wordtovec_writeup

December 11, 2019

1 Trump/Word2Vec

We trained a similar neural network with Trump's tweets to that used to create Word2Vec. Word2vec is a group of models that are used to generate word embeddings. These models are, two-layer neural networks that are trained to reconstruct linguistic contexts of words. Word2vec takes as its input a corpus of text, in our case Trumps tweets, and reduces the dimensionality to 100-dimensional vectors. These vectors retain semantic meaning as we'll see!

I'm adapting code from a course in Udacity to our problem. The original code is here: https://github.com/udacity/deep-learning-v2-pytorch/tree/master/word2vec-embeddings

2 Readings and Videos

- An easy introduction to Recurrent Neural Networks: https://www.youtube.com/watch?v=UNmqTiOnRfg
- A nice lecture on LSTM, which is the type of RNN used in this algorithm: https://www.youtube.com/watch?v=iX5V1WpxxkY&t=1s
- A really good conceptual overview of Word2Vec from Chris McCormick
- First Word2Vec paper from Mikolov et al.
- Neural Information Processing Systems, paper with improvements for Word2Vec also from Mikolov et al. This explains their choice for subsampling and negative sampling, which makes the algorithm run much faster.
- Visualizing the dataset: https://towardsdatascience.com/google-news-and-leo-tolstoy-visualizing-word2vec-word-embeddings-with-t-sne-11558d8bd4d

3 Word embeddings

Embeddings are a a fully connected layer inside a neural network, which takes as input a one-hot-encoded vocabulary, and reduces the dimensionality of the words. We call this layer the embedding layer. We can use this layer, which has as dimensions the number of words in your vocabulary vs a chosen dimension, (in our case 100), as a lookup table. This lookup table is trained just like you would train any layer in a neural network.

Word2Vec trains by looking at each words surrounding context. You feed the network a word and a set of surrounding words. Words that show up in similar contexts, such as "coffee", "tea", and

"water" will have vectors near each other. Different words will be further away from one another, and relationships can be represented by distance in vector space.

3.1 Subsampling

- This part is a suggestion in Google's paper http://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdfText to speed up and improve the algorithm,
- Some words such as "the", "or", etc appear very often and don't provide much context for neighboring words. By discarding some of them, we can train our network faster and get better results. At the same time, you don't want to discard all of them, as they do provide information about the syntax of tweets. For each word w_i in the training set, we'll discard it with probability given by :

$$P(w_i) = 1 - \sqrt{\frac{t}{f(w_i)}}$$

Here, t is a threshold parameter and $f(w_i)$ is the frequency of word w_i in the total dataset. Note that $P(w_i)$ is the probability that a word is discarded.

For each word in the text, we want to grab all the words in a window around that word, with size C.

3.2 Batches

From Mikolov et al.:

"Since the more distant words are usually less related to the current word than those close to it, we give less weight to the distant words by sampling less from those words in our training examples… If we choose $C = 7$, for each training word we will select randomly a number R in range $[1 : C]$, and then use R words from history and R words from the future of the current word as correct labels."

3.3 Log-loss function through negative sampling

A typical way to train this model would be to look at an output softmax layer, to assign probabilities to words near our input word. This can take a long time as we calculate and backpropagate through thousands of words. We can approximate the loss from the softmax layer by only updating a small subset of all the weights at once. We'll update the weights for the correct example, but only a small number of incorrect, or noise, examples. This is called "negative sampling".

There are two modifications we need to make. First, since we're not taking the softmax output over all the words, we're really only concerned with one output word at a time. Similar to how we use an embedding table to map the input word to the hidden layer, we can now use another embedding table to map the hidden layer to the output word. Now we have two embedding layers, one for input words and one for output words. Secondly, we use a modified loss function where we only care about the true example and a small subset of noise examples.

$$-\log\sigma\left({u_{w_O}}^\top v_{w_I}\right) - \sum_i^N \mathbb{E}_{w_i\sim P_n(w)} \log\sigma\left(-{u_{w_i}}^\top v_{w_I}\right)$$

The first term says we take the log-sigmoid of the inner product of the output word vector and the input word vector.

The second term says we're going to take a sum over words w_i drawn from a noise distribution $w_i \sim P_n(w)$. The noise distribution is all those words in Trump's vocabulary that aren't in the context of the input word. We can randomly sample words from our vocabulary to get these. We can choose $P_n(w)$ ro be any distribution, such as a uniform distribution. We could also pick it according to the frequency that each word shows up in our vocabulary. This is called the unigram distribution $U(w)$. The authors found the best distribution to be $U(w)^{3/4}$, empirically.

Finally, in we take the log-sigmoid of the inner product of a noise vector with the input vector.

The first term in the loss function pushes the probability that our network will predict the correct word w_O towards 1. In the second term, we're pushing the probabilities of the noise words towards 0.

3.4 Word prediction

The final output of the neural network trained with a final softmax layer would give us probabilities for the sourrounding words of a particular word in Trump's vocabulary. We could train a Trump Bot that generated tweets! Unfortunately, since we used an approximation of the loss function, created specifically to obtain high quality embeddings, the resulting bot isn't of very high quality. Even though Trump's tweet corpus isn't very extensive, we attempted to train the model in a time series fashion, using the tweets we had, from 2014, to the end of every month starting on early 2017. This was a long process and having the quicker algorithm was very useful to us.