNN [13]

- At layer 1: hidden units will get input from $x^{(t)}$ and its previous time step $h_1^{(t-1)}$
- At layer 2: hidden units will get input from $x^{(t)}$ and its previous time step $h_2^{(t-1)}$ and also input from first layer $h_1^{(t)}$

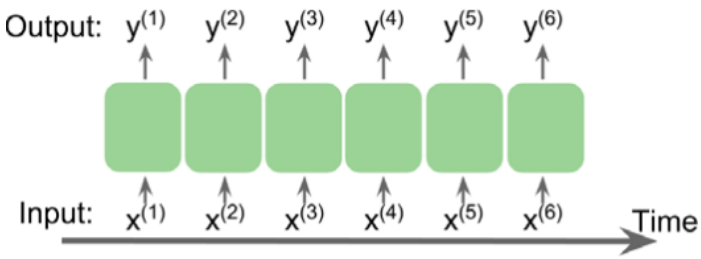

Figure 15: Time sequence [13]

2. **Activation computing in RNN**

   To make it simple, we will consider RNN with single layer. In the figure, there are edges that connected boxes to each other with specific weight values are not dependent on time but shared across the network.

   - W_{xh} is the weight for input layer and the hidden layer
   - W_{hh} is the weight for previous time step hidden unit with the current unit
   - W_{hy} is the weight for current hidden unit with output layer

   Calculating the activation for RNN is similar to other feed-forward network. Then we have the activation of hidden units at time step t as below:

$$h^{(t)} = \phi_h(W_{xh}x^{(t)} + W_{hh}h^{(t-1)} + b_h)$$

   where b_h is the bias for hidden unit and $\phi(.)$ is the activation function of hidden unit.

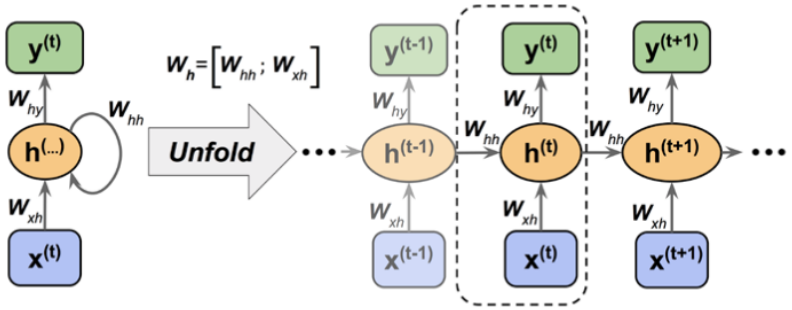

Figure 16: Activation of RNN [13]

3.3.2 RNN model with pre-trained `GloVe`

1. **Data Preparation**

   First, we have to clean up our dataset before doing the training. In the Kaggle competition, we have three different datasets namely labeled train data, unlabeled test data, and finally the unlabeled train data which is used for creating the embeddings with method like `word2vec`. But here we are doing the RNN model, hence the unlabeled train and test data won't help.

   The test dataset which contains no labels but only the review details, can't be used in this training process as there are no actual labels for us to compare with our predictions. As result, the training data will be split into train and validation and test sets.

   We will define our `clean_sentence` function to clean up the data by removing HTML tags, extra space, non-characters. But here we don't break down each review into a list of sentences, we consider each of them as a single training example. This cleaning will be applied to all reviews in the dataset.

```
i dont know why people think this is such a bad movie its got a
    pretty good plot some good action and the change of location for
     harry does not hurt either sure some of its offensive and
    gratuitous but this is not the only movie like that eastwood is
    in good form as dirty harry and i liked pat hingle in this movie
     as the small town cop if you liked dirty harry then you should
    see this one its a lot better than the dead pool 45
```

Listing 16: Example of a review in our data

Next, we have to identify what will be our maximum length for a sequence (or review) to put into the RNN. Because for RNN, it's important that we have input at the same length. As a result, plotting the frequency of sequence length in our dataset is needed. Based on the plot and information below, it's acceptable to set maximum sequence length to `230` words.

