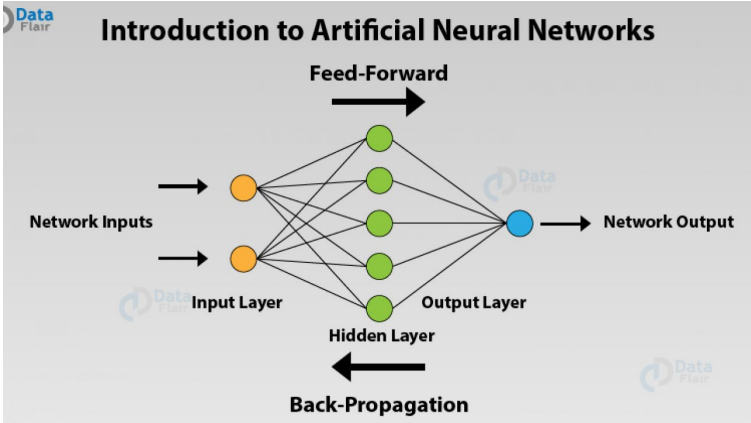
Understanding the Word Embeddings

Promothesh Chatterjee*

Word Embeddings for word2vec

Previously, we have looked at the basic idea underlying word embeddings. These are nothing but vector representation of words over a large number of dimensions (typically from 100 to 500). The way we derive values for different dimensions of word embeddings is by training a simple neural network with a single hidden layer for a particular task, and learn the weights of the hidden layer. These weights are nothing but the "word embedding values". Let me quickly give the high level view of a neural network without going into much details.

Neural networks try to imitate how supposedly the human brain works. Neural networks are made up of layers of neurons. These neurons are the core processing units of the network; first, we have the input layer which receives the input, final layer is the output layer which predicts our final output. In between, exist the hidden layers which perform most of the computations required by our network. Let's look at this figure from dataflair.com:

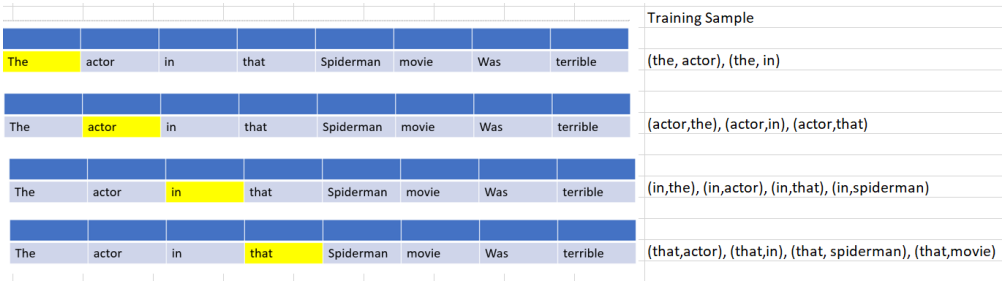


Typically, information is fed as input to each neuron of the first layer. These neurons of one layer are connected to neurons of the next layer through connection links. The signal is passed between neurons via connection links. Each of these links is assigned a numerical value known as the weight. The input signals are multiplied to the corresponding weights and sent as input to the neurons in the hidden layer. Each of these neurons is associated with a numerical value called the bias which is then added to the input, and then passed through a threshold function called the activation function. The result of the activation function determines if the particular neuron will get activated or not an activated. In this manner, the data is propagated through the network (forward propagation). In the output layer the neuron with the highest value fires and determines the output.

How does the network figure whether the output value is correct or not? During the training process, the network is fed both, the predicted output and the actual output. If there is an error in prediction, the information is then transferred backward through the network (back propagation). The weights are again adjusted and the cycle of forward propagation and back propagation is iteratively performed until network can make correct predictions.

Word2vec comes in two variants–the skip-gram model and the Continuous Bag-of-Words (CBOW) model. We will look at the skip-gram method first. In the skip-gram method, we will train the neural network to perform the following task. Given a particular word, look at the context window (say 10 words, remember the "window size" argument that we specified in our R code) and pick one randomly. The neural network will

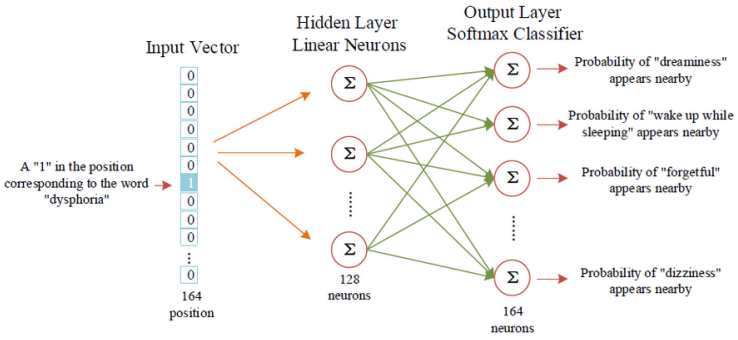
give us the probability for every word in the vocabulary of being the "nearby word". Look at the sentence below to understand the idea of context window. For illustration I have just taken a sample of the words with the context window as 2.



Training the neural network would entail inputting the word pairs from the training documents. The input word is highlighted in yellow. The neural network is going to learn from the frequency of each pairing. For instance, the algorithm will probably get many more pairs of ("Spiderman", "Movie") in the training set than it will of ("Spiderman", "Dinosaur"). Hence, if you input "Spiderman" in the trained network, the output "Movie" will have a higher probability than "Dinosaur".

