Natural Language Processing 2

StartOnAI

1 Overview

In this tutorial, we will be covering some of the most important techniques in natural language processing. We will begin by introducing what word embeddings are and how they work. Then, we will cover how word embeddings can be learned and created using a popular technique called word2vec. Finally, we will also mention some of the limitations to the techniques we discuss.

2 Word Embeddings

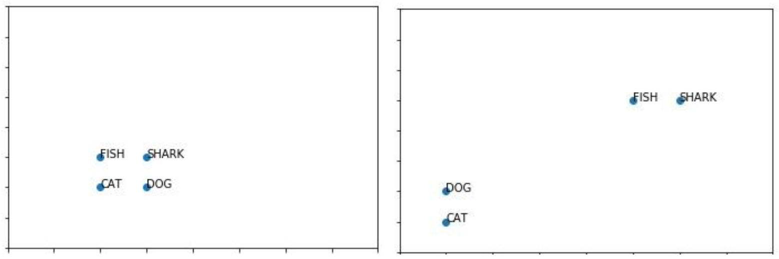
At the most basic level, word embeddings are numerical representations of words that can be passed into a machine learning model. Word embeddings are generally multi-dimensional vectors that can be used to represent a specific word from a given set or dictionary. Let's take the example sentences: *"Machine Learning is fun. Machine Learning is enjoyable."*. The basic and most obvious way to assign vector representations to these words would be as follows: *Machine* $\rightarrow \begin{bmatrix}1 & 0 & 0 & 0 & 0\end{bmatrix}$, *Learning* $\rightarrow \begin{bmatrix}0 & 1 & 0 & 0 & 0\end{bmatrix}$, *is* $\rightarrow \begin{bmatrix}0 & 0 & 1 & 0 & 0\end{bmatrix}$, *fun* $\rightarrow \begin{bmatrix}0 & 0 & 1 & 0 & 0\end{bmatrix}$, *enjoyable* $\rightarrow \begin{bmatrix}0 & 0 & 0 & 1 & 0\end{bmatrix}$.

As you can see, these are 4 dimensional vectors that are distinct for each word. This technique is called one-hot encoding. This is a valid approach to represent each word numerically, however, each vector does not give us much information about the word. In particular, the vectors for *fun* and *enjoyable* are as different (in terms of distance) as the vectors for *Learning* and *is.* Clearly, this should not be the case because "fun" and "enjoyable" are almost synonyms, so their vectors should be similar.

In order to intelligently create vector representations for words, we use a procedure called Word2Vec.

3 Word2Vec

As you can tell from its name, the purpose of Word2Vec is to take a word and represent it as a vector. The input would be the text corpus (full set of words) and the algorithm should output a vector for each distinct word. However, the goal is to place similar words together in the vectorspace and different words farther away. This idea of intelligently creating vectors was developed by Tomas Mikolov et al, in 2013. They showed that you can use a neural network architecture to create word embeddings that maintain the context of the word (Figure 1b).



(a) (b)

Figure 1: Embeddings without word2vec in (a) show less context compared to word2vec embeddings in (b). Credits: HackDeploy

Since the vectors for the words are created based on their context, we can do math operation on the words and get pretty interesting results. In Figure 1, we do not have many words and the vectors are only 2 dimensional for the sake of visualization. However, in reality, there would be over a 1000 words which are represented in high dimensional vectors. A classic example would be as follows:

$$w_{king} - w_{man} + w_{woman} = w_{queen}$$

That statement makes sense logically, and Figure 2 shows a nice graphical representation of the same equation. w_n represents a vector for a given word n.

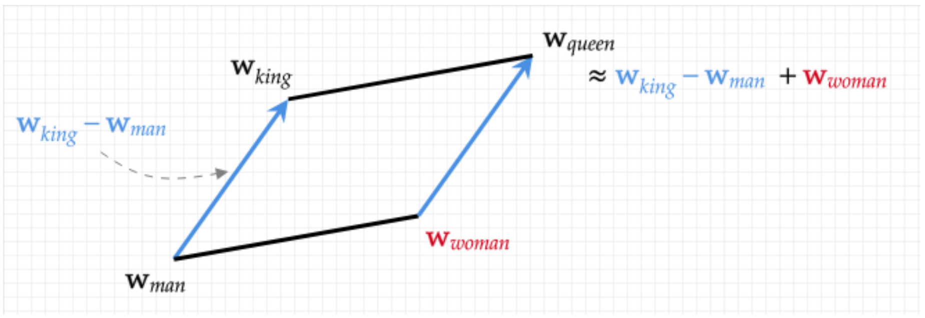

$\mathbf{w}_{queen}$
$\approx \mathbf{w}_{king} - \mathbf{w}_{man} + \mathbf{w}_{woman}$
$\mathbf{w}_{king}$
$\mathbf{w}_{king} - \mathbf{w}_{man}$
$\mathbf{w}_{woman}$
$\mathbf{w}_{man}$

Figure 2: Note how math operations can be used to interpret words. Credits: University of Edinburgh

Now, you should be able to see what it means to store context in the vector representation of a word. While analyzing words, you will often need to know how similar two words are. There are two ways to mathematically define this.

