Implementing Word2Vec:

Continuous Bag of Words (CBOW) version

Uzair Ahmad

Let's implement the Continuous Bag of Words (CBOW) version of the word2vec algorithm. The idea behind CBOW is that we use context words (words around a target word) to predict the target word itself.

For the sake of simplicity:

1. We'll limit our vocabulary.
2. We'll use a small embedding size.
3. We'll not use hierarchical softmax or negative sampling; instead, we'll use simple softmax.

Let's start:

Step 1: Preparing the Data

```
import numpy as np

corpus = [
    "I love deep learning",
    "Deep learning is fascinating",
    "Natural language processing is a subfield of deep learning",
    "Embeddings are used in deep learning",
    "Word embeddings are powerful",
    "CBOW and Skip-gram are word2vec models",
    "Neural networks power deep learning",
    "Machine learning is broader than deep learning",
    "Python is a popular programming language for deep learning",
    "TensorFlow and PyTorch are deep learning frameworks"
]

# Tokenizing and building vocabulary
tokens = [sentence.split() for sentence in corpus]
vocabulary = set([word for sentence in tokens for word in sentence])

# Assign an ID to each word
word_to_id = {word: i for i, word in enumerate(vocabulary)}
id_to_word = {i: word for word, i in word_to_id.items()}

VOCAB_SIZE = len(vocabulary)
```

Step 2: Model Definition

Let's keep our embeddings and context size small for simplicity.

```
EMBEDDING_SIZE = 5
CONTEXT_SIZE = 2  # using two words from left and two from right

# Initialize random weights
W1 = np.random.rand(VOCAB_SIZE, EMBEDDING_SIZE)
W2 = np.random.rand(EMBEDDING_SIZE, VOCAB_SIZE)
```

These two lines of code initialize the weight matrices (W1) and (W2), which are crucial components of the CBOW word2vec model. Let's break down their roles and why they're set up the way they are:

1. **Initialization of Weights:**
    - In neural networks or models like word2vec, the initial values of weights are typically set to small random values. This random initialization ensures that the model doesn't get stuck during training and can learn a good representation of the data.
2. **W1 - Input to Hidden Layer Weights (`VOCAB_SIZE` x `EMBEDDING_SIZE`):**
    - `W1` connects the input layer (representing words in our vocabulary) to the hidden layer (representing the embeddings or dense vector representations of words).
    - Each row of `W1` corresponds to a word in our vocabulary, and each row is a vector of size `EMBEDDING_SIZE`. This vector is essentially the "embedding" or dense representation of the corresponding word.
    - `VOCAB_SIZE` is the number of unique words in our corpus, so we have an embedding vector for each word.
    - `EMBEDDING_SIZE` is a user-defined parameter representing the dimensionality of the embedding space (how many numbers we use to represent each word).
3. **W2 - Hidden to Output Layer Weights (`EMBEDDING_SIZE` x `VOCAB_SIZE`):**
    - `W2` connects the hidden layer (the embeddings) to the output layer.
    - The output layer has a size of `VOCAB_SIZE` because we're trying to predict the likelihood of each word in our vocabulary being the target word.
    - Given an embedding (from the hidden layer), we multiply it by `W2` to produce scores for each word in the vocabulary. These scores are then turned into probabilities using the softmax function.

In Simple Terms:

- Imagine you're trying to describe every word in a language using a handful of numbers (this handful is `EMBEDDING_SIZE`).
- `W1` contains these sets of numbers. So, for every word in our language (`VOCAB_SIZE`), there's a corresponding set of numbers in `W1`.
- Now, when predicting a word based on its context, we use these number sets (embeddings) and `W2` to figure out which word is the most likely fit.

