**Sentiment Analysis of Text using Deep Learning**

CECS 590 Project Report - Fall 2017

Presenters:

Sumeet Sunil Gadkari CSULB ID - 016115942  
Deepesh Kumar Singh CSULB ID - 016149963

**Abstract:**

Convolutional Neural Networks (CNNs) have recently been shown to achieve impressive results on the practically important task of sentence categorization. We propose to explore CNN in NLP domain to classify text sentences. This method will provide a better alternative to existing techniques of text classification namely SVM and Logistic Regression. We would be using Movie Review dataset. Our project would output a single probability value indicating how good or bad a review is for each review of a movie. Probability close to one indicates review for a movie is very good and close to zero indicates the review is not good. This report describes the step we followed in dataset preparation, Training and Testing.

**Introduction** :

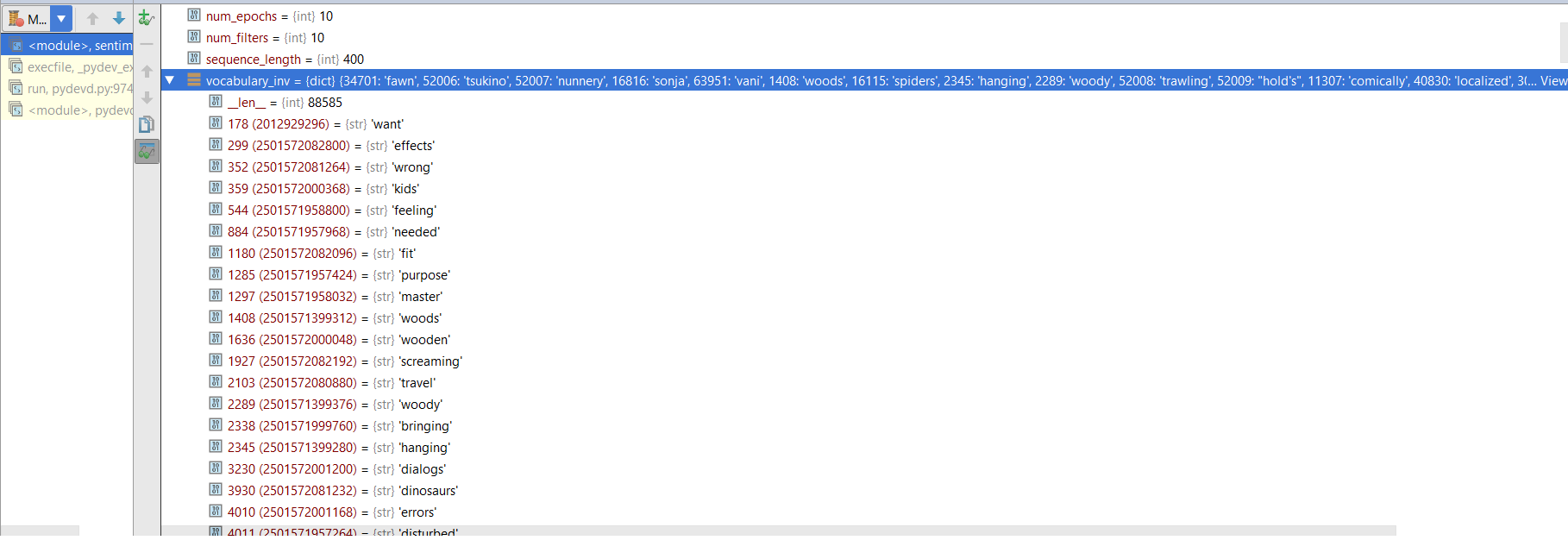
In recent years we have seen amazing results in computer vision and speech recognition using Deep learning models. Inside natural language processing (NLP) handling, a great part of the work with deep learning strategies have included learning word vector representation through neural models and performing composition over the learned word vectors for various classification problems. In word vectors, words are converted into lower dimensional vector representation to act as semantic features of words in its dimensions. Words with semantically same meaning are considered to be close in this lower dimensional dense representation. CNN model uses layers with convolving filters that are applied to local features. CNN was originally created for computer vision. But over the period of time it is now getting adopted in NLP and shown good results in traditional NLP task like semantic parsing etc. In the present work, we would be using word vectors as an input to train a simple CNN with one layer of convolution in an unsupervised neural language model. The word vectors are generated using word2vec open source lib trained on 100 billion words of Google News. At first we keep the word vectors as static and got the other parameters of the model. We see that in spite of little tuning of hyperparameters, our model achieves good results. It suggested that word vectors are good for using in feature extraction in various classification tasks. Task specific vector learning through fine tuning give us a further improvement. In our model we would be using pre-trained and context based vectors. Our work is based on the research paper by Kim, Y. given in first paper of reference section.

