Word Embeddings

Classical Approaches To Word Embeddings

1. WordNet

WordNet relies on an external lexical knowledge base that encodes the information about the definition, synonyms, ancestors, descendants, and so forth of a given word. It allows a user to infer various infomation for a given word, such as the aspects of a word discussed in the preceding sentence and the similarity between two words.

Though WordNet is an amazing resource that anyone can use to learn meanings of word in the NLP tasks, there are quite a few drawbacks in using WordNet for this.

- Missing nuances is a key problem in WordNet. There are both theoretical and practical reasons why this is not viable for WordNet. From a theoretical perspective, it is not well-posed or direct to model the definition of the subtle difference between two entities. Practically speaking, defining nuances is subjective. For example, the words "want" and "need" have similar meanings, but one of them (need) is more assertive. This is considered to be a nuance.
- Next, WordNet is subjective in itself as WordNet was designed by a relatively small community. Therefore, depending on what you are trying to solve, WordNet might be suitable or you might be able to perform better with a loose definition of words.
- There also exists the issue of maintaining WordNet, which is labor-intensive. Maintaining and adding new synsets, definitions, lemmas, and so on, can be very expensive. This adversely affects the scalability of WordNet, as human labor is essential to keep WordNet up to date.
- Developing WordNet for other languages can be costly. There are also some efforts to build WordNet for other languages and link it with the Engilish WordNet as MultiWordNet (MWN), but they are yet incomplete.

2. One-hot Encoded Representation

If we have vocabulary of size V, for each i^{th} word w_i we will represent the word w_i with a V-long vector $[0, 0, \ldots 0, 1, 0, 0 \ldots]$ where the i^{th} element is 1 and other elements are zero.

Howevery, this representation does not encode the similarity between words in any way and completely ignores the context in which the words are used.

Term Frequency-Inverse Document Frequency (TF-IDF) Method

TF-IDF is a frequency-based method that takes into account the frequency with which a word appears in a

corpus.

$$TF(w_i) = \frac{\text{number of times } w_i \text{ appear}}{\text{total number of words}}$$

$$IDF(w_i) = log(\frac{\text{total number of documents}}{\text{total number of documents with } w_i \text{ in it}})$$

$$TF - IDF(w_i) = TF(w_i) * IDF(w_i)$$

For example:
Document 1: *this is about cats. Catsare great companions.*
Document 2: *This is about dogs. Dogs are very loyal.*
TF-IDF(cats,doc1) = (2/8)*log(2/1) = 0.075
TF-IDF(this,doc1) = (1/8)*log(2/2) = 0.0

There for the word "cats" is informative while "this" is not.

3. Word2vec

Skip-Gram

The skip-gram model uses the following approach to design such a dataset:

1. For a given word w_i, a context window size m is assumed. By context window size, we mean the number of words considered as context on a single side. Therefore, for w_i, the context window (including the target word w_i) will be of size 2m+1 and will look like this: $[w_{i-m}, \ldots w_{i-1}, w_i, w_{i+1}, \ldots, w_{i+m}]$.
2. Next, input-output tuples are formed as $[\ldots, (w_i, w_{i-m}), \ldots, (w_i, w_{i-1}), (w_i, w_{w_{i+1}}), \ldots (w_i, w_{i+m})]$; here, $m + 1 \leq i \leq N - m$, and N is the number of words in the text to get a practical insight.

Let's assume the following sentence and text window size (m) of 1:
The dog barked at the mailman.
For this example, the dataset would be as follows:
[(dog,The),(dog,barked),(barked,dog),(barked,at),...,(the,at),(the,mailman)]

Once the data is in the (input,output) format, we can use a neural network to learn the word embedding. First, let's identify the variables we need to learn the word embeddings. To store the word embeddings, we need a $V \times D$ matrix, where V is the vocabulary size and D is the dimensionality of the word embeddings (that is, the number of elements in the vector that represents a single word). D is a user defined hyperparameter. The higher D is, the more expressive the word embeddings learned will be. This matrix will be referred to as the embedding space or the embedding layer. Next we have a softmax layer with weights of size $D \times V$, a bias of size V.