By training the model, we adjust these sets of numbers in `W1` and the weights in `W2` such that the model gets better at its predictions. Once training is done, the sets of numbers in `W1` represent the meaning of words in a way the computer can understand and use.

```
EMBEDDING_SIZE = 5
CONTEXT_SIZE = 2  # using two words from left and two from right

```

```
# Initialize random weights
W1 = np.random.rand(VOCAB_SIZE, EMBEDDING_SIZE)
W2 = np.random.rand(EMBEDDING_SIZE, VOCAB_SIZE)

def forward(context_word_ids):
    # Sum the vectors of the context words
    h = np.mean([W1[id] for id in context_word_ids], axis=0)

    # Produce output
    u = np.dot(h, W2)
    y_pred = np.exp(u) / np.sum(np.exp(u))

    return y_pred, h
```

Step 3: Training

Goal: The aim of the CBOW model is to predict a target word based on its surrounding context words. In other words, given some words around a blank space, we want our model to guess the word that fits in that space.

Steps in the Training Process:

1. **Preparing Context and Target:**
    - For each sentence in our dataset, we slide a window across it. This window captures a few words on the left, a target word in the middle, and a few words on the right. The words on the sides are our context, and the word in the middle is what we want to predict (the target).
2. **Making a Prediction**:
    - We convert our context words into vectors using a set of weights (let's call these weights `W1`). These vectors represent the meaning of the words.
    - We average these vectors to get a single vector that represents the combined meaning of all context words.
    - We then use another set of weights (`W2`) to transform this average vector into a prediction of the target word.
    - **Line 20:** h is the hidden layer's activations (or outputs), which can be thought of as the word vector representation of the context words. It is essentially the row from the input weight matrix `W1` corresponding to the context words.
3. **Measuring How We Did:**
    - We compare our prediction to the actual target word. If our prediction is perfect, great! But if it's not, we'll have some error.

      ```
      # Calculate loss/error
      e = y_pred
      e[target_id] -= 1
      ```

      The purpose of `e[target_id] -= 1` is to compute the error for the predicted word probabilities. For the correct word (i.e., the actual target word), we subtract 1 from its predicted probability. This is because the ideal prediction for the correct word would be 1, and for all other words, it would be 0. **Explanation**: Assuming we have a simple vocabulary of three words: `["apple", "banana", "cherry"]`. The mapping from words to their respective indices is:

- ```
  word_to_id = {"apple": 0, "banana": 1, "cherry": 2}
  ```

  Let's assume our target word for a specific context is "banana". The ideal one-hot encoded representation for this target word would be:

  ```
  [0, 1, 0]
  ```

  This vector indicates a 100% certainty (or probability of 1) for the word "banana", and 0% certainty (probability of 0) for the other words.

  Now, let's say our model predicts the probabilities for this context as:

  ```
  y_pred = [0.2, 0.5, 0.3]
  ```

  This means the model estimates:

  - A 20% chance that the word is "apple".
  - A 50% chance that the word is "banana".
  - A 30% chance that the word is "cherry".

  To compute the error for each word, we can derive it from the difference between the prediction and the desired output:

  ```
  e = y_pred
  e[word_to_id["banana"]] -= 1
  ```

  Post-execution, the error vector `e` will be:

  ```
  [0.2, -0.5, 0.3]
  ```

  Breaking it down:

  - For "apple", there's no error related to the target word, so the error remains as the predicted value: 0.2.
  - For "banana", the model predicted a probability of 0.5 when it should have been 1. This gives an error of (0.5 - 1 = -0.5).
  - For "cherry", there's no error related to the target word, so the error remains as the predicted value: 0.3.

  This `e` vector represents the difference between the model's predictions and the desired outputs. It serves as a foundation for backpropagation, guiding the adjustments made to the model's weights to reduce this discrepancy in future predictions.

  After this line, `e` essentially contains the errors (or differences) between the predicted probabilities and the desired outputs for each word in the vocabulary. For the correct word, this error will be `y_pred[target_id] - 1`, and for all other words, the error will be simply their predicted probabilities (since the ideal prediction for non-target words is 0).