**Implementation Details and Findings:**

**1) Dataset and Data Preprocessing:**

Our project was implemented using movie review dataset from IMDB. Initially, we had proposed to use dataset from rotten tomatoes. However, that dataset was very small and had about 10k movie reviews. In contrast, the IMDB dataset which we ended using had 50k rows. Keras provides this dataset as a part of its API. This provided us with substantial reviews to build vocabulary of words. After loading the dataset, we were tasked with cleaning it. Movie reviews have lot of special symbols, stop words. These were removed using regular expression in python. Each movie review has a different length and must be either reduced or padded so as to form consistent word vector. We found that each review of length around 500 words gives decent amount of words per review to work with. Reviews with length less than 500 words were padded with blank space character. Finally, we had to divide dataset into training and testing set. We used a split of 80% for training and 20% for testing. Instead of creating separate dataset for validation, we decided to use the technique of cross-validation.

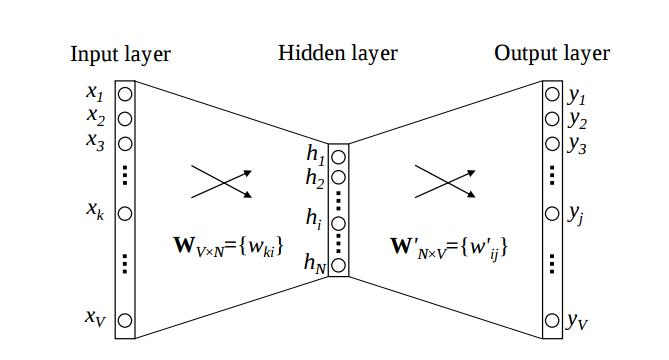




**2) Word Embeddings / Embedding Layer:**

Machine learning model works on numerical data. The major task in our project was to convert each text review into numerical representation. We created vocabulary of words by considering full corpus of movie review dataset. Vocabulary built by us had around 87k unique words. Two approaches were used to create word embeddings - count based and context based. Frequency based approach calculates the number of times the words is present in a review. However, the vector formed using this method was sparse and gave accuracy around 60%. When we were using one hot encoding we were getting a very large vectors representation with lot of ‘one’ and ‘zeros’. Also, when we talk about word and the context in which the words are used then one hot encoding won’t be a good choice of method for deep learning models. In one-hot encoding, words have no natural notion of similarity. Ideally, we would want dot products (since we are dealing with vectors) of synonym / similar words to be close to one.

