A Deep Dive into Word Embeddings for Data Science Interviews

This guide provides a comprehensive overview of word embeddings, a foundational concept in Natural Language Processing (NLP). It begins with the classic Word2Vec models (Skip-gram and CBOW), delving into their mathematical underpinnings, including the challenges of the Softmax function and the efficiency of Negative Sampling. The guide then expands to cover other key static embedding models like GloVe and FastText, comparing their strengths and weaknesses. Finally, it positions these classic techniques in the context of modern, contextual embeddings like BERT and provides a rich set of theoretical and practical interview questions with detailed answers and code implementations in Python and PyTorch.

Knowledge Section

1. Introduction to Word Embeddings

In NLP, we need a way to represent words numerically so that machine learning models can process them. A naive approach is **One-Hot Encoding**, where each word in the vocabulary is represented by a very long vector with a single '1' at its corresponding index and '0's everywhere else.

Problems with One-Hot Encoding:

- **Curse of Dimensionality:** For a vocabulary of 100,000 words, each word vector would have 100,000 dimensions, which is computationally inefficient and requires vast amounts of data to learn from.
- **No Semantic Relationship:** The dot product between any two distinct one-hot vectors is always zero. This means the representation contains no information about the similarity between words (e.g., "king" is no more similar to "queen" than it is to "apple").

Word Embeddings solve these problems. They are dense, low-dimensional vector representations of words that capture their semantic relationships. In an embedding space, similar words like "king" and "queen" or "happy" and "joyful" are located close to each other. These embeddings are learned from large text corpora based on the distributional hypothesis: **"a word is characterized by the company it keeps."**

2. Word2Vec (Google, 2013)

Word2Vec is a seminal framework for learning word embeddings. It's not a single algorithm but a collection of two model architectures (**CBOW** and **Skip-gram**) and two training optimizations (**Hierarchical Softmax** and **Negative Sampling**).

2.1. Model Architectures: CBOW vs. Skip-gram

The core idea is to train a shallow neural network to perform a proxy task. The learned weights of the hidden layer become the word embeddings.

- **Continuous Bag-of-Words (CBOW):** This model predicts a target (center) word based on its surrounding context words. For the sentence "The quick brown fox jumps over", if "fox" is the target word and the window size is 2, the context is `{"The", "quick", "brown", "jumps"}`. CBOW uses these context words to predict "fox". It effectively smooths over the distributional information of the context.
    - **Pros:** Faster to train and generally better for frequent words.
    - **Cons:** Can be less effective for infrequent words.
- **Skip-gram:** This model does the inverse of CBOW. It uses a target (center) word to predict its surrounding context words. Using the same example, Skip-gram would use "fox" to predict `{"The", "quick", "brown", "jumps"}`. This is typically framed as multiple prediction tasks: `fox -> The`, `fox -> quick`, etc.
    - **Pros:** Works well with small amounts of training data and represents even rare words or phrases well. It is generally considered to produce higher-quality embeddings.
    - **Cons:** Slower to train than CBOW.

Word2Vec implementations, like in `gensim`, often default to the Skip-gram model due to its superior performance on most tasks.

2.2. The Challenge: The Softmax Output Layer

Let's consider the Skip-gram model. Given a center word w_I (the input), we want to predict a context word w_O (the output). Each word has two vector representations:

- v_{w_I}: The "input" vector for word w_I (from the input-hidden weight matrix).
- v'_{w_O}: The "output" vector for word w_O (from the hidden-output weight matrix).

The probability of predicting the context word w_O given the center word w_I is defined by the **softmax function**:

$$P(w_O|w_I) = \frac{\exp({v'_{w_O}}^T v_{w_I})}{\sum_{j=1}^{|V|} \exp({v'_{w_j}}^T v_{w_I})}$$

Here, $|V|$ is the size of the vocabulary.

The Problem: The denominator requires summing over every single word in the vocabulary. If $|V|$ is large (e.g., 1 million), calculating this normalization term for every training step is computationally prohibitive.

