Neural Network and Word2vec Basics

Final Project Report for CS349 Natural Language Processing

Wanyi Li
Wellesley College
21 Wellesley College Road
Wellesley, MA 02481

May 14, 2016

Abstract

In this report, I carefully explain the mathematics behind artificial neural networks used for natural language processing (NLP), assuming that readers have a basic conceptual understanding of neural networks. The report touches on the topics including gradient descent, sigmoid function, softmax function and loss functions for methods like negative sampling, Continuous Bag of Words and Skip-gram. I will also describe a project using neural network to generate word vectors on two languages (English and Chinese) and discuss some of the observations.

Introduction & Background

I want to use this project as an opportunity to explore the topic of the application of artificial neural networks on natural language processing and to educate myself on this subject. Deep Learning has become a hot topic in recent years. I want to demystify the concept of "deep learning" and really try to understand the subject from scratch. My goal for this project is to understand the linear algebra behind the neural nets and how to implement words as vectors using neural nets.

When I started learning about neural nets, I did not know our class would touch the basics of neural nets. So, I will just go into the specific methods related using neural nets to build word vectors, which are continuous bag of words (CBOW) and Skip-gram. I used the Stanford Computer Science course CS224d Deep Learning for NLP as my major source. I completed a third of the course material up to assignment 1 (they have three assignments in the course). The assignment has a written component and a programming component (done in iPython Notebook). It is essential for one to complete and understand the written component first before writing any code. A lot of math in my report follows the questions asked in the written component of the assignment, mixed with the flow of my logic when I approached the topic. Though one may find the solution of the assignment online, I write out detailed steps for each derivation, not only to show my work, but also to serve as a resource for other neural nets beginners to thoroughly understand the derivations because often online resources have no detailed explanation.

Neural Network Basics

The most simple neural net has three layers, though some people call it a single layer neural net because there is only one hidden layer. Let's standardize our notation – the three layers are the *input* layer vector $\boldsymbol{x}$ with dimension D_x, the hidden layer h with dimension N, and the output layer $\hat{\boldsymbol{y}}$ with dimension D_y. Between the input and hidden layer, the nodes work as neurons which has an activation function. In class, we learned $tanh$, but here we are going to use sigmoid function $\boldsymbol{\sigma}$ which is defined to be,

$$\sigma(x) = \frac{1}{1+e^{-x}}. \tag{1}$$

Before we go further, I want us to pause here and look at two useful tricks about sigmoid function which will help us tremendously later.

$$\begin{aligned}\sigma(x)' &= \frac{d}{dx}(\frac{1}{1+e^{-x}}) = -\frac{1}{(1+e^{-x})^2}(-e^{-x}) \\ &= \frac{1}{1+e^{-x}}\frac{1+e^{-x}-1}{1+e^{-x}} = \sigma(x)(1-\sigma(x)),\end{aligned} \tag{2}$$

$$\sigma(-x) = \frac{1}{1+e^{x}} = \frac{e^{x}}{e^{x}+1} = \frac{1+e^{-x}-1}{1+e^{-x}} = 1-\sigma(x). \tag{3}$$

This sigmoid function serves as the non-linear activation to get the hidden layer,

$$\boldsymbol{h} = \text{sigmoid}(\boldsymbol{x}\boldsymbol{W_1} + \boldsymbol{b_1}), \tag{4}$$

where $\boldsymbol{W_1}$ is the weight matrix with dimension $D_x \times N$.