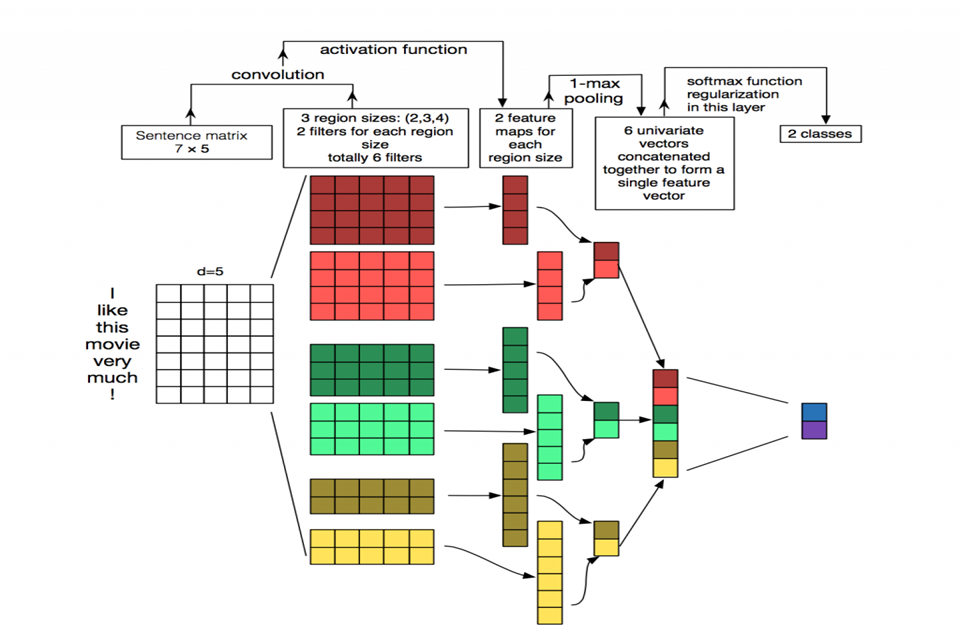
We then shifted our focus to word embeddings using word2vec. Word2vec approach is highly suggested to use while solving sentiment analysis problems. Word2vec is a two layer neural network pre-trained (as shown in above figure) on 100 billion words of Google news. This neural network learns weight between input layer and hidden layer using training data of words. The embeddings are values denoted obtained by multiplying input with the learned weights. Word2vec output is a vector space, typically of several hundred dimensions, with each unique word in the corpus being assigned a corresponding vector in the space. Word vectors are positioned in the vector space such that words that share common contexts in the corpus are located in close proximity to one another in the space.



Above figure shows word2vec model we would be using in our deep learning architecture. Number of neurons in hidden layer represents the embedding dimension. We used word2vec implemented in Gensim library. At first, small random initialization of word vectors is used. Our predictive model learns the vectors by minimizing the loss function. In Word2vec, this happens with a feed-forward neural network and optimization techniques such as Stochastic gradient descent. For training word2vec model, we have used embedding dimension of 50 and maximum sentence length of 400. If the length of the sentences is less than 400 we would pad it and if the length of the sentences is greater than 400 then we prune it. Embedding dimension of 50 will enable the word2vec model to create feature vector of length 50. At the output layer we would get matrix of 400 rows and 50 column where column would represents feature vectors and each row represents each word. We have used continuous bag of words method in word2vec with context window size of 10. All these parameters are provided during training.

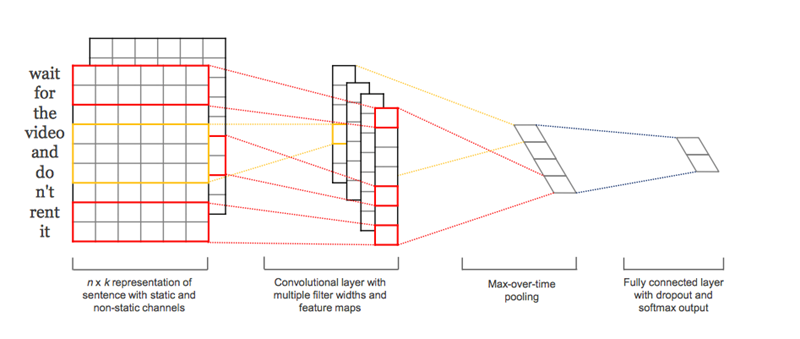
In summary, converting words into vectors, which deep learning algorithms can ingest and process, helps to formulate a much better understanding of natural language.

**3) Convolutional Layer:**

****

Our model uses CNN for sentiment analysis instead of shallow machine learning models. Idea behind CNN is to apply multiple features to input vector and extract key features from it. The embedding layer used above gives one dimensional vector of length 50 for every word. And our input sentence has 400 words. Thus for every sentence we get a matrix of dimension 50\*400. This matrix is flatten into one dimension and is given as input to convolutional layer. When using CNN on images, we apply filters to small pixel regions but in this case due sparse data we have applied filter to full row and stride the filter forward. Keras provides with 1-D convolution for different filters. We tested with various filters of size between 1 to 8 and observed that filter with size 3 and 8 extracted good features. Below figure depict the architecture used for extracting features using filter.