Each word will be represented as a one-hot encoded vector of size V with one element being 1 and all the others being 0. Therefore, an input word and the corresponding output words would each be of size V. Let's refer to the i^{th} input as x_i, the corresponding embedding of x_i as z_i, and the corresponding output as y_i.

$$logit(x_i) = z_i W + b$$

$$\hat{y} = softmax(logit(x_i))$$

We want to define our loss function as follow:

$$J(\theta) = -(1/N - 2m) \sum_{i=m+1}^{N-m} \sum_{j \neq i,\, j=i-m}^{i+m} logit(x_n)_{w_j} - log\Big(\sum_{w_k \in vocabulary} exp\big(logit(x_n)_{w_k}\big) \Big)$$

but the loss function has a performance bottleneck because computing the loss for a single example requires computing logits for all the words in the vocabulary.
We can use two popular choices of approximations:

- Negative sampleing
- Hierarchical softmax

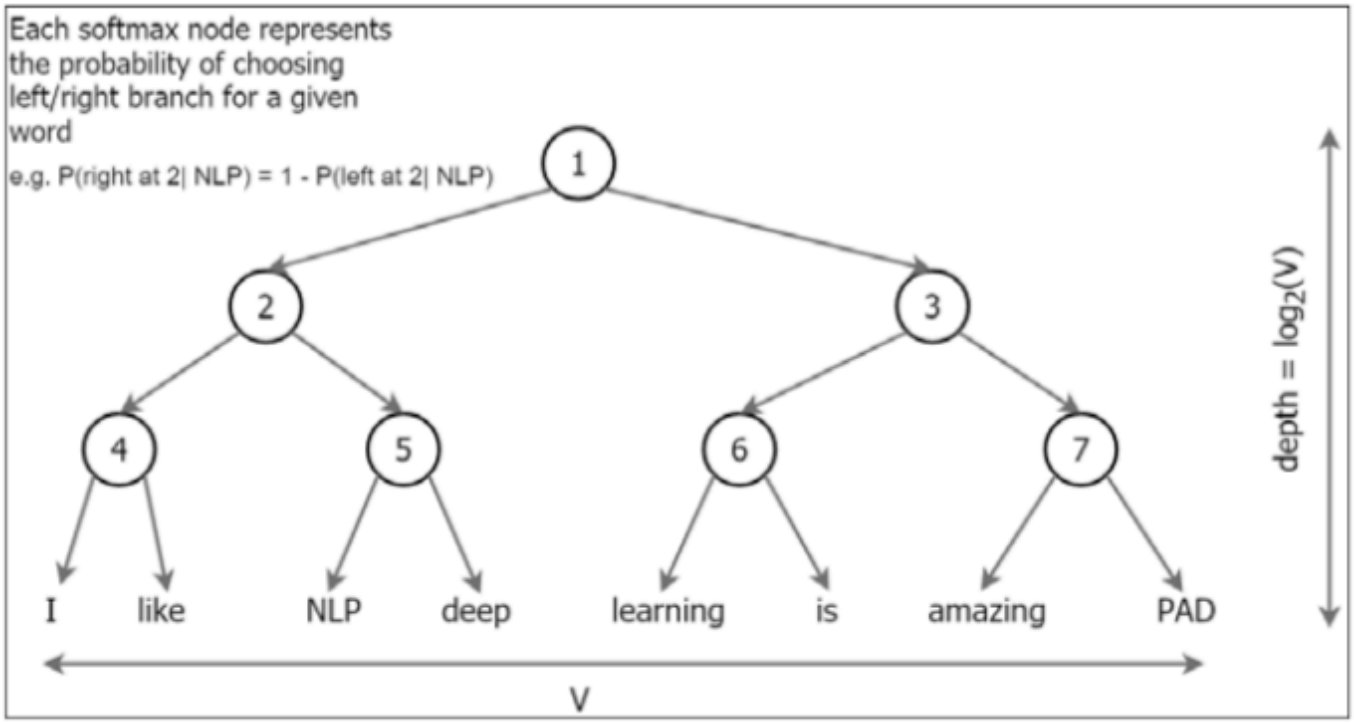
Negative sampleing

$$J(\theta) = -(1/N - 2m) \sum_{i=m+1}^{N-m} \sum_{j \neq i,\, j=i-m}^{i+m} log\Big(\sigma\big(logit(x_n)_{w_j}\big)\Big) + \sum_{q=1}^{k} E_{w_q - vocabulary - (w_i, w_j)}\, log\Big(\sigma\big(- logit(x_n)_{w_q}\big)\Big)$$

