CS224n: Natural Language Processing with Deep Learning [1]

Lecture Notes: Part I
Word Vectors I: Introduction, SVD and Word2Vec [2]
Winter 2019

[1] Course Instructors: Christopher Manning, Richard Socher

[2] Authors: Francois Chaubard, Michael Fang, Guillaume Genthial, Rohit Mundra, Richard Socher

Keyphrases: Natural Language Processing. Word Vectors. Singular Value Decomposition. Skip-gram. Continuous Bag of Words (CBOW). Negative Sampling. Hierarchical Softmax. Word2Vec.

This set of notes begins by introducing the concept of Natural Language Processing (NLP) and the problems NLP faces today. We then move forward to discuss the concept of representing words as numeric vectors. Lastly, we discuss popular approaches to designing word vectors.

1 Introduction to Natural Language Processing

We begin with a general discussion of what is NLP.

1.1 What is so special about NLP?

What's so special about human (natural) language? Human language is a system specifically constructed to convey meaning, and is not produced by a physical manifestation of any kind. In that way, it is very different from vision or any other machine learning task.

Natural language is a discrete/symbolic/categorical system

Most words are just symbols for an extra-linguistic entity : the word is a *signifier* that maps to a *signified* (idea or thing).

For instance, the word "rocket" refers to the concept of a rocket, and by extension can designate an instance of a rocket. There are some exceptions, when we use words and letters for expressive signaling, like in "Whooompaa". On top of this, the symbols of language can be encoded in several modalities : voice, gesture, writing, etc that are transmitted via *continuous* signals to the brain, which itself appears to encode things in a continuous manner. (A lot of work in philosophy of language and linguistics has been done to conceptualize human language and distinguish words from their references, meanings, etc. Among others, see works by Wittgenstein, Frege, Russell and Mill.)

1.2 Examples of tasks

There are different levels of tasks in NLP, from speech processing to semantic interpretation and discourse processing. The goal of NLP is to be able to design algorithms to allow computers to "understand" natural language in order to perform some task. Example tasks come in varying level of difficulty:

Easy

- Spell Checking
- Keyword Search
- Finding Synonyms

Medium

- Parsing information from websites, documents, etc.

Hard

- Machine Translation (e.g. Translate Chinese text to English)
- Semantic Analysis (What is the meaning of query statement?)
- Coreference (e.g. What does "he" or "it" refer to given a document?)
- Question Answering (e.g. Answering Jeopardy questions).

1.3 How to represent words?

The first and arguably most important common denominator across all NLP tasks is how we represent words as input to any of our models. Much of the earlier NLP work that we will not cover treats words as atomic symbols. To perform well on most NLP tasks we first need to have some notion of similarity and difference between words. With word vectors, we can quite easily encode this ability in the vectors themselves (using distance measures such as Jaccard, Cosine, Euclidean, etc).

2 Word Vectors

There are an estimated 13 million tokens for the English language but are they all completely unrelated? Feline to cat, hotel to motel? I think not. Thus, we want to encode word tokens each into some vector that represents a point in some sort of "word" space. This is paramount for a number of reasons but the most intuitive reason is that perhaps there actually exists some N-dimensional space (such that $N \ll 13$ million) that is sufficient to encode all semantics of our language. Each dimension would encode some meaning that we transfer using speech. For instance, semantic dimensions might indicate tense (past vs. present vs. future), count (singular vs. plural), and gender (masculine vs. feminine).

So let's dive into our first word vector and arguably the most simple, the **one-hot vector**: Represent every word as an $\mathbb{R}^{|V| \times 1}$ vector with all 0s and one 1 at the index of that word in the sorted english language. In this notation, $|V|$ is the size of our vocabulary. Word vectors in this type of encoding would appear as the following:

One-hot vector: Represent every word as an $\mathbb{R}^{|V| \times 1}$ vector with all 0s and one 1 at the index of that word in the sorted english language.

$$w^{aardvark} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, w^{a} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, w^{at} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}, \cdots\ w^{zebra} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}$$

Fun fact: The term "one-hot" comes from digital circuit design, meaning "a group of bits among which the legal combinations of values are only those with a single high (1) bit and all the others low (0)".

We represent each word as a completely independent entity. As we previously discussed, this word representation does not give us directly any notion of similarity. For instance,

$$(w^{hotel})^T w^{motel} = (w^{hotel})^T w^{cat} = 0$$

So maybe we can try to reduce the size of this space from $\mathbb{R}^{|V|}$ to something smaller and thus find a subspace that encodes the relationships between words.

