Practice 4

Word2Vec: Skip-Gram vs CBOW

Word2Vec

This approach was released back in 2013 by Google researchers, and it took the NLP industry by storm. In a nutshell, this approach uses the power of a simple Neural Network to generate word embeddings.

How does Word2Vec improve over frequency-based methods?

In TF-IDF, we saw that every word was treated as an individual entity, and semantics were completely ignored. With the introduction of Word2Vec, the vector representation of words was said to be contextually aware, probably for the first time ever.

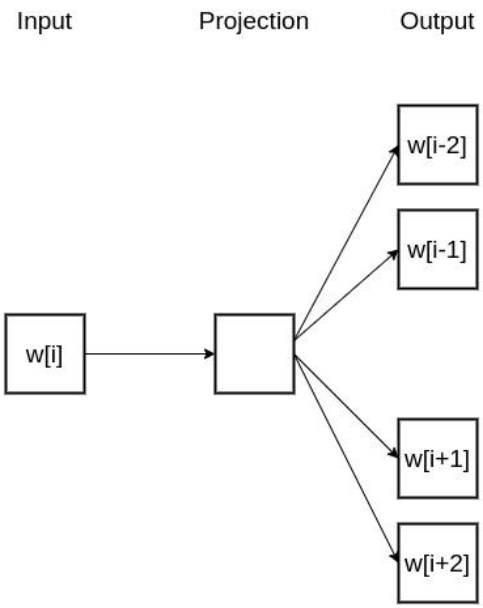
Perhaps, one of the most famous examples of Word2Vec is the following expression:

king – man + woman = queen

Since every word is represented as an n-dimensional vector, one can imagine that all of the words are mapped to this n-dimensional space in such a manner that words having similar meanings exist in close proximity to one another in this hyperspace.

There are mainly two ways to implement Word2Vec, let's take a look at them one by one:

I. So, the first one is the Skip-Gram method in which we provide a word to our Neural Network and ask it to predict the context. The general idea can be captured with the help of the following image:



Here w[i] is the input word at an ‘i’ location in the sentence, and the output contains two preceding words and two succeeding words with respect to ‘i’.

Technically, it predicts the probabilities of a word being a context word for the given target word. The output probabilities coming out of the network will tell us how likely it is to find each vocabulary word near our input word.

This shallow network comprises an input layer, a single hidden layer, and an output layer, we’ll take a look at that shortly.

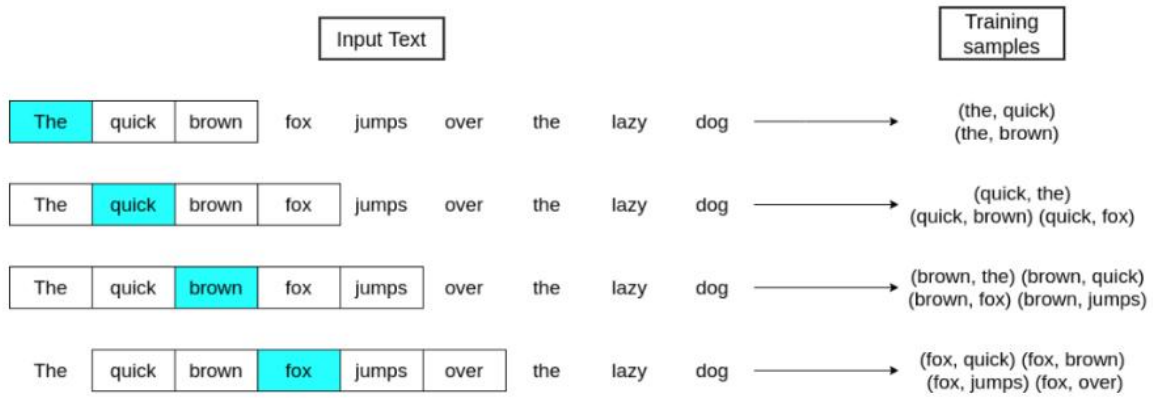
However, the interesting part is, we don’t actually use this trained Neural Network. Instead, the goal is just to learn the weights of the hidden layer while predicting the surrounding words correctly. These weights are the word embeddings.

How many neighboring words the network is going to predict is determined by a parameter called “window size”? This window extends in both the directions of the word, i.e. to its left and right.

Let’s say we want to train a skip-gram word2vec model over an input sentence:

“The quick brown fox jumps over the lazy dog”

The following image illustrates the training samples that would generate from this sentence with a window size = 2.



