Introduction to Word Representation

1. Motivation

Deep learning model has a good performance in many numerical tasks, but it doesn't work well in natural language processing tasks at the beginning because the language materials are **discrete**, so we want to find a way to **represent the language materials in a continuous space**, for example, **vector**.

2. Symbolic Representation

2.1. Wordnet

Idea: Professionals manually annotate the semantic meaning of the words, building up a data structure, including the semantic relations between the word. Based on this data structure, people could vectorize the words with the features they concerned.

Key structure:

1. Synset: A set of synonyms.
2. Hypernym: A word is a hypernym of another word if it is a more general concept.
3. Hyponym: A word is a hyponym of another word if it is a more specific concept.
4. Antonym: A word is an antonym of another word if it is the opposite of the other word.
5. ...

2.2. One-hot Encoding

Idea: Represent each word as a unique vector of zeros, except for a one at the position corresponding to the word in a vocabulary.

Example

We have a sentence: "I like to eat apple". For this sentence, we have a vocabulary: ["I", "like", "to", "eat", "apple"]. Then, we can represent this sentence as a one-hot encoding vector:

$$\begin{aligned} \text{I} &\to [1,0,0,0,0] \\ \text{like} &\to [0,1,0,0,0] \\ &\vdots \\ \text{apple} &\to [0,0,0,0,1] \end{aligned}$$

Code Example

```py
vocabulary = ["I", "like", "to", "eat", "apple"]
sentence = "I like to eat apple"

def one_hot_encode(word, vocab):
    vector = [0] * len(vocab)
    if word in vocab:
        vector[vocab.index(word)] = 1
    return vector

for word in sentence.split():
    encoded = one_hot_encode(word, vocabulary)
```

```
12      print(f"{word}: {encoded}")
```

Disadvantages:

1. **Dimension disaster**: The datas we have contain a lot of words, so the vocabulary is very large. For a passage of 100000 words, the dimension of the one-hot encoding vector $\boldsymbol{v}_i \in \mathbb{R}^{100000}$. (High-dimensional $\rightarrow$ High-dimensional)
2. **Lack of semantic meaning**: It is difficult to capture the semantic meaning of the words ($v_i^T v_j = 0, \forall i \neq j$).
3. **Sparse representation**: The vector is mostly zeros, so useful information is little.

3. Distributed Representation

In one-hot encoding, all the information of single word is stored in a unique position of the vector, and there are many disadvantages. Another way to represent the words is **Distributed Representation**, which distributedly stores the information of each word in different positions of the vector and the dimension is relatively smaller.

3.1. Word Embedding

Word embedding is a subfield of Distributed Representation. It is a technique to represent words as dense vectors of continuous values.

- **Idea**: Use an embedding matrix to store the high-dimensional symbolic representation (one-hot encoding vector) into a low-dimensional dense vector
- For example, a one-hot encoding representation of a vocabulary of size m is a $m \times m$ matrix, we would like to squeeze the features into an n dimension vector, where $n \ll m$. That is, $\mathbb{R}^{m \times m} \rightarrow \mathbb{R}^{m \times n}$

Example of Word Embedding

Now we use a long paragraph: Valorant

1. Clean the text
2. Find the word pairs
3. Train the embedding matrix
4. Get the embedding vector of each word
5. Test the embedding vector (whether embedding vector of "close" words are close to each other)

In this example, the size of vocabulary is $m = 162$, and we want to squeeze the features into an $n = 7$ dimension vector. So we train a 162×7 matrix. It is also worth noticing that, the goal we want to achieve is

$$\max \sum_{i,j \in \text{ pos}} \boldsymbol{v}_i' \cdot \boldsymbol{v}_j + \sum_{i,j \in \text{ neg}} \left(-\boldsymbol{v}_i' \cdot \boldsymbol{v}_j\right)$$

Here, pos means the positive word pairs, and neg means the negative word pairs.

Through this embedding process, we could capture the semantic meaning of the words.

3.2. Word2Vec

Word2Vec is a model to generate the embedding vector of the words. In Word2Vec, there are two models: **Skip-gram** and **CBOW**. Skip-gram is used to predict the context words of the center word, and CBOW is used to predict the center word of the context words. Simply speaking, both aim to get

an embedding matrix to represent the words in a low-dimensional dense vector. However, they have different features.

For Skip-gram, what we do is actually train a model by using context words, just like we have a student (center word) and teachers (context words), each teacher would give information to the student. Therefore, the student would be great since he has learned from all the teachers. But the cost is high.

For CBOW, what we do is actually train a model by using center words, just like we have students (context words) and only one teacher (center word), each student would learn from the teacher. It is more efficient but each student would learn less information compared with Skip-gram.

In the following, we would introduce some preliminaries of Word2Vec and dive into the details of Skip-gram and CBOW.

3.2.1. N-gram (Preliminary)

Goal: Given a sentence containing N words, we would like to calculate the probability of the sentence.