1. Euclidean Distance
2. Cosine Similarity

The Euclidean Distance is pretty basic because all it does is find the distance between two points in the vectorspace. The distance between n-dimensional vectors $\vec{p}$ and $\vec{q}$ is shown in Equation 1.

$$d(\vec{p}, \vec{q}) = \sqrt{\sum_{i=1}^{n}(p_i - q_i)^2} \tag{1}$$

Going back to Figure 1b, we can now use the euclidean distance to prove that the word "shark" is more similar to "fish" than "cat". The cosine similarity metric is more suited when applied to higher dimensional vectors. In plain english, the cosine similarity is the angle between two vectors. This can also be mathematically defined as shown in Equation 2.

$$\text{cosine similarity} = cos(\theta) = \frac{\vec{p} \cdot \vec{q}}{\|\vec{p}\| \, \|\vec{q}\|} = \frac{\sum_{i=1}^{n} p_i q_i}{\sum_{i=1}^{n} p_i^2 \cdot \sum_{i=1}^{n} q_i^2} \tag{2}$$

In 3.1 and 3.2, we will go into more details on how to produce the word vectors given a corpus of text (i.e. set of sentences). We will start with the *Continuous Bag of Words* or CBOW model.

3.1 CBOW

In order to train a CBOW model, there must be a corpus of words that we can fit our data to. Most likely, a corpus will contain over 10,000 words, however, for simplicity, consider the following sentence:

"The dog played in the sun"

Let's denote the one-hot encoded vectors for the ith word as X_i.

$$\begin{aligned}
\text{the} = X_1 &= \begin{bmatrix}1 & 0 & 0 & 0 & 0 & 0\end{bmatrix}^T \\
\text{dog} = X_2 &= \begin{bmatrix}0 & 1 & 0 & 0 & 0 & 0\end{bmatrix}^T \\
&\vdots \\
\text{sun} = X_6 &= \begin{bmatrix}0 & 0 & 0 & 0 & 0 & 1\end{bmatrix}^T
\end{aligned}$$

You will probably notice that in a real dataset, these vectors will have a dimension of almost 10,000! That is another reason why intelligently creating vectors is helpful, because we will be able to create compressed representations with more meaning.

As emphasized, the whole point of CBOW is to maintain context in the vector representations. The way this is done is by looking at windows of text. We will refer to the size of a window as c. If $c = 2$, then the windows of text we would analyze/pass through the model would be as follows:

$\boxed{\text{The dog played in the}}$ sun

The $\boxed{\text{dog played in the sun}}$

As you can see, each window of text contains 5 words: a center word, c words to the left, and c words to the right. Understanding what a window of text is important for both CBOW and skip-gram models, and we will use them in Section 3.2 again.

In a CBOW model, the goal is to be able to predict the center word y, given the surrounding context words X. This makes the entire process unsupervised because we are doing everything based off of a given sentence.

The structure of CBOW is very similar to an autoencoder. We have a input vector and output vector, and again like an autoencoder, the most useful part is the bottleneck layer. The hidden layer in the center is where we get our new vectors.

While building a CBOW model, the only layer that you can tweak is the single hidden layer. It is important to choose the correct number of neurons for the hidden layer because this will end up being the dimension of the final word vectors. Let's assume the hidden dimension is N.

Before learning about the forward propagation process, see if you can understand the architecture shown in Figure 3. Keep in mind that there are $2c$ input vectors which are each one-hot encoded.