To get from the hidden layer to the output layer, we use the softmax function,

$$\text{softmax}(x)_i = \frac{e^{x_i}}{\sum_j e^{x_j}}. \tag{5}$$

The immediate question I raise to myself is that, softmax and sigmoid seems some what similar - why do we apply one onto the hidden layer while the other to the output layer? Softmax function normalize the whole output vector, which is essentially a probability distribution over all the components, but sigmoid function is just activation function which works similar to tanh or the logistic function that makes the hidden layer works like neurons. There is, again, a trick for the softmax function that is good to know:

$$\text{softmax}(\boldsymbol{x}) = \text{softmax}(\boldsymbol{x} + c), \tag{6}$$

to see that,

$$\frac{e^1}{e^1+e^2} = \frac{e^1}{e^1+e^2}\frac{e^2}{e^2} = \frac{e^3}{e^3+e^4}. \tag{7}$$

To generate the output layer, we use:

$$\boldsymbol{\hat{y}} = \text{softmax}(\boldsymbol{hW_2} + \boldsymbol{b_2}), \tag{8}$$

where W_2 is a matrix with dimension $N \times D_y$. With sigmoid and softmax function, we have the so-called forward propagation. After we read the output layer, we want to know how much error there is in our prediction $\boldsymbol{\hat{y}}$ from the actual labels $\boldsymbol{y}$. We call this error a loss function J or a cost functiom. The most simple loss function is cross entropy as we learnt in class which is defined as,

$$CE(\boldsymbol{y}, \boldsymbol{\hat{y}}) = -\sum_i y_i log(\hat{y}_i). \tag{9}$$

This may not be obvious, but y is an one-hot vector which is only one of its component is 1 and the rest is 0 - of course, because only one of the word is the correct work and the rest are all wrong. This really simplifies our calculation later on because we can take off the sum:

$$CE(\boldsymbol{y}, \boldsymbol{\hat{y}}) = -log(\boldsymbol{\hat{y}}). \tag{10}$$

The goal is to minimize the error, which is the loss function. There are different ways to minimize the error. The most efficient way is to do gradient descent on loss function. There are two ways to do gradient descent: numerical and analytical. With some knowledge from numerical analysis, we know that numerical solution is slow ($O(n)$ while $n \approx 10^6$ if the vocabulary size is that much), approximate but easy, yet analytical solution is fast, exact but error-prone. So, we want to implement a gradient check method to make sure the analytical solution is good. In calculus, we have learnt

$$\frac{df}{dx} = \lim_{h\to 0}\frac{f(x+h)-f(x)}{h}. \tag{11}$$

If we set h really small (on the order of 10^{-4}), then we can implement the numerical gradient check. However, in reality, it is better to implement the following "centered different formula":

$$\frac{df}{dx} = \lim_{h\to 0}\frac{f(x+h)-f(x-h)}{2h}. \tag{12}$$

Now, we will go back to analytic gradient check where we will derive the gradient of the loss functions in details.

Start with deriving the gradient with regard to the inputs of a softmax function when cross entropy loss is used for evaluation. Denote the softmax input vector θ and the prediction is made by $\boldsymbol{\hat{y}} = \text{softmax}(\boldsymbol{\theta})$. Remember that $\boldsymbol{y}$ is a one-hot vector, say oth component is the correct word, so we have,

$$\frac{\delta CE(\boldsymbol{y}, \boldsymbol{\hat{y}})}{\delta\theta_o} = -\frac{1}{\hat{y}_o}\frac{d\hat{y}_o}{d\theta_o} = -\frac{\hat{y}_o(1-\hat{y}_o)}{\hat{y}_o} = \hat{y}_o - 1. \tag{13}$$

This expression is equivalent with

$$\frac{\delta CE(\boldsymbol{y}, \boldsymbol{\hat{y}})}{\delta \boldsymbol{\theta}} = \boldsymbol{\hat{y}} - \boldsymbol{y} \tag{14}$$

because when $i \neq o$, $y_i = 0$.