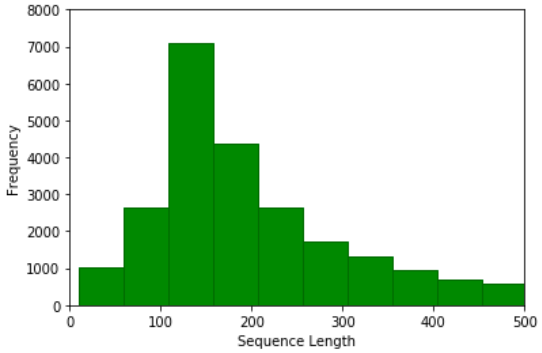

Figure 17: Frequency of review length

2. **`GloVe` embedding:**

   We will use the pre-trained file taken from `https://nlp.stanford.edu/projects/glove/`. The embedding matrix is trained from Wikipedia 2014 + Gigaword 5 (6B tokens, 400K vocab) with file name `glove.6B.zip`.

   There are three different dimensions for us to choose from: `50d`, `100d`, `200d`, and `300d` vectors. The more dimension, the more information we will have but also lead to expensive computation time when training the model. Let's select `50d` to start with.

3. **Tokenize**

   A tokenized list of words will be created from our training data which will be used to convert the text reviews into index numbers which corresponding with its positon in the tokenized word list.

```
9th review in words:
```

```
2
3   this movie is full of references like mad max ii the wild one and
      many others the ladybugs face its a clear reference or tribute
    to peter lorre this movie is a masterpiece well talk much more
    about in the future
4
5 Same review in index:
6
7   [10, 16, 6, 360, 4, 2076, 37, 1154, 2395, 1537, 1, 1313, 27, 2,
    105, 378, 1, 33761, 386, 28, 3, 781, 2887, 40, 3316, 5, 822,
    8900, 10, 16, 6, 3, 989, 69, 733, 72, 49, 41, 7, 1, 692]
```

Listing 17: Example of review in number sequence

4. **Reviews truncating**

   We will apply padding for all reviews in the dataset as mentioned earlier, the maximum sequence length is `230` words.

   When building the TensorGraph, we will feed in our embedding containing all vectors for each word in the tokenized word list with corresponding index, making sure that when TensorFlow doing looking up, it will get the correct vector for specific word at that index.

   Unfortunately, up until now we only have our embedding in the form of dictionary that holds both words and vectors as keys and values when we read the pre-trained file.

   As a result, we have to extract the vector values for all tokenized words and save them into an array to feed into the graph later. The first position our array will be reserved for padding position with zero vectors. Details about coding can be found in the notebook `RNN with GloVe Pre-trained Embedding`.

5. **Create TensorGraph**

   - `Placeholders`: As mentioned in the TensorFlow theory section, in order to feed in data we need to create `placeholder` for values that we want to feed in during training process. Here we need three `placeholders`:
     - Input x: This will be used to feed in our training reviews. Due to large number of traning examples, we will divide training set into smaller batches of size `64`. This and maximum sequence length are also hyper-parameters to be tuned during training.
     - Input y: This is our labels for input training examples. Also comes in batches.
     - Drop-out: This is used for preventing our model from over-fitting by applying probability to drop out 50 percent of hidden units. This is also a hyper-parameter.