Denotational semantics: The concept of representing an idea as a symbol (a word or a one-hot vector). It is sparse and cannot capture similarity. This is a "localist" representation.

3 SVD Based Methods

For this class of methods to find word embeddings (otherwise known as word vectors), we first loop over a massive dataset and accumulate word co-occurrence counts in some form of a matrix X, and then perform Singular Value Decomposition on X to get a USV^T decomposition. We then use the rows of U as the word embeddings for all words in our dictionary. Let us discuss a few choices of X.

3.1 Word-Document Matrix

As our first attempt, we make the bold conjecture that words that are related will often appear in the same documents. For instance, "banks", "bonds", "stocks", "money", etc. are probably likely to appear together. But "banks", "octopus", "banana", and "hockey" would probably not consistently appear together. We use this fact to build a word-document matrix, X in the following manner: Loop over billions of documents and for each time word i appears in document j, we add one to entry X_{ij}. This is obviously a very large matrix ($\mathbb{R}^{|V| \times M}$) and it scales with the number of documents (M). So perhaps we can try something better.

Distributional semantics: The concept of representing the meaning of a word based on the context in which it usually appears. It is dense and can better capture similarity.

3.2 Window based Co-occurrence Matrix

The same kind of logic applies here however, the matrix X stores co-occurrences of words thereby becoming an affinity matrix. In this method we count the number of times each word appears inside a window of a particular size around the word of interest. We calculate this count for all the words in corpus. We display an example below. Let our corpus contain just three sentences and the window size be 1:

Using Word-Word Co-occurrence Matrix:

- Generate $|V| \times |V|$ co-occurrence matrix, X.
- Apply SVD on X to get $X = USV^T$.
- Select the first k columns of U to get a k-dimensional word vectors.
- $\frac{\sum_{i=1}^{k} \sigma_i}{\sum_{i=1}^{|V|} \sigma_i}$ indicates the amount of variance captured by the first k dimensions.

1. I enjoy flying.
2. I like NLP.
3. I like deep learning.

The resulting counts matrix will then be:

$$X = \begin{array}{c} \\ I \\ like \\ enjoy \\ deep \\ learning \\ NLP \\ flying \\ . \end{array} \begin{array}{c} \begin{array}{cccccccc} I & like & enjoy & deep & learning & NLP & flying & . \end{array} \\ \begin{bmatrix} 0 & 2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

3.3 Applying SVD to the cooccurrence matrix

We now perform SVD on X, observe the singular values (the diagonal entries in the resulting S matrix), and cut them off at some index k based on the desired percentage variance captured:

$$\frac{\sum_{i=1}^{k} \sigma_i}{\sum_{i=1}^{|V|} \sigma_i}$$

We then take the submatrix of $U_{1:|V|,1:k}$ to be our word embedding matrix. This would thus give us a k-dimensional representation of every word in the vocabulary.

Applying SVD to X:

$$|V| \overset{|V|}{\begin{bmatrix} & & \\ & X & \\ & & \end{bmatrix}} = |V| \overset{|V|}{\begin{bmatrix} | & | & \\ u_1 & u_2 & \cdots \\ | & | & \end{bmatrix}} |V| \overset{|V|}{\begin{bmatrix} \sigma_1 & 0 & \cdots \\ 0 & \sigma_2 & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}} |V| \overset{|V|}{\begin{bmatrix} - & v_1 & - \\ - & v_2 & - \\ & \vdots & \end{bmatrix}}$$

Reducing dimensionality by selecting first k singular vectors:

$$\begin{array}{c} \\ |V| \end{array} \begin{array}{c} |V| \\ \begin{bmatrix} & & \\ & \hat{X} & \\ & & \end{bmatrix} \end{array} = \begin{array}{c} \\ |V| \end{array} \begin{array}{c} k \\ \begin{bmatrix} | & | & \\ u_1 & u_2 & \cdots \\ | & | & \end{bmatrix} \end{array} \begin{array}{c} \\ k \end{array} \begin{array}{c} k \\ \begin{bmatrix} \sigma_1 & 0 & \cdots \\ 0 & \sigma_2 & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix} \end{array} \begin{array}{c} \\ k \end{array} \begin{array}{c} |V| \\ \begin{bmatrix} - & v_1 & - \\ - & v_2 & - \\ & \vdots & \end{bmatrix} \end{array}$$

