Ex-9 Lab Manual: Implement Word Embeddings for IMDB Dataset

---

1. Objective

To understand and implement **word embeddings** for the **IMDB movie reviews dataset**, using deep learning frameworks such as TensorFlow/Keras. The goal is to convert text data into dense vector representations that capture semantic relationships.

2. Introduction to Word Embeddings

Word embeddings are a type of word representation that allows words with similar meaning to have a similar representation in vector space. Unlike **one-hot encoding**, word embeddings provide **dense and lower-dimensional** vector representations, capturing relationships between words.

Popular Word Embedding Techniques:

- Word2Vec
- GloVe (Global Vectors for Word Representation)
- FastText
- Pre-trained embeddings (e.g., Word2Vec, GloVe, BERT)
- Embeddings learned during training (Keras Embedding Layer)

In this lab, we will use **Keras' Embedding Layer** to train word embeddings on the **IMDB movie reviews dataset**.

3. System Requirements

Hardware:

- Computer with at least **4GB RAM** (8GB recommended)
- CPU/GPU support for faster computation (optional)

Software:

- Python (>=3.6)
- TensorFlow/Keras
- NumPy, Matplotlib (for visualization)

Install missing libraries using:

```
pip install numpy tensorflow matplotlib
```

4. Step-by-Step Procedure

Step 1: Import Required Libraries

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.datasets import imdb
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Flatten, Dense
import matplotlib.pyplot as plt
```

Step 2: Load and Preprocess IMDB Dataset

The IMDB dataset contains **50,000 movie reviews** labeled as **positive** or **negative**.

```
# Load IMDB dataset with only the top 10,000 most frequent words
vocab_size = 10000
max_length = 100  # Maximum words per review

(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=vocab_size)

# Pad sequences to ensure uniform length
x_train = pad_sequences(x_train, maxlen=max_length, padding='post')
x_test = pad_sequences(x_test, maxlen=max_length, padding='post')
```

Step 3: Define the Model with an Embedding Layer

```
embedding_dim = 16  # Dimension of word embedding vectors
model = Sequential([
    Embedding(input_dim=vocab_size, output_dim=embedding_dim, input_length=max_length),
    Flatten(),  # Converts 2D embedding output to 1D
    Dense(16, activation='relu'),
```

```
    Dense(1, activation='sigmoid')  # Binary classification (positive/negative review)
])

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
model.summary()
```

Step 4: Train the Model

```
history = model.fit(x_train, y_train, epochs=10, batch_size=32, validation_data=(x_test, y_test))
```

Step 5: Visualize Word Embeddings

After training, we can extract and visualize the word embeddings.

```
# Extract embeddings from the first layer
embedding_layer = model.layers[0]
weights = embedding_layer.get_weights()[0]
print("Shape of the embedding matrix:", weights.shape)
```

Step 6: Evaluate the Model

```
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"Test Accuracy: {test_acc:.4f}")
```

Step 7: Plot Training History

```
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

5. Observations and Results

- The **Embedding Layer** successfully converts words into dense vectors.
- Training accuracy should improve over epochs, with validation accuracy indicating generalization.
- Embeddings can be extracted and further analyzed.

6. Troubleshooting

- **Overfitting?** Reduce model complexity or apply dropout layers.
- **Low accuracy?** Increase embedding dimensions or train for more epochs.
- **Memory issues?** Reduce vocab_size or max_length.

7. Additional Tasks

- Use **pre-trained embeddings** (Word2Vec, GloVe) instead of learning new ones.
- Visualize embeddings using **t-SNE or PCA**.
- Experiment with different embedding dimensions.

8. Conclusion

Word embeddings significantly improve text classification by capturing relationships between words. Unlike one-hot encoding, embeddings provide compact and meaningful word representations, improving model performance.

End of Lab Manual

Word Embeddings: A Detailed Explanation

1. Introduction

Word embeddings are a technique used in **Natural Language Processing (NLP)** to represent words in a **continuous vector space**, where words with similar meanings have similar numerical representations. Unlike traditional text representation methods like **one-hot encoding**, word embeddings capture the **semantic relationships** between words.

