CS224d: Deep Learning for NLP
Lecture Notes

Francois Chaubard

Introduction to Natural Language Processing

We begin with a general discussion of what is NLP. 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, pdfs, etc.

Hard:

- Machine Translation(go from Chinese text to English),
- Semantic Analysis (what is the meaning of query statement),
- Coreference (What does "He" or "it" refer to given a document),
- Q/A (Answering Jeopardy questions).
- Dialog (Siri)

The first and arguably most important common denominator across all NLP tasks is how we represent words as input to any and all 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. (distance measures! cosine, euclidean, etc)

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<<13 Million) that is sufficient to encode all semantics

of our language! Each dimension would encode something that we use in speech, (i.e. one dimension might be past vs present vs future, one might represent plural vs singular, another might be masculine vs feminine, etc.)

So lets dive into our first word vector and arguably the most simple. The **"one-hot"** vector.

> *Idea: Represent every word as a |V|x1 vector with all 0's and one 1 at the index of that word in the sorted english language.*

For all these notes, we represent the size of the language we are analyzing as |V|.

$$
\begin{aligned}
&v_{aardvark} \quad [1\ 0\ 0\ 0\ \dots\ 0] \quad 1x|V| \\
&v_{a} \quad\quad [0\ 1\ 0\ 0\ \dots\ 0] \quad 1x|V| \\
&v_{at} \quad\quad [0\ 0\ 1\ 0\ \dots\ 0] \quad 1x|V| \\
&\dots \\
&v_{zebra} \quad [0\ 0\ 0\ 0\ \dots\ 1] \quad 1x|V|
\end{aligned}
$$

> *(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.

$$v_{hotel}^T v_{motel} == v_{hotel}^T v_{cat} == 0$$

So maybe we can try to reduce the size of this space from dimension size |V| to something smaller, find a subspace that encodes the relationships between words.

SVD Based Methods

For this class of methods to find word embeddings, we first loop over a massive dataset and accumulate word co-occurrence counts in some form of a matrix **X**, then perform SVD on **X** to get **U*S*V**^{T} and use the rows of **U** as the word embeddings for all words in our dictionary. We go through a few choices of **X**.

Word-Document matrix

For our first attempt, we make the bold conjecture that words that are related will often appear in the same documents. *Banks, bonds, stocks, money, etc* are probably likely to appear together. But *Banks, Octopus, Banana, and Hockey*, would probably not consistently appear together. So idea number 1. Build a word-document matrix, **X**. Loop over billions of documents and for every time word i appears in document j, we add one to entry X(i,j). This is obviously a pretty large matrix if we have billions of documents! So perhaps we can try something better.

Word-Word co-occurrence matrix with context size C

The same kind of logic applies here however, the matrix we accumulate counts in, actually becomes an affinity matrix. We display an example one below.

counts	I	like	enjoy	deep	learning	NLP	flying	.
I	0	2	1	0	0	0	0	0
like	2	0	0	1	0	1	0	0
enjoy	1	0	0	0	0	0	1	0
deep	0	1	0	0	1	0	0	0
learning	0	0	0	1	0	0	0	1
NLP	0	1	0	0	0	0	0	1
flying	0	0	1	0	0	0	0	1
.	0	0	0	0	1	1	1	0

Figure 1: Word-Word co-occurrence matrix

We now perform SVD on **X**, observe the Singular Values, and cut them off at some index k, and take the submatrix of U(1:|V|,1:k) to be our word embedding matrix.

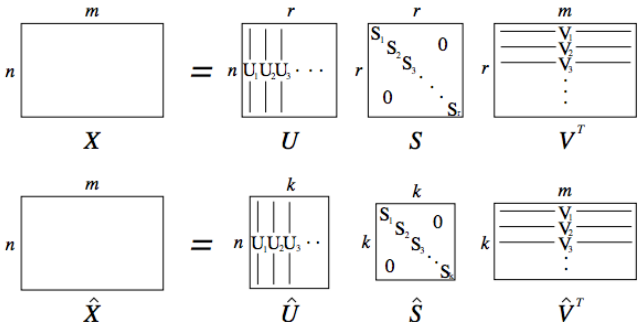

Figure 2: Latent Semantic Analysis

These methods both give us word vectors that are more than sufficient to encode semantic and syntactic (part of speech) information!

Some problems with SVD based methods:

- The dimensions of the matrix change all the time (new words are added very frequently!)
- Sparsity!
- Very high dimension in general! (millions^2 = too big to store on a typical laptop!)
- Quadratic cost to train (or perform SVD).
- Need to incorporate some sort of hacks to **X** to account for drastic imbalance in word frequency (bombastic is far less frequent than the word the)
    - max count!
    - ignore function words.. “the”, “he”, “has”..
    - ramp window..
- Pearson correl and set negative counts to 0 vs just raw count tends to give much better performance.