2.3. The Solution: Negative Sampling

Instead of updating weights for all $|V|$ words, **Negative Sampling** reformulates the problem. For each training pair of `(center_word, context_word)`, which is a positive sample, we generate k negative samples. A negative sample is a pair `(center_word, random_word)`, where `random_word` is not a true context word.

The model is now trained to solve a binary classification problem:

- Given a word pair `(w_I, w_O)`, is it a true context pair? (Label = 1)
- Given a word pair `(w_I, w_k)`, is it a negative sample? (Label = 0)

We use the sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$ to model the probability. The objective function for a single positive pair (w_O, w_I) and its k negative samples $w_k \sim P_n(w)$ is to maximize:

$$J(\theta) = \log \sigma({v'_{w_O}}^T v_{w_I}) + \sum_{i=1}^{k} \mathbb{E}_{w_i \sim P_n(w)} [\log \sigma(-{v'_{w_i}}^T v_{w_I})]$$

- The first term maximizes the probability of the true context word.

- The second term maximizes the probability of the negatively sampled words *not* being the context word.

The negative samples are chosen using a unigram distribution raised to the power of 3/4 ($P_n(w) \propto U(w)^{3/4}$), which was found empirically to increase the probability of sampling less frequent words.

This approach is vastly more efficient because at each step, we only update the weights for the positive pair and the k negative samples ($1 + k$ updates) instead of all $|V|$ words.

3. GloVe: Global Vectors for Word Representation (Stanford, 2014)

Word2Vec is a "predictive" model that learns from local context windows. **GloVe** is a "count-based" model that leverages global corpus statistics.

Core Idea: The ratio of co-occurrence probabilities can encode meaning. For example, consider the words "ice" and "steam".

- The ratio $P(\text{solid}|\text{ice})/P(\text{solid}|\text{steam})$ will be very large.
- The ratio $P(\text{gas}|\text{ice})/P(\text{gas}|\text{steam})$ will be very small.
- The ratio $P(\text{water}|\text{ice})/P(\text{water}|\text{steam})$ will be close to 1.

GloVe's objective is to learn word vectors such that their dot product equals the logarithm of their co-occurrence probability.

Objective Function:
The cost function for GloVe is:

$$J = \sum_{i,j=1}^{|V|} f(X_{ij})(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log X_{ij})^2$$

Where:

- X_{ij} is the number of times word j appears in the context of word i (the co-occurrence matrix).
- w_i and $\tilde{w}_j$ are the word vectors and context word vectors, respectively.

- b_i and $\tilde{b}_j$ are bias terms.
- $f(X_{ij})$ is a weighting function that gives less weight to very common and very rare co-occurrences.

GloVe vs. Word2Vec:

- **Training:** Word2Vec learns by iterating through sentences. GloVe first builds a global co-occurrence matrix and then learns the embeddings to reconstruct it.
- **Performance:** They generally perform similarly, with each having a slight edge on different tasks. GloVe can be faster to train as it's a regression on a fixed matrix.

4. FastText (Facebook, 2016)

FastText is an extension of the Word2Vec model. Its key innovation is treating each word as a collection of **character n-grams**.

Core Idea: The word "where" with n=3 would be represented by the n-grams `<wh`, `whe`, `her`, `ere`, `re>` (plus special start/end of word tokens) and the whole word itself `<where>`. The final vector for "where" is the sum of the vectors for its constituent n-grams.

Advantages:

1. **Out-of-Vocabulary (OOV) Words:** FastText can construct a vector for an unseen word by summing the vectors of its n-grams. Word2Vec and GloVe would assign it a generic "UNK" (unknown) vector. This is a massive advantage for morphologically rich languages (like German or Turkish) and for domains with novel words (like Twitter or medical records).
2. **Morphology:** It captures subword information, so words like "nation" and "national" will share n-grams and thus have similar embeddings.

5. Practical Usage with `gensim`

`gensim` is a popular Python library for topic modeling and NLP, providing an efficient implementation of Word2Vec.

