2019-1 Deep Learning Homework #3

Fabian Tan Hann Shen (ID 20185112)

May 6, 2019

1.

(a) **Porter Stemming**
LSTM is used on the imdb datasets to perform sentiment analysis. The imdb datasets comprise of training and testing sets with both of the positive and negative reviews. Each of the reviews' folders has 12,500 reviews. On the following, the model is trained with LSTM using the preprocessing techniques, known as Porter's stemming algorithm. It is a process of removing the commoner morphological and inflexional endings from words in English. The main gist is to normalize the tokenized word. Stemming's job is to strip the suffixes of a word into its base word, giving the features for the classification. Fig. 1 to Fig.4 shows the model accuracy loss plot of the training data. With the trained model, it is then applied with the testing data to evaluate the accuracy.

However, the accuracy score has been slightly degraded after using the Porter's stemming algorithm. The accuracy score on the testing set with Porter's stemming algorithms shows 83.53% whereas the accuracy without stemming algorithms shows 85.15%, as shown in Table 1.

The result of the LSTM performing on the imdb sentiment analysis with the Porter stemming has lower accuracy compared to that of without stemming algorithm.

Table 1: Accuracy of the LSTM on the IMDB Reviews

Porter Stemming Algo	Accuracy	ROC AUC score
Before	85.15%	0.94
After	83.15%	0.92

(b) (c)

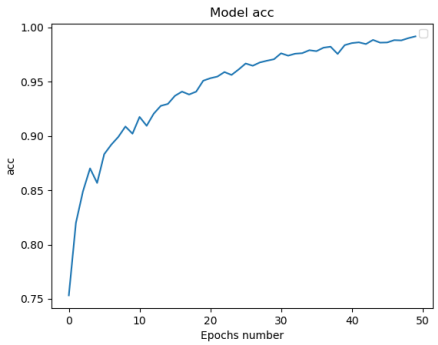

Figure 1: shows the accuracy plot for the training data(without Porter stemming)

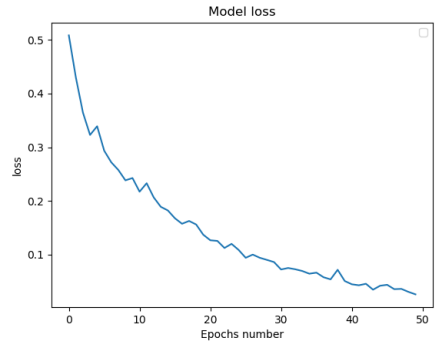

Figure 2: shows the loss plot for the training data(without Porter stemming)

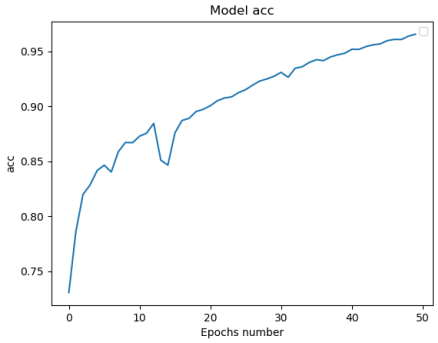

Figure 3: shows the accuracy plot for the training data(with Porter Stemming)

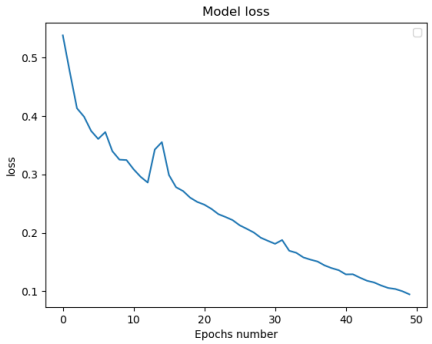

Figure 4: shows the loss plot for the training data(with Porter Stemming)

2.