'The' becomes the first target word and since it's the first word of the sentence, there are no words to its left, so the window of size 2 only extends to its right resulting in the listed training samples.
As our target shifts to the next word, the window expands by 1 to the left because the presence of a word on the left of the target.
Finally, when the target word is somewhere in the middle, training samples get generated as intended.

The Neural Network

Now let's talk about the network which is going to be trained on the aforementioned training samples.

Intuition

If you're aware of what autoencoders are, you'll find that the idea behind this network is similar to that of an autoencoder.
You take an extremely large input vector, compress it down to a dense representation in the hidden layer, and then instead of reconstructing the original vector as in the case of autoencoders, you output probabilities associated with every word in the vocabulary.

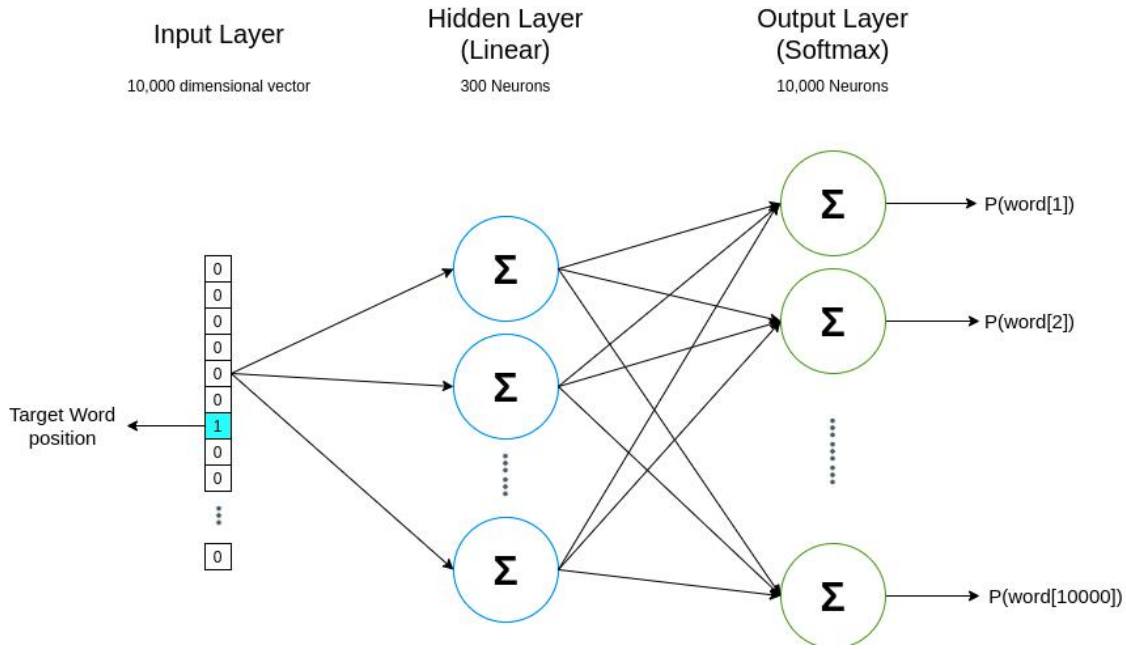
Input/Output

Now the question arises, how do you input a single target word as a large Vector?

The answer is One-Hot Encoding.

Let's say our vocabulary contains around 10,000 words and our current target word 'fox' is present somewhere in between. What we'll do is, put a 1 in the position corresponding to the word 'fox' and 0 everywhere else, so we'll have a 10,000-dimensional vector with a single 1 as the input.

Similarly, the output coming out of our network will be a 10,000-dimensional vector as well, containing, for every word in our vocabulary, the probability of it being the context word for our input target word.
Here's the architecture of how our neural network is going to look like this:



As it can be seen that input is a 10,000-dimensional vector given by our vocabulary size =10,000, containing a 1 corresponding to the position of our target word.
The output layer consists of 10,000 neurons with the Softmax activation function applied, so as to obtain the respective probabilities against every word in our vocabulary.
Now the most important part of this network, the hidden layer is a linear layer i.e. there's no activation function applied there, and the optimized weights of this layer will become the learned word embeddings.
For example, let's say we decide to learn word embeddings with the above network. In that case, the hidden layer weight matrix shape will be M x N, where M = vocabulary size (10,000 in our case) and N = hidden layer neurons (300 in our case).
Once the model gets trained, the final word embedding for our target word will be given by the following calculation:
1×10000 input vector * 10000×300 matrix = 1×300 vector