After convolutional layer, we have applied 1-D max pooling to extract important features from the vector. One feature was extracted from every 2 rows and it is followed for every feature vector extracted from convolutional filter. In max-pooling we extracted maximum value present among the vectors obtained from convolutional step. These extracted values acts as input to fully connected layer.

****

To prevent overfitting to training data, we considered applying regularization to the network randomly. With this approach activation of many neurons is set zero thereby reducing their contribution to zero. We found that regularization adds value to the model by decreasing the weights of learned model.

Above architecture shows our entire model. We have embedding layer which denotes words in numerical format with each word having a unique value in feature space. This followed by convolutional layer and max-pooling which gives a features for training. Then we have used one fully connected layer which uses backpropagation algorithm to learn weights of neural network and then followed by output layer which gives us the probability of goodness for every review.

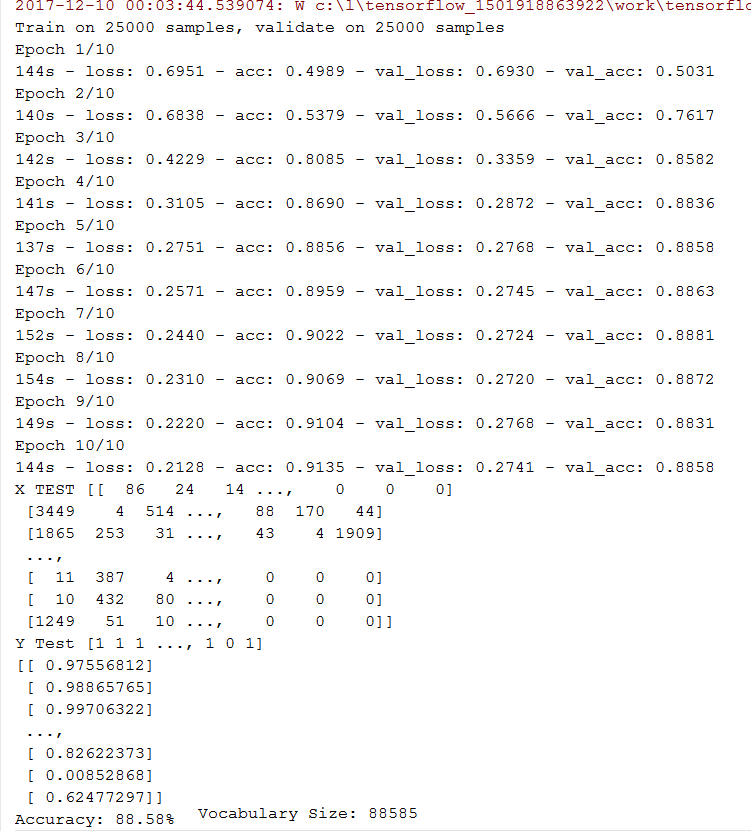
**4) Fully Connected Layer:**

One fully connected layer was used in our model to map extracted features to sentiment. We tested model with 10, 15, 40, 50, 70 neurons in hidden layer. But with 50 neurons in hidden layer the computation time required was very less and accuracy was beyond 82%. Activation used is relu for hidden layer. At output we would get a single value indicating the probability of how good/bad a review is. Probability is given by sigmoid function which outputs value between 0 to 1.

**5) Training and Testing:**

We initially started with 5 epochs followed by 10 and 15. We then finalized with 10 epochs. We divided training data into batches of 64. There are various optimizers available for backpropagation but we found out adam optimizer to be most efficient. Loss function is used to check out error for the model. Cross entropy is one of them and is used in our algorithm. Training was done on 80% of data. We opted for cross-validation to tune hyperparameters. After training graph was finished we got a neural network model. We tested remaining data against this model.Labels for test data is either 0 and 1. We got an accuracy between 85% to 90%. During creation of word embeddings we ran program once for every hyperparameter. Once we have created word2vec, this neural network is was saved to memory instead of training it every time for same parameters

**Results And Analysis**

****

Above figure shows the training and testing process. There are 10 epochs used and as we can see that as the number of training epochs increase the loss parameter decreases. Accuracy initially is less since the model is yet to learn appropriate weights. After few epochs the network learns the weight and hence accuracy becomes almost close to constant. Y Test denotes the label for corresponding input test feature. ‘1’ indicates the review is positive while ‘0’ indicates a negative review. For the first ‘1’ in test data we got 0.97 probability as output which indicates a very positive review. Review before last is negative, and hence we got 0.0085 as output. This indeed conveys the accuracy of our model because the network correctly identified both positive and negative reviews. Moreover, a review which is slightly positive is given the probability of 0.624!!