4. **Learning from Our Mistakes**:
  ```

- We figure out how wrong each part of our model was (both `W1` and `W2` weights). This is a bit like asking, "Which parts of our machine need tuning to make better predictions next time?"
- A bit more detail on the **backpropagation process**:

```
dW2 = np.outer(h, e)
EH = np.dot(e, W2.T)
```

These lines continue the backpropagation process, computing the gradients for the model's parameters. Let's break down these calculations:

1. `dW2 = np.outer(h, e)`:
   - `np.outer(h, e)` computes the outer product of vectors `h` and `e`.
   - Here, `h` is the output of hidden layer (the average of the embedding vectors of the context words in the case of CBOW).
   - `e` is the error term discussed previously.
   - The outer product gives a matrix whose dimensions are `(size_of_h, size_of_vocabulary)`. This matches the dimensions of `W2`, the weight matrix connecting the hidden layer to the output layer.
   - `dW2` represents the gradient of the loss with respect to `W2`.

   Here's what we've done: calculate gradient of the loss with respect to the weights `W2`

   `dW2 = np.outer(h, e)`

   Explanation, Given: An error vector (e) that represents the error for each neuron in the output layer:

$$e = \begin{bmatrix} 0.5 \\ -0.3 \\ 0.2 \\ -0.4 \\ 0.1 \end{bmatrix}$$

An activations vector (h) from the hidden layer representing the activations from the hidden layer's neurons:

$$h = \begin{bmatrix} 0.7 \\ 0.9 \end{bmatrix}$$

The gradient of the loss with respect to the weights `W2` is

$$\nabla W2 = e \times h^T$$

The result of `np.outer(h, e)` is:

$$\nabla W2 = \begin{pmatrix} 0.7 \times 0.5 & 0.7 \times (-0.3) & 0.7 \times 0.2 & 0.7 \times (-0.4) & 0.7 \times 0.1 \\ 0.9 \times 0.5 & 0.9 \times (-0.3) & 0.9 \times 0.2 & 0.9 \times (-0.4) & 0.9 \times 0.1 \end{pmatrix}$$

$$\nabla W2 = \begin{pmatrix} 0.35 & -0.21 & 0.14 & -0.28 & 0.07 \\ 0.45 & -0.27 & 0.18 & -0.36 & 0.09 \end{pmatrix}$$

So, the gradient matrix $\nabla W2$ gives you how much the error will change for a small change in the respective weight. If you're training the network using gradient descent, you would then subtract some fraction of this gradient from the weights to minimize the error.

2. `EH = np.dot(e, W2.T)` computes the dot product of the error term `e` and the transpose of the `W2` matrix. `EH` represents the error in the hidden layer. Remember that `e` has the dimension `(1, size_of_vocabulary)`, and `W2.T` (transpose of `W2`) has the dimensions `(size_of_vocabulary, size_of_h)`. The resulting dot product `EH` will be a vector of size `(1, size_of_h)`. This computation effectively backpropagates the error from the output layer back to the hidden layer.

   After computing these gradients, they are used to update the model's weights (`W1` and `W2`) in the gradient descent process, adjusting the model to minimize the loss.

We then make small adjustments to `W1` and `W2` based on our findings. These adjustments are done in the direction that reduces our error. In the gradient update for `W1`, we distribute this error to each context word's embedding. This is why we divide `EH` by the number of context words.

```
for word_id in context_ids:
divide by the number of context words
                W1[word_id] -= learning_rate * EH / len(context_ids)
```

In the gradient update for `W2`, we distribute `dw2` error to each context word's embedding.

```
W2 -= learning_rate * dW2
```

5. **Rinse and Repeat**:

   We keep repeating steps 2-4 for each window in each sentence, over and over again (often called epochs), until our model gets better at predicting the target word from its context or until we feel the model is "good enough".

By the end of the training, we have a model that's good at filling in blanks based on surrounding words. Additionally, the weights `W1` that we've been adjusting throughout training? They contain our word embeddings – vectors that capture the meanings of words.