5.1 Model Training

```python
import gensim
from gensim.models import Word2Vec

# Sentences should be a list of lists of tokens (words).
# For real-world use, this would come from a large text file.
sentences = [['the', 'quick', 'brown', 'fox', 'jumps'],
             ['the', 'lazy', 'dog', 'sleeps']]

# Train a Word2Vec model
# vector_size: The dimensionality of the word vectors.
# window: The maximum distance between the current and predicted word within a sentence.
# min_count: Ignores all words with a total frequency lower than this.
# workers: Use these many worker threads to train the model (=faster training with multic
# sg=1 for Skip-gram (default), sg=0 for CBOW.
model = Word2Vec(sentences, vector_size=100, window=5, min_count=1, workers=4, sg=1)

# Save the model for later use
model.save("word2vec.model")
```

5.2 Incremental Training

You can update a trained model with new data, which is useful for domain-specific applications.

```
# Load a pre-trained model
model = Word2Vec.load("word2vec.model")

# New sentences for updating the model
new_corpus = [['a', 'fast', 'brown', 'cat', 'jumps']]

# Update the vocabulary with the new words
model.build_vocab(new_corpus, update=True)

# Further train the model on the new corpus
# Note: In modern gensim, model.iter is deprecated. Use epochs.
model.train(new_corpus, total_examples=model.corpus_count, epochs=model.epochs)

# Save the updated model
model.save("word2vec_updated.model")
```

5.3 Using the Embeddings

Once trained, the model can be used to find word similarities or the vector for a word.

```
# Load a model
model = Word2Vec.load("word2vec.model")

# Get the vector for a word
vector = model.wv['fox']
print("Vector for 'fox':", vector.shape)

# Find the most similar words
# The result is a list of (word, cosine_similarity) tuples.
similar_words = model.wv.most_similar('fox', topn=5)
print("Words similar to 'fox':", similar_words)

# Calculate similarity between two words
similarity = model.wv.similarity('fox', 'dog')
print(f"Similarity between 'fox' and 'dog': {similarity:.4f}")
```

6. Limitations and The Path to Modern NLP

(Information current as of early 2024)

Static embeddings like Word2Vec, GloVe, and FastText have a major limitation: **they generate a single, context-independent vector for each word**. The word "bank" has the same vector in "river bank" and "investment bank".

This limitation led to the development of **contextual embeddings**.

- **ELMo (Embeddings from Language Models):** Generates embeddings for a word based on the entire sentence it appears in. It uses a bidirectional LSTM to achieve this.
- **Transformers (Attention is All You Need, 2017):** Introduced the self-attention mechanism, allowing models to weigh the importance of different words in a sequence when representing a given word. This architecture is far more parallelizable than LSTMs.
- **BERT (Bidirectional Encoder Representations from Transformers):** A Transformer-based model pre-trained on a massive corpus. It learns deep bidirectional representations by considering both left and right context simultaneously. BERT revolutionized NLP by providing powerful, pre-trained models that could be fine-tuned for specific tasks, achieving state-of-the-art results.
- **GPT (Generative Pre-trained Transformer):** Another Transformer-based architecture, typically used for generative tasks. Models like GPT-3 and GPT-4 are decoder-only models that excel at text generation, summarization, and few-shot learning.

For most modern, high-performance NLP tasks, pre-trained contextual models like BERT or its variants (RoBERTa, ALBERT) are the standard starting point, not Word2Vec. However, understanding Word2Vec is crucial as it lays the conceptual groundwork for the entire field of representation learning in NLP.

Interview Questions

Theoretical Questions

Question 1: What are word embeddings and why are they generally superior to one-hot encoding for NLP tasks?

Answer:

Word embeddings are dense, low-dimensional vector representations of words that capture their semantic meaning and relationships. They are learned from large text corpora based on the principle that words appearing in similar contexts have similar meanings.

They are superior to one-hot encoding for three primary reasons:

1. **Dimensionality Reduction:** One-hot vectors are extremely high-dimensional (equal to the vocabulary size), making them computationally expensive and memory-intensive (the "curse of dimensionality"). Word embeddings are low-dimensional (e.g., 100-300 dimensions), making them far more efficient.
2. **Capture of Semantic Similarity:** The mathematical structure of one-hot vectors is orthogonal, meaning the dot product of any two different word vectors is zero. This implies that there is no notion of similarity between words. In contrast, word embeddings are learned such that semantically similar words (e.g., "cat" and "dog") are close to each other in the vector space, as measured by cosine similarity or Euclidean distance.
3. **Generalization:** Because embeddings capture relationships, a model trained on them can generalize better. If a model learns something about the word "cat," it can apply some of that knowledge to the word "dog" because their vectors are similar. This is impossible with one-hot vectors, where every word is an isolated entity.