Iteration Based Methods

Lets 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 will be able to encode the ***probability*** of a ***word*** given its ***context*** (terminology we will explain soon). We can set up this probabilistic model of known and unknown parameters and take one training example at a time and try 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. This idea is a very old one (1986!). We call this method "backpropagating" the errors (see *David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. 1988.* ***Learning representations by back-propagating errors***.)

Language Models (Unigram, Bigram, etc)

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

"The cat jumped over the puddle." (1)

A good language model will give this sentence a high probability because in our language this is a completely ok sentence. 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:

$$P(w_1, w_2, \ldots w_n) \quad (2)$$

We can take the unary language model approach and break apart this probability by assuming the words are completely independent which means:

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

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:

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

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 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, lets observe some example models that could learn these probabilities.

CBOW and Skip-Gram

Lets return to our original objective of designing an iteration based algorithm that will do better than the SVD based approaches in the places they were weak.

CBOW Model

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

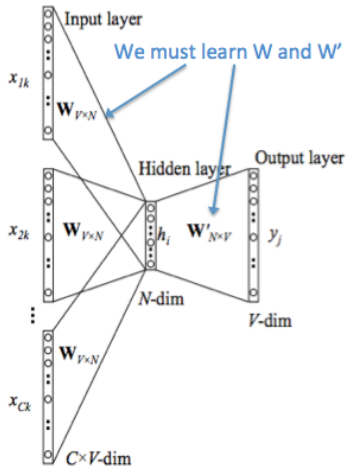

Figure 3: The CBOW model.

Lets 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 vectors. The input one hot vectors or ***context*** we will represent with an $\boldsymbol{x}_{\textbf{subscript}}$. And the output as $\boldsymbol{y}_{\textbf{subscript}}$ 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 center word. Now lets define our unknowns in our model. We create two

matrices, $W \in \Re^{n \times |V|}$ and $W' \in \Re^{|V| \times n}$. Where n is an arbitrary size which defines the size of our embedding space. W is the ***input*** word matrix such that the i[th] column of W is the n-dimensional embedded vector for word i when it is an input to this model. We denote this nx1 vector as v_{w_i}. Similarly, we call the rows of W' the ***output*** word matrix such that the j[th] row of W' is the n-dimensional embedded vector for word j when it is an output of the model. We denote this row of W' as v'_{w_j}. Note that we do infact learn **two** vectors for every word!

We breakdown the way this model works in these steps:

1) We generate our one hot input vectors *(x_1, x_2, … x_c)* for context size c.
2) We get our embedded word vectors for the context
   ($v_{w_1} = Wx_1$, $v_{w_2} = Wx_2$, … , $v_{w_c} = Wx_c$)
3) Average them to get h. ($h = \frac{1}{C}(v_{w_1} + v_{w_2} + ... + v_{w_C})$)
4) Generate a score vector, u. ($u = W'h$)
5) Turn the scores into probabilities, $\hat{y}$. ($\hat{y} = softmax(u)$)
6) We desire our probabilities generated to match the true probabilities which is y, the one hot vector of the actual output.

So now that we have an understanding of how our model would work if we had a W and W', 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 invoke a common choice for such a metric, Cross Entropy H(p,q). Do not worry why for this part since its not that important but we invite you to look up the wikipedia on Cross Entropy if you are interested. All you need to know is that the important term for our optimization program is a minus log probability function of the form:

$$
\begin{aligned}
minimize\ E &= -log(\ p(w_o|w_{I_1}, w_{I_2} ... ,\ w_{I_C})\ ) \\
&= -log(\ p(w_o|\ h)\ ) \\
&= -log(\ \frac{exp({v'_{w_o}}^T h)}{\sum_{j=1}^{|V|} exp({v'_{w_j}}^T v_{w_I})}\ ) \\
&= -(\ {v'_{w_o}}^T h\ ) + log(\ \sum_{j=1}^{|V|} exp({v'_{w_j}}^T v_{w_I}))
\end{aligned}
$$

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.

(WILL INSERT GRADIENTS HERE WHEN ASSIGNMENT ONE IS GRADED)

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***. And we call this type of model a Skip-Gram model.

