Explaining Attract-Repel Decompositions Explained

Tommy Ly

June 30, 2023

1 Introduction

Word embedding refers to a group of natural language processing techniques where words and phrases from a vocabulary are represented as real-valued vectors. It is a way of mapping words to high-dimensional vectors such that semantically similar words or words used in comparable contexts are positioned close to each other. Word embeddings capture a wealth of information about words, including their meanings, the types of contexts they appear in, their grammatical functions, and even some semantic relationships between words (such as analogies). They essentially bridge the gap between human and machine understanding of language by representing words numerically. There are various methods for generating word embeddings. The most well-known models for producing word embeddings are Word2Vec, developed by Google.

Word2Vec is a two-layer neural network that takes text as input and outputs a vector space, with each unique word in the corpus assigned to a corresponding vector in the space. Word vectors are positioned in the vector space so that words sharing common contexts in the corpus are located close to one another. Word2Vec represents words in a high-dimensional vector space where the semantic similarity between words corresponds to the spatial proximity in the vector space. Word2Vec uses two architectures to create these word embeddings: Continuous Bag of Words (CBOW) and Skip-Gram models.

2 Dot Product Embeddings and Low-Rank Decompositions of Adjacency Matrices

2.1 Dot Product Explained Through Graph

The dot product of two undirected graphs represented as adjacency matrices results in a directed graph. An undirected graph in its simplest form has no duplicate edges or loops. This means that the adjacency matrix of an undirected graph will have zeros along the diagonal and that the matrix M is equal to its transpose M^T.

Directed graphs are quite different. A directed graph contains directed edges, meaning that you can only traverse from one node to another if there is an edge directed toward that node. The directions are denoted by arrows on the edges.

In this perspective, $G = (N, E)$ represents a graph with nodes N and edges E. Each node is associated with a vector v_i, and the dot product between two vectors v_i and v_j is a measure of their similarity [PSB]

The dot product between two vectors v_i and v_j is given by:

$$s = v_i \cdot v_j.$$

In the logistic node embedding model, this dot product appears in the probability of an edge between two nodes i and j:

$$p(e_(ij)) = \sigma(v_i \cdot v_j)$$

where σ is the sigmoid function. Alongside the negative sampling scheme specified in [PSB], this model relates to the standard word2vec model.

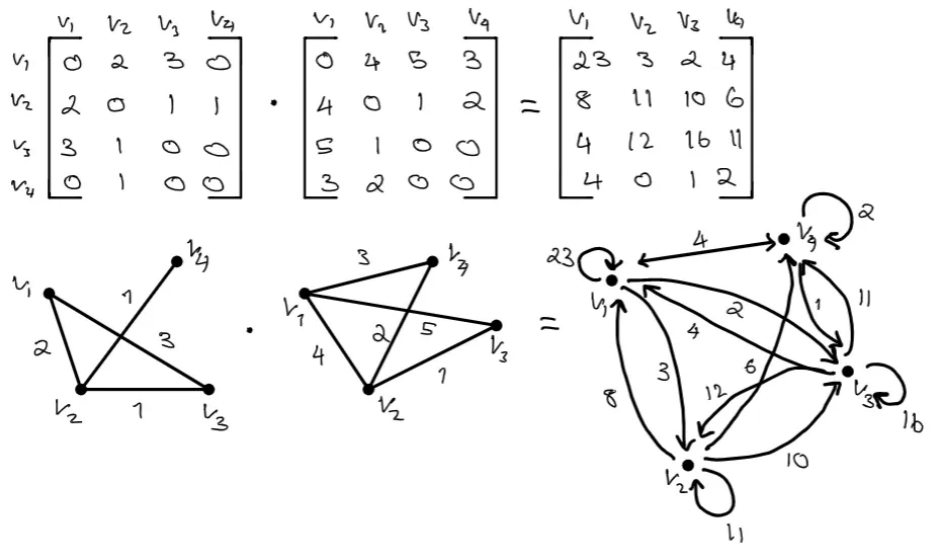

Figure 1: Dot product of two matrices

2.2 Connection between the sigmoid function and the word2vec loss function

The loss function used in Word2Vec, a popular algorithm for generating word embeddings, is called Negative Sampling, a simplification of the Noise Contrastive Estimation (NCE). Word2Vec models, specifically the Skip-gram model, try to maximize the similarity of close words on the basis of their context.