Now let's derive the gradients with respect to the *inputs* $\boldsymbol{x}$ to an one-hidden-layer neural network, which is to find $\frac{dJ}{d\boldsymbol{x}}$ where J is the cost function for the neural network.

$$\begin{aligned}\frac{\delta J}{\delta \boldsymbol{x}} =& \frac{\delta CE}{\delta(\boldsymbol{h}\boldsymbol{W_2} + \boldsymbol{b_2})} \frac{\delta(\boldsymbol{h}\boldsymbol{W_2} + \boldsymbol{b_2})}{\delta \boldsymbol{h}} \frac{\delta \boldsymbol{h}}{\delta(\boldsymbol{x}\boldsymbol{W_1} + \boldsymbol{b_1})} \\ & \frac{\delta(\boldsymbol{x}\boldsymbol{W_1} + \boldsymbol{b_1})}{\delta \boldsymbol{x}} (\boldsymbol{\hat{y}} - \boldsymbol{y})\boldsymbol{W_2^T}\boldsymbol{h}(1 - \boldsymbol{h})\boldsymbol{W_1^T}.\end{aligned} \tag{15}$$

Notice that there are $(D_x + 1) \cdot H + (H + 1) \cdot (D_y + 1)$ parameters in this neural network assuming the input is D_x-dimensional, the output is D_y-dimensional and there are H hidden units.

Word2vec

Now, let's talk specifically about applying neural nets on natural language processing. In class, we have learnt language models like N-gram. Here, we will talk about two models: continuous bag of words (CBOW) and skip gram. The idea for CBOW is simple: there is a context window and we want to predict the center word. Skip-gram is like the opposite of CBOW – the model should be able to predict the surrounding word given the center word (the center word becomes the "context" word). These two models are basically the same, but we switch the $\boldsymbol{x}$ and $\boldsymbol{y}$ in both models. In CBOW, the output vector is an one-hot vector but in Skip-gram, the input vector is an one-hot vector (just one word).

Assume given a predicted word vector $\boldsymbol{v_c}$ corresponding to the center word $\boldsymbol{c}$ for skip-gram, and word prediction is made with the softmax function defined as,

$$\hat{y}_o = p(o|c) = \frac{exp(\boldsymbol{u_o^T}\boldsymbol{v_c})}{\sum_{w=1}^{W} exp(\boldsymbol{u_w^T}\boldsymbol{v_c})}, \tag{16}$$

where $\boldsymbol{o}$ is the expected word where the one-hot vector is one. Use cross entropy as the loss function, to minimize the loss function, let's derive its gradients with respect to $\boldsymbol{v_c}$ and then

$\boldsymbol{U} = [\boldsymbol{u_1}, \boldsymbol{u_2}, \ldots \boldsymbol{u_w}]$ which is the matrix of all the output vectors. When $i = o$.

$$\begin{aligned}
\frac{\delta J}{\delta \boldsymbol{v_c}} &= \frac{\delta}{\delta \boldsymbol{v_c}}(-log(\hat{y}_i)) \\
&= \frac{\delta}{\delta \boldsymbol{v_c}}(-(\boldsymbol{u_i^T} \boldsymbol{v_c} - log \sum_{w=1}^{W} exp(\boldsymbol{u_w^T} \boldsymbol{v_c}))) \\
&= -\boldsymbol{u_i^T} + \frac{\sum_{w=1}^{W} \boldsymbol{u_w^T} \cdot exp(\boldsymbol{u_w^T} \boldsymbol{v_c})}{\sum_{w=1}^{W} exp(\boldsymbol{u_w^T} \boldsymbol{v_c})} \\
&= -\boldsymbol{u_i^T} + \sum_{w=1}^{W} \boldsymbol{u_w^T} \hat{y}_w \\
&= \boldsymbol{U^T}(\boldsymbol{\hat{y}} - \boldsymbol{y}).
\end{aligned} \tag{17}$$

$$\begin{aligned}
\frac{\delta J}{\delta \boldsymbol{u_{w \neq o}}} &= 0 + \frac{\sum_{w=1}^{W} \boldsymbol{v_c} \cdot exp(\boldsymbol{u_w^T} \boldsymbol{v_c})}{\sum_{w=1}^{W} exp(\boldsymbol{u_w^T} \boldsymbol{v_c})} \\
&= \sum_{w=1}^{W} \boldsymbol{v_c} \hat{y}_w,
\end{aligned} \tag{18}$$

$$\frac{\delta J}{\delta \boldsymbol{u_{w=o}}} = -\boldsymbol{v_c} + \sum_{w=1}^{W} \boldsymbol{v_c} \hat{y}_w = \boldsymbol{v_c}(\hat{y}_o - 1). \tag{19}$$