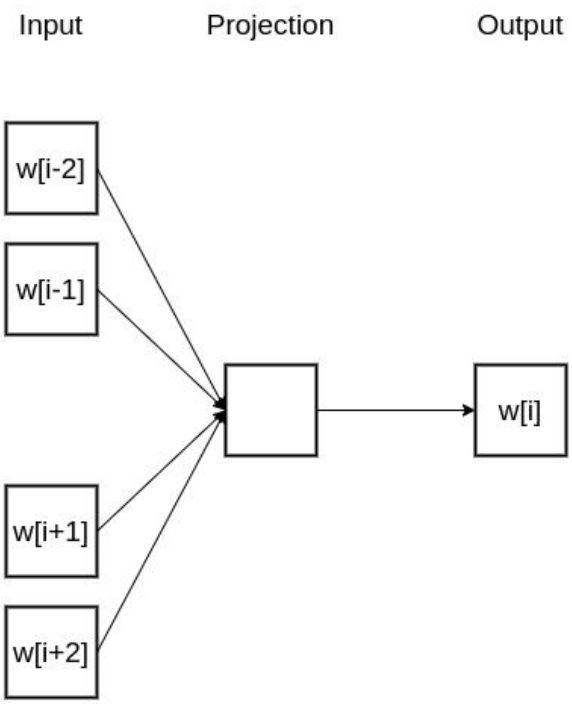
300 hidden layer neurons were used by Google in their trained model, however, this is a hyperparameter and can be tuned accordingly to obtain the best results.
So this is how the skip-gram word2vec model generally works.

Task 1: Describe how to create the word2vec embeddings on the base of a skip-gram.

Time to take a look at it's competitor.

II. CBOW

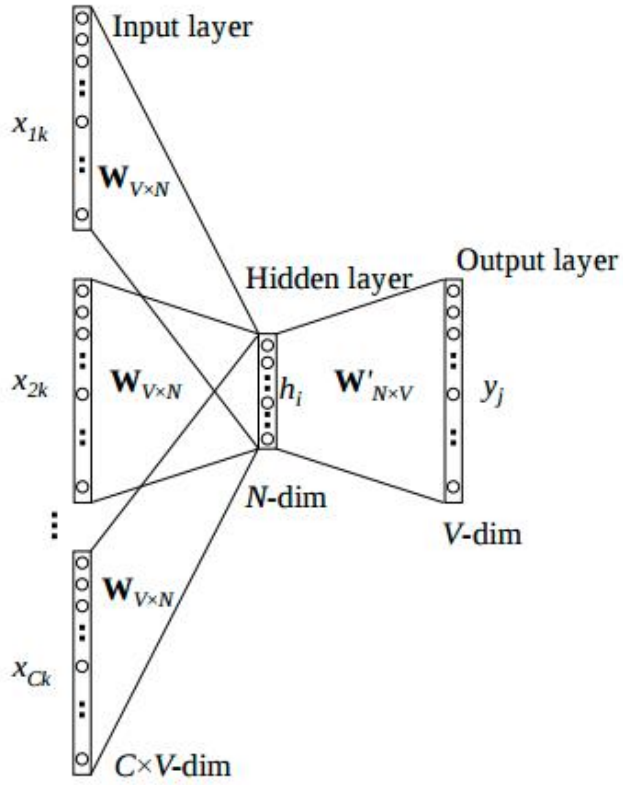
CBOW stands for Continuous Bag of Words. In the CBOW approach instead of predicting the context words, we input them into the model and ask the network to predict the current word. The general idea is shown here:



You can see that CBOW is the mirror image of the skip-gram approach. All the notations here mean exactly the same thing as they did in the skip-gram, just the approach has been reversed.

Now, since we already took a deep dive into what skip-gram is and how it works, we won't be repeating the parts that are common in both approaches. Instead, we'll just talk about how CBOW is different from skip-gram in its working. For that, we'll take a rough look at the CBOW model architecture.

Here's what it looks like:



CBOW Model Architecture

The dimension of our hidden layer and output layer stays the same as the skip-gram model.
However, just as we read that a CBOW model takes context words as input, here the input is C context words in the form of a one-hot encoded vector of size $1\text{x}V$ each, where V = size of vocabulary, making the entire input $C\times V$ dimensional.
Now, each of these C vectors will be multiplied with the Weights of our hidden layer which are of the shape $V\times N$, where V = vocab size and N = Number of neurons in the hidden layer.
If you can imagine, this will result in C, $1\times N$ vectors, and all of these C vectors will be averaged element-wise to obtain our final activation for the hidden layer, which then will be fed into our output softmax layer.
The learned weight between the hidden and output layer makes up the word embedding representation.

