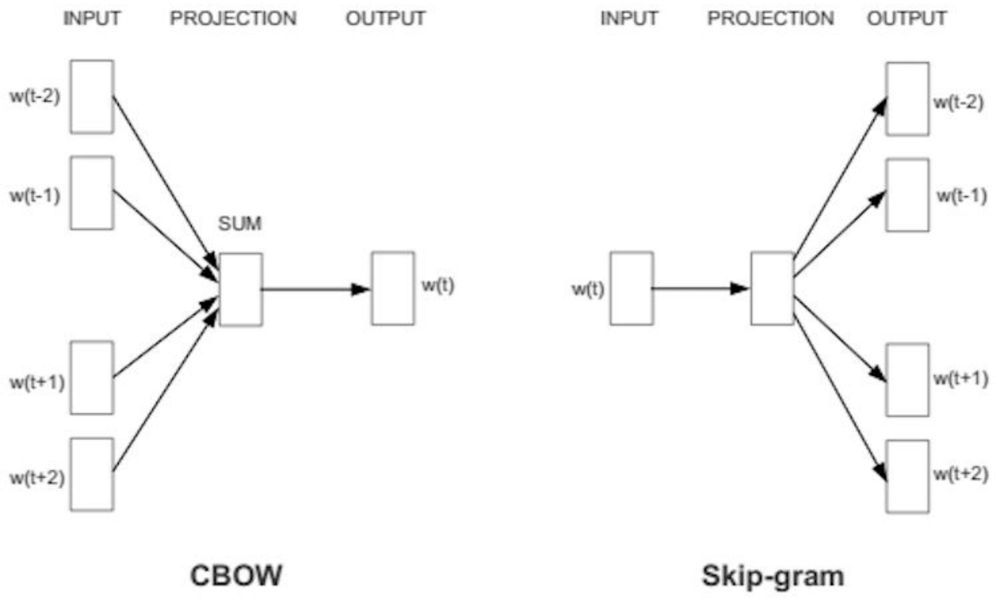
Fun With Word Vectors

Motivation

As a person who's always been better at dealing with numbers than words, I was fascinated when I first heard the idea of capturing a word's meaning through a sequence of numbers. How is this possible was my very next thought which led me to discover word vectors and the various techniques used to create them. While I had an intuitive understanding of how they were formed, I like to really immerse myself in the topics I'm interested in and I thought the best way to learn was to actually build vectors from scratch. Learning by coding. Given we had to complete a project for this course, I couldn't think of a better topic.

Popular Algorithms

WIth the onset of deep learning and improved machinery, it's no surprise that the two most popular techniques at the moment use a neural network architecture. The first one was proposed in Mikolov et al. (2013) - 1 where the authors describe two methods for building word vectors, namely the Continuous Bag-of-Words (CBOW) Model and the Continuous Skip-gram Model. The CBOW model, has a simple architecture with only the projection layer and the output layer (no hidden layers). The goal of this model, is to predict the current word, based on the surrounding words. Thus, this model uses context to its advantage by looking at words before and after the target. On the other hand, the Continuous Skip-gram Model takes a word and tries to predict the surrounding words within a certain window. The input in this case is just a single word and the goal is to assign probabilities for other words to occur within the target's window.



The CBOW architecture predicts the current word based on the context, and the Skip-gram predicts surrounding words given the current word.

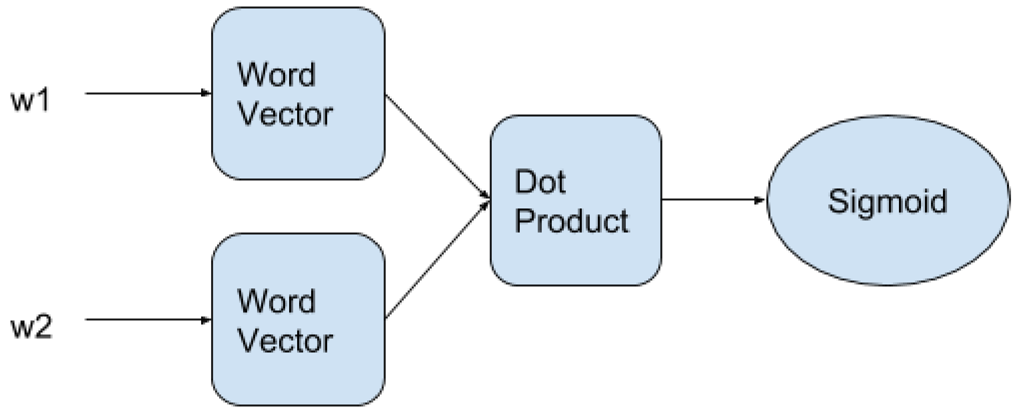
Another popular technique is the Global Vectors for Word Representation (GloVe) introduced in Pennington et al. (2014). It is based on word-word co-occurrence statistics from a corpus. The main idea behind the model is that the ratio of word occurrences have the potential to capture meaning. The training objective of GloVe is to learn word vectors such that their dot product equals the logarithm of the words' probability of co-occurrence.