Both of these methods give us word vectors that are more than sufficient to encode semantic and syntactic (part of speech) information but are associated with many other problems:

- The dimensions of the matrix change very often (new words are added very frequently and corpus changes in size).
- The matrix is extremely sparse since most words do not co-occur.
- The matrix is very high dimensional in general ($\approx 10^6 \times 10^6$)
- Quadratic cost to train (i.e. to perform SVD)
- Requires the incorporation of some hacks on X to account for the drastic imbalance in word frequency

SVD based methods do not scale well for big matrices and it is hard to incorporate new words or documents. Computational cost for a $m \times n$ matrix is $O(mn^2)$

However, count-based method make an efficient use of the statistics

Some solutions to exist to resolve some of the issues discussed above:

- Ignore function words such as "the", "he", "has", etc.
- Apply a ramp window – i.e. weight the co-occurrence count based on distance between the words in the document.
- Use Pearson correlation and set negative counts to 0 instead of using just raw count.

As we see in the next section, iteration based methods solve many of these issues in a far more elegant manner.

4 *Iteration Based Methods - Word2vec*

Let us step back and try a new approach. Instead of computing and storing global information about some huge dataset (which might be billions of sentences), we can 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.

The idea is to design a model whose parameters are the word vectors. Then, train the model on a certain objective. 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. Thus, we learn our word vectors. This idea is a very old

For an overview of Word2vec, a note map can be found here : `https://myndbook.com/view/4900`

A detailed summary of word2vec models can also be found here [Rong, 2014]

Iteration-based methods capture cooccurrence of words one at a time instead of capturing all cooccurrence counts directly like in SVD methods.

one dating back to 1986. We call this method "backpropagating" the errors (see [Rumelhart et al., 1988]). The simpler the model and the task, the faster it will be to train it.

Several approaches have been tested. [Collobert et al., 2011] design models for NLP whose first step is to transform each word in a vector. For each special task (Named Entity Recognition, Part-of-Speech tagging, etc.) they train not only the model's parameters but also the vectors and achieve great performance, while computing good word vectors! Other interesting reading would be [Bengio et al., 2003].

Context of a word:
The context of a word is the set of m surrounding words. For instance, the $m = 2$ context of the word "fox" in the sentence "The quick brown fox jumped over the lazy dog" is {"quick", "brown", "jumped", "over"}.

This model relies on a very important hypothesis in linguistics, *distributional similarity*, the idea that similar words have similar context.

In this class, we will present a simpler, more recent, probabilistic method by [Mikolov et al., 2013] : word2vec. Word2vec is a software package that actually includes :

- **2 algorithms**: continuous bag-of-words (CBOW) and skip-gram. CBOW aims to predict a center word from the surrounding context in terms of word vectors. Skip-gram does the opposite, and predicts the *distribution* (probability) of context words from a center word.

- **2 training methods**: negative sampling and hierarchical softmax. Negative sampling defines an objective by sampling *negative* examples, while hierarchical softmax defines an objective using an efficient tree structure to compute probabilities for all the vocabulary.

4.1 *Language Models (Unigrams, Bigrams, etc.)*

First, we need to create such a model that will assign a probability to a sequence of tokens. Let us start with an example:

"The cat jumped over the puddle."

A good language model will give this sentence a high probability because this is a completely valid sentence, syntactically and semantically. Similarly, the sentence "stock boil fish is toy" should have a very low probability because it makes no sense. Mathematically, we can call this probability on any given sequence of n words:

$$P(w_1, w_2, \cdots, w_n)$$

We can take the unary language model approach and break apart this probability by assuming the word occurrences are completely independent:

$$P(w_1, w_2, \cdots, w_n) = \prod_{i=1}^{n} P(w_i)$$

Unigram model:

$$P(w_1, w_2, \cdots, w_n) = \prod_{i=1}^{n} P(w_i)$$

However, we know this is a bit ludicrous because we know the next word is highly contingent upon the previous sequence of words. And the silly sentence example might actually score highly. So perhaps we let the probability of the sequence depend on the pairwise probability of a word in the sequence and the word next to it. We call this the bigram model and represent it as:

$$P(w_1, w_2, \cdots, w_n) = \prod_{i=2}^{n} P(w_i | w_{i-1})$$

Again this is certainly a bit naive since we are only concerning ourselves with pairs of neighboring words rather than evaluating a whole sentence, but as we will see, this representation gets us pretty far along. Note in the Word-Word Matrix with a context of size 1, we basically can learn these pairwise probabilities. But again, this would require computing and storing global information about a massive dataset.

