Tutorial 3 – word vector representations

- Introduction – Word Embedding's
- Word2Vec
- Negative sampling
- Glove

Tomer Golany – tomer.golany@gmail.com
Word Embeddings
What Are Word Embeddings?

- A word embedding is a learned representation for text where words that have the same meaning, have a similar representation.

- Word embeddings are in fact a class of techniques where individual words are represented as real-valued vectors in a predefined vector space.

- Each word is mapped to one vector and the vector values are learned in a way that resembles a neural network.

- Each word is represented by a real-valued vector, often tens or hundreds of dimensions. This is contrasted to the thousands or millions of dimensions required for sparse word representations, such as a one-hot encoding.

We learn the representation with a learning algorithm.

For example, motel and hotel.
Overview

- Word2vec is a group of related models that are used to produce word embedding's.

- Word2vec takes as its input a large corpus of text and produces a vector space, typically of several hundred dimensions, with each unique word in the corpus being assigned a corresponding vector in the space.

- Word vectors are positioned in the vector space such that words that share common contexts in the corpus are located in close proximity to one another in the space.

- We will go through the skip gram model that is used to train word2vec.
The task of the neural network will be as follows: Given a specific word in the middle of a sentence (that is, the input into the network), look at the words nearby and pick one at random.

The network is going to tell us the probability for every word in our vocabulary of being the “nearby word” that we chose.

The output probabilities are going to relate to how likely it is to find each vocabulary word nearby the input word.

We will train a neural network to do this by feeding it word pairs, found in our training documents.

“nearby word” is actually a “window size” parameter to the algorithm. Typical window size is 5, meaning 5 words behind and 5 words ahead – 10 in total.

For example, if the input word is “Soviet”, the output probabilities are going to be much higher for words like “Union” and “Russia” than for unrelated words like “watermelon” and “kangaroo”.

Intro to NLP and Deep Learning - 236605
The Skip-Gram model

The below examples shows training samples (word pairs) we would take from the sentence "The quick brown fox jumps over the lazy dog". For this example the window size is 2.

Source Text

<table>
<thead>
<tr>
<th>Source Text</th>
<th>Training Samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>The quick brown fox jumps over the lazy dog.</td>
<td>(the, quick), (the, brown)</td>
</tr>
<tr>
<td>The quick brown fox jumps over the lazy dog.</td>
<td>(quick, The), (quick, brown), (quick, fox)</td>
</tr>
<tr>
<td>The quick brown fox jumps over the lazy dog.</td>
<td>(brown, the), (brown, quick), (brown, fox), (brown, jumps)</td>
</tr>
<tr>
<td>The quick brown fox jumps over the lazy dog.</td>
<td>(fox, quick), (fox, brown), (fox, jumps), (fox, over)</td>
</tr>
</tbody>
</table>

- The network is going to learn the statistics from the number of times each pairing shows up.
- So, for example, the network is probably going to get many more training samples of ("Soviet", "Union") than it is of ("Soviet", "Sasquatch").
- When the training is finished, if you give it the word "Soviet" as input, then it will output a much higher probability for "Union" or "Russia" than it will for "Sasquatch".
Representing the data of the neural network:

- From the training documents we first build a **vocabulary** – for our example let’s say the vocabulary consists of 10,000 unique words.

- We will represent an **input word** as a **one-hot vector**. In our example, this vector will have 10,000 components (one for every word in the vocabulary). So to represent the word “human” we will place “1” in the position corresponding to the word “human”, and “0” in all the other positions.

- The **output** of the network is a single vector (in our example with 10,000 components) containing, for every word in the vocabulary, the probability that a randomly selected nearby word is that vocabulary word.
Linear activation function in the hidden layer

when you evaluate the trained network on an input word, the output vector will actually be a probability distribution (i.e., a bunch of floating point values, not a one-hot vector)

Softmax activation function at the final layer.
The hidden layer:

- The hidden layer actually determines the dimension of the word embedding presentation.
- In this example, we used 300 neurons and therefore we are going to learn word vectors with 300 features.

- Each neuron in the hidden layer will have 10,000 learning weights (as the size of the input vector)
- We can represent all the hidden layer by a weight matrix with 10,000 rows (one for every word in our vocabulary) and 300 columns (one for each neuron)

- If you look at the rows of this weight matrix, these are actually what will be our word vectors!
If you look at the rows of this weight matrix, these are actually what will be our word vectors:

So the end goal of all of this is really just to learn this hidden layer weight matrix – the output layer we’ll just toss when we’re done.
– you might be asking yourself—“That one-hot vector is almost all zeros... what’s the effect of that?”

– If you multiply a 1 x 10,000 one-hot vector by a 10,000 x 300 matrix, it will effectively just *select* the matrix row corresponding to the “1”. Here’s a small example to give you a visual:

\[
\begin{bmatrix}
    0 & 0 & 0 & 1 & 0 \\
\end{bmatrix}
\times
\begin{bmatrix}
    17 & 24 & 1 \\
    23 & 5 & 7 \\
    4 & 6 & 13 \\
    10 & 12 & 19 \\
    11 & 18 & 25 \\
\end{bmatrix}
\]

\[
= \begin{bmatrix}
    10 & 12 & 19 \\
\end{bmatrix}
\]

– This means that the hidden layer of this model is really just operating as a lookup table. *The output of the hidden layer is just the “word vector” for the input word.*
The output layer:

- The 1x300 (in our example) word vector for "ants" then gets fed to the output layer.
- The output layer is a softmax regression classifier –
  \[ \text{out}_j = \frac{e^{x^T w_j}}{\sum_{k=1}^{10,000} e^{x^T w_k}} \]

- each output neuron (one per word in our vocabulary), will produce an output between 0 and 1, and the sum of all these output values will add up to 1.