**Conclusion and Future work:**

In the present work we have depicted a progression of tests with convolutional neural systems based over word2vec. In spite of small tuning of hyperparameters, a straightforward CNN with one layer of convolution performs surprisingly well. Our outcomes add to the established. Our results add to the well-established evidence that unsupervised pre-training of word vectors is an important ingredient in deep learning for NLP. For future work, we would try to implement Recurrent Neural Network along with improving the results of Convolutional neural network.

**References** -

[1] Kim, Y. (n.d.). Convolutional Neural Networks for Sentence Classification. Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP).

[2] Kalchbrenner, N., Grefenstette, E., & Blunsom, P. (n.d.). A Convolutional Neural Network for Modelling Sentences. Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers).

[3] Wang, P., Xu, J., Xu, B., Liu, C. L., Zhang, H., Wang, F., & Hao, H. (2015). Semantic Clustering and Convolutional Neural Network for Short Text Categorization. In Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Vol. 2, pp. 352- 357).

**APPENDIX:**

**Code:**

**Sentiment\_cnn.py**

===================

*"""*

*Train convolutional network for sentiment analysis.*

*"""*

**import** numpy **as** np

**import** data\_helpers

**from** w2v **import** trainWord2vec

**from** keras.models **import** Sequential, Model

**from** keras.layers **import** Dense, Dropout, Flatten, Input, MaxPooling1D, Convolution1D, Embedding

**from** keras.layers.merge **import** Concatenate

**from** keras.datasets **import** imdb

**from** keras.preprocessing **import** sequence

np.random.seed(0)

dataSource = **"kerasDataSet"**

embeddingDim = 50

filterSizes = (3, 8)

numFilters = 10

dropoutProb = (0.5, 0.8)

hiddenDims = 50

batchSize = 64

numEpochs = 10

sequenceLength = 400

maxWords = 5000

minWordCount = 1

contextWindow = 10

**def** load\_data(dataSource):

**assert** dataSource **in** [**"kerasDataSet"**, **"local\_dir"**], **"Unknown data source"**

**if** dataSource == **"kerasDataSet"**:

(x\_train, y\_train), (x\_test, y\_test) = imdb.load\_data(num\_words=maxWords, start\_char=**None**,

oov\_char=**None**, index\_from=**None**)

x\_train = sequence.pad\_sequences(x\_train, maxlen=sequenceLength, padding=**"post"**, truncating=**"post"**)

x\_test = sequence.pad\_sequences(x\_test, maxlen=sequenceLength, padding=**"post"**, truncating=**"post"**)

vocabulary = imdb.get\_word\_index()

vocabulary\_inv = dict((v, k) **for** k, v **in** vocabulary.items())

vocabulary\_inv[0] = **"<PAD/>"**

**else**:

x, y, vocabulary, vocabulary\_inv\_list = data\_helpers.load\_data()

vocabulary\_inv = {key: value **for** key, value **in** enumerate(vocabulary\_inv\_list)}

y = y.argmax(axis=1)

*# Shuffle data*

shuffle\_indices = np.random.permutation(np.arange(len(y)))

x = x[shuffle\_indices]

y = y[shuffle\_indices]

train\_len = int(len(x) \* 0.9)

x\_train = x[:train\_len]

y\_train = y[:train\_len]

x\_test = x[train\_len:]

y\_test = y[train\_len:]

**return** x\_train, y\_train, x\_test, y\_test, vocabulary\_inv

print(**"Loading Data"**)

x\_train, y\_train, x\_test, y\_test, vocabulary\_inv = load\_data(dataSource)

