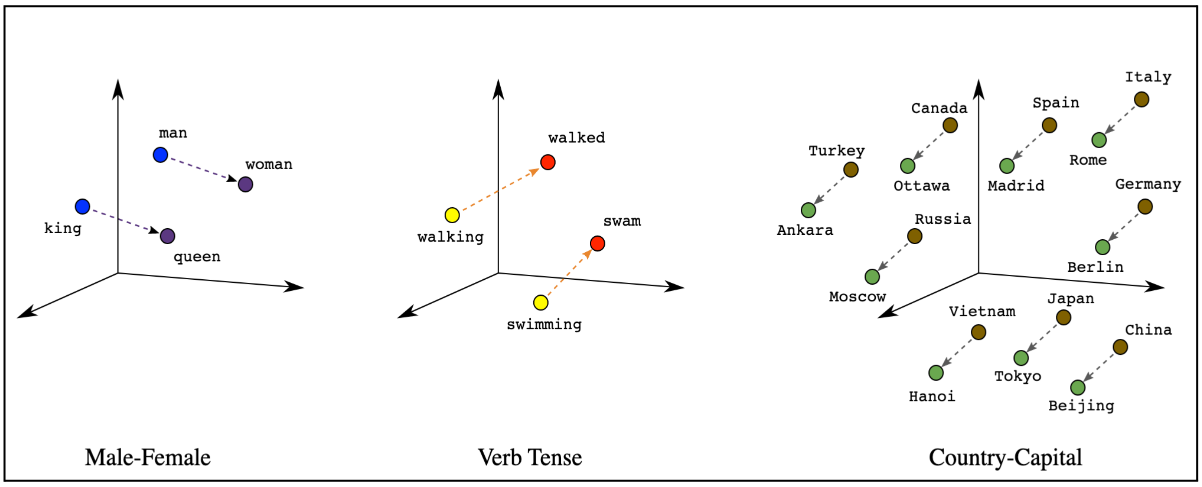
Word Embeddings

Author: **Neeraja Menon**
www.linkedin.com/in/neeraja-j-menon

Let's say that you want to create a machine-learning model that can detect spam emails. The model will work by looking at the words it finds in an email and compare them to a set of words it already knows to be commonly found in spam emails. However, the model will not know every possible word to look out for, so there needs to be a way to detect words of similar meanings. If the words 'prize' and 'lottery' need to be compared, how will it know that these words are related? This is where word embeddings come in.

In Natural Language Processing (NLP), word embeddings are a way to represent relationships between words in a high-dimensional space. It captures the semantics of a word by mapping it to a vector consisting of real numerical values - the closer two vectors are together in this space, the closer their meaning or relation is.



Visual representation of 3 different vector spaces, each displaying a different relationship

The vector spaces above are achieved by training a model on a large corpus of text using a neural network. There are several word embedding techniques which exist, and depending on this, variations in network structures and algorithms. For simplicity, the focus of this article will be on 'Word2Vec'. This is the most commonly used method - the model can either predict the surrounding words given a target word or predict the target word given the surrounding words. Once this vector representation is created, this information can be fed into machines that can use this for applications such as sentiment analysis, text classification, and even language translation. The process of creating these models can be split into creating the initial vector representations of the word, and then training the model.

One-hot Encoding

This is the most basic way to represent a word in a vector format. If you have a dictionary with n words, it creates a binary vector representing each word of size $1 \times n$. The index of the current word will have the value 1, and all other indices will be 0. For the sake of this example let's imagine we have a dictionary of 5 words: Car, Bike, Transport, Orange, and Fruit. The vector representation for each of the words is shown below:

$$\begin{array}{ccccc} \textbf{Car} & \textbf{Bike} & \textbf{Transport} & \textbf{Orange} & \textbf{Fruit} \\ \begin{bmatrix}1\\0\\0\\0\\0\end{bmatrix} & \begin{bmatrix}0\\1\\0\\0\\0\end{bmatrix} & \begin{bmatrix}0\\0\\1\\0\\0\end{bmatrix} & \begin{bmatrix}0\\0\\0\\1\\0\end{bmatrix} & \begin{bmatrix}0\\0\\0\\0\\1\end{bmatrix} \end{array}$$

These vectors are only a starting point. With the current values, all the words are positioned on their own plane - it conveys that the relationship between truck and orange is the same as between orange and fruit, both 0. Mathematically, this can be shown using their cosine similarity:

$$cos(\theta) = \frac{a \cdot b}{|a||b|}$$

Given the input of 2 vectors a and b, the formula outputs the cosine of the angle between them, so the more similar two words are, the higher the cosine similarity between their respective vectors will be. Since all these vectors are currently orthogonal with eachother, their dot products will all be 0, and consequently their cosine similarities will be 0 too. Therefore, these values need to be updated to more accurately respresent their semantics. Neural networks are used for this purpose.