Together, we have

$$\frac{\delta J}{\delta \boldsymbol{U}} = \boldsymbol{v_c}(\boldsymbol{\hat{y}_o} - \boldsymbol{y})^T. \tag{20}$$

Now, let's pause and see if this analytic gradient descent is computationally efficient or not. The sum has W terms which is the size of the vocabulary which is on the order of million. Is there anything we can do to fasten this process? Here, we will introduce the method of negative sampling. For every training step, instead of looping through the whole vocabulary, we take a few wrong samples from a noise distribution $P_n(w)$ which matches the frequency of the vocabulary. In mundane words, if the correct sentence is "I want to stay inside my house all day" and we are trying to predict the word "inside , we feed into some random wrong sentences and learn them as a negative sample. For example, "I want to stay cat my house all day". To fully implement this method into word2vec, we also need to do gradient descent on the negative sampling loss function, defined as,

$$J_{neg}(\boldsymbol{o}, \boldsymbol{v_c}, \boldsymbol{U}) = -log(\sigma(\boldsymbol{u_o^T} \boldsymbol{v_c})) - \sum_{k=1}^{K} log(\sigma(-\boldsymbol{u_k^T} \boldsymbol{v_c})). \tag{21}$$

Notice, now we are just taking K negative samples which will fasten the process. Again, let's derive its gradients against the predicted word vector $\boldsymbol{v_c}$ and the output vector $\boldsymbol{u_k}$

and $\boldsymbol{u_o}$. (Two important tips: 1. Don't forget to use the tricks about sigmoid functions I mentioned earlier in the paper; 2. $k \neq o$ because they are "negative" samples.)

$$\begin{aligned}\frac{\delta J}{\delta \boldsymbol{v_c}} &= -\frac{1}{\sigma(\boldsymbol{u_o^T v_c})}\frac{\delta\sigma(\boldsymbol{u_o^T v_c})}{\delta v_c} - \sum_{k=1}^{K}(1-\sigma(\boldsymbol{u_o^T v_c}))(-\boldsymbol{u_k}) \\ &= -(1-\sigma(\boldsymbol{u_o^T v_c}))\boldsymbol{u_o} - \sum_{k=1}^{K}(\sigma(\boldsymbol{u_o^T v_c})-1)\boldsymbol{u_k},\end{aligned} \tag{22}$$

$$\frac{\delta J}{\delta \boldsymbol{u_o}} = -(1-\sigma(\boldsymbol{u_o^T v_c}))\boldsymbol{v_c}, \tag{23}$$

$$\frac{\delta J}{\delta \boldsymbol{u_k}} = -(\sigma(-\boldsymbol{u_k^T v_c})-1)\boldsymbol{v_c}. \tag{24}$$

So far, we talked about two cost functions and their gradients from a single input vector and a single output vector (softmax with cross entropy and negative sampling). However, the two language models (CBOW and skip-gram) handle more than a single input/output vector. To simplify and to generalize the notation, let's call $J_{softmax-CE}(\boldsymbol{o}, \boldsymbol{v_c}, \boldsymbol{U})$ or $J_{negative-sampling}(\boldsymbol{o}, \boldsymbol{v_c}, \boldsymbol{U})$ as $F(\boldsymbol{o}, \boldsymbol{v_c})$. Skip-gram takes in one word and tries to predict the surrounding word window, so its cost function looks like,

$$J_{skip}(w_{c-m\ldots c+m}) = \sum_{-m\leq j\leq m, j\neq 0} F(\boldsymbol{w_{c+j}}, \boldsymbol{v_c}). \tag{25}$$