Question 2: The standard softmax function in Word2Vec is computationally expensive. Explain why and derive the objective function for Negative Sampling as a more efficient alternative.

Answer:

Part 1: Why is the standard softmax expensive?

The standard softmax function calculates the probability of an output word w_O given an input word w_I:

$$P(w_O|w_I) = \frac{\exp({v'_{w_O}}^T v_{w_I})}{\sum_{j=1}^{|V|} \exp({v'_{w_j}}^T v_{w_I})}$$

The computational bottleneck is the denominator, which is the normalization term. To calculate this term, we must compute the dot product between the input word's vector v_{w_I} and the output vector v'_{w_j} for **every single word w_j in the vocabulary V**. If the vocabulary size $|V|$ is in the hundreds of thousands or millions, this summation becomes prohibitively expensive to compute for every training step. The complexity of a single forward pass for one training pair is $O(|V| \cdot d)$, where d is the embedding dimension.

Part 2: Deriving the Negative Sampling objective function.

Negative Sampling reframes the multi-class prediction problem (predicting the correct word out of $|V|$ possibilities) into a series of independent binary classification problems.

For each "positive" pair (w_I, w_O) (an input word and a true context word), we want to maximize its probability. We model this probability using the sigmoid function:

$$P(y=1|w_O, w_I) = \sigma({v'_{w_O}}^T v_{w_I})$$

Simultaneously, we sample k "negative" words w_i (where $i = 1, \ldots, k$) that are not in the context of w_I. For these negative pairs, we want to maximize the probability that they are *not* a true context pair, which is equivalent to minimizing their sigmoid probability:

$$P(y=0|w_i, w_I) = 1 - \sigma({v'_{w_i}}^T v_{w_I}) = \sigma(-{v'_{w_i}}^T v_{w_I})$$

The objective is to maximize the joint probability of the positive event and all k negative events, assuming they are independent. For a single training instance, the loss function to be minimized (which is the negative log-likelihood of our objective) is:

$$\mathcal{L} = -\left(\log \sigma({v'_{w_O}}^T v_{w_I}) + \sum_{i=1}^{k} \log \sigma(-{v'_{w_i}}^T v_{w_I})\right)$$

This is much more efficient because for each training step, we only need to perform computations and update weights for the positive pair and the k negative samples (a total of $1 + k$ pairs), instead of all $|V|$ words. Since k is usually small (e.g., 5-20), the computational cost is drastically reduced.

Question 3: How does FastText handle out-of-vocabulary (OOV) words, and why is this a significant advantage over Word2Vec or GloVe?

Answer:

FastText handles out-of-vocabulary (OOV) words by representing each word as a bag of character n-grams, in addition to the word itself. For example, the word "apple" with n=3 might be represented by the set of character n-grams: `<ap`, `app`, `ppl`, `ple`, `le>` (where `<` and `>` are special boundary symbols), plus the whole word token `<apple>`.

When the model encounters an OOV word during inference (a word not seen during training), it can still construct a vector for it by summing the vectors of its constituent character n-grams. For instance, if the OOV word is "apple-pie", FastText would break it down into its n-grams (`<ap`, `app`, `ppl`, `ple`, `le-`, `-pi`, `pie`, `ie>`) and sum their learned vectors to create a meaningful representation for "apple-pie".

This is a significant advantage over Word2Vec and GloVe for two main reasons:

1. **Robustness to OOV:** Word2Vec and GloVe can only assign a single, generic "UNK" (unknown) vector to all OOV words. This means the model loses all information about these words. FastText, by contrast, can generate a reasonable vector that captures morphological similarities, greatly improving performance on tasks with many novel or rare words (e.g., social media text, technical domains).
2. **Morphological Awareness:** Because related words share n-grams (e.g., "learn", "learning", "learned" all share "learn"), their final vectors will be inherently similar. This allows the model to understand morphological relationships without having seen every form of a word, which is especially powerful for morphologically rich languages like German, Finnish, or Turkish.