**if** sequenceLength != x\_test.shape[1]:

print(**"Adjusting sequence length for actual size"**)

sequenceLength = x\_test.shape[1]

print(**"x\_train shape:"**, x\_train.shape)

print(**"x\_test shape:"**, x\_test.shape)

print(**"Vocabulary Size: {:d}"**.format(len(vocabulary\_inv)))

embedding\_weights = trainWord2vec(np.vstack((x\_train, x\_test)), vocabulary\_inv, num\_features=embeddingDim,

minWordCount=minWordCount, context=contextWindow)

input\_shape = (sequenceLength,)

model\_input = Input(shape=input\_shape)

z = Embedding(len(vocabulary\_inv), embeddingDim, input\_length=sequenceLength, name=**"embedding"**)(model\_input)

z = Dropout(dropoutProb[0])(z)

conv\_blocks = []

**for** sz **in** filterSizes:

conv = Convolution1D(filters=numFilters,

kernel\_size=sz,

padding=**"valid"**,

activation=**"relu"**,

strides=1)(z)

conv = MaxPooling1D(pool\_size=2)(conv)

conv = Flatten()(conv)

conv\_blocks.append(conv)

z = Concatenate()(conv\_blocks) **if** len(conv\_blocks) > 1 **else** conv\_blocks[0]

z = Dropout(dropoutProb[1])(z)

z = Dense(hiddenDims, activation=**"relu"**)(z)

model\_output = Dense(1, activation=**"sigmoid"**)(z)

model = Model(model\_input, model\_output)

model.compile(loss=**"binary\_crossentropy"**, optimizer=**"adam"**, metrics=[**"accuracy"**])

weights = np.array([v **for** v **in** embedding\_weights.values()])

print(**"Initializing embedding layer with word2vec weights, shape"**, weights.shape)

embedding\_layer = model.get\_layer(**"embedding"**)

embedding\_layer.set\_weights([weights])

model.fit(x\_train, y\_train, batchSize=batchSize, epochs=numEpochs,

validation\_data=(x\_test, y\_test), verbose=2)

model.save(**'my\_model.h5'**)

**del** model

model = load\_model(**'my\_model.h5'**)

print(**"X TEST"**, x\_test)

print(**"Y Test"**, y\_test)

test2 = model.predict(x\_test)

print(test2)

scores = model.evaluate(x\_test, y\_test, verbose=0)

print(**"Accuracy: %.2f%%"** % (scores[1]\*100))

**W2v.py**

=========

**from** \_\_future\_\_ **import** print\_function

**from** gensim.models **import** word2vec

**from** os.path **import** join, exists, split

**import** os

**import** numpy **as** np

**def** trainWord2vec(sentence\_matrix, vocabulary\_inv,

num\_features=300, min\_word\_count=1, context=10):

modelDir = **'models'**

modelName = **"{:d}features\_{:d}minwords\_{:d}context"**.format(num\_features, min\_word\_count, context)

modelName = join(modelDir, modelName)

**if** exists(modelName):

embeddingModel = word2vec.Word2Vec.load(modelName)

print(**'Load Word2Vec model for CNN \'%s\''** % split(modelName)[-1])

**else**:

num\_workers = 2 *# Number of threads to run in parallel*

downsampling = 1e-3 *# Downsample setting for frequent words*

print(**'Training Word2Vec model...'**)

sentences = [[vocabulary\_inv[w] **for** w **in** s] **for** s **in** sentence\_matrix]

embeddingModel = word2vec.Word2Vec(sentences, workers=num\_workers,

size=num\_features, min\_count=min\_word\_count,

window=context, sample=downsampling)

embeddingModel.init\_sims(replace=**True**)

**if not** exists(modelDir):

os.mkdir(modelDir)

print(**'Saving Word2Vec model for later backpropogation use \'%s\''** % split(modelName)[-1])

embeddingModel.save(modelName)

embeddingWeights = {key: embeddingModel[word] **if** word **in** embeddingModel **else**

np.random.uniform(-0.25, 0.25, embeddingModel.vector\_size)

**for** key, word **in** vocabulary\_inv.items()}

**return** embeddingWeights

**if** \_\_name\_\_ == **'\_\_main\_\_'**:

**import** data\_helpers

print(**"Loading data..."**)

x, \_, \_, vocabulary\_inv\_list = data\_helpers.load\_data()

vocabulary\_inv = {key: value **for** key, value **in** enumerate(vocabulary\_inv\_list)}

w = trainWord2vec(x, vocabulary\_inv)