Among the above two techniques, I first tried to construct vectors using the GloVe algorithm as it's simpler in principle. However, I was having difficulty setting up the model in keras and given that the computational costs would be high, I didn't think it was prudent to build a model without using a powerful backend like tensorflow. I eventually went with the skip-gram model because keras has useful wrapper functions to make building the model a little easier.

Architecture Details

The type of skip-gram model I used was the one which was later published by the original authors in the paper Mikolov et al. (2013) - 2. The problem with the original model was that, for every single word in the corpus, the probabilities for getting all the other words had to be computed and it was impractical to put into practice. So an alternative proposed by the authors was to take negative samples. The fundamental idea behind this method is that a good model can distinguish between the right surrounding words and the wrong ones. The new objective now is, given a target word to predict whether a context word lies within a certain window. The output layer, which was originally a softmax in the original paper, has now been reduced to a sigmoid layer making it computationally feasible.

With negative sampling, we reduce the problem to binary classification. The positive examples tell us that the context word lies within the window of the target somewhere in the corpus. The negative examples tells us the opposite and the objective is to build a binary classifier that can correctly distinguish the context words. In the process, the word vector is continuously updated and the relationship between the words is captured.



The architecture. Note that there are no hidden layers and that the output layer is a sigmoid layer. Also the same word vector is used for both words.

The algorithm I wrote was as follows:

```
for i in range(iterations):
    randomly select word pairs of target word (w1) and context word (w2)
    get word representations for w1 and w2 using the word vector
    get the dot product of the word representations
    calculate sigmoid(dot product)
    compare with label and compute cost
    make updates to word vector
```

With the help of keras, generating the positive and negative samples was easy. This task, which would have normally taken several lines of code was done in a single line. Also, creating and updating the model is simple using their API and with tensorflow in the backend, it's able to compute gradients efficiently.

About My Code

There are three scripts that I've written namely WordVector.py, create_word_vectors.py and word_game.py. WordVector is the class I created for creating word vectors. I've also added the functionality of storing previously trained vectors as well.

For creating word vectors from a piece of text the steps are as follows:

1. Get token details and replace all words by their rank in the text. Here rank is based on frequency of word in the text.
2. Get word pairs and their respective labels using the useful keras functions.
3. Create placeholder tensors for the model and initialize the word vector.
4. Run the algorithm as described above in the 'Architecture Details'.

The client code for interacting with the user is in create_word_vectors.py. Here the user is asked to provide the path to a text file on which the user wants to build word vectors. The user has the option of using the default settings for the vector or specifying their own. Once the model has run through the iterations, the user is then asked to provide the location of an output text file to save the word vector.

Challenges

Halfway through this project, I realized I'd set myself with a huge task. Just understanding how these models worked was a challenge. Writing code to match the algorithms was another level of difficulty altogether. I trained a model on 'Emma' in the Gutenberg corpus and while it showed a few interesting properties, it was nowhere near usable for other NLP tasks. The main problem I faced was that, while these algorithms are effective at learning relationships between words, they require massive amounts of data and industrial computational resources. For example, ‘Emma’ had a vocabulary size of around 9000 and had nearly 200,000 tokens. This may seem like a lot and my computer took hours to work with this but the smallest GloVe vector, has a vocabulary size of 400,000 and was trained on 6 billion tokens. I have kept a subset of this vector in the repository which is used for the word game described below.

Can You Guess The Word?

With the functionality of my WordVector class, I was able to load the pre-trained GloVe embeddings and create a small word game. The idea behind the game is to show the user the interesting relationships that can be captured with word vectors. The property I wanted to shed light on was that of associativity. The purpose of this game is to beat the vector in a word association challenge. An example would be, king : man :: woman : ?. The correct answer is queen. One way of looking at it is, what do you get when you remove the man in king and instead put woman? The game can be played by running word_game.py. Enjoy!

References

1. Efficient Estimation of Word Representations in Vector Space
2. Distributed Representations of Words and Phrases and their Compositionality
3. GloVe: Global Vectors for Word Representation
4. Sequence Models
5. Word2Vec word embedding tutorial in Python and TensorFlow
6. Word2Vec Tutorial - The Skip-Gram Model
7. Paper Dissected: “Glove: Global Vectors for Word Representation” Explained
8. An overview of gradient descent optimization algorithms
9. Cosine similarity
10. Keras Documentation
11. StackOverflow