In the following Word2Vec experiment, Gensim was used to handle large text collections and the dataset used was OpinRank dataset. OpinRank dataset consists of full reviews for cars and hotels which aggregated from the well-known application - Tripadvisor. Features extracted from car reviews are dates, author names, favorites and full textual reviews are extracted whereas hotel reviews are date, review title and also full reviews.
A list of tokenized sentences (reviews_data.txt) will be read and passed into the model as the input data. Each of the word from the review sentences has been converted into a list of lowercase tokens via the simple_preprocess function. These processed tokens will be turned into a series of words and internally created a unique vocabulary. Gensim trains the model to predict the targeted word based on the neighboring words for 10 epochs, and the throughout the training process, the it gradually learns and update the weight embeddings.
As seen in Fig. 5, it returns top-k similar words for the "shocked" word from the most_similar call function. In addition, we can compute the similarity score for two words as illustrated in Fig. 6. It uses the cosine similarity - a measure of similarity between two non-zero vectors, measuring the cosine of the angle between two vectors. The smaller the angle, more likely the chances they are belong to the same class. Lastly, we can use Word2Vec to filter the odd out of the lists as shown in Fig. 7. For instance, bed belongs to the 'furniture' class and does not belong to any of the 'sport' class.

3.

In Continuous Bag of Words (CBOW), instead of processing one word, a batch of (input, output) tuples with a user-defined numWords(hyperparameter) in a single sliding window are fed into the model. The gist of CBOW is to average the embedding vectors of all the input words as the input to the model.

```
In [24]: w1 = ["shocked"]
         model.wv.most_similar (positive=w1)

Out[24]: [('horrified', 0.8087976574897766),
          ('amazed', 0.7938255667686462),
          ('astonished', 0.7776336073875427),
          ('stunned', 0.7753317952156067),
          ('appalled', 0.7656266093254089),
          ('dismayed', 0.7639718651771545),
          ('astounded', 0.7184687852859497),
          ('suprised', 0.7047489881515503),
          ('surprised', 0.7014123797416687),
          ('mortified', 0.687908947467804)]
```