Now that we understand how we can think about a sequence of tokens having a probability, let us observe some example models that could learn these probabilities.

Bigram model:

$$P(w_1, w_2, \cdots, w_n) = \prod_{i=2}^{n} P(w_i | w_{i-1})$$

4.2 *Continuous Bag of Words Model (CBOW)*

One approach is to treat {"The", "cat", ’over", "the’, "puddle"} as a context and from these words, be able to predict or generate the center word "jumped". This type of model we call a Continuous Bag of Words (CBOW) Model.

CBOW Model:
Predicting a center word from the surrounding context

For each word, we want to learn 2 vectors
- v: (input vector) when the word is in the context
- u: (output vector) when the word is in the center

Let’s discuss the CBOW Model above in greater detail. First, we set up our known parameters. Let 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 $x^{(c)}$. And the output as $y^{(c)}$ and in the CBOW model, since we only have one output, so we just call this y which is the one hot vector of the known center word. Now let’s define our unknowns in our model.

We create two matrices, $\mathcal{V} \in \mathbb{R}^{n \times |V|}$ and $\mathcal{U} \in \mathbb{R}^{|V| \times n}$. Where n is an arbitrary size which defines the size of our embedding space. $\mathcal{V}$ is the input word matrix such that the i-th column of $\mathcal{V}$ is the n-dimensional embedded vector for word w_i when it is an input to this model. We denote this $n \times 1$ vector as v_i. Similarly, $\mathcal{U}$ is the output word matrix. The j-th row of $\mathcal{U}$ is an n-dimensional embedded vector for word w_j when it is an output of the model. We denote this row of $\mathcal{U}$ as u_j. Note that we do in fact learn two vectors for every word w_i (i.e. input word vector v_i and output word vector u_i).

Notation for CBOW Model:

- w_i: Word i from vocabulary V
- $\mathcal{V} \in \mathbb{R}^{n \times |V|}$: Input word matrix
- v_i: i-th column of $\mathcal{V}$, the input vector representation of word w_i
- $\mathcal{U} \in \mathbb{R}^{|V| \times n}$: Output word matrix
- u_i: i-th row of $\mathcal{U}$, the output vector representation of word w_i

We breakdown the way this model works in these steps:

1. We generate our one hot word vectors for the input context of size $m : (x^{(c-m)}, \ldots, x^{(c-1)}, x^{(c+1)}, \ldots, x^{(c+m)} \in \mathbb{R}^{|V|})$.

2. We get our embedded word vectors for the context ($v_{c-m} = \mathcal{V}x^{(c-m)}$, $v_{c-m+1} = \mathcal{V}x^{(c-m+1)}$, $\ldots$, $v_{c+m} = \mathcal{V}x^{(c+m)} \in \mathbb{R}^n$)
3. Average these vectors to get $\hat{v} = \frac{v_{c-m}+v_{c-m+1}+\ldots+v_{c+m}}{2m} \in \mathbb{R}^n$
4. Generate a score vector $z = \mathcal{U}\hat{v} \in \mathbb{R}^{|V|}$. As the dot product of similar vectors is higher, it will push similar words close to each other in order to achieve a high score.
5. Turn the scores into probabilities $\hat{y} = \text{softmax}(z) \in \mathbb{R}^{|V|}$.
6. We desire our probabilities generated, $\hat{y} \in \mathbb{R}^{|V|}$, to match the true probabilities, $y \in \mathbb{R}^{|V|}$, which also happens to be the one hot vector of the actual word.

The **softmax** is an operator that we'll use very frequently. It transforms a vector into a vector whose i-th component is $\frac{e^{\hat{y}_i}}{\sum_{k=1}^{|V|} e^{\hat{y}_k}}$.
- exponentiate to make positive
- Dividing by $\sum_{k=1}^{|V|} e^{\hat{y}_k}$ *normalizes* the vector ($\sum_{k=1}^{n} \hat{y}_k = 1$) to give probability

So now that we have an understanding of how our model would work if we had a $\mathcal{V}$ and $\mathcal{U}$, how would we learn these two matrices? Well, we need to create an objective function. Very often when we are trying to learn a probability from some true probability, we look to information theory to give us a measure of the distance between two distributions. Here, we use a popular choice of distance/loss measure, cross entropy $H(\hat{y}, y)$.