Because of having multiple context words, averaging is done to calculate hidden layer values. After this, it gets similar to our skip-gram model, and learned word embedding comes from the output layer weights instead of hidden layer weights.

When to use the skip-gram model and when to use CBOW?

Skip-gram works well with small datasets and can better represent rare words. However, CBOW is found to train faster than skip-gram and can better represent frequent words.
So the choice of skip-gram vs CBOW depends on the kind of problem that we're trying to solve.

Task 2 :Describe how to create the Word2Vec embeddings on the base of CBOW.

The whole algorithm for Word2Vec:

1. Create a list of texts;
2. Create a vocabulary, reduce it's size if it is necessary;
3. Create a neural network according to the implemented model (skip-gram or CBOW)
4. Translate words to one-hot encoding vector.

Gensim library

Gensim is a Python library for topic modelling, document indexing and similarity retrieval with large corpora. Target audience is the natural language processing (NLP) and information retrieval (IR) community [**https://www.machinelearningplus.com/nlp/gensim-tutorial/**].

Using Gensim

You can use gensim in any of your python scripts just by importing it like any other package. Just use the following import:

```
import gensim
```

Develop Gensim Word2Vec Embedding

In order to work on text documents, Gensim requires the words (tokens) be converted to unique ids. In order to achieve that, Gensim lets you create a `Dictionary` object that maps each word to a unique id. So, how to create a `Dictionary`? By converting your text/sentences to a [list of words] and pass it to the `corpora.Dictionary()` object. The input text typically comes in 3 different forms:

1. As sentences stored in python's native list object
2. As one single text file, small or large.
3. In multiple text files, grouped in one folder
   [https://www.machinelearningplus.com/nlp/gensim-tutorial/].

```
from gensim import corpora

# How to create a dictionary from a list of sentences?
documents = ['data science is one of the most important fields of
science',
             'this is one of the best data science courses',
             'data scientists analyze data']

# Tokenize(split) the sentences into words
texts = [[text for text in doc.split()] for doc in documents]

# Create dictionary
dictionary = corpora.Dictionary(texts)

# Get information about the dictionary
print(dictionary)
```

Show the ids of words in dictionary:

```
# Show the word to id map
print(dictionary.token2id)
```

We have successfully created a Dictionary object. If you get new documents in the future, it is also possible to update an existing dictionary to include the new words.

```
from gensim.models import Word2Vec

# train model
model = Word2Vec(texts, min_count=1)
# summarize the loaded model
print(model)
# summarize vocabulary
words = list(model.wv.vocab)
print(words)
# access vector for one word
print(model['scientists'])
# save model
model.save('model.bin')
# load model
new_model = Word2Vec.load('model.bin')
print(new_model)
```

Let's run the code, we are expecting vector for each word:

```
Word2Vec(vocab=14, size=100, alpha=0.025)
['data', 'science', 'is', 'one', 'of', 'the', 'most', 'important',
'fields', 'this', 'best', 'courses', 'scientists', 'analyze']
[ 8.0946906e-05 -2.3447643e-05  3.0631042e-04  3.5974982e-03
 -2.3266664e-03  7.3776109e-04  1.6859109e-03  2.8264525e-03
  2.6968149e-03  1.0197462e-03  4.4770944e-03  2.5953732e-03
 -1.4906763e-03 -6.7264057e-04 -3.1194722e-03 -7.1081921e-04
  4.2864135e-03 -2.1385239e-03  3.4250619e-04 -2.8121686e-03
  4.2369459e-03 -4.3218499e-03 -3.3128243e-03  2.8898765e-04
 -3.2172739e-03 -1.3250931e-03 -2.8357239e-04 -1.9586461e-03
  8.7642699e-04  2.7913270e-03  2.4418011e-03 -2.6495145e-03
  2.1393497e-03  4.9795946e-03  2.6344779e-04  2.0657177e-03
 -3.4472232e-03  4.1874223e-03 -2.0517923e-03 -3.6642856e-03
 -4.9046422e-03  4.2103827e-03  3.1076695e-03  4.5919777e-03
 -3.9744992e-03 -8.5270405e-04  3.1025568e-03  2.9759650e-04
  8.1874325e-04 -5.8604893e-04  3.1681990e-03  4.0135449e-03
 -3.2999497e-03  4.2036376e-03  2.2640333e-03  2.0973913e-03
 -4.3956912e-03 -4.3523381e-03 -2.6643889e-03  3.8661286e-03
  3.6859985e-03 -2.3312182e-03  2.5218364e-03 -4.9016201e-03
  1.2470389e-03  2.8178850e-03 -4.2354967e-03  5.2679694e-05
  4.8665288e-03  3.3792746e-03 -1.4594697e-03  4.2065335e-04
 -8.4669143e-04  2.0868680e-03  1.7894120e-03  4.9348390e-03
  5.1669358e-05  4.8590628e-03  1.5331771e-04  1.1289392e-03
  1.8111974e-03  4.3893661e-03  2.6811608e-03 -3.7260819e-03
 -4.9275705e-03  9.8996423e-04 -4.8519056e-03  2.6469470e-03
  2.1196834e-03 -6.9309864e-04 -1.2348744e-03  2.3128139e-03
 -2.5820716e-03 -3.6399348e-03  2.8450666e-03 -1.6017255e-03
  9.8955445e-04  3.1314211e-03 -2.0322208e-03  1.2194500e-03]
```