Recall that CBOW takes a context window and output one center word prediction, so the input vector $\boldsymbol{v_c}$ needs to become the sum of the window word vectors, denoted as $\hat{\boldsymbol{v}}$:

$$\hat{\boldsymbol{v}} = \sum_{-m\leq j\leq m, j\neq 0} \boldsymbol{v_{c+j}}. \tag{26}$$

Its loss function will just be F itself,

$$J_{CBOW}(w_{c-m\ldots c+m}) = F(\boldsymbol{w_c}, \boldsymbol{v_c}). \tag{27}$$

Now we have implemented gradient descents for all the models, but gradient descent is still very expensive to implement. There is a more efficient version of gradient descent called mini batch gradient descent. As it is called, we only run gradient descent on a small batch of samples. In the Stanford CS224d course assignment, stochastic gradient descent (SGD) is implemented. SGD is an extension or a more extreme version of mini-batch gradient descent. Instead of taking a small amount of sample, SGD just only sample one single file. One can read the specific implementation of SGD in the iPython notebook of the assignment in my GitHub repository.

Word2vec on English and Chinese Wikipedia

In this section, I want to talk about my effort on applying neural nets on two different languages. Though most of my effort is spent on learning the theoretical aspects of neural nets, I do want to apply it and see how the results varies from word vectors using linear methods (for example, singular value decomposition for dimension reduction). Before going into my observation, I want to talk about my motivation for this side project and the tools that I have used because I believe this will be able to serve as a collections of resources for the audience to use as well.

I used Google's project word2vec to do the machine learning of neural networks. It's based in C. Conveniently, there is a GitHub repository that is a Python interface to Google word2vec where training is done through C while other functionality is based on numpy, etc. I got my data from wikipedia dump – please see reference & resources section for the link to download the data. The data format is in raw XML from wikipedia dump, so I had to use to a package called Wikiextractor to generate plain text and discard other information in the XML files.

When we map English word vectors, we can see things like clusters of words with similar meanings, or structures like "women" - "girl" = "men" - "boy". Will we find similar structures in the mappings across different languages? If we visualize word vectors in two dimensions for different languages, after some appropriate phase shift and projection, will we be able to find words with similar meanings in the similar positions? I found this paper written by folks at CMU on this subject. They used English, Spanish, German and French which are all Western languages. Chinese is a very different language as in it is not based in word, but rather in character. One character can translate into one English word, or more often, two characters together can translate into one English word. How would the character vector space look like and how does it differ from English?

The training for English Wikipedia took about 5 hours and for Chinese was much faster. The vocabulary size for English was 1832169 and the word vector dimensions was 100. The Chinese vocabulary size was much smalled, 14984 only, because they are just characters and combination of characters makes up as words.

I realise that this project was too ambitious, so I did not have time to produce much interesting results. In the iPython notebook 'trained_models.ipynb', I will just give a few examples of sets of closest word vectors to the English/Chinese words that share the same meaning, and we can see how their closest words differ.

Conclusion

This project gave me a great opportunity to explore my intellectual interest and learn the linear algebra behind the neural networks. Though I spent a lot of time watching the videos lectures and reading notes, it was not until I derived the results step by step and translate the math into code, that I actually begin to understand. This has been a truly rewarding experience!

References & Resources

Stanford CS224d Deep Learning for NLP course site;
Deep Learning for Java (great online resource for reading);
Google's word2vec in C;
Python's wrapper for Google's word2vec;
Chinese Wikipedia dump;
Wikipedia Extractor;
Improve Vector Space Word Representation Using Multilingual Correlation.

Acknowledgments

Thanks Sravana for guiding me to the right resources for this project and provide me with the English Wikipedia data! It has been an amazing and very rewarding class.