Implement Word2Vec with word order from scratch

Ankur Shrivastava
201405551

Ritesh Modi
201405518

Ruchir Sharma
201405558

Sourabh Dhanotia
201405605

ABSTRACT

This paper provides implementation details and various testing results of **Continuous Bag of Words** (CBOW) variant using **Word2Vec** model from scratch. Word2Vec is a two layer neural net that processes text. Its input is a text corpus and output is a set of vectors: feature vectors for each word in that corpus. The learnt feature vectors are evaluated on intrinsic tasks like Analogy Completion and Extrinsic Tasks like sentiment analysis.

Keywords

Deep Learning; Neural Network; Word Vectors; CBOW

1. INTRODUCTION

Word2Vec is the name given to a class of neural network models that, given an unlabelled training corpus, produce a vector for each word in the corpus that encodes its semantic information. These vectors are usefull for two main reasons.

1. We can measure the semantic similarity between two words by calculating the cosine similarity between their corresponding word vectors.

2. We can use these word vectors as features for various supervised NLP tasks such as document classification, named entity recognition, and sentiment analysis. The semantic information that is contained in these vectors make them powerful features for these tasks.

Instead of computing and storing global information about some huge dataset (which might be billions of sentences), we try to create a model that will be able to learn one iteration at a time and eventually be able to encode the probability of a word given its context. We can set up this probabilistic model of known and unknown parameters and take one training example at a time in order to learn just a little bit of information for the unknown parameters based on the input, the output of the model, and the desired output of the model. At every iteration we run our model, evaluate the errors, and follow an update rule that has some notion of penalizing the model parameters that caused the error.

In Continuous Bag of Words (CBOW) Model the approach is to treat "Hyderabad", "is", "of", "Andhra", "Pradesh" as a context and from these words, be able to predict or generate the center word "capital".

WOODSTOCK '97 El Paso, Texas USA

DOI: 10.475/123_4

2. PROBLEM

The word2vec tool takes a text corpus as input and produces the word vectors as output. It first constructs a vocabulary from the training text data and then learns vector representation of words. The resulting word vector file can be used as features in many natural language processing and machine learning applications.

There are two main learning algorithms in word2vec : continuous bag-of-words and continuous skip-gram. Both algorithms learn the representation of a word that is useful for prediction of other words in the sentence We were given the task of implementing **word2vec** with word order from scratch.

3. APPROACH

3.1 CBOW Model

Consider the Sentence:

The cat jumped over the puddle

This model treats "The", "cat", "over", "the", "puddle" as a context and from these words, it predicts center word "jumped".

First, we set up our known parameters. The known parameters in our model be the sentence represented by one-hot word vectors. The input one hot vectors or context we will represent with an $x^{(i)}$ And the output as y.

We create two matrices,$W^{(1)} \in R^{nX|V|}$ and $W^{(2)} \in R^{|V|Xn}$. Where n is an arbitrary size which defines the size of our embedding space. $W^{(1)}$ is the input word matrix such that the i-th column of $W^{(1)}$ is the n-dimensional embedded vector for word $W^{(i)}$ when it is an input to this model. We denote this nX1 vector as $u^{(i)}$. Similarly,$W^{(2)}$ is the output word matrix. The j-th row of $W^{(2)}$ is an n-dimensional embedded vector for word $W^{(j)}$ when it is an output of the model. We denote this row of $W^{(2)}$ as $v^{(j)}$.We learn two vectors for every word $w^{(i)}$ (i.e. input word vector $u^{(i)}$ and output word vector $v^{(i)}$.

We breakdown the way this model works in these steps:

1. We generate our one hot word vectors $x^{(i-C)}$, . . . , $x^{(i-1)}$, $x^{(i+1)}$, . . ,$x^{(i+C)}$

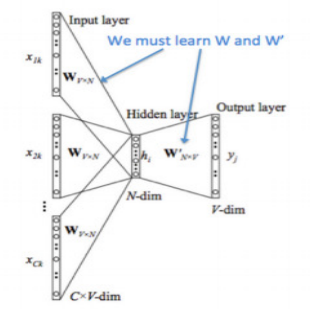

Figure 1: This image demonstrates how CBOW works and how we must learn the transfer matrices

Table 1: Skipgram with negative Sampling

Google/Self	Window	Epochs	Semantic	Syntactic
Self	5	3	12.92	5.87
Google	5	3	16.41	8.63
Self	5	1	6.85	2.53
Google	5	1	3.05	7.79

2. We get our embedded word vectors for the context $u^{(i-C)} = W^1 x^{(i-C)}$, $u^{(i-C+1)} = W^1 x^{(i-C+1)}$, . . ., $u^{(i+c)} = W^{(1)} x^{(i+C)}$ for the input context of size C. 3. Average these vectors to get

h = [u^{i-C}+u^{i-C+1}+...+u^{i+C}] / 2C

4. Generate a score vector z = $W^{(2)}$h

5. Turn the scores into probabilities y=softmax(z)

6. We desire our probabilities generated, y , to match the true probabilities, y, which also happens to be the one hot vector of the actual word.

We have experimented with window sizes of 5 and 8 , epochs from 1 to 5 , learning rate 0.001 to 0.5. Our program parses the input data , Reads the window size specified and treats the equal no of surrounding words as the context and generates word embedding accordingly in order to predict the center word from the given surrounding context.

