Word embedding

Xing Fang

What is word embedding?

In NLP, word embedding is a process of mapping a word into a fixed-length real vector.

Why do we care about it?

Word embedding is an indispensable step for deep learning algorithms in NLP. Without word embedding, text information cannot be transformed into numeric values that are used as the input information for neural networks.

Word embedding techniques

1. One-hot or One-of-V encoding

Given a set of text information, a vocabulary is a set of all words that appear in the information set. The length (dimension) of a word vector is the size of the vocabulary, where the value on a certain index is set to be 1, if the vector is denoting a word of the index.

Pros:

* Easy to implement.
* Works for certain scenarios.

Cons:

* Lack of meanings.
* Sparsity can cause problems during training processes of certain deep learning algorithms.

1. Randomly generated real vectors

E.x.: embedding = np.random.randn(N,128), where N is the size of the vocabulary, 128 is the size of each word vector.

Pros:

* Easy to implement.
* It is indeed the technique used in some of the current deep learning libraries, such as TensorFlow and PyTorch.

Cons:

* Lack of meanings.

The meaning of a word vector:

Distributional similarity

How to obtain the similarity?

Intuition: We can get a lot of value by representing a word by means of its neighbors. “You shall know a word by the company it keeps.”

Idea: We define a model that aims to predict the probability of context words, given a center word.

Let m = 2 be the length of a word window, considering the following sentence:

Where the words: the, cat, on, the are four context words and the word, sat, is the center word ().

For each step, take one word as the center word, then try to predict the context words based on a window size. Using the notation from the Maximum Likelihood Estimation (MLE), we have:

Concretely, if the center word is sat and m=2, then the above equation is computed as:

To generalize, for each word , we will predict its surrounding words in a window of “radius”, m. The objective is to maximize the probability of any context word given the current center word:

Therefore, the objective function is the negative log-likelihood function:

To compute the posterior probability, , let the index of the context word, , be o and the index of the center word, , be c, consider the following:

Where and are word vectors for the context word and the center word respectively.

To compute the gradient, let us consider the simple case, where there is only one context word and only one center word . Hence the objective function becomes:

The skip-gram algorithm:

Step 1: Randomly generate two embedding matrices, U and V, where both matrices have the size of . N is the size of the vocabulary, d is the size of the embedding dimension, usually 64, 128, 256, or 512. Matrix U contains all context word vectors and matrix V contains all center word vectors.

Choose a fixed window size, m, where .

Step 2: For a center word, , compute for all in m. To do this, follow the steps below:

* 1. Get from V and make a column vector.
  2. Compute P = np.dot(U,), P is now an N by one vector.
  3. Apply the softmax function to P: P = softmax(P). P is now

Step 3: Compute the gradient for and the gradient for every in m. Cumulate the gradient of every

Step 4: Apply gradient descent to update matrix U based on the gradients of computed in Step 3; apply gradient descent to update matrix V based on the gradients of computed in Step 3.

Step 5: Compute J and the average of J (J divided by the total number of context words).

Step 6: For each word in the training file, treat the word as , repeat steps 2 to 5, until J coverages.

Step 7: Compute the final embedding matrix by adding U to V.