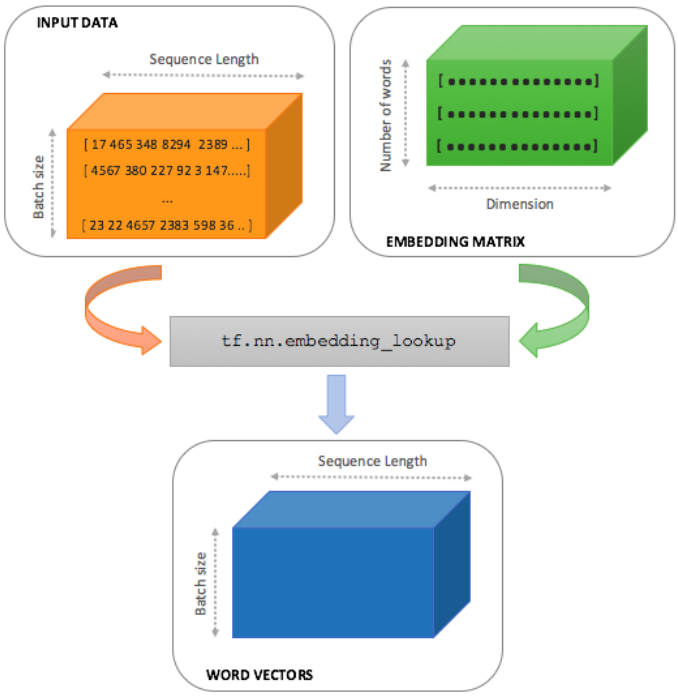

Figure 18: Looking up for vectors of words in TensorFlow

- Embedding layer: In order to make things easier, we will directly use our embedding vectors (`word_vector`) inside the `tf.nn.embedding_lookup` function
- Long Short Term Memory (LSTM) cells: This is used to prevent problem with vanishing gradient. Moreover, in RNN, dependencies is important among words in a sequence, by using LSTM helps preserve the long term dependencies in the network.

  The use of `MultiRNNCell` function is for multiple RNN layers but due to the expensive computation, we will only use single layer RNN.
- Initial state: The initial state will be set to zero.

  Function `dynamic_rnn` can be replaced by `static_rnn` when we have same sequence length for all inputs. However, I decided to use the first function just in case later in the future I want to do follow-up work and extend the model to accepting input with different lengths.

  When using batches, the LSTM output state of previous batch will be provided as the input state for the next batch.
- Labels: to predict the labels of the input, we use sigmoid function as we only have 0 or 1 in final output.
- Cost: Function `sigmoid_cross_entropy_with_logits` is used to calculate the cost.
- Optimization: We used `GradientDescentOptimizer` with decay learning

rate to gradually reduce learning rate through the training process. Starting point is `0.1`.

- Accuracy: This is computed by comparing the predicted labels with the actual labels using function `equal` and `reduce_mean`.

6. **TensorBoard**

   In order to keep track of the training process, we created two scalars for `cost` and `accuracy`. This is a very powerful tool of TensorFlow, by keeping track of the board we know how well our training is going on. It helps us detect if the cost or the accuracy is hovering around some values or changing through time. The board also gives us information about the training time as well, we will know how long it has been running. We can also choose to compare different models that were trained before with our current model.

7. **Train RNN model**

   It's time to train the model. Here we will build single layer RNN with 64 LSTM hidden units. The output activation function will be sigmoid function as we only need '1' (for positive review) or '0' (negative review).

   - Create a new TensorFlow session
   - Define three different writers for our TensorBoard:
     - Main writer:
       To keep track of our cost and accuracy values for every training batch we feed into the network. This is to make sure that our cost will be decreasing whereas the opposite is true for accuracy.
     - Train writer:
       To keep track of the training cost and accuracy of training set after every epoch.
     - Validation writer:
       To keep track of cost and accuracy for validation set after epoch. The reason we need train and validation writer is that by looking at the board, we know when the cost of the training set begins to decrease and the cost of validation set starts to increase to detect over-fitting.
     - Saving models:
       We will save models after every epoch, so if over-fitting happens, we can either stop the training or let it finish and pick the model before the over-fitting starts.

8. **Model selection**

   After training for five hours, 80 epochs, we have about 80 models to select from. It is obvious in the graph below that the final accuracy is quite high.

   Figure 19 and 20 are for cost and accuracy of the training process.