An n-gram model is realized through the following procedure

N-gram model

1. First, we introduce the **Markov Assumption**: The probability of the sentence is only related to the previous $n-1$ words.
2. Then we have the probability of the sentence (if $n=2$ here):

$$P(\boldsymbol{S}) = P(w_1, w_2, ..., w_N) = \prod_{i=1}^{N+1} P(w_i|w_{i-1})$$

   Note that we could add the symbol $< s >$ and $< /s >$ to the beginning and the end of the sentence, which is $w_0 =< s >$ and $w_{N+1} =< /s >$
3. To calculate the conditional probability, we use the sliding window technique:
   1. For each word in the sentence, we consider it as the center word and the context words are the words in the window, and collect the word pairs. For example, if the size of the window is 1, then the context words are the words in the range of $[-1, 1]$ around the center word. For example, the sentence "I like to eat apple" will be transformed into:

$$[(\text{I}, \text{like}), (\text{like}, \text{to}), (\text{to}, \text{eat}), (\text{eat}, \text{apple})]$$

   2. Then we calculate the probability of the word pairs:

$$P(w_i, w_{i-1}) = \frac{C(w_i, w_{i-1})}{\sum_{w' \in V} C(w_{i-1}, w')}$$

      where $C(w_i, w_{i-1})$ is the count of the word pair (w_i, w_{i-1}) in the training data.
4. Finally, for given words, we calculate the probability of all possible word, and choose the word with the highest probability as the next word.

- To measure the performance of the n-gram model, we could use the **perplexity** (PP) as the metric. Given a test words set of size M, the perplexity is defined as:

$$\text{Perplexity}(\boldsymbol{S}) = P(w_1, w_2, ..., w_M)^{-\frac{1}{M}}$$

  The smaller the perplexity is, the better the model is.

- Key problem: The **sparsity** of the probability table. Sometimes there are a lot of words pairs which only appear little times in the training data, so the probability is too small.

3.2.2. BoW (Preliminary)

Bag of Words is a simple method to represent the document as a vector.

Idea: Count the frequency of each word in the sentence, and use the frequency as the feature of the word.

Bag of Words

1. Build a vocabulary from the training data.
2. Count the frequency of each word in the sentence.
3. Use the frequency as the feature of the word.

For example:

$$\text{I like to eat apple and I like to eat banana} \Rightarrow [2\ \ 2\ \ 2\ \ 2\ \ 1\ \ 1]$$

- Key problem:
  1. The **order** and **semantic** of the sentence are ignored.
  2. The **sparsity** of the vectorize result, some words may have appeared only once.

3.2.3. TF-IDF (Preliminary)

Term Frequency-Inverse Document Frequency is a method to measure the importance of a word in a document.

Idea: Use the **term frequency** to measure the importance of the word in the document, and use the **inverse document frequency** to measure the importance of the word in the whole corpus.

TF-IDF

d is the document, t is the word, D is the corpus.

1. Term Frequency (TF): The higher the TF is, the more important the word is, it could capture the topic of the document.

$$\text{TF}(t, d) = \frac{C(t, d)}{\sum_{t' \in d} C(t', d)}$$

2. Inverse Document Frequency (IDF): The lower the IDF is, the more important the word is and it's to make sure some common words are not included, such as "the", "and", "is", etc.

$$\text{IDF}(t) = \log\left(\frac{\sum_{d \in \boldsymbol{D}} \boldsymbol{I}(t \in d)}{|\boldsymbol{D}|}\right)$$

3. TF-IDF:

$$\text{TF-IDF}(t, d) = \text{TF}(t, d) \times \text{IDF}(t)$$

TF-IDF also loses the **order** of the words. For the **vectorization task** of a document, we could use the **bag of words** to represent the frequency of the words, and use the **TF-IDF** to represent the importance of the words, and we put them together to represent the document.

3.2.4. Skip-gram

Idea: Use the **center** word to predict the **context** words.

Skip-gram

Goal: Given a word and a sliding window size, predict the context words. For a sliding window of size m, we have $m-1$ context words and 1 center word.

1. Data preprocessing:
   1. Clean the text
   2. Build the vocabulary and the frequency table
   3. Remove the words with low frequency and some high frequency words with little information, such as "the", "and", "is", etc.
2. Model Building:
   1. Network design: It's a shallow neural network model with one hidden layer. Input layer is the one-hot encoding vector of the center word, the hidden layer is the embedding matrix and the output layer is a softmax layer which output the probability distribution of the context words.
   2. Loss function: $L = -\log P\left(\{w_i\}_{t-m+1}^{t+m-1} | w_t\right)$
3. Model training:
   1. Initialize the embedding matrix
   2. Calculate and update the loss function
   3. Repeat the process until the loss function is minimized

3.2.5. CBOW

Idea: Use the **context** words to predict the **center** word.

CBOW

Goal: Given a word and a sliding window size, predict the center word. For a sliding window of size m, we have $m-1$ context words and 1 center word.