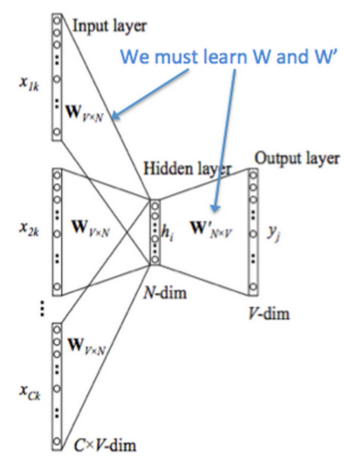

Figure 1: This image demonstrates how CBOW works and how we must learn the transfer matrices

The intuition for the use of cross-entropy in the discrete case can be derived from the formulation of the loss function:

$$H(\hat{y}, y) = -\sum_{j=1}^{|V|} y_j \log(\hat{y}_j)$$

Let us concern ourselves with the case at hand, which is that y is a one-hot vector. Thus we know that the above loss simplifies to simply:

$$H(\hat{y}, y) = -y_i \log(\hat{y}_i)$$

In this formulation, c is the index where the correct word's one hot vector is 1. We can now consider the case where our prediction was perfect and thus $\hat{y}_c = 1$. We can then calculate $H(\hat{y}, y) = -1\log(1) = 0$. Thus, for a perfect prediction, we face no penalty or loss. Now let us consider the opposite case where our prediction was very bad and thus $\hat{y}_c = 0.01$. As before, we can calculate our loss to be $H(\hat{y}, y) = -1\log(0.01) \approx 4.605$. We can thus see that for probability distributions, cross entropy provides us with a good measure of distance. We thus formulate our optimization objective as:

$\hat{y} \mapsto H(\hat{y}, y)$ is minimum when $\hat{y} = y$. Then, if we found a $\hat{y}$ such that $H(\hat{y}, y)$ is close to the minimum, we have $\hat{y} \approx y$. This means that our model is very good at predicting the center word!

To learn the vectors (the matrices U and V) CBOW defines a cost that measures how good it is at predicting the center word. Then, we optimize this cost by updating the matrices U and V thanks to stochastic gradient descent

$$\begin{aligned}
\text{minimize } J &= -\log P(w_c | w_{c-m}, \ldots, w_{c-1}, w_{c+1}, \ldots, w_{c+m}) \\
&= -\log P(u_c | \hat{v}) \\
&= -\log \frac{\exp(u_c^T \hat{v})}{\sum_{j=1}^{|V|} \exp(u_j^T \hat{v})} \\
&= -u_c^T \hat{v} + \log \sum_{j=1}^{|V|} \exp(u_j^T \hat{v})
\end{aligned}$$

We use stochastic gradient descent to update all relevant word vectors u_c and v_j.

Stochastic gradient descent (SGD) computes gradients for a window and updates the parameters
$\mathcal{U}_{new} \leftarrow \mathcal{U}_{old} - \alpha \nabla_{\mathcal{U}} J$
$\mathcal{V}_{old} \leftarrow \mathcal{V}_{old} - \alpha \nabla_{\mathcal{V}} J$

Skip-Gram Model:
Predicting surrounding context words given a center word

4.3 *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.

Let's discuss the Skip-Gram model above. 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 $y^{(j)}$. We define $\mathcal{V}$ and $\mathcal{U}$ the same as in CBOW.

Notation for Skip-Gram Model:

- w_i: Word i from vocabulary V
- $\mathcal{V} \in \mathbb{R}^{n \times |V|}$: Input word matrix
- v_i: i-th column of $\mathcal{V}$, the input vector representation of word w_i
- $\mathcal{U} \in \mathbb{R}^{n \times |V|}$: Output word matrix
- u_i: i-th row of $\mathcal{U}$, the output vector representation of word w_i

We breakdown the way this model works in these 6 steps:

1. We generate our one hot input vector $x \in \mathbb{R}^{|V|}$ of the center word.
2. We get our embedded word vector for the center word $v_c = \mathcal{V}x \in \mathbb{R}^n$
3. Generate a score vector $z = \mathcal{U}v_c$.
4. Turn the score vector into probabilities, $\hat{y} = \text{softmax}(z)$. Note that $\hat{y}_{c-m}, \ldots, \hat{y}_{c-1}, \hat{y}_{c+1}, \ldots, \hat{y}_{c+m}$ are the probabilities of observing each context word.
5. We desire our probability vector generated to match the true probabilities which is $y^{(c-m)}, \ldots, y^{(c-1)}, y^{(c+1)}, \ldots, y^{(c+m)}$, the one hot vectors of the actual output.