Question 4: Compare and contrast the underlying training philosophies of Word2Vec and GloVe.

Answer:

Word2Vec and GloVe both produce high-quality static word embeddings, but their training philosophies are fundamentally different. Word2Vec is a **predictive, local context window-based** model, while GloVe is a **count-based, global statistics** model.

- **Word2Vec (Predictive/Local):**
  - **Philosophy:** It learns embeddings by training a neural network on a proxy prediction task. For Skip-gram, the task is to predict local context words given a center word.
  - **Data Processing:** It processes text by sliding a window across the corpus and generating training samples on the fly. It never computes or stores global co-occurrence counts explicitly.
  - **Learning:** It learns implicitly. The model is never directly told about word co-occurrence counts; it infers these relationships by repeatedly trying to make correct local predictions.
- **GloVe (Count-based/Global):**
  - **Philosophy:** It's based on the idea that ratios of global co-occurrence probabilities hold the key to meaning. The model is explicitly designed to learn vectors whose dot products are related to the logarithm of their global co-occurrence count.
  - **Data Processing:** The first step is to iterate through the entire corpus once to build a large, global word-word co-occurrence matrix (X). This matrix stores how many times each word appears in the context of every other word.
  - **Learning:** The model learns by performing a weighted least-squares regression on the log-co-occurrence counts from the matrix X. It directly optimizes the embeddings to capture these global statistics.

In summary: Word2Vec learns from many small, local context windows, indirectly capturing global statistics through repeated updates. GloVe first aggregates all global co-occurrence statistics into a matrix and then learns embeddings that directly explain those statistics.

Practical & Coding Questions

Question 1: Implement the Skip-gram with Negative Sampling model from scratch in PyTorch.

Answer:

Here is a complete, well-commented implementation of Skip-gram with Negative Sampling in PyTorch. This example covers data preparation, dataset creation, the model architecture, and the training loop.

```python
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
from collections import Counter
import random

# --- 1. Hyperparameters & Data Preparation ---
# Update information: This implementation uses modern PyTorch (version 2.0+)
# and standard practices as of early 2024.

# Sample corpus
corpus_text = """
natural language processing and machine learning are fascinating fields
machine learning models can learn from data
natural language processing allows machines to understand human language
word embeddings represent words as dense vectors
skip gram and cbow are two popular word2vec models
negative sampling is an efficient training method
"""

# Hyperparameters
EMBEDDING_DIM = 50
WINDOW_SIZE = 2  # Context window is 2 words to the left and 2 to the right
BATCH_SIZE = 16
LEARNING_RATE = 0.01
EPOCHS = 50
NEGATIVE_SAMPLES = 5 # Number of negative samples per positive sample

# Preprocessing
words = corpus_text.lower().split()
word_counts = Counter(words)
vocab = sorted(word_counts, key=word_counts.get, reverse=True)
word_to_idx = {word: i for i, word in enumerate(vocab)}
idx_to_word = {i: word for i, word in enumerate(vocab)}
VOCAB_SIZE = len(vocab)

# Generate training data (center_word, context_word) pairs
training_data = []
```

```python
for i, center_word in enumerate(words):
    for j in range(max(0, i - WINDOW_SIZE), min(len(words), i + WINDOW_SIZE + 1)):
        if i == j:
            continue
        context_word = words[j]
        training_data.append((word_to_idx[center_word], word_to_idx[context_word]))

print(f"Vocabulary Size: {VOCAB_SIZE}")
print(f"Number of training pairs: {len(training_data)}")

# --- 2. Negative Sampling Distribution ---
# Create a sampling distribution based on word frequencies (as in the paper)
word_freqs = np.array([word_counts[word] for word in vocab])
unigram_dist = word_freqs / word_freqs.sum()
sampling_dist = unigram_dist ** 0.75
sampling_dist /= sampling_dist.sum()

def get_negative_samples(center_word_idx, num_samples):
    """Draw negative samples from the sampling distribution."""
    neg_samples = []
    while len(neg_samples) < num_samples:
        sample_idx = random.choices(range(VOCAB_SIZE), weights=sampling_dist, k=1)[0]
        # Ensure the negative sample is not the center word itself (though context word is
        if sample_idx != center_word_idx:
            neg_samples.append(sample_idx)
    return neg_samples


# --- 3. The PyTorch Model ---
class SkipGramNegativeSampling(nn.Module):
    def __init__(self, vocab_size, embed_dim):
        super(SkipGramNegativeSampling, self).__init__()
        self.vocab_size = vocab_size
        self.embed_dim = embed_dim

        # Two embedding layers: one for center words, one for context words
        self.in_embeddings = nn.Embedding(vocab_size, embed_dim)
        self.out_embeddings = nn.Embedding(vocab_size, embed_dim)
        # Initialize weights
        self.in_embeddings.weight.data.uniform_(-1, 1)
        self.out_embeddings.weight.data.uniform_(-1, 1)

    def forward(self, center_words, context_words, negative_words):
        # Get embeddings for the words
        # center_words: [batch_size]
        # context_words: [batch_size]
        # negative_words: [batch_size, num_negative_samples]

        center_embeds = self.in_embeddings(center_words)    # [batch_size, embed_dim]
        context_embeds = self.out_embeddings(context_words)  # [batch_size, embed_dim]
        negative_embeds = self.out_embeddings(negative_words) # [batch_size, num_negative_

        # Calculate score for positive pairs (center, context)
        # We want the dot product, so we sum over the embedding dimension
        positive_score = torch.sum(center_embeds * context_embeds, dim=1) # [batch_size]
        positive_loss = -torch.log(torch.sigmoid(positive_score) + 1e-10)

        # Calculate score for negative pairs (center, negative)
        # Use bmm (batch matrix multiplication) for efficiency
        # center_embeds.unsqueeze(1): [batch_size, 1, embed_dim]
        # negative_embeds.transpose(1, 2): [batch_size, embed_dim, num_negative_samples]
        # Resulting negative_score: [batch_size, 1, num_negative_samples] -> squeeze to [
        negative_score = torch.bmm(negative_embeds, center_embeds.unsqueeze(2)).squeeze(2
        negative_loss = -torch.sum(torch.log(torch.sigmoid(-negative_score) + 1e-10), dim

        # Total loss is the sum of positive and negative losses, averaged over the batch
        total_loss = torch.mean(positive_loss + negative_loss)
        return total_loss

# --- 4. Training Loop ---
model = SkipGramNegativeSampling(VOCAB_SIZE, EMBEDDING_DIM)
optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE)

for epoch in range(EPOCHS):
    total_loss = 0

    # Create batches
    batch_indices = np.random.choice(len(training_data), size=len(training_data), replace=

    for i in range(0, len(training_data), BATCH_SIZE):
        batch_slice = batch_indices[i:i+BATCH_SIZE]
        if not len(batch_slice): continue

        batch_center_words = []
        batch_context_words = []
        batch_negative_words = []

        for j in batch_slice:
            center_idx, context_idx = training_data[j]
            neg_samples = get_negative_samples(center_idx, NEGATIVE_SAMPLES)

            batch_center_words.append(center_idx)
            batch_context_words.append(context_idx)
            batch_negative_words.append(neg_samples)

        # Convert to tensors
        center_tensor = torch.LongTensor(batch_center_words)
        context_tensor = torch.LongTensor(batch_context_words)
        negative_tensor = torch.LongTensor(batch_negative_words)

        # Training step
        optimizer.zero_grad()
        loss = model(center_tensor, context_tensor, negative_tensor)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    if (epoch + 1) % 10 == 0:
        print(f"Epoch: {epoch+1}/{EPOCHS}, Loss: {total_loss / len(training_data):.6f}")

# --- 5. Inspecting the Embeddings ---
def find_similar_words(word, top_n=5):
    word_idx = word_to_idx[word]
    # We use the input embeddings as the final word vectors
    embeddings = model.in_embeddings.weight.data.cpu().numpy()
    word_vec = embeddings[word_idx]

    similarities = {}
    for i, vec in enumerate(embeddings):
        if i == word_idx: continue
        # Cosine similarity
        cos_sim = np.dot(word_vec, vec) / (np.linalg.norm(word_vec) * np.linalg.norm(vec)
        similarities[idx_to_word[i]] = cos_sim

    sorted_sim = sorted(similarities.items(), key=lambda item: item[1], reverse=True)
    return sorted_sim[:top_n]

print("\n--- Testing Similarity ---")
print("Words similar to 'language':", find_similar_words('language'))
print("Words similar to 'learning':", find_similar_words('learning'))
```