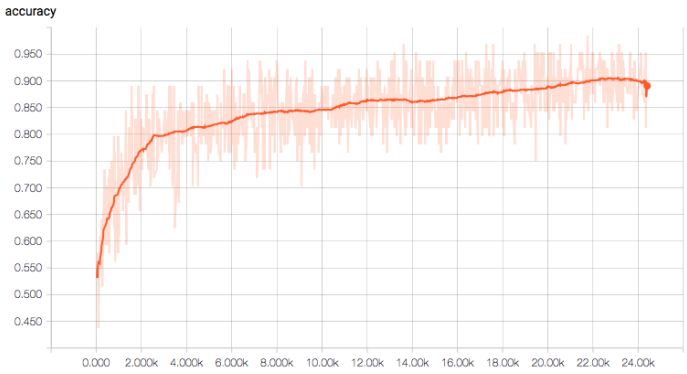

Figure 19: `GloVe` - Accuracy for each training batch after 80 epochs - about 24k iterations

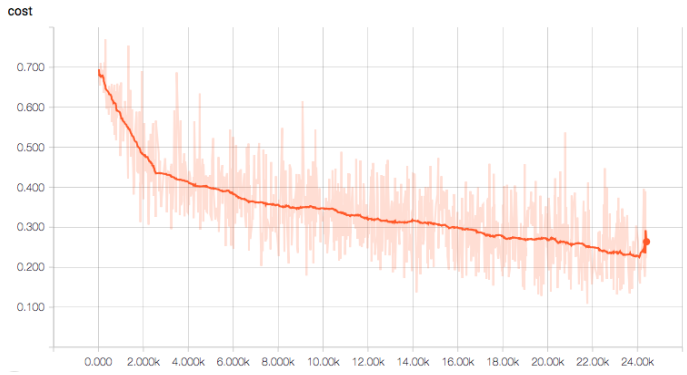

Figure 20: `GloVe` - Cost for each training batch after 80 epochs - about 24k iterations

Unfortunately, we are not sure if we were over-fitting our training data. This is when the graph for training and validation costs come in (figure 21). Normally, in theory, we expect the validation cost to be higher than the training cost but here the result is not the same, the reason for training cost to be higher than validation cost is because we used drop-out during training and no drop-out during validation checking. The model before we over-fitted our data is at epoch 56, after this, cost of validation set outgrows training set. The numbers can be checked in our notebook training log.

- Blue line: training cost
- Purple line: validation cost

9. **Testing model with test set**

   Our accuracy result when testing the selected model 56 with our test set is `83.669%` which is slightly higher than our last model at epoch 80 (`83.568%`).

10. **Testing model with our own review**

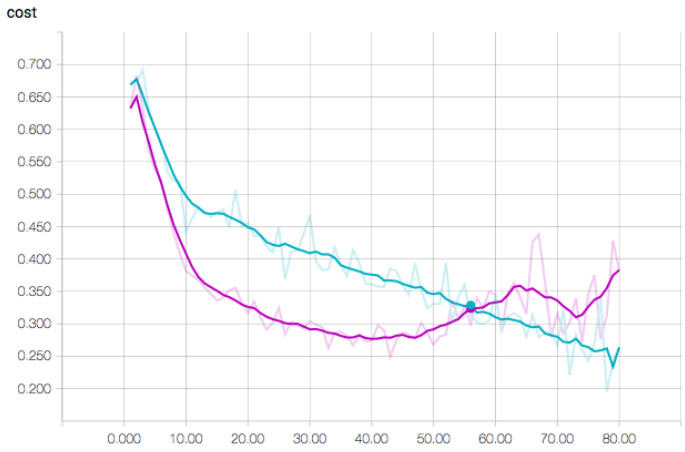

Figure 21: `GloVe` - Accuracy for training and validation set each epoch

In this part, we will test the sentiment of our own review.

Function `format_user_review` is defined to format input string from user to array of words and find corresponding index of each word in our `word_list` and do padding for review with length less than/ or more than 230 words. However, to feed this into the graph, we have to format it into `[batch_size, seq_len]` which is `[64, 230]`. To put it another way, our input array will have 64 rows with our last row is the review, the rest will be zeros.

```
[[  0   0   0 ...,   0   0   0]
 [  0   0   0 ...,   0   0   0]
 [  0   0   0 ...,   0   0   0]
 ...,
 [  0   0   0 ...,   0   0   0]
 [  0   0   0 ...,   0   0   0]
 [  0   0   0 ...,  16   6  75]]
```

Listing 18: Example of input for user input review `"Movie is bad."`

3.3.3 RNN model with `word2vec`

Details about coding can be found in the notebook `RNN with word2vec Gensim Embedding`.