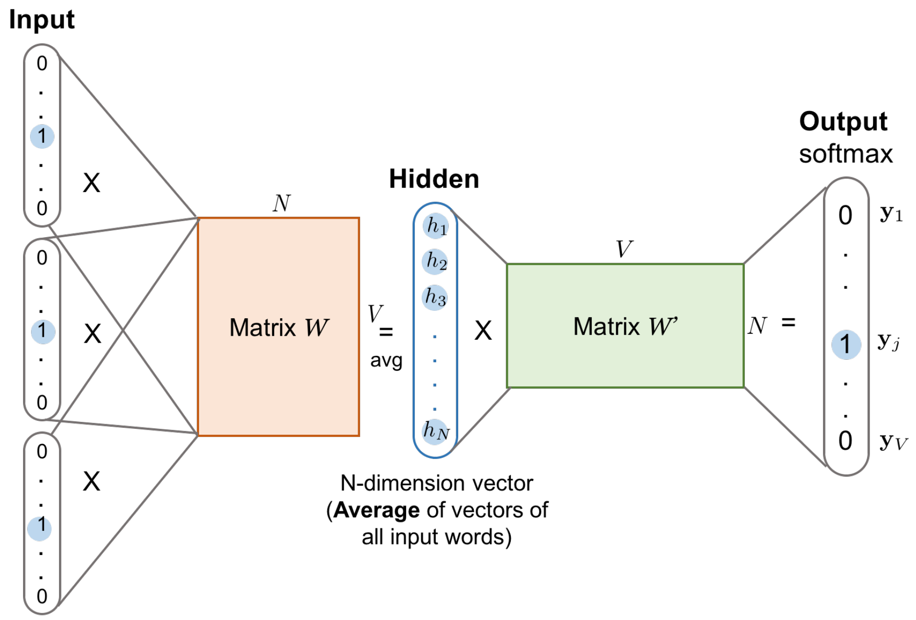


Figure 3: CBOW architecture consists of one hidden layer in addition to the input and output layers. Adapted from TowardsDataScience

Forward Propagation

As shown in the image, there are two weight matrices in the CBOW model, W and W'. Pay close attention to W in particular since it is almost like a lookup table for word vectors once trained. Let's walk through the process using the first window:

$$\boxed{\text{The dog played in the}}\ \text{sun}$$

We are trying to predict the center word "played". Note how the y vector can be taken directly from our input vector X because of the unsupervised nature of this algorithm.

$$y = X_3 = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}^T$$

Finally, because $c = 2$, we have 4 input vectors X_1, X_2, X_4, and X_5 which have been defined earlier. All the variables have now been defined, and understanding how these values pass through the model is relatively simple because there are only 3 layers. The steps can be outlined in a 5 step process.

1. Multiply each input vector X_n with weight matrix W to get $2c$ or 4 vectors of length N denoted as p.

$$p_n = X_n \cdot W$$

2. Average the p vectors to get the hidden layer v. Of course, because p has dimension N, v has the same size.

$$v = \frac{p_1 + p_2 + p_4 + p_5}{4}$$

3. Multiply vector v with W' with shape NxV to yield a vector z of length V. V is the number of words in the corpus of text. In other words, it is the size of the one-hot vectors.

$$z = W' \cdot v$$

4. Because the y vector is one-hot encoded, we can apply the softmax activation function to assign each neuron from z to have a probability of being 1.

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{V} e^{x_j}}$$

$$\hat{y}_i = \text{softmax}(z_i)$$

Getting the Final Word Vector

The most important trainable parameter in the entire model is the weight matrix W. That is the only necessary component in order to generate our word vectors. If we want to generate the word vector w for a given word n, we use the one-hot encoded representation of the word and the matrix W.

$$w_n = X_n \cdot W$$

Since X_n is a one-hot encoded vector, the above equation can be simply stated by saying w_n is the nth column in W. As you can see, once we have a properly trained a CBOW model, getting the word vector with context is very simple.

CBOW models can be trained using the categorical cross-entropy loss defined in Equation 3.

$$CE = -\sum_{i=1}^{V} y_i \log(\hat{y}_i) \tag{3}$$

The only problem with this loss function is that we are iterating V times. V can be the total number of words in an English dictionary. Iterating a million times is very computationally expensive. The same issue applies with the softmax activation function, as it requires a summation over all V dimensions. This is one of the down sides of the CBOW architecture, however, researchers have figured out ways to get around this obstacle. For example, Mikolov et al. presented a technique called negative sampling to approximate the loss function rather than calculating the exact cross-entropy value called Negative Sampling.

This tutorial will not cover Negative Sampling in detail, however, we will cover another common architecture used to produce word2vec embeddings known as Skip Gram.

3.2 Skip Gram

Often times, when you look at a tutorial on skip gram and CBOW, they will start by showing the following image.