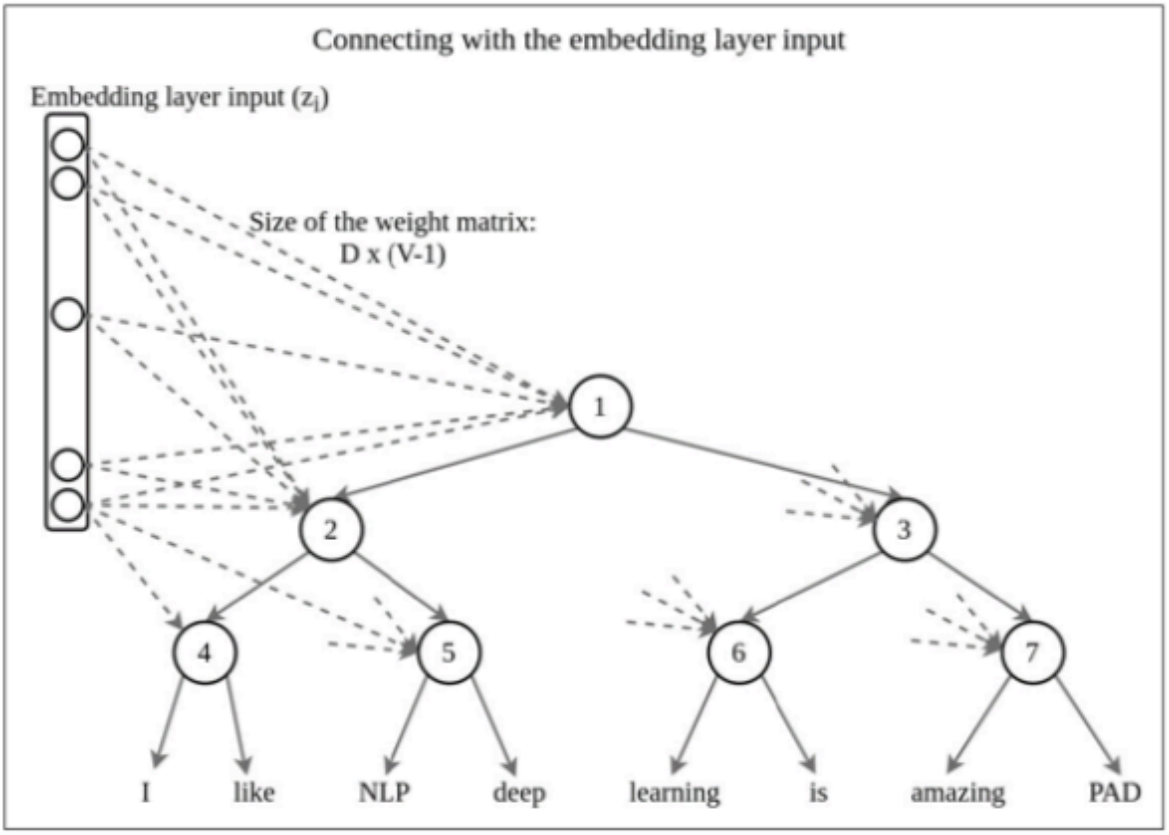
Here w_j represents a context word of w_i and w_q represents a noncontext word of w_i. To minimize $J(\theta)$ we want $logit(x_n)_{w_j}$ to be large positive value and $logit(x_n)_{w_q}$ to be large negative value.