1. **Data preparation**

   In this part, the data cleaning is a little bit different. Instead of doing tokenizing for training data as in RNN using `GloVe`, we will find index for each word in each training review by looking for its corresponding index in the vocabulary `word_list` which is the result from our embedding learning with `Gensim`.

The reason behind is that with `GloVe` pre-trained embedding, there are about 400,000 words making it expensive for us to feed in TensorGraph with that big amount of data knowing that we won't really need that many words for the movie context. Therefore, using tokenized word list from our training data and getting vectors for each words from pre-trained embedding is less heavy for us.

Fortunately, with our embedding learned with `Gensim`, we used our own data from the training and unlabeled data to get a list of vocabularies and vectors, limiting the number of words to only movie context. As a result, feeding this embedding to our TensorGraph is acceptable.

We will do this with the help of function `format_train_review`:

(a) Clean up data by removing special characters (`clean_sentence` function).

(b) Cut any review that is longer than `max_seq_len`, any review less than that will be the same (padding with zero for short review will be done later).

(c) Look up for index of each word in our `word_list`. Any word that is not available in the list will take index of `unk` word (last word in the vocabulary with zero vector values).

After formatting is done, we will do padding for review less than `max_seq_len`. Data will be split into three parts: training, validation and testing set and to batches as in our first RNN model.

For the TensorGraph and training section, it will be the same as in our first model. So we are going to skip this part and go directly to the result of the model after training 80 epochs.

2. **Model selection**

Figure 22 and 23 are for cost and accuracy of the training process. As we can

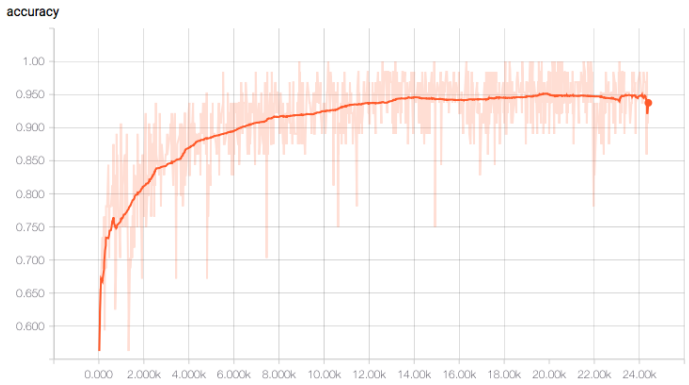

Figure 22: `word2vec` - Accuracy for each training batch after 80 epochs - about 24k iterations

see from figure 24 that cost of validation cost tends to increase from epoch 20

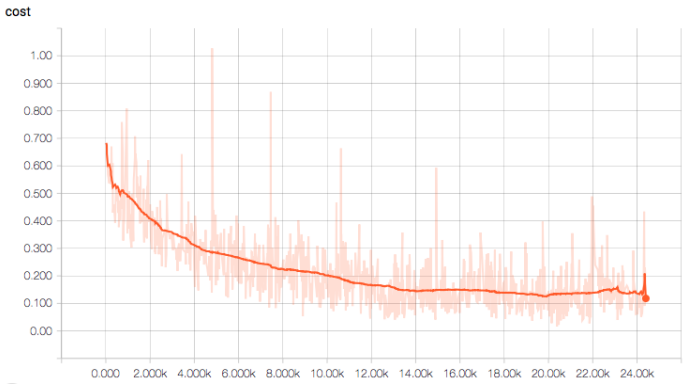

Figure 23: `word2vec` - Cost for each training batch after 80 epochs - about 24k iterations