```
learning_rate = 0.05
epochs = 1000

for epoch in range(epochs):
    total_loss = 0
    for sentence in tokens:
        for i, word in enumerate(sentence):
            # Prepare context words
            start = max(0, i - CONTEXT_SIZE)
            end = min(len(sentence), i + CONTEXT_SIZE + 1)
            context = [sentence[j] for j in range(start, end) if j != i]

            # Skip if we don't have enough context
            if len(context) < 2 * CONTEXT_SIZE:
                continue

            context_ids = [word_to_id[w] for w in context]
            target_id = word_to_id[sentence[i]]
            # y_pred: the predicted probabilities P(target | context)
```

```
            # Prob of all words in the vocabulary being the target word given the context.
            y_pred, h = forward(context_ids)

            # Calculate the loss: -log of the probability of the correct word
            total_loss += -np.log(y_pred[target_id])

            # Calculate loss/error
            e = y_pred
            e[target_id] -= 1

            # calculate gradient of the loss with respect to the weights W2
            dW2 = np.outer(h, e)
            # spread the error back to the hidden layer
            EH = np.dot(e, W2.T)

            for word_id in context_ids:
                # divide by the number of context words
                W1[word_id] -= learning_rate * EH / len(context_ids)
            '''
            # equivalent
            for word_id in context_ids:
            # Explicitly compute gradient dW1 for the specific context word
            # Note: In this case, the word's one-hot representation would be a vector
            # with a '1' at the index 'word_id' and '0' everywhere else.
            # When multiplied with EH, it effectively selects the corresponding row from W1.
                dW1_word = np.outer(np.eye(1, len(word_to_id), word_id), EH)
                W1[word_id] -= learning_rate * dW1_word
            '''
            W2 -= learning_rate * dW2


    if epoch % 100 == 0:
        print(f"Epoch {epoch}, Loss: {total_loss}")
```

Step 4: Retrieving Embeddings

```
word_embeddings = {word: W1[word_to_id[word]] for word in vocabulary}
```

This is a basic and straightforward implementation. To improve efficiency and accuracy:

1. Introduce negative sampling.
2. Use hierarchical softmax.
3. Fine-tune hyperparameters.
4. Use a larger and cleaner dataset.

Remember, in a real-world scenario, you'd want to use optimized libraries like TensorFlow or PyTorch, which offer automatic differentiation, GPU acceleration, and other conveniences. But this example should give you an insight into the inner workings of the CBOW model.

Step 5: Visualize

Once you've trained your word vectors (which are rows in `W1`), you can visualize them using Principal Component Analysis (PCA) or t-SNE (t-distributed Stochastic Neighbor Embedding).

Here's a simple step-by-step guide on how to visualize your word vectors using PCA and matplotlib:

1. First, you'll need to import necessary libraries:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
```

2. Extract the word vectors from the `W1` matrix and prepare a list of words. The rows of `W1` represent the vectors for words.
3. Apply PCA to reduce the dimensionality:

```
pca = PCA(n_components=2)
word_vectors_2d = pca.fit_transform(W1)
```

4. Visualize the vectors:

```
plt.figure(figsize=(12,12))

for i, word in enumerate(id_to_word.values()):  # Assuming you have `id_to_word` which is the inverse of `word_to_id`
    plt.scatter(word_vectors_2d[i, 0], word_vectors_2d[i, 1])
    plt.annotate(word, xy=(word_vectors_2d[i, 0], word_vectors_2d[i, 1]))

plt.show()
```

If your vocabulary is very large, visualizing all word vectors at once might make the plot cluttered. In such cases, you can either:

- Choose a subset of interesting words.
- Use other visualization techniques like t-SNE, which can better handle the complexity of high-dimensional data, but can be more computationally intensive.

If you decide to use t-SNE, replace the PCA part with:

```
from sklearn.manifold import TSNE
word_vectors_2d = TSNE(n_components=2).fit_transform(W1)
```

This should give you a 2D visualization of your word vectors!