**3) data\_helpers.py**

========================

**import** numpy **as** np

**import** re

**import** itertools

**from** collections **import** Counter

**def** cleanStr(string):

string = re.sub(**r"[^A-Za-z0-9(),!?\'\`]"**, **" "**, string)

string = re.sub(**r"\'s"**, **" \'s"**, string)

string = re.sub(**r"\'ve"**, **" \'ve"**, string)

string = re.sub(**r"n\'t"**, **" n\'t"**, string)

string = re.sub(**r"\'re"**, **" \'re"**, string)

string = re.sub(**r"\'d"**, **" \'d"**, string)

string = re.sub(**r"\'ll"**, **" \'ll"**, string)

string = re.sub(**r","**, **" , "**, string)

string = re.sub(**r"!"**, **" ! "**, string)

string = re.sub(**r"\("**, **" \( "**, string)

string = re.sub(**r"\)"**, **" \) "**, string)

string = re.sub(**r"\?"**, **" \? "**, string)

string = re.sub(**r"\s{2,}"**, **" "**, string)

**return** string.strip().lower()

**def** load\_data\_and\_labels():

positiveExamples = list(open(**"./data/rt-polarity.pos"**).readlines())

positiveExamples = [s.strip() **for** s **in** positiveExamples]

negativeExamples = list(open(**"./data/rt-polarity.neg"**).readlines())

negativeExamples = [s.strip() **for** s **in** negativeExamples]

x\_text = positiveExamples + negativeExamples

x\_text = [cleanStr(sent) **for** sent **in** x\_text]

x\_text = [s.split(**" "**) **for** s **in** x\_text]

positive\_labels = [[0, 1] **for** \_ **in** positiveExamples]

negative\_labels = [[1, 0] **for** \_ **in** negativeExamples]

y = np.concatenate([positive\_labels, negative\_labels], 0)

**return** [x\_text, y]

**def** pad\_sentences(sentences, padding\_word=**"<PAD/>"**):

sequence\_length = max(len(x) **for** x **in** sentences)

padded\_sentences = []

**for** i **in** range(len(sentences)):

sentence = sentences[i]

num\_padding = sequence\_length - len(sentence)

new\_sentence = sentence + [padding\_word] \* num\_padding

padded\_sentences.append(new\_sentence)

**return** padded\_sentences

**def** buildVocab(sentences):

word\_counts = Counter(itertools.chain(\*sentences))

vocabulary\_inv = [x[0] **for** x **in** word\_counts.most\_common()]

vocabulary = {x: i **for** i, x **in** enumerate(vocabulary\_inv)}

**return** [vocabulary, vocabulary\_inv]

**def** buildInputData(sentences, labels, vocabulary):

x = np.array([[vocabulary[word] **for** word **in** sentence] **for** sentence **in** sentences])

y = np.array(labels)

**return** [x, y]

**def** load\_data():

sentences, labels = load\_data\_and\_labels()

sentencesPadded = pad\_sentences(sentences)

vocabulary, vocabulary\_inv = buildVocab(sentencesPadded)

x, y = buildInputData(sentencesPadded, labels, vocabulary)

**return** [x, y, vocabulary, vocabulary\_inv]

**def** batchIter(data, batch\_size, num\_epochs):

data = np.array(data)

data\_size = len(data)

num\_batches\_per\_epoch = int(len(data) / batch\_size) + 1

**for** epoch **in** range(num\_epochs):

shuffle\_indices = np.random.permutation(np.arange(data\_size))

shuffled\_data = data[shuffle\_indices]

**for** batch\_num **in** range(num\_batches\_per\_epoch):

start\_index = batch\_num \* batch\_size

end\_index = min((batch\_num + 1) \* batch\_size, data\_size)

**yield** shuffled\_data[start\_index:end\_index]

**Link to Code and Final PPT:**

[**https://drive.google.com/open?id=1G2RDr0UB05JTUPI01Dgxs1xnDUF4UQis**](https://drive.google.com/open?id=1G2RDr0UB05JTUPI01Dgxs1xnDUF4UQis)