OPTIMIZATION OF SKIP-GRAM MODEL

Jishen Yin
Department of Statistical Sciences
Duke University
Durham, NC, US
jishen.yin@duke.edu

Chenxi Wu
Department of Statistical Sciences
Duke University
Durham, NC, US
chenxi.wu@duke.edu

May 2, 2020

ABSTRACT

This paper explores Skip-gram model in word2vec and two training algorithms for learning distributed vector representations of words, namely hierarchical softmax and negative sampling. We also implemented the model in both Python NumPy and PyTorch, which achieved good performance. The paper is divided into five parts: The first part is the introduction of the intuition and history of word2vec method. The second part focuses on the derivation of Skip-gram model which is one of the most efficient method for learning word representations that are good at predicting surrounding words. The third and fourth part illustrate hierarchical softmax and negative sampling algoritm. Lastly, we conducted two empirical experiments on the trained word vectors and evaluated the result.

***K*eywords** Word Embedding · Skip-gram · Hierarchical Softmax · Negative Sampling

1 Introduction

Word2Vec is an unsupervised NLP tool developed by Google in 2013. The method assigns a condensed vector to word so that the relationship between words can be quantified. Word2Vec is based on the statistical assumption that word can be predicted from its context. It mainly contains two models, Skip-gram model and continuous bag of words (CBOW) model, and two efficient methods of training, hierarchical softmax and negative sampling.

The essence of the method is word embedding, which maps the word into a high-dimensional vector space. Previously researchers used one-hot representation of words in which the length of each word vector equals to the size of the entire vocabulary. Each vector contains only one '1' which is the position of the word and '0' everywhere else. This method is extremely inefficient especially when the corpus is large. The distributed representation of word vectors solves this problem. By training the one-hot representation, we map each word into a shorter vector whose size can be determined by ourselves. The condensed representation of words are not only efficient, but also captures many linguistic regularities and patterns. Mikolov et al.(2013) pointed out that many linguistic patterns can be represented as linear translations. That being said, simple addition of word vectors can produce meaningful results. For example, vec("Russia") + vec("river") is close to vec("Volga River"), and vec("Germany") + vec("capital") is close to vec("Berlin"). But a natural question would be, how to train these vectors? A traditional method is to use neural networks which requires dense matrix multiplications. Mikolov et al. (20) introduced Skip-gram model which makes the training process much more efficient.

2 Skip-gram Model

The Skip-gram model aims at training word representations that are good at predicting surrounding words. Suppose we have a sequence of training words $w_1, w_2, ..., w_T$. Given the representation of the input word $w_{I,t}$, the model is expected to output the word representations of its nearby words $w_{O,t-c}, ..., w_{O,t+c}$. The window size $2c$, which is the number of surrounding words we want to predict, can be determined on our own. Specifically, we want to obtain the softmax probability of every word in the corpus and the nearby words should have the highest probabilities among

them all. Inversely (given these contextual words), we can train the vector representations of the input word as well as parameters within the model.

Softmax function is a widely-used activation function in multi-classification machine learning problem, which outputs a probability distribution for the input numbers. The basic softmax probability is defined as follows:

$$p(w_O \mid w_I) = \frac{exp(v^{'}_{w_O} \top v_{w_I})}{\sum_{w=1}^{W} exp(v^{'}_{w_O}) \top v_{w_I})} \tag{1}$$

where w_O and w_I are output word, input word respectively. And v_w, $v^{'}_w$ are the "input" and "output" vector representations of w. W is the number of unique words in the corpus. This formula is inefficient because the cost of computing $\bigtriangledown p(w_O \mid w_I)$ is proportional to W, which is often large. Thus, hierarchical softmax is proposed to substitute the full softmax.