Visualize Word Embedding

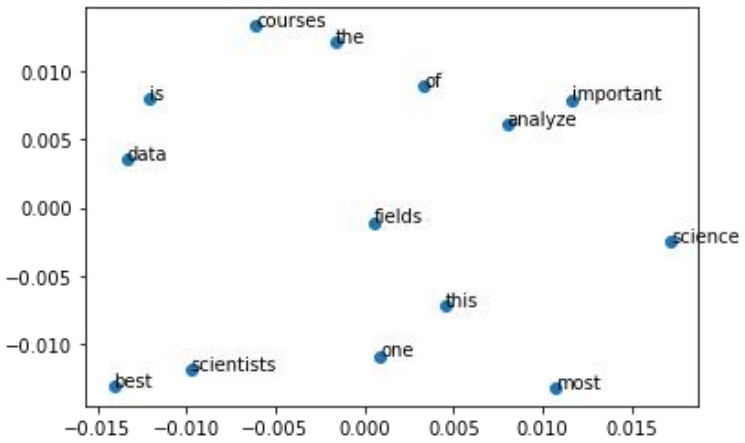
We can see several vectors for every word in our training data and it is definitely hard to understand. Visualizing can help us in this scenario:

```
from gensim.models import Word2Vec
from sklearn.decomposition import PCA
from matplotlib import pyplot

# fit a 2d PCA model to the vectors
X = model[model.wv.vocab]
pca = PCA(n_components=2)
result = pca.fit_transform(X)
# create a scatter plot of the projection
pyplot.scatter(result[:, 0], result[:, 1])
words = list(model.wv.vocab)
for i, word in enumerate(words):
	pyplot.annotate(word, xy=(result[i, 0], result[i, 1]))
pyplot.show()
```

Task 3: Analyze the line: `X = model[model.wv.vocab]`. Shortly describe the PCA method. What is it used for?

Let's start the program. So we've reduced the dimension of our word vectors and deduced their first 2 major (most important) components to see word proximity:



Load Google's Word2Vec Embedding

Using an existing pre-trained data may not be the best approach for an NLP application but it can be really a time consuming and difficult task to train your own data at this point as it requires a lot of computer RAM and time of course. So we are using Google's data for this example. For this example, you'll be needing a file which you can find here.

Download the file, unzip it and we'll use the binary file inside.

Here is a sample program:

```
from gensim.models import KeyedVectors
# load the google word2vec model
filename = 'GoogleNews-vectors-negative300.bin'
model = KeyedVectors.load_word2vec_format(filename, binary=True)
# calculate: (king - man) + woman = ?
result = model.most_similar(positive=['woman', 'king'], negative=['man'],
topn=1)
print(result)
```

The above example loads google's word to vec data and then calculates **king-man + woman=?.** We should expect the following:

```
[('queen', 0.7118192315101624)]
```

Task 5: Use Gensim library to define 10 most similar words to word "happy".

Task 6: Use Gensim library to train your own model on the true-fake news corpus. Don't forget to make the preprocessing step.

Task 7. Output top 15 most similar words to one word (on your choice) from the corpus. Visualize the results on plot.

To use the trained languge model for word prediciton:

```
print(model.predict_output_word(['good', 'candidate', 'is']))
```

Task 8. Try to predict the next word with trained model.

Questions for Practice 4:

1. Describe the advantages of Word2Vec and CBOW methods.
2. Get the definition to one-hot encoding.
3. Describe the algorithm word2vec embeddings on the base of neural network.
4. Describe the algorithm CBOW embeddings on the base of neural network.
5. Give the definition to language model.