Question 2: You have trained a `gensim` Word2Vec model. Write a Python function to visualize the relationships between a given list of words using PCA for dimensionality reduction.

Answer:

This function demonstrates a common and practical task: interpreting the learned embeddings. By reducing their high dimensionality (e.g., 100D) down to 2D, we can plot them and visually inspect their relationships.

```python
import gensim
from gensim.models import Word2Vec
import numpy as np
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt

def visualize_word_embeddings(model, words_to_plot):
    """
    Visualizes word embeddings for a given list of words using PCA.

    Args:
        model (gensim.models.Word2Vec): A trained Word2Vec model.
        words_to_plot (list): A list of strings of words that exist in the model's vocab.
    """

    # Step 1: Get the vectors for the specified words
    word_vectors = []
    valid_words = []
    for word in words_to_plot:
        if word in model.wv:
            word_vectors.append(model.wv[word])
            valid_words.append(word)
        else:
            print(f"Warning: '{word}' not in vocabulary. Skipping.")

    if len(word_vectors) < 2:
        print("Error: Need at least two valid words to plot.")
        return

    # Convert list of vectors to a NumPy array
    word_vectors_np = np.array(word_vectors)

    # Step 2: Reduce dimensionality from N -> 2 using PCA
    # PCA is a linear dimensionality reduction technique that seeks to preserve
    # as much variance in the data as possible.
    pca = PCA(n_components=2)
    vectors_2d = pca.fit_transform(word_vectors_np)

    # Step 3: Create the plot
    plt.style.use('seaborn-v0_8-whitegrid')
    fig, ax = plt.subplots(figsize=(12, 10))

    # Scatter plot of the 2D vectors
    ax.scatter(vectors_2d[:, 0], vectors_2d[:, 1], s=50, alpha=0.7)

    # Annotate each point with its corresponding word
    for i, word in enumerate(valid_words):
        ax.annotate(word,
                    (vectors_2d[i, 0], vectors_2d[i, 1]),
                    xytext=(5, 2),
                    textcoords='offset points',
                    ha='right',
                    va='bottom',
                    fontsize=12)

    ax.set_title('2D PCA of Word Embeddings', fontsize=16)
    ax.set_xlabel('Principal Component 1', fontsize=12)
    ax.set_ylabel('Principal Component 2', fontsize=12)
    plt.show()


# --- Example Usage ---
# First, let's train a simple model for demonstration
sentences = [
    ['king', 'is', 'a', 'strong', 'man'],
    ['queen', 'is', 'a', 'wise', 'woman'],
    ['boy', 'is', 'a', 'young', 'man'],
    ['girl', 'is', 'a', 'young', 'woman'],
    ['prince', 'is', 'a', 'young', 'king'],
    ['princess', 'is', 'a', 'young', 'queen'],
    ['france', 'paris', 'capital'],
    ['germany', 'berlin', 'capital'],
    ['spain', 'madrid', 'capital']
]

# Train a small Word2Vec model
w2v_model = Word2Vec(sentences, vector_size=100, window=3, min_count=1, sg=1)
w2v_model.train(sentences, total_examples=len(sentences), epochs=50)
```

```
# List of words we want to see the relationships of
words_to_visualize = ['king', 'queen', 'man', 'woman', 'prince', 'princess', 'paris', 'ber

# Call the visualization function
visualize_word_embeddings(w2v_model, words_to_visualize)
```