---

2. What are Word Embeddings?

Word embeddings are dense numerical representations of words in a multi-dimensional space. Instead of treating words as isolated symbols, embeddings allow models to understand the **context** and **similarity** between words.

For example:

- In a **one-hot encoding** representation, the words *king*, *queen*, and *man* would be unrelated.
- In a **word embedding** representation, the vectors for *king* and *queen* will be closer to each other, while *king* and *man* will share some similarities but also have distinct differences.

Example of Word Embeddings in a 3D Space

Word	Vector Representation (Example)
King	[0.7, 0.5, 0.3]
Queen	[0.6, 0.4, 0.3]
Man	[0.5, 0.6, 0.2]
Woman	[0.4, 0.5, 0.3]

---

3. Why Use Word Embeddings?

Traditional text representations like **one-hot encoding** and **bag-of-words (BoW)** suffer from:

- **High dimensionality** (one-hot encoding requires a vector of size equal to the vocabulary).
- **Lack of semantic meaning** (words are treated as independent symbols).

- **Sparse representations** (mostly zeros in the vectors).

Word embeddings solve these issues by:

✅ **Reducing dimensionality** – Instead of thousands of dimensions, embeddings use smaller fixed-size vectors (e.g., 50, 100, or 300).

✅ **Capturing relationships** – Similar words have similar vector representations.

✅ **Efficient learning** – Models can generalize better to unseen words.

---

4. Types of Word Embeddings

4.1 Pre-trained Embeddings

These embeddings are **pre-trained** on large corpora and can be used directly in models. Examples include:

- **Word2Vec** – Developed by Google, uses **CBOW** (Continuous Bag of Words) and **Skip-Gram** methods.
- **GloVe (Global Vectors for Word Representation)** – Developed by Stanford, based on word co-occurrence.
- **FastText** – Developed by Facebook, extends Word2Vec by considering subwords.
- **BERT (Bidirectional Encoder Representations from Transformers)** – Contextual embeddings trained using deep learning.

4.2 Trainable Embeddings

Embeddings can also be learned **during training** of a model. This is common in **deep learning models** using frameworks like **TensorFlow/Keras**.

---

5. Implementing Word Embeddings in Python

5.1 Using Keras Embedding Layer (Trainable Embeddings)

```
import tensorflow as tf
from tensorflow.keras.layers import Embedding
import numpy as np
# Example: Vocabulary size = 5000, Embedding dimension = 50, Sentence length = 10
embedding_layer = Embedding(input_dim=5000, output_dim=50, input_length=10)
# Example sentence (integer-encoded)
sentence = np.array([[1, 45, 23, 678, 345, 9, 27, 4, 89, 2]])
```

```
# Get the embedded representation
embedded_output = embedding_layer(sentence)
print(embedded_output.shape)  # Output: (1, 10, 50)
```

5.2 Using Pre-trained Word2Vec Embeddings (Gensim)

```
import gensim.downloader as api
# Load pre-trained Word2Vec embeddings
word2vec_model = api.load("word2vec-google-news-300")
# Get vector for a word
vector = word2vec_model["king"]
print("Vector for 'king':", vector[:10])  # Print first 10 values
```

---

6. Applications of Word Embeddings

Word embeddings are widely used in:

✅ **Text classification** (e.g., sentiment analysis, spam detection).
✅ **Machine translation** (e.g., English to French translation).
✅ **Chatbots and virtual assistants** (e.g., Siri, Alexa).
✅ **Named Entity Recognition (NER)** (e.g., identifying names, dates, locations).
✅ **Search engines** (e.g., Google uses embeddings for query expansion).

---

7. Limitations of Word Embeddings

❌ **Lack of context** – Traditional embeddings like Word2Vec and GloVe assign the same vector to a word, regardless of its meaning in different contexts.
❌ **Require large training data** – To get meaningful embeddings, large corpora are needed.
❌ **Sensitive to rare words** – Words that appear infrequently may get poor-quality embeddings.