Input and the Output Layer

The first question is how do we input the words? One-hot encoding is typically used for the task. Say, the vocabulary of the corpus is 5000 words. Each word is represented as a vector of 5000 dimensions, where the position of the specific word gets a value of 1 and all other positions get a value of 0. The output of the neural network is also a single vector (again with 5000 components) containing the probability that a randomly selected nearby word is that vocabulary word, in other words, the output vector will actually be a probability distribution (not a one-hot vector). Let's look at a figure from Hu et al.(2020) where we are taking an input a particular word "dysphoria" and the outout is probability of different words nearby such as "dreaminess", "forgetful" etc.



Hidden Layer

When creating word embeddings, we have to specify the number of dimensions (for instance, word2vec trained over google news corpus has 300 dimensions). In our case, the hidden layer will be a weight matrix featuring 5000 rows (for each word in the vocabulary) and 300 columns/dimensions (one for every hidden neuron). The number of features or dimensions is a hyperparameter that you can play around with for best results. Our word embeddings are nothing but rows of this weight matrix. Now if you multiply the 1 x 5000 one-hot vector by a 5000 x 300 weight matrix, it tantamounts to selecting the matrix row corresponding to

the "1" or the target word.

Softmax Regression in the Output Layer

The 1 x 300 word vector for "dysphoria" then gets fed to the softmax regression in the outer layer. Softmax regression (or multinomial logistic regression) is a generalization of logistic regression to the case where we want to handle multiple classes. The softmax regression takes each output neuron (one per word in our vocabulary!) and will produce an output between 0 and 1; the sum of all these output values will add up to 1.

Of course, the skip-gram neural network comprises a humongous quantity of weights. In our instance, 300 dimensions and 5000 words will make 1.5 million weights in both, the hidden layer and the output layer. Some shortcuts are needed to make training less cumbersome.

Running gradient descent (check this short video for the basic idea of gradient descent: https://www.youtube.com/watch?v=Gbz8RljxIHo&t=26s) on this large a neural network will be slow and cumbersome. Plus we run into the issue of overfitting. To avoid these the authors of word2vec implemented a sampling technique. This is a brief overview of the method:

1. First, subsample frequent words (for instance, stopwords) to decrease the number of training examples. This is intuitively similar to the TFIDF idea.
2. Second, shrink the context window randomly, this generates greater weight to neighboring context words.
3. Third, used a technique called "Negative Sampling", which causes each training sample to update only a small percentage of the model's weights.

This gives pretty accurate results.

CBOW Variant of the Word2vec Model

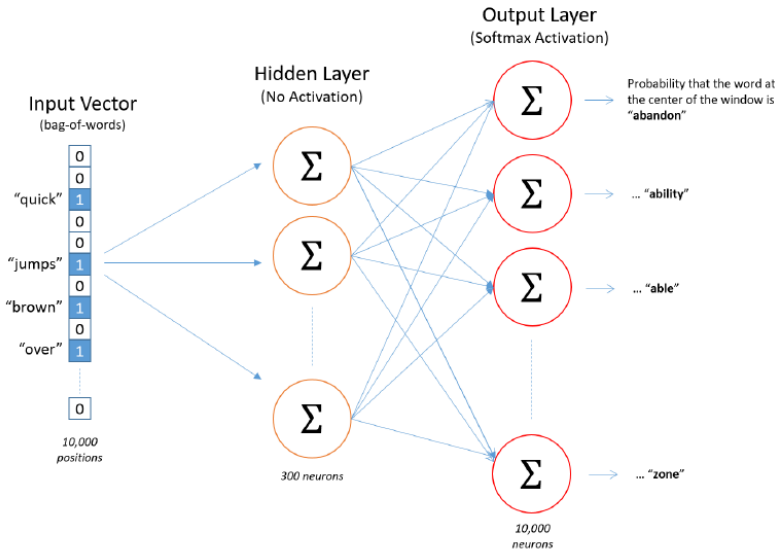
In the Continuous Bag-Of-Words variant of the word2vec model, we simply flip over the task from skip-gram model. Instead of predicting the near by words, in the CBOW version given a certain context window, what is most likely the word at the center? For example, say we have a window size of 2 on the following sentence. Given the words ("actor", "that","_____", "movie", "terrible"), the aim is to predict the word "spiderman".

Thus, the CBOW training set would look something like this:

								CBOW Training Sample
The	actor	in	that	Spiderman	movie	Was	terrible	((actor,in), (the))
The	actor	in	that	Spiderman	movie	Was	terrible	((the, in, that), (actor))
The	actor	in	that	Spiderman	movie	Was	terrible	((the, actor,that,spiderman),(in))
The	actor	in	that	Spiderman	movie	Was	terrible	((actor,in,spiderman,movie),(that))

Obviously, instead of a "one hot" vector, we use "bag-of-words" vector. It is pretty much the same concept, except that we put 1s in multiple positions (corresponding to the context words). Look at this popular

example:



With a context window of 2, skip-gram will generate (up to) four training samples per center word, whereas CBOW only generates one.

In the skip-gram model, multiplying the hidden layer weight matrix with one-hot vector results in selecting a single row. In the bag-of-words vector since the multiplication will result in multiple rows, we sum them up together and divide by the number of context words to create an average word vector.

The only other difference here is that instead of negative sampling as in skip-gram, we use hierarchical softmax here which is an alternative to negative Sampling. As a recap, both these methods reduce the cost of training by sampling only a smaller subset in the output layer which makes it easier to compute gradients for.

In general, skip-grams often perform better when small training set is small whereas CBOW performs a little better at capturing syntactic relationships (i.e., recognizing different conjugations of a verb like run, ran, running, runs)and at representing the more frequent words. The default method in most packages is CBOW.

References

Fang Hu, Liuhuan Li, Xiaoyu Huang, Xingyu Yan, Panpan Huang. Symptom Distribution Regularity of Insomnia: Network and Spectral Clustering Analysis.JMIR Medical Informatics (http://medinform.jmir.org), 16.04.2020