Let's start with the sigmoid function which is defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

The derivative of this sigmoid function is:

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

For Word2Vec, the goal is to maximize the log probability of a word given its context. For a target word w_O and a context word w_I, their vectors are v_{w_O} and v_{w_I} respectively. The probability of a context word given a target word is modeled as a sigmoid function:

$$P(w_I|w_O) = \sigma(v_{w_O} \cdot v_{w_I})$$

where $v_{w_O} \cdot v_{w_I}$ is the dot product of the vectors, which measures their similarity.

For negative sampling, we sample K negative examples (words that are not in the context). Let w_n be one of these negative examples. The objective is to maximize the probability of the context word and minimize the probabilities of the negative examples. Therefore, the loss function (negative log likelihood) is:

$$L = -\log(\sigma(v_{w_O} \cdot v_{w_I})) - \sum_{k=1}^{K} \log(\sigma(-v_{w_O} \cdot v_{w_{n_k}}))$$

The formula is the loss function for a variant of the Word2Vec model that uses a technique called "negative sampling". Word2Vec is a group of models used for generating word embeddings, which are high-dimensional vectors that represent words in such a way that words with similar meanings are close together in the vector space.

Let's break down the formula:

v_{w_O} and v_{w_I} are the vector representations of the output word and the input word, respectively. $v_{w_O} \cdot v_{w_I}$ is the dot product of these two vectors, which gives a measure of their similarity.

σ is the sigmoid function, which maps its input to a value between 0 and 1. In this context, it's used to transform the dot product of the word vectors into a probability.

$\log(\sigma(v_{w_O} \cdot v_{w_I}))$ is the natural logarithm of the probability that v_{w_I} is a context word of v_{w_O}. Taking the negative of this quantity (as we do in the formula) gives a measure of the "loss" or "error" associated with this prediction.

$v_{w_{n_k}}$ is the vector representation of a word that is not in the context of w_O (a "negative" sample). $\log(\sigma(-v_{w_O} \cdot v_{w_{n_k}}))$ is the natural logarithm of the probability that $v_{w_{n_k}}$ is not a context word of w_O.

$\sum_{k=1}^{K} \log(\sigma(-v_{w_O} \cdot v_{w_{n_k}}))$ is the sum of these quantities for all K negative samples.

$L = -\log(\sigma(v_{w_O} \cdot v_{w_I})) - \sum_{k=1}^{K} \log(\sigma(-v_{w_O} \cdot v_{w_{n_k}}))$ is the total loss. In the context of training the Word2Vec model, the aim is to minimize this total loss. This is done by adjusting the word vectors in such a way that the probabilities associated with the true context words are increased, and the probabilities associated with the negative samples are decreased.

Therefore, this formula essentially quantifies the error in the current word vectors with respect to their ability to correctly predict context-word relationships. This error is used to update the vectors during the training process.

So, in this context, the sigmoid function is used to squash the output of the dot product of the input and output word vectors into a probability between 0 and 1. The sigmoid function is an integral part of the Word2Vec algorithm, as it is used to convert raw score values into probabilities, making the scores interpretable and usable for computing the loss and updating the model parameters.

2.3 The Pseudo-Euclidean Similarity

The pseudo-Euclidean similarity between two nodes i and j in a graph is given by:

$$a_i \cdot a_j - r_i \cdot r_j$$

It does this by taking the attract vectors a of nodes i and j and calculating their dot product. This gives the attract component, measuring the strength of connection between the nodes. Taking the repel vectors r of nodes i and j and calculating their dot product. This gives the repel component, measuring the strength of repulsion between the nodes. Subtracting the repel component from the attract component. This gives the overall pseudo-Euclidean similarity between the two nodes. This equation is measuring whether nodes i and j have a direct connection, based on their attract and repel components. If similarity(i, j) is high, that indicates the nodes have a strong connection. If it's low or negative, that indicates repulsion between the nodes.

Python code for computing the pseudo-Euclidean similarity:

```python
import torch
import numpy as np

# List of words
word_list = ["apple", "banana", "orange", "grape"]

# Word embeddings (vector representations) for each word
word_embeddings_a = {
    "apple": np.array([0.2, 0.5, -0.1]),
    "banana": np.array([0.8, 0.3, 0.2]),
    "orange": np.array([0.6, -0.4, 0.9]),
    "grape": np.array([0.1, 0.7, 0.5])
}

word_embeddings_r = {
    "apple": np.array([0.4, 0.3, -0.2]),
    "banana": np.array([0.7, 0.1, 0.3]),
    "orange": np.array([0.3, -0.6, 0.8]),
    "grape": np.array([0.2, 0.8, 0.4])
}