Solution: Use Contextual Embeddings (BERT, ELMo), which adjust word vectors based on context.

Comparison of Word2Vec, GloVe, and FastText

Word embeddings are a key component of **Natural Language Processing (NLP)**. Among the most popular embedding techniques are **Word2Vec**, **GloVe**, and **FastText**. Below is a comparison based on their working principles, advantages, and disadvantages.

---

1. Overview of the Three Methods

Method	Developer	Year	Key Idea
Word2Vec	Google	2013	Learns word vectors based on their context in a sentence. Uses **Skip-Gram** and **CBOW** models.
GloVe	Stanford	2014	Learns word embeddings based on word **co-occurrence** in a large corpus.
FastText	Facebook	2016	Extends Word2Vec by learning **subword** representations, making it effective for rare or misspelled words.

---

2. Working Mechanism

Feature	Word2Vec	GloVe	FastText
Training Approach	Predicts words based on context (CBOW) or predicts context words from a given word (Skip-Gram).	Uses a word co-occurrence matrix to generate embeddings.	Similar to Word2Vec but breaks words into character n-grams (subwords).
Model Type	Predictive	Count-based	Predictive (like Word2Vec)
Corpus Dependency	Trained on local context (window-based approach).	Trained on global statistics (word co-occurrence).	Uses subwords, making it robust for small datasets.
Representation of Words	Single vector per word.	Single vector per word.	Multiple vectors per word (subword-based).

---

3. Key Differences

3.1 Handling of Out-of-Vocabulary (OOV) Words

✅ **FastText handles unseen words well** because it learns embeddings for subwords (e.g., “play” and “playing” share common subwords like “pla” and “lay”).
❌ **Word2Vec and GloVe do not handle OOV words**, meaning new words not seen during training will have no vector representation.

3.2 Contextual Information

✅ **GloVe captures global context better** since it learns word relationships based on co-occurrence across the entire corpus.
✅ **Word2Vec captures local context better**, focusing on word sequences within a short window.
✅ **FastText captures both global and local information** due to its use of subwords.

3.3 Computational Efficiency

✅ **Word2Vec is faster to train than GloVe** because it does not rely on large matrix factorization.
✅ **GloVe requires more memory and computation** due to its reliance on word co-occurrence matrices.
❌ **FastText is computationally heavier than Word2Vec** due to the added complexity of subword representations.

---

4. Example of Word Representations

Word2Vec Example (Google's Word2Vec Pre-trained Model)

- **King - Man + Woman ≈ Queen**
- Captures syntactic and semantic relationships well.

GloVe Example (Stanford GloVe Pre-trained Model)

- **France - Paris + Berlin ≈ Germany**
- Captures co-occurrence patterns across large text corpora.

FastText Example (Facebook's FastText Model)

- **Play, Played, Playing** share common subwords like "pla", "lay".
- Can generate vectors for rare or misspelled words like "misstake" (instead of “mistake”).

---

5. Strengths and Weaknesses

Feature	Word2Vec	GloVe	FastText
Captures Context	✅ Local Context	✅ Global Context	✅ Local + Global Context
Handles OOV Words	❌ No	❌ No	✅ Yes
Computational Efficiency	✅ Fast	❌ Slower (Matrix Factorization)	❌ Slower (Subword Representation)
Pre-trained Models Available?	✅ Yes	✅ Yes	✅ Yes
Works Well for	Small datasets, word analogies, local context.	Large datasets, overall word similarity.	Morphologically rich languages, rare words.

---

6. When to Use Each Method?

Use Case	Recommended Method
General NLP tasks (Text Classification, Sentiment Analysis)	Word2Vec or GloVe
Handling unseen words, spelling variations, morphological differences	FastText
Small datasets with strong local context needs	Word2Vec
Large-scale corpus with strong global word relationships	GloVe
Languages with complex morphology (e.g., German, Finnish)	FastText

---