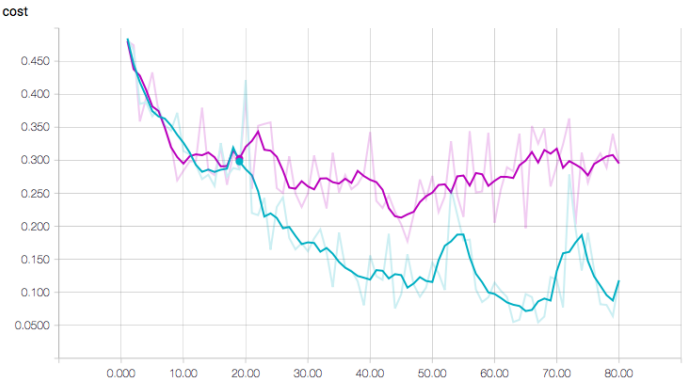

Figure 24: `word2vec` - Accuracy for training and validation set each epoch

whereas that of training cost continues to decrease. This is a sign of over-fitting training data. Hence, we will pick model 19 as our final model.

3. **Testing model with test set**

   Using selected model on the test data, we reached our accuracy at `88.4073%`.

3.3.4 RNN model with `fastText`

For `fastText` the data cleaning is similar to that of `word2vec` so I won't spend time to talk about that. Inside the notebook `RNN with word2vec Gensim Embedding`, I provided details about the coding and results of training, validation and testing. User can also try their own reviews inside the notebook as well.

1. **Model selection**

   Similar to our previous models, we also pick model before over-fitting training data and model 37 is our choice. purple color is our validation cost and blue color is our training cost.

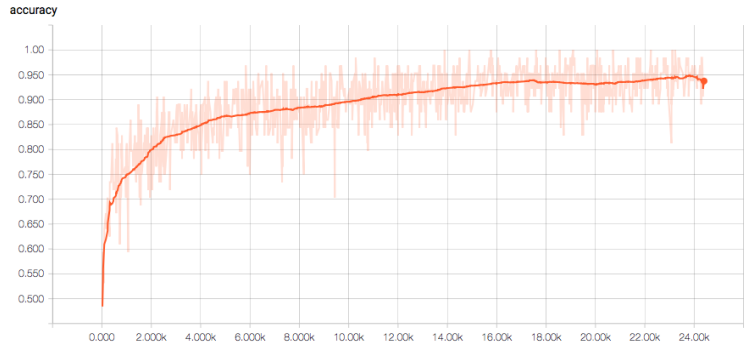

Figure 25: `fastText` - Accuracy for each training batch after 80 epochs - about 24k iterations

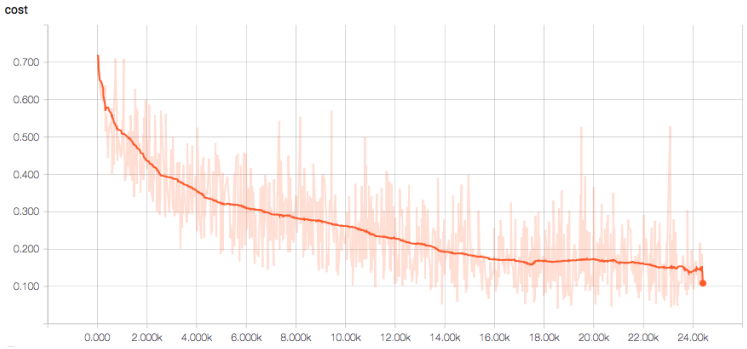

Figure 26: `fastText` - Cost for each training batch after 80 epochs - about 24k iterations

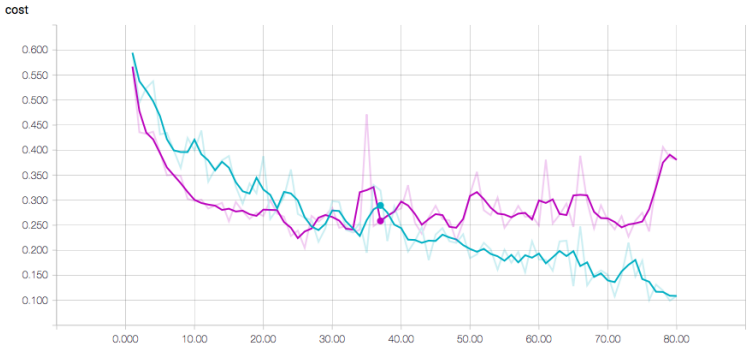

Figure 27: `fastText` - Accuracy for training and validation set each epoch

2. **Testing model with test set**

   Our overall accuracy of the selected on testing data is `86.9960%` which is the second highest accuracy among the three models.