```

```
# Ground truth similarity labels
labels = np.array([[1.0, 0.5, 0.2, 0.3],
                   [0.5, 1.0, 0.4, 0.1],
                   [0.2, 0.4, 1.0, 0.6],
                   [0.3, 0.1, 0.6, 1.0]])

# Convert labels to PyTorch tensor
labels_tensor = torch.tensor(labels)

# Compute similarity using the provided formula
similarity_matrix = np.zeros((len(word_list), len(word_list)))
# Iterate over each pair of words
for i in range(len(word_list)):
    for j in range(len(word_list)):
        lhs_a = torch.tensor(word_embeddings_a[word_list[i]]).unsqueeze(0)
        rhs_a = torch.tensor(word_embeddings_a[word_list[j]]).unsqueeze(0)
        lhs_r = torch.tensor(word_embeddings_r[word_list[i]]).unsqueeze(0)
        rhs_r = torch.tensor(word_embeddings_r[word_list[j]]).unsqueeze(0)

        # Compute similarity using the formula
        a_sim = torch.matmul(lhs_a, rhs_a.T).sum(1)
        r_sim = torch.matmul(lhs_r, rhs_r.T).sum(1)
        sim = a_sim - r_sim

        # Store the similarity value in the matrix, excluding diagonal elements
        if i != j:
            similarity_matrix[i, j] = sim.item()

# Convert similarity matrix to PyTorch tensor
similarity_tensor = torch.tensor(similarity_matrix)
print(similarity_tensor)

# Create a mask to exclude diagonal elements
loss_mask = torch.ones_like(labels_tensor)
torch.diagonal(loss_mask).fill_(0)

# Compute the loss using mean squared error
loss = torch.nn.MSELoss()(labels_tensor, similarity_tensor * loss_mask)

print("Loss:", loss.item())
```

2.4 Attract-Repel Decomposition

However, while these mathematical equivalences are helpful to understand what could be going on at a theoretical level in models like skip-gram embeddings with negative sampling, there are easier ways to compute embeddings.

For example, [PSB] state in Section 7.5.2: "We compute the exact low rank (k=125) AR decomposition of this matrix" referring to a dataset of 180,000+ recipes.

Python code for computing the AR decomposition:

```
# 1. Creating a simple graph
G = nx.Graph()
G.add_edge('A', 'B')
G.add_edge('A', 'C')
G.add_edge('B', 'C')
G.add_edge('B', 'D')

```

Table 1: Algorithm 1: Construct Minimal AR Decomposition

Steps
Solve the convex problem: $\min \|\hat{M}\|_*$ s.t. $\hat{M}_{ij} = e_{ij} \quad \forall i \neq j$
Compute the eigendecomposition of $\hat{M} = Q'DQ$.
If low rank is desired, then truncate the $n - k$ smallest eigenvalues to 0.
Let D^- be the strictly negative eigenvalues and D^+ be the strictly positive ones.
Let Q^- correspond to the eigenvectors with negative $\hat{M}_{ij} = e_{ij}$ for $i \neq j$, and let Q^+ be the eigenvectors with positive eigenvalues.
Set $A = Q^+\sqrt{D^+}$ and set $R = Q^-\sqrt{-D^-}$.
Rows of A are a_i, and rows of R are r_i.

```python
# 2. Constructing adjacency matrix
A = nx.adjacency_matrix(G)

# Convert to dense matrix
A_dense = A.toarray()

# 3. Solve the nuclear norm minimization problem
N = A_dense.shape[0]
M = cp.Variable((N, N), symmetric=True)
constraints = [M[i, j] == A_dense[i, j] for i in range(N) for j in range(N) if i != j]
problem = cp.Problem(cp.Minimize(cp.norm(M, "nuc")), constraints)
problem.solve()

M_hat = M.value

# Compute the eigendecomposition
eigenvalues, eigenvectors = np.linalg.eigh(M_hat)

# Separate negative and positive eigenvalues/eigenvectors
negative_indices = eigenvalues < 0
positive_indices = eigenvalues > 0

D_negative = np.diag(np.abs(eigenvalues[negative_indices]))
D_positive = np.diag(eigenvalues[positive_indices])

Q_negative = eigenvectors[:, negative_indices]
Q_positive = eigenvectors[:, positive_indices]

# Define A and R
A = np.dot(Q_positive, np.sqrt(D_positive))
R = np.dot(Q_negative, np.sqrt(D_negative))
```

To explain the code above:

1. $\|\hat{\mathbf{M}}\|_*$ is the nuclear norm of the matrix M, which is the sum of its singular values. $\hat{M}_{ij} = e_{ij} \quad \forall i \neq j$ is the constraint that the off-diagonal entries of M must equal the corresponding entries of E. e_{ij} denotes the entries of the matrix E.