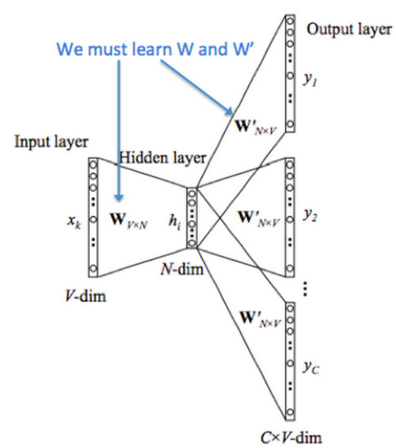

Figure 2: This image demonstrates how Skip-Gram works and how we must learn the transfer matrices

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.

$$\begin{aligned}
\text{minimize } J &= -\log P(w_{c-m}, \ldots, w_{c-1}, w_{c+1}, \ldots, w_{c+m} | w_c) \\
&= -\log \prod_{j=0, j\neq m}^{2m} P(w_{c-m+j}|w_c) \\
&= -\log \prod_{j=0, j\neq m}^{2m} P(u_{c-m+j}|v_c) \\
&= -\log \prod_{j=0, j\neq m}^{2m} \frac{\exp(u_{c-m+j}^T v_c)}{\sum_{k=1}^{|V|} \exp(u_k^T v_c)} \\
&= -\sum_{j=0, j\neq m}^{2m} u_{c-m+j}^T v_c + 2m \log \sum_{k=1}^{|V|} \exp(u_k^T v_c)
\end{aligned}$$

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.

Note that

$$\begin{aligned}
J &= -\sum_{j=0, j\neq m}^{2m} \log P(u_{c-m+j}|v_c) \\
&= \sum_{j=0, j\neq m}^{2m} H(\hat{y}, y_{c-m+j})
\end{aligned}$$

Only one probability vector $\hat{y}$ is computed. Skip-gram treats each context word equally : the models computes the probability for each word of appearing in the context independently of its distance to the center word

where $H(\hat{y}, y_{c-m+j})$ is the cross-entropy between the probability vector $\hat{y}$ and the one-hot vector y_{c-m+j}.

4.4 *Negative Sampling*

Loss functions J for CBOW and Skip-Gram are expensive to compute because of the softmax normalization, where we sum over all $|V|$ scores!

Lets take a second to look at the objective function. Note that the summation over $|V|$ is computationally huge! Any update we do or evaluation of the objective function would take $O(|V|)$ time which if we recall is in the millions. A simple idea is we could instead just approximate it.

For every training step, instead of looping over the entire vocabulary, we can just sample several negative examples! We "sample" from a noise distribution ($P_n(w)$) whose probabilities match the ordering of the frequency of the vocabulary. To augment our formulation of the problem to incorporate Negative Sampling, all we need to do is update the:

- objective function

- gradients
- update rules

Mikolov et al. present **Negative Sampling** in Distributed Representations of Words and Phrases and their Compositionality. While negative sampling is based on the Skip-Gram model, it is in fact optimizing a different objective. Consider a pair (w, c) of word and context. Did this pair come from the training data? Let's denote by $P(D = 1|w, c)$ the probability that (w, c) came from the corpus data. Correspondingly, $P(D = 0|w, c)$ will be the probability that (w, c) did not come from the corpus data. First, let's model $P(D = 1|w, c)$ with the sigmoid function:

The **sigmoid** function
$\sigma(x) = \frac{1}{1+e^{-x}}$
is the 1D version of the softmax and can be used to model a probability

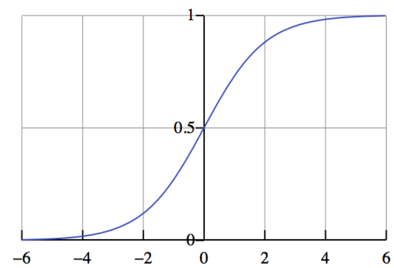

Figure 3: Sigmoid function

$$P(D = 1|w, c, \theta) = \sigma(v_c^T v_w) = \frac{1}{1 + e^{(-v_c^T v_w)}}$$

Now, we build a new objective function that tries to maximize the probability of a word and context being in the corpus data if it indeed is, and maximize the probability of a word and context not being in the corpus data if it indeed is not. We take a simple maximum likelihood approach of these two probabilities. (Here we take θ to be the parameters of the model, and in our case it is $\mathcal{V}$ and $\mathcal{U}$.)