3.3.5 Discussion

In this part we will spend some time to discuss about the results of three models above. Discussion points are as followed:

- Choosing parameters: During the training, I used the same parameters for all of them and I did that on purpose because before choosing these parameters I have tried many different values and even multi-layer RNN but the training time was very expensive and the accuracy was not good, especially when it comes to testing with user-input review.

- Accuracy: Among the three models, `word2vec` has highest accuracy over the test set (Table 7) whereas the opposite is true for pre-trained `GloVe` embeddings. The difference here is not that big but the explanation is because we used pre-trained embedding for `GloVe` which was created from billions of words from Wikipedia and Gigaword.

  However, our problem here is sentiment analysis for movie reviews, hence, all the words in the training examples are mainly about movie context and the pre-trained embedding was learned from different context where words might have different dependencies in sentences.

  With our own `Gensim` embeddings like `word2vec` or `fastText`, we trained the embeddings from our own movie review data, even when the unlabeled training set was not used in the RNN models, it's still about movie reviews where words related to each other in the similar context, making the embedding more closely to our problem.

- Over-fitting: As we can see in previous figures about the trends of cost during training of train set and validation set, the RNN models with pre-trained `GloVe` took more time to detect over-fitting over training data (after epoch 56). `word2vec` and `fastText` showed a sign of being over-fitted right after epoch 19 and 37 respectively, making training time shorter than the first one. When over-fitting was found, we can choose to stop the training and pick the model before over-fitting point.

- Predicting user input review: If users have two reviews as below:

  Negative: `"Content is very boring and this is a waste of time to see it."`

  Positive: `"Movie is about a spy, which is not new subject. Content is very good and this is a great time to see it."`

  The prediction is correct for both of them with all three models. However, if we change the affirmative sentence into negative sentence by adding `not` after the word for verbs or using `don't, doesn't, didn't, won't, etc.` before the verbs, model will not understand this as a negative review but also predict it as positive one. This is a problem with negation.

Table 7: Accuracy of RNN using different embeddings

Pre-trained GloVe	word2vec	fastText
83.669	88.4073	86.9960

For short reviews containing one or two words, models will not understand if this is a good or bad review and will output negative result. This happens because when we fed in training data for our RNN models, we didn't have a large number of reviews with one, two or three words (our average number of words per review is 230 - Figure 13). As a result, the final model tends to work well with longer reviews. Actually, in reality, when people write reviews for something, they usually say their own opinions and explain about the reasons why they have that thoughts, it's not always the case that they just write a short word like `hate, bad, love, etc.` in the review textfield (some websites also set minimum number of words).

In order to make the checking of project details easier, I decided to create 4 different separate Jupyter Notebooks for each of the sub-section in the practical section, even though there are many similarities among the three models in terms of cleaning data, building the TensorGraph, making prediction for user input review.

4 Conclusion

In summary, I have finished the project in terms of theory and practice. Thanks to this project, now I feel more confident in using RNN and word embeddings in NLP. As this is the first time I have done a completed Deep Learning application, and very new to TensorFlow, I believe this is not a perfect application but it reflects my time and efforts spent for it.

Three months ago when I proposed this project, I didn't have much knowledge about words embeddings, RNN or anything about TensorFlow. I have come a long way to reach the position where I am now. Thanks to this project, I have learned a lot of things that I have never known before. The learning process was never easy for me. Before starting with the theory, I spent more than three weeks to attend courses about Deep Learning from Coursera by Andrew Ng. as I needed to understand better about Deep Learning especially Sequence Modeling. For me Natural Language Processing (NLP) is a very new and interesting field, I got inspired a lot when learning about word embeddings and this is a very good starting point for me if I ever want to follow down this path and learn more about NLP.

The most difficult thing in the theory part is to understand the papers written at higher level than I have. A lot of searching on different websites, Youtube videos was done to help me more familiar with this new field. To be honest, I still can't fully understand those papers.