Figure 5: shows the most similar word for top-10 words

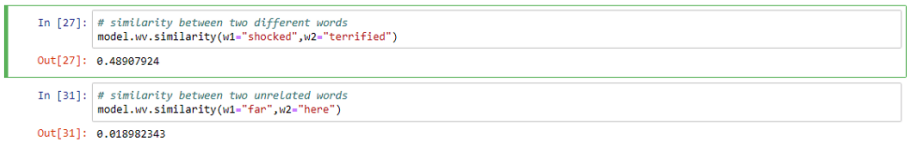

Figure 6: shows the similarity score for two words

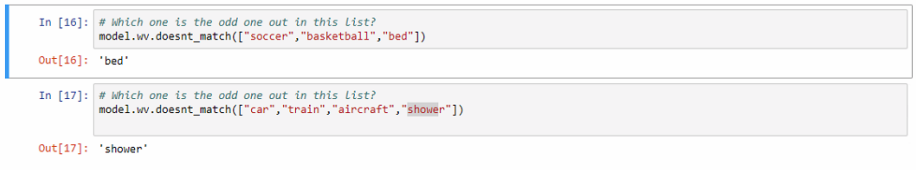

Figure 7: shows the odd items out of the pairing list

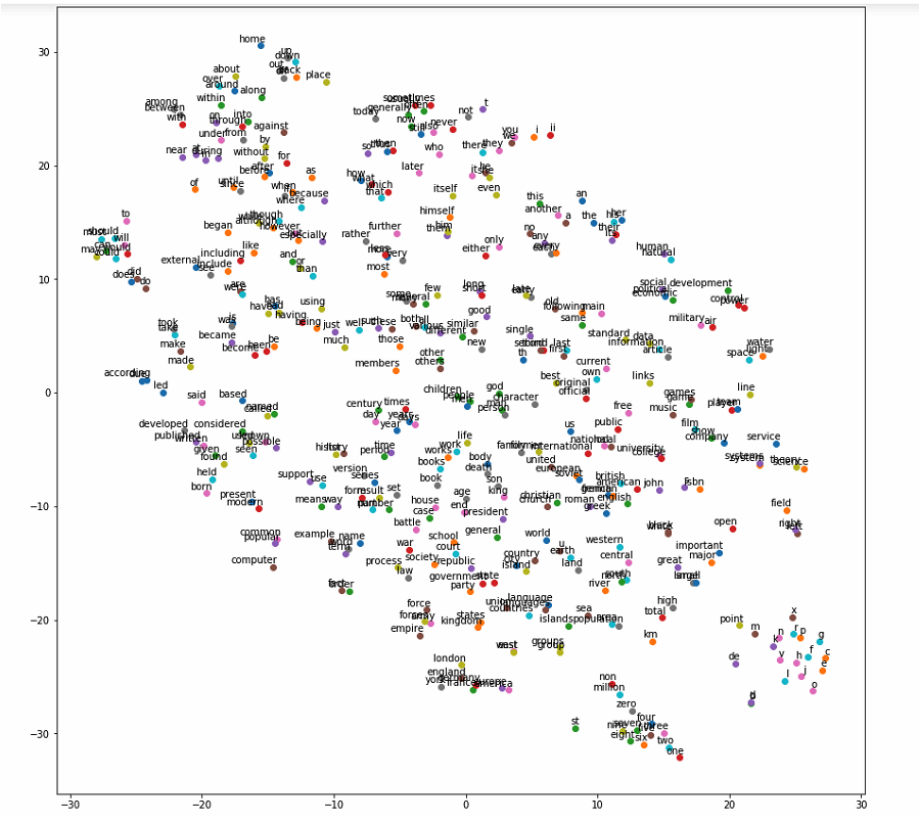

Figure 8: shows the embedding plots for CBOW

In the tutorial, it creates a skip window of 1, which spans over the words from the left to right side of the targeted word, and also number of skips of 2. For instance, if the skip window = 1, then it would capture the immediate neighbor to the left and to the right of the given targeted word and train the model to make predictions. The window is moved at every timestep. Moreover, the embedding size of the model is set as 128. The words from every slide windows that stored as embeddings will be averaged using the tensorflow built-in reduce mean function.
Model loss is minimized and updated by performing back-propagation using Adagrad optimizer, and iterated for 100000 steps. Lastly, the similarity of the words uses the cosine distance to compute. In Fig. 8, it shows the word embeddings in a 2-dimensional plot. Words that share similar embeddings are stayed closed to each other in the graph.

4.

Skip-gram
Skip-gram will have a window looking back and forth for the related word based on the targeted instance(single input), which is different from the CBOW above. This window is called skip-window, and they will form a tuple comprises of input and output. The context are the immediate neighbours of the target and they will be then converted into vector representation. Next, train the model to gradually learn the vector representation of the targeted word. The closeness of the target word and the context word is computed via the dot product between input-embedding and the output embedding.
In the tutorial, the dataset used is the same as the CBOW and the skip window and the number of skips are set as 2 and 4 respectively. Embedding size is configured as 128. Next, train the model with the aggregated tuples without the labeled data. The loss of the model is minimized and updated by performing back-propagation using Adagrad optimizer, and iterated for about 100000 steps. Finally, in order to compute the similarity of the word, it uses the cosine distance based on the learnt embedding.
It returned the top-k word for the targeted word based on the similarity score. As seen in the result, the similarity score was improved gradually throughout every interval of training steps. Fig. 9, it shows the word embeddings in a 2-dimensional plot. Words that have the similar vector representations are placed relatively nearer to each other in the graph.

5.

GloVe(Global Vector) is considered as one of the unsupervised algorithms to obtain vector representation. It captures the counts of the overall statistics of how often these word appear. It is trained with global word co-occurence statistics from the corpus, which tabulates how frequently words co-occur with one another.
In the tutorial, the dataset used is the same as the CBOW. It employs the global statistics of the corpus to the model. The embedding size of the experiment is 192 and pick a valid size of 16 to evaluate the similarity.

The loss function is given as

$$J = \sum_{i=1}^{V} \sum_{j=1}^{V} f(X_{ij})(w_i^T w_j + b_i + b_j - log(1 + X_{ij}))^2$$

where V is the vocabulary size, w_i is the target word, w_j is the context word and b_i is the bias for word w_i.