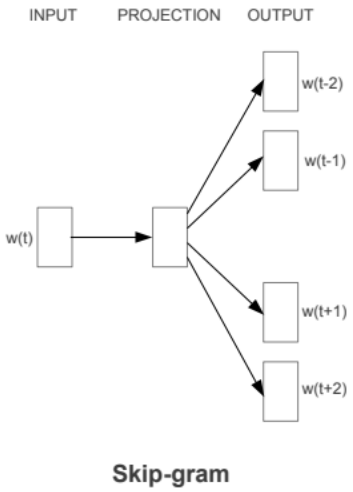

Figure 1: Schematic Diagram of Skip-gram Model

3 Hierarchical Softmax

The traditional neural network word vector representation model takes one-hot representation of words as input, via embedding layer and a hidden layer to the output layer where softmax probability is computed. The method requires us to compute softmax probability of every word, which is often inefficient. To make the training process more computational efficient, we apply hierarchical softmax which uses a binary tree representation of the hidden and output layer. The structure of the tree has a considerable effect on the training performance. In this paper we use Huffman binary tree in which each leaf of the tree corresponds to a word in the corpus. And each internal node explicitly represents the relative probabilities of its child nodes. In Huffman tree, more frequent words are placed closer to the root so that they have shorter "route code". We define the Huffman code of nodes as 1 if it is a left child node and 0 if it is a right child node. This way for each word, we will have a sequence of codes indicating the route from the root to this word.

3.1 Huffman Tree

The process of building a Huffman tree for the corpus is summarized below:

Input: n weights $f_1, f_2, ..., f_n$ (The frequency of each word in the corpus)
Output: The corresponding Huffman tree

(1) Treat $f_1, f_2, ..., f_n$ as a forest with n trees (Each tree has only one node);
(2) In the forest, select the two trees with the smallest weights to merge as the left and right subtrees of a new tree. And the weight of the root node of this new tree is the sum of the weights of the left and right child nodes;

(3) Delete the two selected trees from the forest and add the new trees to the forest;
(4) Repeat steps (2) and (3) until there is only one tree left in the forest, and the tree is the Huffman tree that is sought.

For example, suppose we have a corpus that contains eight words: "the", "a", "model", "chain", "markov", "jib", "cuba" and "hierarchical", sorted by descendent frequency. Following the algorithm above, we will get the Huffman tree as Figure 2.

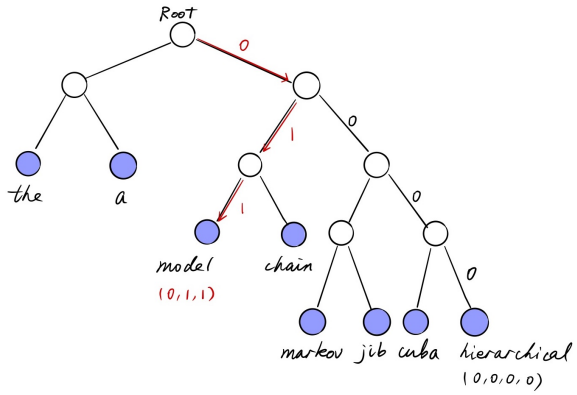

Figure 2: Huffman Tree of Eight Words

The leaves are denoted by purple circle, each represents a word in the corpus. We can see that more frequent words are closer to the root with a shorter route code. Rare words are put further with longer code, like the word "hierarchical" with code "0000". With such tree, we can arrive at any word in the corpus from the root. And the softmax probability can be computed along the tree.

3.2 Derivation

As an improvement of neural network, hierarchical softmax model adopts a projection layer to replace the hidden layer in neural nets, which simply takes the average of all input vectors. In skip-gram model where the input layer contains just one vector for the given word, the projection layer is simply the input layer. The second improvement is that instead of computing the softmax probability of all words, the model applies a binary tree to represent the output layer. So the softmax probability can be computed along the tree. Take Figure 2 as an example, to compute the probability of the word "model" given the word "hierarchical", we navigate along the route denoted by the red arrow in Figure 2. So $Pr(\text{"}model\text{"} \mid \text{"}hierarchical\text{"})$ can be calculated as the product of probabilities of all nodes along the route. See detail below:

Consider a given word w in the corpus, we denote:

- x_w: the vector representation after projection layer of word w;
- r^w: the path from root to word w;
- l^w: the number of nodes along r^w(including root node and leaf node);
- $n(w,1),...,n(w,l_w-1)$: the nodes along r^w
- $\theta_1^w, \theta_2^w, ..., \theta_{l_w-1}^w$: the vector representation of non-leaf nodes along r^w;
- $d_2^w, d_3^w, ..., d_{l_w}^w \in \{0,1\}$: Huffman code for word w (the root node does not have Huffman code)