Hierarchical softmax

To understand hierarchical softmax, let's consider an example:
I like NLP. Deep learning is amazing.
With this vocabulary, we will build a binary tree, where all the words in the vocabulary are present as leaf nodes.
We will also add a special token PAD to make sure that all the tree leaves have two members:

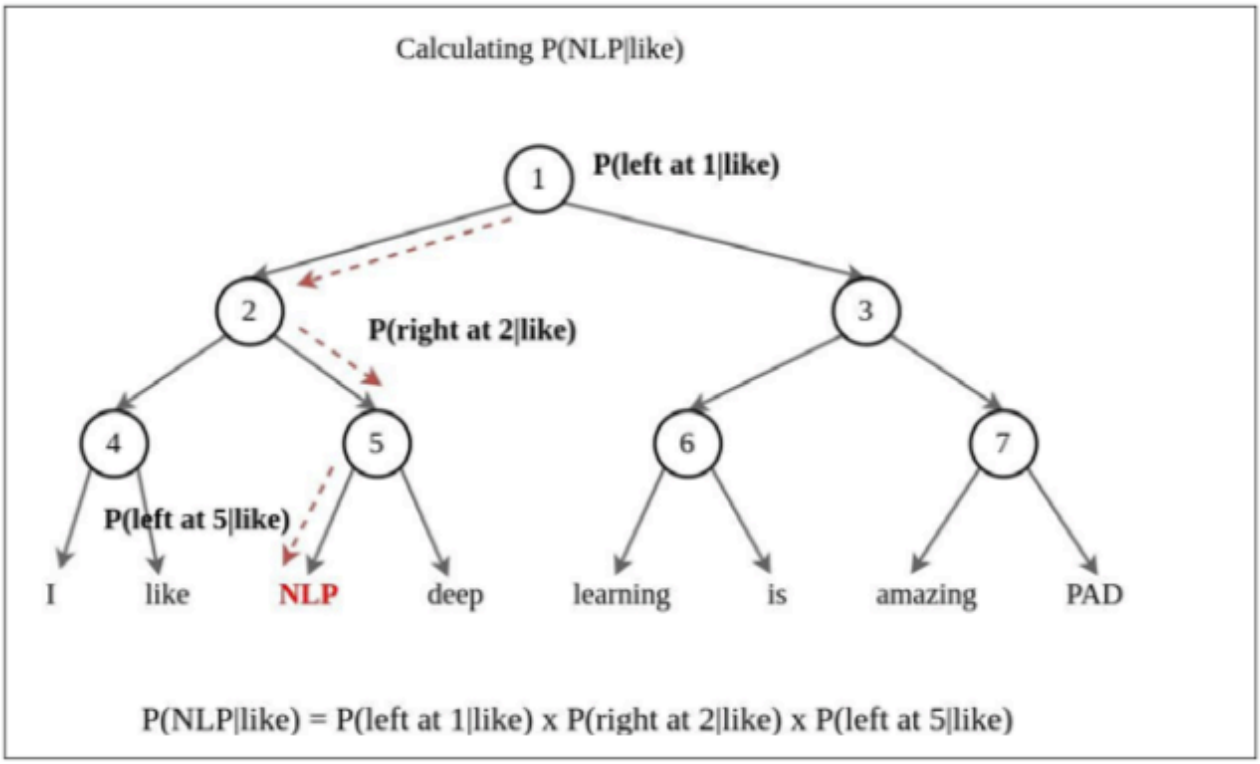


Then, our last hidden layer will be fully connected to all the nodes in the hierarchy



Then the probability of word "NLP" given word "like" will be

$$P(NLP \mid like) = P(\text{left at } 1 \mid like) * P(\text{right at2} \mid like) * P(\text{left at } 5 \mid like)$$



So the loss function will be:

$$J(\theta) = -(1/N - 2m) \sum_{i=m+1}^{N-m} \sum_{j \neq i,\, j=i-m}^{i+m} logP(w_j \mid w_i)$$

We can initialize the hierarchy randomly or use WordNet to determine the hierarchy.

Continuous Bag-of-Words

The CBOW model has a working similar to the skip-gram algorithm with one significant change in the problem formulation. In the skip-gram model, we predicted the context words from the target word. However, in the CBOW model, we will predict the target from contextual words. Let's compare what data looks like for skip- gram and CBOW by taking the previous example sentence:
The dog barked at the mailman.
For skip-gram, data tuples—(input word, output word)—might look like this:
(dog, the), (dog, barked), (barked, dog), and so on.
For CBOW, data tuples would look like the following:
([the, barked], dog), ([dog, at], barked), and so on.
Consequently, the input of the CBOW has a dimensionality of 2 × m × D, where m is the context window size and D is the dimensionality of the embeddings.