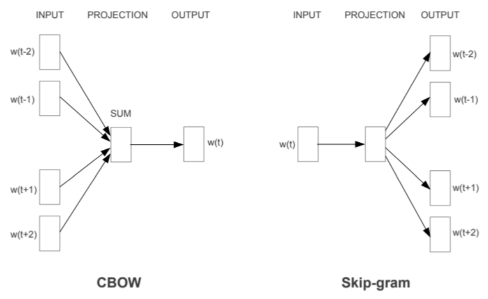
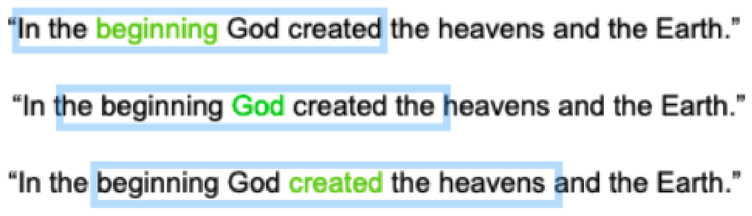


Figure 4: This image shows the similarities between Skip Gram and CBOW. Credits: TowardsDataScience

As shown in Figure 4, the Skip Gram model starts with one word, and tries to predict the context words from the window. Take the sentences given below, for example. The goal of skip gram is to take in the word “beginning” as input, and produce the one-hot vectors for the words “in”, “the”, “God”, and “created” (assuming $c = 2$).



“In the beginning God created the heavens and the Earth.”

“In the beginning God created the heavens and the Earth.”

“In the beginning God created the heavens and the Earth.”

Figure 5

Again, as with the CBOW architecture, the Skip Gram model uses the input and output layers only while training, after that, the only layer that is important is the hidden layer which will eventually contain the word embeddings. Let’s take a look at the architecture of Skip-gram in Figure 6.

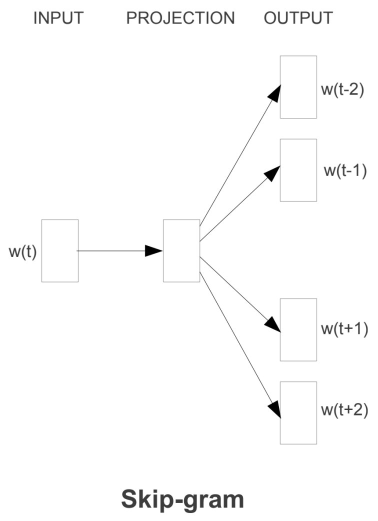


Figure 6: Skip-gram model architecture. Credits: Mikolov et al.

Forward Propagation

Let’s go through the step-by-step process of how a Skip-gram model produces word embeddings just like we did with CBOW. First, we start with the one-hot encoded vector for the center word X_2. With the new text corpus from Figure 5, X_2 would represent the word “beginning”.

$$X_2 = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^T$$

Notice how this time the one-hot encoded vector has a dimension 8 even though the sentence has 10 words in it. This is because the word “the” is repeated 3 times so we only have 8 distinct words in the corpus.

Now, similar to last time, the first step is to multiply the X_2 by the weight matrix W of shape VxN which yields the vector v.

$$v = X_2 \cdot W$$

Unlike the CBOW process, the Skip-gram does not require us to find an average vector. Since we only have one input, we can directly get to the hidden layer. Now, the output is a little trickier because of the number of vectors we produce. Because we chose a window size of 2, we will have 4 output vectors. In general, you will have $2c$ output vectors each corresponding to a word from that window.

For each respective output word, we must multiply v with W' which has a shape of NxV yielding a score vector of size V just as we wanted. After applying the softmax activation function to each of the $2c$ score vectors, we have our predictions for word vectors. The goal is to predict each context word correctly.

Again, the loss function is very similar to before. We can use the categorical cross-entropy (3) and treat each of the V dimensions as a separate class.

Although Skip-gram is often a little less intuitive that CBOW, getting the word vectors is extremely simple. Since we only have one input word this time, all we do is pass in a word, and the hidden layer is the final word vector!

Both of the models suffer from the fact that the softmax activation function must be applied causing each iteration to have $\mathcal{O}(V)$ time-complexity. Regardless of their very similar architecture, experiments have shown that each model has its own pros and cons. For example, Skip-grams often perform better than CBOW if there is less data available, however, CBOW is extremely fast to train compared to the Skip-gram.

In this tutorial, a lot of information was covered and we focused on two of the most popular models used to produce word2vec embeddings. This is still a very new field of research since word2vec was introduced less than a decade ago! New techniques have been created such as GloVe which also create word embeddings while maintaining a words’ context. We hope this tutorial provided a good intro into some of the most fascinating parts of Natural Language Processing!