As mentioned above, every internal node j has a Huffman code d_j^w to denote if it is a left child node or right child node. The path from root to word "model" is equivalent to a 3-step binary classification problem. Then we need to classify each non-leaf node along the path as "go left" or "go right", in which Huffman code could be of use. We use sigmoid function to decide whether to go left (+) or go right (-).

$$Pr(+) = \sigma(x_w^T\theta) = \frac{1}{1+e^{x_w^T\theta_i}}$$

$$Pr(-) = 1 - Pr(+)$$

In the example above, w is "hierarchical".

$$Pr(\text{"model"} \mid \text{"hierarchical"}) = \prod_{i=1}^{3} P(n(w,i)) = (1 - \frac{1}{1+e^{x_w^T\theta_1}})(\frac{1}{1+e^{x_w^T\theta_2}})(\frac{1}{1+e^{x_w^T\theta_3}})$$

We can see that the factor that influence the probability to go left or right is node parameter θ and current word vector x_w. In this example our expected output is word "model", so we would train θ and x_w to maximize the likelihood $Pr(\text{"model"} \mid \text{"hierarchical"})$.

Generally speaking, Let u be the surrounding words of w, i.e. $u \in Context(w)$. For each train sample (w, u), our goal is to maximize the likelihood

$$\prod_{u \in Context(w)} Pr(u|w)$$

For one sample (w, u),

$$Pr(u|w) = \prod_{j=2}^{l^u} p(d_j^u | x_w, \theta_{j-1}^u) = \prod_{j=2}^{l^u} [\sigma(x_w^T \theta_{j-1}^u)]^{1-d_j^u} [1 - \sigma(x_w^T \theta_{j-1}^u)]^{d_j^u}$$

Plugging in,

$$\prod_{u \in Context(w)} Pr(u|w) = \prod_{u \in Context(w)} \prod_{j=2}^{l^u} [\sigma(x_w^T \theta_{j-1}^u)]^{1-d_j^u} [1 - \sigma(x_w^T \theta_{j-1}^u)]^{d_j^u}$$

Putting everything together, for $w \in$ Corpus C

$$\mathcal{L} = \sum_{w \in C} log \prod_{u \in Context(w)} Pr(u|w) = \sum_{w \in C} \sum_{u \in Context(w)} \sum_{j=2}^{l^u} \{(1 - d_j^u) log[\sigma(x_w^T \theta_{j-1}^u])] + d_j^u log[1 - \sigma(x_w^T \theta_{j-1}^u)]\}$$

Write the likelihood for every training sample (w, u) as

$$L(w, u, j) = (1 - d_j^u) log[\sigma(x_w^T \theta_{j-1}^u])] + d_j^u log[1 - \sigma(x_w^T \theta_{j-1}^u)]$$

For a given word w, u d_j^u and are fixed. Compute the derivative with regard to x_w and θ_{j-1}^u

$$\frac{\partial L(w, u, j)}{\partial \theta_{j-1}^u} = [1 - d_j^u - \sigma(x_w^T \theta_{j-1}^u)] x_w$$

Applied linear search, which is $x_{k+1} = x_k + \eta \nabla f(x_k)$

We can obtain the update function of θ_{j-1}^u

$$\theta_{j-1}^u = \theta_{j-1}^u + \eta [1 - d_j^u - \sigma(x_w^T \theta_{j-1}^u)] x_w$$

In a similar manner,

$$\frac{\partial L(w, u, j)}{\partial x_w} = [1 - d_j^u - \sigma(x_w^T \theta_{j-1}^u)] \theta_{j-1}^u$$

$$x_w = x_w + \eta [1 - d_j^u - \sigma(x_w^T \theta_{j-1}^u)] \theta_{j-1}^u$$

4 Negative Sampling

Although we use Huffman tree instead of traditional neural network to improve the efficiency of model training, if the central word w in our training sample is a very hidden word, then we need to go down along the Huffman tree a lot because the the rarer the word, the further it is away from the root. Negative sampling is an alternative method proposed to solve this problem.