$$\begin{aligned}
\theta &= \underset{\theta}{\operatorname{argmax}} \prod_{(w,c)\in D} P(D = 1|w, c, \theta) \prod_{(w,c)\in \tilde{D}} P(D = 0|w, c, \theta) \\
&= \underset{\theta}{\operatorname{argmax}} \prod_{(w,c)\in D} P(D = 1|w, c, \theta) \prod_{(w,c)\in \tilde{D}} (1 - P(D = 1|w, c, \theta)) \\
&= \underset{\theta}{\operatorname{argmax}} \sum_{(w,c)\in D} \log P(D = 1|w, c, \theta) + \sum_{(w,c)\in \tilde{D}} \log(1 - P(D = 1|w, c, \theta)) \\
&= \underset{\theta}{\operatorname{argmax}} \sum_{(w,c)\in D} \log \frac{1}{1 + \exp(-u_w^T v_c)} + \sum_{(w,c)\in \tilde{D}} \log(1 - \frac{1}{1 + \exp(-u_w^T v_c)}) \\
&= \underset{\theta}{\operatorname{argmax}} \sum_{(w,c)\in D} \log \frac{1}{1 + \exp(-u_w^T v_c)} + \sum_{(w,c)\in \tilde{D}} \log(\frac{1}{1 + \exp(u_w^T v_c)})
\end{aligned}$$

Note that maximizing the likelihood is the same as minimizing the negative log likelihood

$$J = -\sum_{(w,c)\in D} \log \frac{1}{1 + \exp(-u_w^T v_c)} - \sum_{(w,c)\in \tilde{D}} \log(\frac{1}{1 + \exp(u_w^T v_c)})$$

Note that $\tilde{D}$ is a "false" or "negative" corpus. Where we would have sentences like "stock boil fish is toy". Unnatural sentences that should get a low probability of ever occurring. We can generate $\tilde{D}$ on the fly by randomly sampling this negative from the word bank.

For skip-gram, our new objective function for observing the context word $c-m+j$ given the center word c would be

$$-\log \sigma(u_{c-m+j}^T \cdot v_c) - \sum_{k=1}^{K} \log \sigma(-\tilde{u}_k^T \cdot v_c)$$

To compare with the regular softmax loss for skip-gram
$-u_{c-m+j}^T v_c + \log \sum_{k=1}^{|V|} \exp(u_k^T v_c)$

For CBOW, our new objective function for observing the center word u_c given the context vector $\hat{v} = \frac{v_{c-m}+v_{c-m+1}+\ldots+v_{c+m}}{2m}$ would be

$$-\log \sigma(u_c^T \cdot \hat{v}) - \sum_{k=1}^{K} \log \sigma(-\tilde{u}_k^T \cdot \hat{v})$$

To compare with the regular softmax loss for CBOW
$-u_c^T \hat{v} + \log \sum_{j=1}^{|V|} \exp(u_j^T \hat{v})$

In the above formulation, $\{\tilde{u}_k | k = 1 \ldots K\}$ are sampled from $P_n(w)$. Let's discuss what $P_n(w)$ should be. While there is much discussion of what makes the best approximation, what seems to work best is the Unigram Model raised to the power of 3/4. Why 3/4? Here's an example that might help gain some intuition:

$$\text{is: } 0.9^{3/4} = 0.92$$
$$\text{Constitution: } 0.09^{3/4} = 0.16$$
$$\text{bombastic: } 0.01^{3/4} = 0.032$$

"Bombastic" is now 3x more likely to be sampled while "is" only went up marginally.

4.5 *Hierarchical Softmax*

Mikolov et al. also present hierarchical softmax as a much more efficient alternative to the normal softmax. In practice, hierarchical softmax tends to be better for infrequent words, while negative sampling works better for frequent words and lower dimensional vectors.

Hierarchical Softmax uses a binary tree where leaves are the words. The probability of a word being the output word is defined as the probability of a random walk from the root to that word's leaf. Computational cost becomes $O(\log(|V|))$ instead of $O(|V|)$.