AdaGrad optimizer is used to optimize the softmax weights and the embedding loss. After training against the model, the target words stay closer will

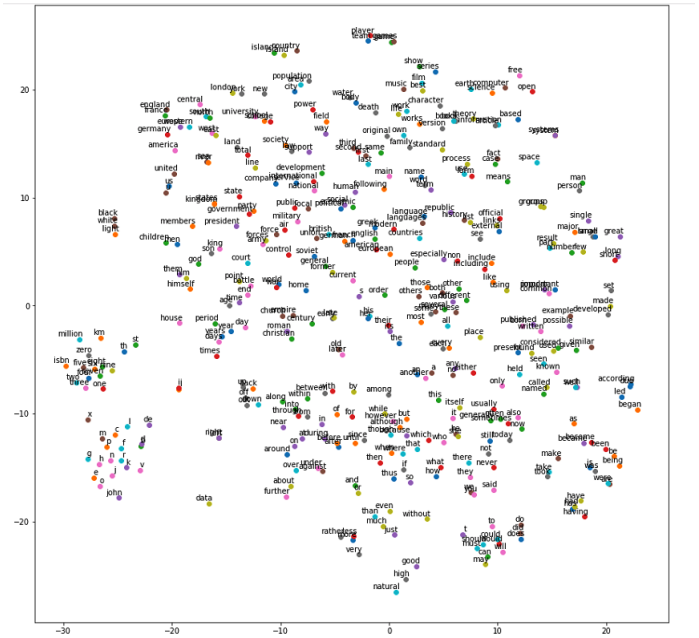

Figure 9: shows the embedding plots for skip-gram

have similar embeddings/representation, and hence the result in Fig. 10.
The similarity score is computed using the cosine distance, and returns the top-k similar word based on the targeted word.

6.

POS shorts for Part-Of-Speech, is a grammatical category of a word, which labels each of the word with a POS tag such as noun, verb, adj, preposition, pronoun, adverb, conjunction or interjection.
In the POS tutorial, NLTK library is utilized on training the model and output the possible POS tags for individual word. The dataset used here was CONLL-2000 which obtained from the nltk. Firstly, we extracted the train and testing dataset and train them by mapping the POS tag to unique ID in the form of dictionary. The network finally outputted a probability distribution over the possible tags with the softmax layer.

7.

Sequence-to-sequence network embeds two main networks, which is the encoder - encodes the information of the input into a vector, and decoder network - unfolds or decodes the vector into another sequence. Furthermore, it employs

```
Nearest to troops: solve, firing, nsdap, silla, resentful, boots, cavea
ts, triumvirate,
Nearest to older: ahead, bornu, maine, seizures, rodeo, musketeers, pau
sanias, distancing,
Nearest to brother: big, his, father, works, work, own, its, their,
Nearest to powers: gulf, bran, mael, marginalized, valkyrie, montreal,
laches, newsstand,
Nearest to bbc: news, radio, top, side, on, earth, convention, based,
Nearest to running: attar, shuts, heck, praising, catania, sustainable,
wah, seems,
Nearest to articles: finegold, goblin, interstellar, drogheda, migratin
g, xa, anastasia, frankfurter,
```