In negative sampling, for a given word w, there are $2c$ surrounding words that is in the context of w. We randomly select one word u from its surrounding words, so u and w compose one "positive sample". The negative sample would be to use this same u, we randomly choose a word from the dictionary that is not w. For example, in the sentence "I hope the corona virus will disappear soon", we choose "virus" as the target word and "corona" as its context word. So "corona" and "virus" is a positive sample which should be marked as 1. And any word except for "virus" would form a negative sample with "corona", like "sample", "book", "of" , etc. We choose neg number of words to form negative samples $NEG(w)$ as indicated in Table 1 where $neg = 3$. Then we use the one positive sample and neg negative samples to run binary logistic regression to obtain the word representation for each word w_i and model parameter θ_i.

Table 1: Example Negative Samples

context	word	mark
corona	virus	1
corona	sample	0
corona	book	0
corona	of	0

4.1 Derivation

Negative sampling also adopts binary logistic regression to solve for the model parameters. Specifically, the positive sample should satisfies:

$$Pr(u, w_i) = \sigma(x_u^T \theta^{w_i}), d_i = 1, i = 0 \tag{2}$$

We expect negative samples should satisfies:

$$Pr(u, w_i) = 1 - \sigma(x_u^T \theta^{w_i}), d_i = 0, i = 1, 2, ..., neg \tag{3}$$

So the log-likelihood for one context word is:

$$L = \sum_{i=0}^{neg} d_i log(\sigma(x_u^T \theta^{w_i}) 1 + 1 - d_i log(1 - \sigma(x_u^T \theta^{w_i})) \tag{4}$$

Similar to hierarchical softmax, we can use the gradient descent method to update parameters.

4.2 Sampling Details

Intuitively, the sampling method should ensure that words with higher frequency should be easier to be included in the sample. Suppose the the size of the vocabulary is V. Map all words to a line that is length 1, where each word occupies a segment of the line. The length of line occupied each word w is defined below:

$$len(w) = \frac{count(w)^{3/4}}{\sum_{u \in C} count(w)^{3/4}} \tag{5}$$

Then we can sample negative words with probability based on $len(w)$.

5 Empirical Experiment

We implemented the above algorithms in Python NumPy. To speed up the training process, we also built a PyTorch version of negative sampling model (See source code). Word2vec is essentially a matrix factorization model. The matrix depicts the correlation between each word and the set of words in its context. The learned word vectors represent the semantics of words, and can be used for classification, clustering, and similarity calculation of words. To test the validity of our model, we trained our model i=on different data sets and conducted two types of empirical analysis in NLP, sentiment analysis and synonyms detection.

5.1 Sentiment Analysis

Using the PyTorch code of Skip-gram model, we conducted a sentiment analysis on IMDB movie reviews dataset. The dataset stores IMDB movie reviews for Sentiment Analysis. We aimed at classifying the reviews to "positive" and "negative" based on the word representations we trained.

There are 98469 reviews that are labeled "neg", "pos" or "unsup", where each category accounts for 50%, 25% and 25% of the data respectively. We used all data points for training word representations, and select only the "pos" or "neg" observations for sentiment classification. We also applied sub-sampling to discard some frequent words each iteration to improve efficiency. The training process takes about 20 minutes. Using the word vectors we obtained after training, we calculated the average of word vectors of all words in each review to obtain the "document vector" that can represent the semantic characteristic of the review. For short text classification, linear combination of the vectors corresponding to all the words in the document, as a text feature training classifier, has achieved good performance. The number of reviews after pre-possessing of the original text is $100,000$. The document vector has the same length as single word vector, resulting in a feature space of $50,000 \times 100$. Since it's a binary classification problem, we applied logistic regression to the feature space. The accuracy score for train and test set is 0.70768 and 0.70048 respectively. The confusion matrix is displayed as below:

Table 2: Confusion Matrix of Sentiment Analysis

	Positive	Negative
1	17550	7256
0	7450	17744

We also extracted the most "negative" review from the corpus – "I thought this movie was horrible. I was bored and had to use all the self control I have to not scream at the screen. Mod Squad was beyond cheesy, beyond cliche, and utterly predictable." And the most "positive" review is "Sorry to disagree with you, but I found the DKC series to be quite engaging. So much so that I invested in the SNES system and my own copies of the games. This is, mind you, almost ten years after the initial release of DKC 1. The graphics were ground-breaking for their time, the first vector graphics games for home systems. The music and characters are all memorable, and the games brought myself and my girlfriend dozens of hours of entertainment. True, the second game was better than the first, and the third was perhaps lacking the 'edge' of the second installment. But all three offered different play, and I enjoy them to this day. By the way, I'm old enough to remember when there were NO video games whatsoever (and TVs were black and white!)."