1. Data preprocessing:
   1. Clean the text
   2. Build the vocabulary and the frequency table
   3. Remove the words with low frequency and some high frequency words with little information, such as "the", "and", "is", etc.
2. Model Building:
   1. Network design: Similar to Skip-gram. But since for a center word we could have m context words. Therefore, the input layer is the average of the one-hot encoding vectors of the context words, and the rest is the same as Skip-gram. Let $\{x_i\}_1^m$ be the one-hot encoding vectors of the context words.

$$x = \frac{1}{m} \sum_{i=1}^{m} x_i$$

   1. Loss function: we may use cross-entropy loss function as the loss function. We aim to maximize the probability of the center word and minimize the probability of other words.

$$L = -\log P\left(w_t | \{w_{t-m+1}\}^{t+m-1}\right)$$

3. Model training:
   1. Initialize the embedding matrix
   2. Calculate and update the loss function
   3. Repeat the process until the loss function is minimized

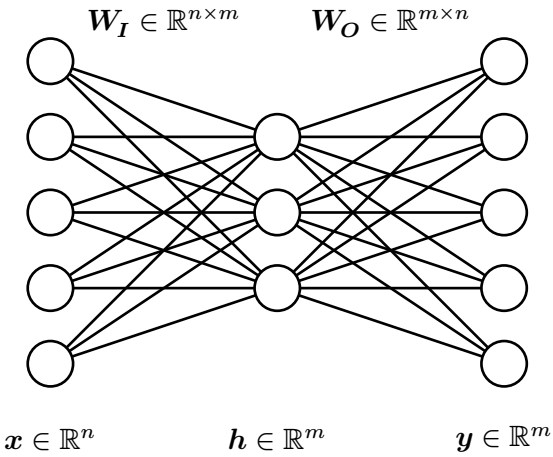
3.2.6. Modifications in Training Process

- Negative Sampling

  Before, in each round of training, we would update the embedding matrix for all the words. However, in Negative Sampling, we would randomly sample some words as negative words, and update the embedding matrix for the center word and the negative words. Negative words are the words not intended to be the output.

- Hierarchical Softmax

  It is a method to speed up the training process, especially for the output layer. Assume we have the following neural network structure:



  where $\boldsymbol{h}$ is the word vector. The way we obtain $\boldsymbol{y}$ is:

$$y_i = \frac{e^{\boldsymbol{h}^T \boldsymbol{W}_{O_i}}}{\sum_{j=1}^{N} e^{\boldsymbol{h}^T \boldsymbol{W}_{O_j}}}$$

  This is a softmax function. The denominator term is computationally expensive, so we use the hierarchical softmax to speed up this part instead of doing softmax all the time. The hierarchical softmax is based on the Huffman tree.

Huffman Tree

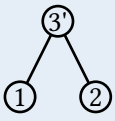
We need some concepts, for a tree:

- Tree's path length: The sum of the number of edges from root to all nodes.
- Tree's weighted path length: The sum of the products of the tree's path length and the frequency of the leaf nodes.

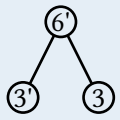
Given the two concepts, assume we have a group of frequencies $F = \{f_1, f_2, ..., f_N\}$, we could build a Huffman tree from the frequencies by putting all the frequencies as the leaf nodes. The Huffman tree is a binary tree whose tree's weighted path length is the smallest.

For example, we have a group of frequencies $F = \{1, 2, 3, 4\}$, we could build a Huffman tree as follows:

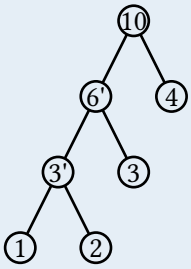
1. First, we put all the frequencies as the leaf nodes.
2. Then we continuously combine the two smallest frequencies until there is only one node left.



Then we find 3 and 3 are the two smallest frequencies, so we combine them and get a new node with frequency 6.



Then we find 6 and 4 are the two smallest frequencies, so we combine them and get a new node with frequency 10. Continue this process until there is only one node left.



The weighted tree's path length is $\sum \text{freq} \times \text{path length} = 1 \times 3 + 2 \times 3 + 3 \times 2 + 4 \times 1 = 19$. Note that the nodes with prime symbol are the internal nodes and the leaf nodes are our input.

Before, our softmax is $O(V)$ since we do a sum over operation. Now, by building a Huffman tree, we could speed up the operation to $O(\log(V))$.

The procedure is as follows:

1. We get the **frequency list** of the vocabulary, $F = \{f_1, f_2, ..., f_N\}$
2. Then we build a **Huffman tree** from the frequencies.
3. For each internal node $\boldsymbol{\theta}_k \in \{\boldsymbol{\theta}_i |\ i \in [1, V-1]]\}$ we create (actually there are $V-1$ internal nodes), our network train a corresponding binary classifier parameter $\boldsymbol{w}_k$.
4. For each binary classifier parameterized by $\boldsymbol{\theta}_k$, we use a sigmoid function to calculate the probability of **going left** and **going right**:

$$P(\text{left}) = \sigma(\boldsymbol{h}^T \boldsymbol{w}_k)$$
$$P(\text{right}) = 1 - \sigma(\boldsymbol{h}^T \boldsymbol{w}_k)$$

5. To calculate the probability of a word w_i, we calculate the probability of going the path from the root to the leaf node w_i.

The structure of Huffman tree ensures that high frequency words are closer to the root and low frequency words are closer to the leaf nodes. Therefore, we do not need that much calculation for high frequency words, which we may need to calculate many times.