Figure 10: shows the similar words for the targeted word after 100000 timesteps

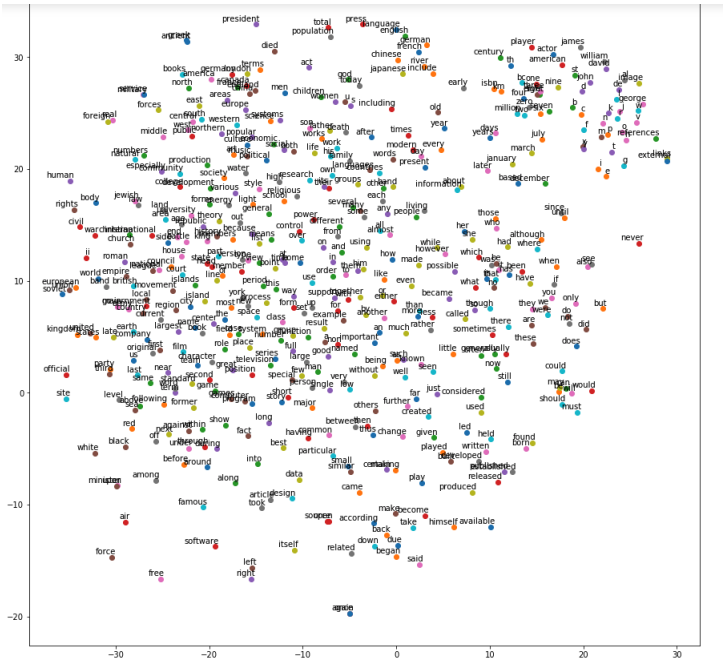

Figure 11: shows the embedding plots for GloVe

```
Out[17]: (<__main__.Lang at 0x266b3176518>,
          <__main__.Lang at 0x266b3176ef0>,
          [['go .', 'va !'],
           ['run !', 'cours !'],
           ['run !', 'courez !'],
           ['wow !', 'ca alors !'],
           ['fire !', 'au feu !'],
           ['help !', 'a l aide !'],
           ['jump .', 'saute .'],
           ['stop !', 'ca suffit !'],
           ['stop !', 'stop !'],
           ['stop !', 'arrete toi !'],
           ['wait !', 'attends !'],
           ['wait !', 'attendez !'],
           ['i see .', 'je comprends .'],
           ['i try .', 'j essaye .'],
           ['i won !', 'j ai gagne !'],
           ['i won !', 'je l ai emporte !']
```

Figure 12: shows the preprocessed words of the language pairs

```
Reading lines...
Read 135842 sentence pairs
Trimmed to 10599 sentence pairs
Counting words...
Counted words:
fra 4345
eng 2803
['nous sommes plutot occupes ici .', 'we re kind of busy here .']
```

Figure 13: shows the preprocessed words of the language pairs

attention mechanism to focus decoder on particular range of the input. First, we will get the english-to-france dataset on the web, and preprocess them including punctuation trimming, ASCII conversion and making them case-insensitive, and lastly pair them according to the language, as illustrated in Fig. 12 and Fig. 13.

The encoder of the seq2seq network will generate a vector and a hidden state, and deploys the hidden state for the subsequent input word. Internal network uses gated recurrent network (GRU). Whereas decoder of the seq2seq network will generate a sequence of words by consuming the encoded output vector to produce the translation part. In addition to that, we have the attention layer which create a weighted combination from the encoded output vectors.
During the model evaluation, the decoder prediction is fed back to itself in every interval of timestep. It predicts every word added and eventually stops predicting when EOS token is retrieved. The seq2seq nets uses 256 hidden nodes and one GRU layer, and iterated for 75000 iterations. As shown in the Fig. 14, the model loss is minimized over the iterations.

```
hidden_size = 256
encoder1 = EncoderRNN(input_lang.n_words, hidden_size).to(device)
attn_decoder1 = AttnDecoderRNN(hidden_size, output_lang.n_words, dropout_p=0.1).to(device)

trainIters(encoder1, attn_decoder1, 75000, print_every=5000)
```

```
4m 22s (- 61m 12s) (5000 6%) 2.8303
8m 39s (- 56m 15s) (10000 13%) 2.2774
12m 58s (- 51m 55s) (15000 20%) 1.9446
17m 18s (- 47m 35s) (20000 26%) 1.7321
21m 37s (- 43m 15s) (25000 33%) 1.5115
25m 58s (- 38m 58s) (30000 40%) 1.3595
30m 20s (- 34m 40s) (35000 46%) 1.2128
34m 40s (- 30m 20s) (40000 53%) 1.0762
39m 3s (- 26m 2s) (45000 60%) 1.0058
43m 26s (- 21m 43s) (50000 66%) 0.8989
47m 50s (- 17m 23s) (55000 73%) 0.7941
52m 13s (- 13m 3s) (60000 80%) 0.7136
```

Figure 14: shows the training loss of the seq2seq nets