5.2 Synonyms Detection

Since Skip-gram model uses the context information of the word, the trained representations contain information about "similarities" of words. The words whose representations are similar should play similar roles in a sentence. To test this hypothesis, we applied our model on CBC Coronavirus News dataset. The dataset contains 2,755 unique news on coronavirus from CBC. The number of sentences after pre-possessing is 203,400. We trained the corpus on both our PyTorch code and "word2vec" model in gensim package, and compared the performance of the two versions with regard to extracting the most similar words of "virus", "drug", "china" and "of". The similarity between word vectors are measured by their cosine distance. (See source code)

Using the vectors trained on our PyTorch code, the top 8 similar words of "virus" are "spread", "covid", "spreading", "coronavirus", "illness", "infections", "exacerbate" and "disease"; While using gensim package, the top 8 words are 'illness', 'coronavirus', 'unknowingly', 'novel', 'disease', 'spread', 'infection' and 'hospitalizing'. We can see that the similar words detected for "virus" are similar across the two models, whereas the similar words extracted by our model make more sense.

We tested two models on more examples, although our code takes longer time to train, the accuracy of our code with regard to synonym detection are better than gensim package. The top 8 similar words of "drug" from PyTorch code are "drugs" , "treat", "shortages", "xue", "ticks", "fda", "medicine" and "treatments"; The result from Gensim package are 'drugs', 'canadacanada', 'lopinavir', 'ritonavir', 'heartburn', 'treatments', 'hydroxychloroquine', 'antiviral'. The top 8 similar words to "china" are "countries", "chinese", "mainland", "wuhan", "canada", "iran", "surging", "outbreaks" from PyTorch, and 'iran''wuhan', 'italy', 'chinas', 'mainland', 'hubei', 'dailyreuters', 'originated' from gensim package. The top 8 similar words to "of" are "the", "and", "a", "to", "in", "with", "says" and "on" from Pytorch, and 'the', 'in', 'a', 'ruhi', 'and', 'fulford', 'jest', 'stopscovid' from gensim package.

It is notable that the similar words given by both models capture the semantics of words relatively well, but not syntactic role. To capture syntax, word tagging might be useful in improving accuracy.

6 Conclusion

Natural language is a complex system for expressing meaning. In this system, words are the basic unit of meaning. Word vectors are vectors used to represent words, and can also be considered as feature vectors or representations of words. The technique of mapping words into vectors of real numbers is also called word embedding. In recent years, word embedding has gradually become the basic knowledge of natural language processing. In this project, we explored and derived all the methods in the original paper "Distributed Representations of Words and Phrases and their Compositionality" by Mikolov et.al, 2013. We also implemented the method in Python, with NumPy and PyTorch respectively. The PyTorch implementation achieves both training efficiency and accuracy.

The first experiment, sentiment classification of movie reviews, suggest that the trained vector is useful in capturing the sentiment in words. We used logistic regression for simplicity and achieved more than 70% accuracy in both training and test set. Using more complex classifier, like SVM or non-linear classifier, should achieve more accuracy. The second experiment on synonyms also obtains good results, even better than the "gensim" package in Python. But it can be seen that word2 vec has limitations in capturing the syntactic role of words, which can be a research direction of word embedding.

References

[1] Mikolov, Tomas Sutskever, Ilya Chen, Kai Corrado, G.s Dean, Jeffrey. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems*, 26. 2013.

[2] Guy Hadash, Einat Kermany, Boaz Carmeli, Ofer Lavi, George Kour, and Alon Jacovi. Estimate and replace: A novel approach to integrating deep neural networks with existing applications. *arXiv preprint arXiv:1804.09028*, 2018.