Hierarchical softmax uses a binary tree to represent all words in the vocabulary. Each leaf of the tree is a word, and there is a unique path from root to leaf. In this model, there is *no output representation for words*. Instead, each node of the graph (except the root and the leaves) is associated to a vector that the model is going to learn.

In this model, the probability of a word w given a vector w_i, $P(w|w_i)$, is equal to the probability of a random walk starting in the root and ending in the leaf node corresponding to w. The main advantage in computing the probability this way is that the cost is only $O(\log(|V|))$, corresponding to the length of the path.

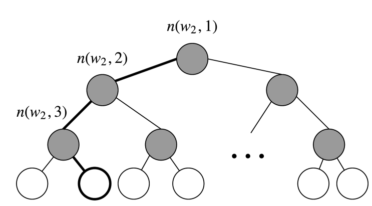

Figure 4: Binary tree for Hierarchical softmax

Let's introduce some notation. Let $L(w)$ be the number of nodes in the path from the root to the leaf w. For instance, $L(w_2)$ in Figure 4 is 3. Let's write $n(w, i)$ as the i-th node on this path with associated vector $v_{n(w,i)}$. So $n(w,1)$ is the root, while $n(w,L(w))$ is the father of w. Now for each inner node n, we arbitrarily choose one of its children and call it $ch(n)$ (e.g. always the left node). Then, we can compute the probability as

$$P(w|w_i) = \prod_{j=1}^{L(w)-1} \sigma([n(w,j+1) = ch(n(w,j))] \cdot v_{n(w,j)}^T v_{w_i})$$

where

$$[x] = \begin{cases} 1 \text{ if } x \text{ is true} \\ -1 \text{ otherwise} \end{cases}$$

.

and $\sigma(\cdot)$ is the sigmoid function.

This formula is fairly dense, so let's examine it more closely.

First, we are computing a product of terms based on the shape of the path from the root $(n(w,1))$ to the leaf (w). If we assume $ch(n)$ is always the left node of n, then term $[n(w,j+1) = ch(n(w,j))]$ returns 1 when the path goes left, and -1 if right.

Furthermore, the term $[n(w,j+1) = ch(n(w,j))]$ provides normalization. At a node n, if we sum the probabilities for going to the left and right node, you can check that for any value of $v_n^T v_{w_i}$,

$$\sigma(v_n^T v_{w_i}) + \sigma(-v_n^T v_{w_i}) = 1$$

The normalization also ensures that $\sum_{w=1}^{|V|} P(w|w_i) = 1$, just as in the original softmax.

Finally, we compare the similarity of our input vector v_{w_i} to each inner node vector $v_{n(w,j)}^T$ using a dot product. Let's run through an example. Taking w_2 in Figure 4, we must take two left edges and then a right edge to reach w_2 from the root, so

$$\begin{aligned} P(w_2|w_i) &= p(n(w_2,1),\text{left}) \cdot p(n(w_2,2),\text{left}) \cdot p(n(w_2,3),\text{right}) \\ &= \sigma(v_{n(w_2,1)}^T v_{w_i}) \cdot \sigma(v_{n(w_2,2)}^T v_{w_i}) \cdot \sigma(-v_{n(w_2,3)}^T v_{w_i}) \end{aligned}$$

To train the model, our goal is still to minimize the negative log likelihood $-\log P(w|w_i)$. But instead of updating output vectors per word, we update the vectors of the nodes in the binary tree that are in the path from root to leaf node.

The speed of this method is determined by the way in which the binary tree is constructed and words are assigned to leaf nodes. Mikolov et al. use a binary Huffman tree, which assigns frequent words shorter paths in the tree.

References

[Bengio et al., 2003] Bengio, Y., Ducharme, R., Vincent, P., and Janvin, C. (2003). A neural probabilistic language model. *J. Mach. Learn. Res.*, 3:1137–1155.

[Collobert et al., 2011] Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., and Kuksa, P. P. (2011). Natural language processing (almost) from scratch. *CoRR*, abs/1103.0398.

[Mikolov et al., 2013] Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781.

[Rong, 2014] Rong, X. (2014). word2vec parameter learning explained. *CoRR*, abs/1411.2738.

[Rumelhart et al., 1988] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1988). Neurocomputing: Foundations of research. chapter Learning Representations by Back-propagating Errors, pages 696–699. MIT Press, Cambridge, MA, USA.