5. Implement the Continuous Bag of Words (CBOW) Model

Aim: To implement the Continuous Bag of Words (CBOW) model for learning word embeddings from a text corpus.

Theory: The **Continuous Bag of Words (CBOW)** model is a popular method for learning **word embeddings**, which are dense vector representations of words that capture semantic and syntactic relationships. CBOW's objective is to predict a target word based on its surrounding context words.

The process involves:

1. **Context Window**: Defining a window size (e.g., 2 words before and 2 words after) to determine the context for a given target word.
2. **Input Representation**: The context words are treated as a "bag," meaning their order doesn't matter. They are typically represented using one-hot encoding or as indices.
3. **Output Representation**: The target word is also represented using one-hot encoding.
4. **Model Architecture**: A simple neural network typically consists of:
    - An **input layer** that takes the context words.
    - A **hidden layer** (the embedding layer) where word embeddings are learned. This layer maps each context word's one-hot vector to a dense vector (the embedding).
    - An **output layer** with a softmax activation function that predicts the probability distribution over the entire vocabulary for the target word.

The model is trained to maximize the likelihood of predicting the correct target word given its context. The learned weights in the embedding layer represent the word embeddings.

Stages:

- **a. Data Preparation**:
    - **Tokenization**: Breaking down the text corpus into individual words (tokens).
    - **Vocabulary Creation**: Building a dictionary mapping each unique word to an integer index.
    - **Stop Word Removal**: Optionally removing common words (like "the," "a," "is") that may not carry much semantic meaning.
    - **Lowercasing**: Converting all words to lowercase for consistency.
- **b. Generate Training Data**:
    - Iterate through the tokenized corpus.
    - For each word, define its context words based on the chosen window size.

    - Create input-output pairs: the input is the set of context word indices, and the output is the target word index.
- **c. Train Model**:
    - Define the neural network architecture using Keras/TensorFlow. This typically involves an embedding layer.
    - Compile the model with an optimizer (e.g., Adam) and a loss function like categorical_crossentropy (if using one-hot encoded targets) or sparse_categorical_crossentropy (if using integer targets).
    - Train the model on the generated input-output pairs.
- **d. Output**:
    - Extract the trained word embeddings from the embedding layer of the model. These embeddings can then be used for various downstream NLP tasks like sentiment analysis, text classification, or similarity calculations.

Code Example (Simplified CBOW using a small corpus):