The program reads the input data as a stream from hard drive and trains the model on the fly instead of storing the whole input matrix in-memory.This allows us to process large data inputs without being constrained by the available physical memory.

3.2 Skip-Gram Model

Another approach is to create a model such that given the center word "jumped", the model will be able to predict or generate the surrounding words "The", "cat", "over", "the", "puddle". Here we call the word "jumped" the context. We call this type of model a Skip-Gram model.

The setup is largely the same but we essentially swap our x and y i.e. x in the CBOW are now y and vice-versa. The input one hot vector (center word) we will represent with an x (since there is only one). And the output vectors as

Table 2: Continuous Bag of Words(CBOW) with Word Order

Google/Self	Window	Epochs	Semantic	Syntactic
Self	5	1	0.19	0.48
Google	5	1	0.97	0.56

$$\begin{aligned}\text{minimize } J &= -\log P(w^{(i-C)},\ldots,w^{(i-1)},w^{(i+1)},\ldots,w^{(i+C)}|w^{(i)}) \\ &= -\log \prod_{j=0,j\neq C}^{2C} P(w^{(i-C+j)}|w^{(i)}) \\ &= -\log \prod_{j=0,j\neq C}^{2C} P(v^{(i-C+j)}|u^{(i)}) \\ &= -\log \prod_{j=0,j\neq C}^{2C} \frac{\exp(v^{(i-C+j)T}h)}{\sum_{k=1}^{|V|}\exp(v^{(k)T}h)} \\ &= -\sum_{j=0,j\neq C}^{2C} v^{(i-C+j)T}h + 2C\log\sum_{k=1}^{|V|}\exp(v^{(k)T}h)\end{aligned}$$

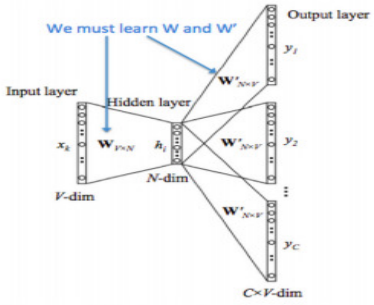

Figure 2: This image demonstrates how Skip-Gram works and how we must learn the transfer matrices

$y^{(j)}$. We define $W^{(1)}$ and $W^{(2)}$ the same as in CBOW. We breakdown the way this model works in these 6 steps:

1. We generate our one hot input vector x.
2. We get our embedded word vectors for the context $u^{(i)} = W^{(1)}$
3. Since there is no averaging, just set h = $u^{(i)}$?
4. Generate 2C score vectors, $v^{(i-C)}$, . . . , $v^{(i-1)}$, $v^{(i+1)}$, . . . ,$v^{(i+C)}$ using v=$W^{(2)}$h
5. Turn each of the scores into probabilities, y = softmax(v)
6. We desire our probability vector generated to match the true probabilities which is $y^{(i-C)}$, . . . , $y^{(i-1)}$, $y^{(i+1)}$, . . . ,$y^{(i+C)}$, the one hot vectors of the actual output.

As in CBOW, we need to generate an objective function for us to evaluate the model. A key difference here is that we invoke a Naive Bayes assumption to break out the probabilities. If you have not seen this before, then simply put, it is a strong (naive) conditional independence assumption. In other words, given the center word, all output words are completely independent.

With this objective function, we can compute the gradients with respect to the unknown parameters and at each iteration update them via Stochastic Gradient Descent.

4. RESULTS AND LEARNINGS

Maintaining word order in the sentences preserves the natural occurence of words, hence providing more context for the prediction of the center word. The algorithm proceeds by shifting a window of specified size and then training the neural network model on the basis of left and right window words for the center word prediction. Some of the key learnings of the project are: capturing context is efficient when

the window sizes are larger, but increase in window size also increases the computation and training time of the model. The learning rate, LR, applies a greater or lesser portion of the respective adjustment to the old weight. If the factor is set to a large value, then the neural network may learn more quickly, but if there is a large variability in the input set then the network may not learn very well or at all. To train a neural network well, a large amount of test data and a very low learning rate is required. Also, iterating on the data set multiple times also increases the quality of learning. But since we are limited by computation power, we could only try out with max 5 iterations starting a learning rate of 0.001. For the same, number of iterations and learning rate, our results are close to what google's word2vec implementation does.

Github Repo: https://github.com/ankur263/W2V_major

5. REFERENCES

1. http://cs224d.stanford.edu/lecture_notes/LectureNotes3.pdf.
2. https://www.youtube.com/watch?v=sU_Yu_USrNc.
3. http://tylerneylon.com/a/learn-lua/.
4. http://www.lua.org/manual/5.3/.
5. http://cs224d.stanford.edu/lecture_notes/LectureNotes1.pdf.
6. http://alexminnaar.com/word2vec-tutorial-part-ii-the-continuous-bag-of-words-model.html.
7. https://code.google.com/p/word2vec/.
8. https://github.com/torch/nn/blob/master/doc/training.md#nn.traningneuralnet.dok
9. https://github.com/torch/tutorials/tree/master/coursera_neural_nets/assignment2/solution_torch_nn
10. https://github.com/yoonkim/word2vec_torch