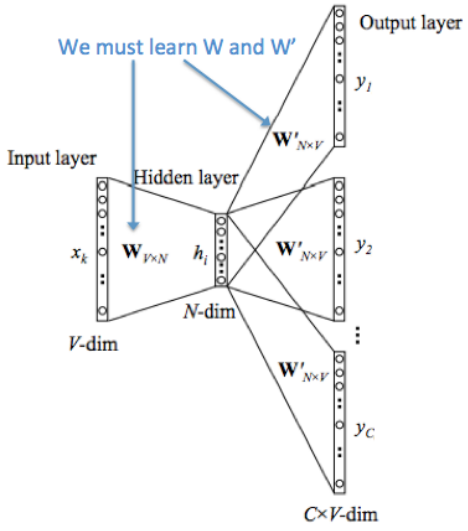

Figure 4: The Skip-Gram Model

Lets discuss the Skip-Gram model above. The set-up is largely the same but we essentially swap our ***x***'s and ***y***'s. i.e. what are ***x***'s in the CBOW are now ***y***'s and visa versa. The input one hot vector or ***context*** we will represent with an ***x*** (since there is only one). And the output vectors as $\boldsymbol{y}_{\textbf{subscript}}$. We define W and W' 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
   ($v_w = Wx$)
3. Since there is no averaging, just set ($h = v_w$)
4. Generate C score vectors, $u_1, u_2, \ldots, u_C$. ($u = W' h$)
5. Turn the scores into probabilities, $\hat{y}$. ($\hat{y} = softmax(u)$)
6. We desire our probability vector generated to match the true probabilities which is $y_1, y_2, \ldots y_C$, the one hot vectors of the actual output. (Note, since the score vectors are all equal: $u = u_1 = u_2 = \ldots = u_C$ the output probabilities also are: $\hat{y} = \hat{y}_1 = \hat{y}_2 = \hat{y}_3 .. \hat{y}_C$)

Similarly, as in CBOW, we need to generate an objective function for us to evaluate the model. A key difference here is, we invoke a Naive-Bayes assumption to break out the

probabilities. If you have not seen this before, then simply, it is a strong (naive) conditional independence assumptions. (Given the center word, all output words are completely independent.)

$$
\begin{aligned}
minimize\ E &= -log(\ p(w_{O_1}, w_{O_2} \dots,\ w_{O_C} | \ w_I)\ ) \\
&= -log(\ \prod_{i=1}^{C} p(w_{O_i} | \ w_I\ )\ ) \\
&= -log(\ \prod_{i=1}^{C} \frac{exp({v'_{w_{o_i}}}^T v_{w_I})}{\sum_{j=1}^{|V|} exp({v'_{w_j}}^T v_{w_I})}\ ) \\
&= -\sum_{i=1}^{C} ({v'_{w_{o_i}}}^T v_{w_I}) + C\ log(\ \sum_{j=1}^{|V|} exp({v'_{w_j}}^T v_{w_I}))
\end{aligned}
$$

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.

(WILL INSERT GRADIENTS HERE WHEN ASSIGNMENT ONE IS GRADED)

Negative Sampling

Lets take a second to look at the objective function in the Skip-Gram Model. 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:

1. objective function
2. gradients
3. update rules

Mikolov et al. presents 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 altogether. Consider a pair (w, c) of word and context. Did this pair come from the training data? With what probability did it come from our corpus? Let's denote this 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, lets model P(D = 1|w, c) with the sigmoid function.

$$p(D = 1|w, c; \theta) = \frac{1}{1 + e^{-v_c \cdot v_w}}$$

(Here we take θ to be the parameters of the model, and in our case it is W(:) concatenated with 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 Max Likelihood approach of these two probabilities.

$$
\begin{aligned}
&\arg\max_{\theta} \prod_{(w,c)\in D} p(D=1|c,w;\theta) \prod_{(w,c)\in D'} p(D=0|c,w;\theta) \\
=&\arg\max_{\theta} \prod_{(w,c)\in D} p(D=1|c,w;\theta) \prod_{(w,c)\in D'} (1-p(D=1|c,w;\theta)) \\
=&\arg\max_{\theta} \sum_{(w,c)\in D} \log p(D=1|c,w;\theta) + \sum_{(w,c)\in D'} \log(1-p(D=1|w,c;\theta)) \\
=&\arg\max_{\theta} \sum_{(w,c)\in D} \log \frac{1}{1+e^{-v_c\cdot v_w}} + \sum_{(w,c)\in D'} \log(1-\frac{1}{1+e^{-v_c\cdot v_w}}) \\
=&\arg\max_{\theta} \sum_{(w,c)\in D} \log \frac{1}{1+e^{-v_c\cdot v_w}} + \sum_{(w,c)\in D'} \log(\frac{1}{1+e^{v_c\cdot v_w}})
\end{aligned}
$$

Note that ***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 ***D'*** on the fly by randomly sampling this negative from the word bank! Our new objective function would then be:

$$
E = -\log \sigma({\mathbf{v}'_{w_O}}^T \mathbf{h}) - \sum_{i=1}^{K} \log \sigma(-{\mathbf{v}'_{w_i}}^T \mathbf{h})
$$

where $\{w_i | i = 1, \ldots, K\}$ are sampled from $P_n(w)$ }

Lets 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 3/4th power. Why 3/4ths? (toy example)

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

"Bombastic" is now 3x more likely to be sampled while "is" only went up by a small amount. This is what we want as to have a higher chance of sampling rare words.

(WILL INSERT GRADIENTS AND UPDATE RULES HERE WHEN ASSIGNMENT ONE IS GRADED)