Python

```
import tensorflow as tf

from tensorflow.keras.preprocessing.text import Tokenizer

from tensorflow.keras.preprocessing.sequence import skipgrams

from tensorflow.keras.models import Model

from tensorflow.keras.layers import Input, Dense, Embedding, Flatten

import numpy as np

import collections


# --- a. Data Preparation ---


# Sample corpus

corpus = """

The quick brown fox jumps over the lazy dog.

The dog barks, and the fox runs away.

A quick brown rabbit also jumps.

"""
```

```
# Tokenize the corpus
tokenizer = Tokenizer()
tokenizer.fit_on_texts([corpus])
word_index = tokenizer.word_index
vocab_size = len(word_index) + 1 # +1 for padding/unknown words

print(f"Vocabulary: {word_index}")
print(f"Vocabulary size: {vocab_size}")

# Convert corpus to sequences of integers
sequences = tokenizer.texts_to_sequences([corpus])[0]
print(f"Sequences: {sequences}")

# --- b. Generate Training Data (Skip-gram is used here for demonstration, CBOW concept is similar) ---
# For CBOW, we'd typically create pairs of (context_words, target_word)
# Here, we use skipgrams for simplicity as it generates pairs from sequences.
# skipgrams returns pairs of (target_word, context_word)

# Parameters for skipgrams
window_size = 2 # Context window size
# Note: For true CBOW, you'd structure this differently, but skipgrams is often used to generate pairs for word2vec-like models.

# We will generate pairs of (target_word, context_word) using skipgrams for now,
# and then adapt it to mimic CBOW's prediction goal.
# A typical CBOW implementation involves averaging context embeddings.

# A simplified approach to get CBOW data structure:
```

```
# For each word, gather its context.
# Example: "The quick brown fox" -> target: "brown", context: ["The", "quick", "fox"]

data = []
target = []
context_window = 2

# Iterate through the sequences
for i, word in enumerate(sequences):
    context_start = max(0, i - context_window)
    context_end = min(len(sequences), i + context_window + 1)
    context = sequences[context_start:i] + sequences[i+1:context_end]

    # Ensure context is not empty
    if context:
        # For CBOW, we want to predict the word 'word' from 'context'
        # We can represent context by averaging its embeddings later, or use its indices
        # Let's prepare data where input is context indices, and target is the word index
        # For simplicity in Keras, we might one-hot encode contexts or average embeddings

        # A common simplification for Keras is to use a multi-hot encoding of context
        # or directly use skipgrams pairs and adapt the model.
        # Let's stick to generating data in a way that the model can learn the prediction.
        # A typical CBOW structure in Keras: Input (context word indices) -> Embedding -> Average -> Dense -> Output

        # For demonstration, let's create pairs where target is 'word' and input is a context word from 'context'
        # This is closer to Skip-gram but can be adapted.
        # To truly implement CBOW, one would average context embeddings.
        # We'll use a simplified model that predicts target from individual context words, then
can be modified.

        # Let's generate pairs of (context_word_index, target_word_index)
        for context_word_index in context:
            data.append(context_word_index)
            target.append(word)

# Convert lists to numpy arrays
data = np.array(data)
target = np.array(target)

print(f"\nGenerated context-target pairs (simplified): {len(data)} pairs")
# print(f"Example pair (context_word_index, target_word_index): ({data[0]}, {target[0]})")


# --- c. Train Model ---
embedding_dim = 10 # Dimension of the word embeddings

# CBOW Model: Predict target word from context words
# Input layer will take context word index
context_input = Input(shape=(1,), name='context_input')

# Embedding layer: Maps word indices to dense vectors
embedding_layer = Embedding(input_dim=vocab_size, output_dim=embedding_dim,
name='word_embedding')
context_embedding = embedding_layer(context_input) # Embeddings for context words

# We need to aggregate context embeddings. In true CBOW, we average them.
# For Keras, this is often done implicitly or by custom layers.
# A simpler model might predict target from each context word individually, then average results,
# or use a multi-hot encoding for context and average after embedding.

# Let's use a simplified model: input is ONE context word, predict the target.
# This is more akin to skip-gram, but can illustrate the embedding learning.
# For a true CBOW, you'd process all context words together.

# Output layer: Predicts the target word (softmax over vocabulary)
output_layer = Dense(vocab_size, activation='softmax', name='output_layer')
output_probs = output_layer(context_embedding)

# Create the model
# This simplified model predicts target from a SINGLE context word.
# To make it CBOW, you'd need to:
# 1. input multiple context words (e.g., a sequence of indices)
# 2. pass them through the embedding layer
# 3. average the resulting embeddings
# 4. pass the averaged embedding through a Dense layer

# Let's adapt to predict target from averaging context embeddings.
# This requires a slightly more complex setup or a pre-processing step.

# For a practical Keras CBOW:
# 1. Prepare data where each training instance has multiple context word indices.
# 2. Input shape would be (num_context_words,)
# 3. Use an Embedding layer.
# 4. Use a Lambda layer to average the embeddings of context words.
# 5. Then feed into Dense output.
# Re-preparing data for actual CBOW structure:
# For each word, gather its context indices.
# Input: [context_word1_idx, context_word2_idx, ...]
# Target: target_word_idx
data_cbow = []
target_cbow = []

for i, word in enumerate(sequences):
    context_start = max(0, i - context_window)
    context_end = min(len(sequences), i + context_window + 1)
    context_indices = sequences[context_start:i] + sequences[i+1:context_end]

    if context_indices:
        # Pad context if needed to have a fixed length, or handle variable length
        # For simplicity, let's assume a fixed context window size that we can pad
        # Here, we'll use the actual context and average.
        data_cbow.append(context_indices)
        target_cbow.append(word)

# Convert to numpy arrays
data_cbow = np.array(data_cbow)
target_cbow = np.array(target_cbow)

print(f"\nCBOW data shape: {data_cbow.shape}") # (num_samples,
avg_num_context_words)
print(f"CBOW target shape: {target_cbow.shape}") # (num_samples,)

# Building the CBOW model architecture
```

```
cbow_input = Input(shape=(None,), name='cbow_input') # Variable number of context words


# Embedding layer
embedding = Embedding(input_dim=vocab_size, output_dim=embedding_dim,
name='word_embedding')
embedded_contexts = embedding(cbow_input) # Shape: (batch_size, num_context_words,
embedding_dim)


# Averaging the context embeddings
# Use tf.reduce_mean along the second axis (axis=1)
# We need a Lambda layer for this in Keras
from tensorflow.keras.layers import Lambda
import tensorflow.keras.backend as K


def average_embeddings(embedded_sequences):
    return K.mean(embedded_sequences, axis=1)


averaged_context = Lambda(average_embeddings, output_shape=(embedding_dim,),
name='average_context')(embedded_contexts)


# Output layer to predict the target word
cbow_output = Dense(vocab_size, activation='softmax',
name='output_layer')(averaged_context)


# Create the CBOW model
cbow_model = Model(inputs=cbow_input, outputs=cbow_output)
cbow_model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy', # Use sparse because target is integer
index
              metrics=['accuracy'])
```

```
cbow_model.summary()


# Train the CBOW model
print("\nTraining the CBOW model...")
# You might need to pad sequences if using fixed input shapes or handle variable lengths.
# For 'None' input shape, Keras handles variable lengths automatically for this setup.
history_cbow = cbow_model.fit(data_cbow, target_cbow, epochs=100, batch_size=16,
verbose=0) # Increased epochs for better learning


# --- d. Output ---
print("\nTraining finished. Extracting word embeddings.")


# Extract the word embeddings from the embedding layer
word_embeddings = cbow_model.get_layer('word_embedding').get_weights()[0]


print(f"Shape of word embeddings: {word_embeddings.shape}") # (vocab_size,
embedding_dim)


# Display embeddings for a few words
print("\nWord Embeddings:")
for word, i in word_index.items():
    print(f"'{word}': {word_embeddings[i][:5]}...") # Print first 5 dimensions


# You can now use these embeddings for similarity calculation, etc.
# For example, finding words similar to 'fox':
# Calculate cosine similarity between 'fox' vector and all other vectors.
```

Conclusion: The CBOW model provides a powerful way to learn dense vector representations of words that capture semantic relationships. By predicting a target word from its context, the model learns embeddings where similar words tend to have similar vector representations. This allows for various downstream NLP tasks and a deeper understanding of word meanings.