Training the Model

These initial vectors described above are fed into the neural network as the input layer. Depending on the type of word embedding method used, the way the neural networks learn can differ. The most commonly used method is known as 'Word2Vec'.

Word2Vec works by using a shallow neural network (meaning it only has 1 or 2 hidden layers), and learns using the words around the target word (the word we are trying to create the embedding of. Firstly, a 'window size' is determined - this is the number of words before and after the target word in a sentence that will be accounted for. Using this window size, all possible pairs are created for that sentence - a pair consists of the target word and a surrounding word that is present in that chosen window size. For example, if we have the following sentence:

'My preferred modes of transport include car and bike'

A window size of 2 for the target word 'car' would include these words:

'My preferred modes of transport include car and bike'

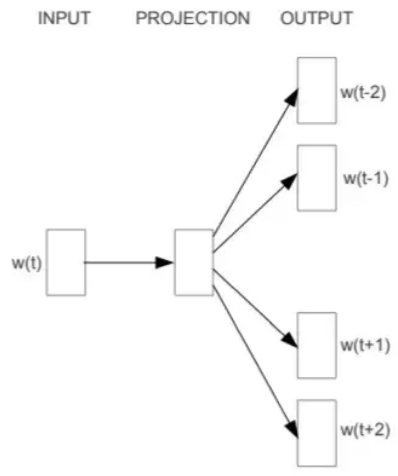
Consequently, the following word pairs can be created:

(car , include) , (car , transport) , (car , and) , (car , bike)

Within Word2Vec, there are 2 ways these pairs can be used to create the word embeddings. It can use 'Skip-Gram' or a 'Continuous Bag Of Words' (CBOW).

Skip-Gram

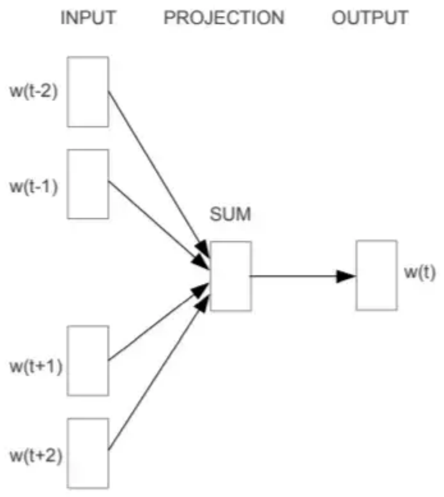
The former works by taking the one-hot encoded vector V of the target word as the input layer, and uses this to predict the words around it. This will have size $1 \times n$. The size of the hidden layer will be $n \times m$, where m is the size of the ideal size of word embeddings (the higher m is, the more information it can learn). The size of the final layer will once again be $1 \times n$ as it outputs the vector representation of the surrounding words. It is run for each target word until all words in the corpus have been given as an input. During each run, the weights for the hidden layer will be updated too. In the end, this will result in a weight that is a matrix of size $n \times m$ which will be the final weight of the hidden layer. This model should now produce word embeddings that better represent the relationships between words.



Skip-gram

CBOW

Conversely, CBOW takes the surrounding words as one-hot encoded vectors for the input layer and trains the model to predict the target word. It works the same way as Skip-Gram does, except since there are multiple input vectors, the weight for the hidden layer will be an average of the hidden layers of each surrounding/input word.



CBOW

Overall, CBOW works better with words that arise more frequently and generally is much faster to train. On the other hand, Skip-Gram is ideal when the model needs to be catered to the problem as it works well with rare words and phrases.

Implement it Yourself!

If you would like to implement this yourself we highly recommend the Tensorflow implementation: https://www.tensorflow.org/tutorials/text/word2vec

Models such as Fasttext(https://github.com/facebookresearch/fastText) and GloVe(https://nlp.stanford.edu/projects/glove/) build on top of Word2Vec.
BERT is a State of Art approach that considers the semantic context, as opposed to Word2Vec.

As you’ve probably gathered, word embeddings are a great tool to solve many problems in NLP. They're even used for music recommendation systems, spam email detection, and survey response analysis. They provide a way for machines to understand the ‘human meaning’ behind words in a sophisticated way. Although other techniques like statistical models and rule-based systems are still used in NLP, word embeddings still play a crucial role, greatly improving performance and optimising tasks.