2. Given a matrix $A \in R^{m \times n}$ with singular values σ_1, σ_2, ..., σ_n, the nuclear norm of A, denoted as $\|A\|_*$, is defined as:

$$\|A\|_* = \sigma_1 + \sigma_2 + \cdots + \sigma_n$$

The rank of a matrix is the number of linearly independent rows or columns it has, which is equal to the number of non-zero singular values. The intuition is that if many singular values are close to zero, their sum (i.e., the nuclear norm) will also be small.

The algorithm you have mentioned seems to be about a matrix factorization approach for approximating a given matrix, which could be used in low-rank approximation or matrix completion problems. It seems to involve spectral decomposition and some other steps to generate the factors of the input matrix $\hat{M}$. Here's the explanation:

3. Compute the eigendecomposition of $\hat{M} = Q'DQ$: This step involves finding the eigenvalues and eigenvectors of the matrix $\hat{M}$. The matrix D is a diagonal matrix that contains the eigenvalues of $\hat{M}$, and Q is a matrix where each column is an eigenvector of $\hat{M}$.

4. If low rank is desired, then truncate the $n - k$ smallest eigenvalues to 0: This step is for the low-rank approximation of $\hat{M}$. By setting the smallest eigenvalues to 0, you effectively reduce the rank of the matrix to k, because the rank of a matrix is the number of its non-zero eigenvalues.

5. Let D^- be the strictly negative eigenvalues and D^+ be the strictly positive ones: This step separates the negative and positive eigenvalues into two separate matrices. Note that D^- and D^+ are still diagonal matrices.

6. Let Q^- correspond to the eigenvectors with negative $\hat{M}_{ij} = e_{ij}$ for $i \neq j$, and let Q^+ be the eigenvectors with positive eigenvalues: Here, we associate the eigenvectors of $\hat{M}$ with their corresponding eigenvalues, separated by the sign of the eigenvalues.

7. Set $A = Q^+\sqrt{D^+}$ and set $R = Q^-\sqrt{-D^-}$: This step is essentially a square root operation on the original matrix, where the square root of a matrix is defined in terms of its eigendecomposition. Here, the square root of the positive and negative eigenvalues is taken, forming two new matrices A and R.

8. Rows of A are a_i, and rows of R are r_i: This last step defines the rows of A and R, which may be used in the next steps of your overall algorithm or analysis. The output of the above algorithm is a pair of matrices (A, R), which might be used to approximate the original matrix $\hat{M}$, or used in subsequent steps of a larger algorithm.

3 Skip-Gram models

Skip-Gram predicts the surrounding context words given a center word. The probability function 'p' is also computed using the softmax function.

The word vectors are learned by training these models on large amounts of text and using optimization techniques like stochastic gradient descent. The objective function to be maximized is:

$$J = \sum_{t=1}^{T} \sum_{j=-m}^{m} \log p(w_{t+j}|w_t)$$

where T is the number of words in the corpus, m is the window size, and $p(w_(t + j)|w_t)$ is the probability of the context word given the center word.

Let's consider the sentence: "The cat sat on the mat." The first step in Skip-gram modeling is to create pairs of center words and context words. Let's say we select a window size of 2 (2 words before and 2 words after the center word). For the center word "sat", the pairs would be:

- (sat, the)
- (sat, cat)
- (sat, on)
- (sat, the)

These pairs are our training data.

The Skip-gram model would then start with random vectors for each word, and use these to predict the context words from the center word. It uses the softmax function as the activation function, which outputs a probability distribution of context words given a center word.

Let's say for the center word "sat", the model initially predicts the following probabilities for the context words:

- P(the—sat) = 0.1
- P(cat—sat) = 0.2
- P(on—sat) = 0.4
- P(mat—sat) = 0.1
- P(is—sat) = 0.1
- P(a—sat) = 0.1

The model's objective is to adjust the word vectors such that the predicted probabilities are as close as possible to the actual distribution, which in this case is 1 for the actual context words and 0 for all other words.

This is done by calculating a loss function, such as cross-entropy loss, which measures the difference between the predicted and actual distributions, and then using backpropagation and gradient descent to adjust the word vectors to minimize this loss.

Over many iterations of this process on large amounts of data, the model eventually learns word vectors that accurately represent the semantic relationships between words.

For example, "cat" and "mat" might end up with similar vectors because they often appear in similar contexts (like "the cat is on the mat"), while "cat" and a completely unrelated word like "spaceship" would have very different vectors.

References

[PSB] Alexander Peysakhovich, Anna Klimovskaia Susmel, and Leon Bottou. Pseudo-Euclidean Attract-Repel Embeddings for Undirected Graphs.