When doing the practical section, I encountered a lot of obstacles because I have never done any RNN models from beginning to end starting with the chosen of dataset, cleaning up the data, learning embeddings, building the model, training it and use it for prediction. Even with the help of TensorFlow framework, things didn't get easier. TensorFlow is a very powerful tool for us but it's also a difficult tool to use. After many times of training failed models, I have realized that everything was all about trying and failing and trying again until we get what we are expecting for. But sometimes, this was very desperate if we don't have help from other people such as Professors and friends. I would like to take this opportunity to thank those who helped me in completing this project.

References

[1] Y. Bengio, R. Ducharme, Vincent, P., and C. Jauvin. 2003. A neural probabilistic language model. Journal of Machine Learning Reseach.

[2] Tomas Mikolov, Wen-tau Yih and Geoffrey Zweig. Linguistic Regularities in Continuous Space Word Representations. In Proceedings of NAACL HLT, 2013.

[3] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781 (2013).

[4] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado and Jeffrey Dean. Distributed Representations of Words and Phrases and their Compositionality. arXiv:1310.4546 (2013).

[5] Amit Mandelbaum, Adi Shalev. Word Embeddings and Their Use In Sentence Classification Tasks. arXiv:1610.08229v1(2017).

[6] Jeffrey Pennington , Richard Socher , Christopher D. Manning. Glove: Global vectors for word representation. In EMNLP (2014).

[7] Armand Joulin, Edouard Grave, Piotr Bojanowski, Tomas Mikolov. Bag of Tricks for Efficient Text Classification. arXiv:1607.01759 (2016)

[8] Piotr Bojanowski, Edouard Grave, Armand Joulin, Tomas Mikolov. Enriching Word Vectors with Subword Information. arXiv:1607.04606 (2017)

[9] Piotr Bojanowski, Edouard Grave, Armand Joulin, Tomas Mikolov. Facebook Research on fastText
`https://research.fb.com/fasttext/`

[10] NSS, Text Classification & Word Representations using FastText (An NLP library by Facebook), 2017.
`https://www.analyticsvidhya.com/blog/2017/07/word-representations-\text-classification-using-fasttext-nlp-facebook/`

[11] Sebastian Ruder, On word embeddings, 2016.
http://ruder.io/word-embeddings-1/index.html

[12] Manjeet Singh, Word Embedding, 2017.
https://medium.com/data-science-group-iitr/
word-embedding-2d05d270b285

[13] Sebastian Raschka, Vahid Mirjalili, Python Machine Learning - Second Edition, 2017.
https://www.packtpub.com/big-data-and-business-intelligence/
python-\machine-learning-second-edition

[14] TensorFlow Introduction.
https://www.tensorflow.org/programmers_guide/low_level_intro#
tensor_values

[15] TensorFlow API Documentation - Placeholder.
https://www.tensorflow.org/api_docs/python/tf/placeholder

[16] Tensor Variables.
https://www.tensorflow.org/programmers_guide/variables

[17] TensorFlow Graphs and Sessions.
https://www.tensorflow.org/programmers_guide/graphs

[18] TensorFlow Save and Restore.
https://www.tensorflow.org/programmers_guide/saved_model

[19] TensorFlow Embeddings.
https://www.tensorflow.org/programmers_guide/embedding

[20] TensorFlow Embeddings - Word2Vec.
https://www.tensorflow.org/tutorials/word2vec

[21] Bag of Words Meets Bags of Popcorn.
https://www.kaggle.com/c/word2vec-nlp-tutorial/data

[22] Predicting Movie Review Sentiment with TensorFlow and TensorBoard.
https://medium.com/@Currie32/predicting-movie-review-sentiment-\
with-tensorflow-and-tensorboard-53bf16af0acf

[23] Perform sentiment analysis with LSTMs, using TensorFlow.
https://www.oreilly.com/learning/perform-sentiment-analysis-\
with-lstms-using-tensorflow

[24] Word2Vec and FastText Word Embedding with Gensim.
https://towardsdatascience.com/word-embedding-with-word2vec-and-fasttext-\
a209c1d3e12c