- Specifically, each output neuron has a weight vector which it multiplies against the word vector from the hidden layer, then it applies the function \( \exp(x) \) to the result.

- Finally, in order to get the outputs to sum up to 1, we divide this result by the sum of the results from all 10,000 output nodes.
Here’s an illustration of calculating the output of the output neuron for the word “car”:

Output weights for “car”

Word vector for “ants”

300 features

\[
\text{softmax} \quad \frac{e^x}{\sum e^x}
\]

= Probability that if you randomly pick a word nearby “ants”, that it is “car”
If two different words have very similar “contexts” (that is, same words are likely to appear around them), then our model needs to output very similar results for these two words.

And one way for the network to output similar context predictions for these two words is if the word vectors are similar. So, if two words have similar contexts, then our network is motivated to learn similar word vectors for these two words.

The network will likely learn similar word vectors for the words “ant” and “ants” because these should have similar contexts.
Negative Sampling
You may have noticed that the skip-gram neural network contains a huge number of weights.

For our example with 300 features and a vocab of 10,000 words, that's 3M weights in the hidden layer and output layer each!

Running gradient descent on a neural network that large is going to be slow. And to make matters worse, you need a huge amount of training data in order to tune that many weights. Millions of weights times billions of training samples means that training this model is going to be a beast.

The authors of Word2Vec addressed these issues by 3 innovations:

• Treating common word pairs or phrases as single “words” in their model.
• Subsampling frequent words to decrease the number of training examples.
• Modifying the optimization objective with a technique they called “Negative Sampling”, which causes each training sample to update only a small percentage of the model’s weights.

A word pair like “Boston Globe” (a newspaper) has a much different meaning than the individual words “Boston” and “Globe”. So it makes sense to treat “Boston Globe”, wherever it occurs in the text, as a single word with its own word vector representation.
Negative Sampling

• Training a neural network means taking a training example and adjusting all of the neuron weights slightly so that it predicts that training sample more accurately.

• In other words, each training sample will tweak all of the weights in the neural network.

• As we discussed above, the size of our word vocabulary means that our skip-gram neural network has a tremendous number of weights, all of which would be updated slightly by every one of our billions of training samples!

• Negative sampling addresses this by having each training sample only modify a small percentage of the weights, rather than all of them.

• When training the network on the word pair (“fox”, “quick”), recall that the “label” or “correct output” of the network is a one-hot vector. That is, for the output neuron corresponding to “quick” to output a 1, and for all of the other thousands of output neurons to output a 0.
With negative sampling, we are instead going to randomly select just a small number of “negative” words (let’s say 5) to update the weights for.

In this context, a “negative” word is one for which we want the network to output a 0 for.

We will also still update the weights for our “positive” word (which is the word “quick” in our current example).

Recall that the output layer of our model has a weight matrix that’s 300 x 10,000.

So we will just be updating the weights for our positive word (“quick”), plus the weights for 5 other words that we want to output 0. That’s a total of 6 output neurons, and 1,800 weight values total.

That’s only 0.06% of the 3M weights in the output layer!

In the hidden layer, only the weights for the input word are updated (this is true whether you’re using Negative Sampling or not).
The “negative samples” (that is, the 5 output words that we’ll train to output 0) are chosen using a “unigram distribution”.

Essentially, the probability for selecting a word as a negative sample is related to its frequency, with more frequent words being more likely to be selected as negative samples.

Each word is given a weight equal to it’s frequency (word count) raised to the 3/4 power. The probability for selecting a word is just it’s weight divided by the sum of weights for all words:

$$p(w_i) = \frac{f(w_i)^{4/3}}{\sum_{j=0}^{n}(f(w_j)^{4/3})}$$

The decision to raise the frequency to the 3/4 power appears to be empirical. in the paper they say it outperformed other functions.

The power makes less frequent words be sampled more often. (see lecture slides for an example)
GloVe is another algorithm for obtaining vector representations.

The GloVe algorithm consists of the following steps:

1. Collect word co-occurrence statistics in a form of word co-occurrence matrix $A$. Each element $A_{ij}$ in the matrix represents how often word $i$ appears in context of word $j$. 

![Window based cooccurrence matrix](image)
GloVe algorithm steps

• Algorithm main steps:

2. Define soft constraints for each word pair:
   \[ w^T_i w_j + b_i + b_j = \log(A_{ij}) \]
   - \( w(i) \) - vector for the main word. \( w(j) \) – vector for the context word. \( b(i), b(j) \) are scalar biases for the main and context words.

3. Define a cost function:
   \[ J = \sum_{i=1}^{V} \sum_{j=1}^{V} f(A_{ij})(w^T_i w_j + b_i + b_j - \log(A_{ij}))^2 \]
   - \( f \) is a weighting function which help us to prevent learning only from the extremely common word pairs:
   \[
   f(A_{ij}) = \begin{cases} 
   \left( \frac{A_{ij}}{A_{\text{max}}} \right)^\alpha & \text{if } A_{ij} < A_{\text{max}} \\
   1 & \text{otherwise}
   \end{cases}
   \]
References:

- http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/
- https://cran.r-project.org/web/packages/text2vec/vignettes/glove.html
- https://nlp.stanford.edu/projects/glove/

- Glove implementation in C:
  - https://github.com/stanfordnlp/GloVe

- Glove paper: