IMDB_Word_Embeddings_Final_1

February 21, 2025

[ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

[ ]:
```
%cd drive/MyDrive/Deep_Learning_With_Tensorflow
```

```
/content/drive/MyDrive/Deep_Learning_With_Tensorflow
```

0.0.1 Link to decription about IMDB Dataset

https://keras.io/api/datasets/imdb/#:~:text=This%20is%20a%20dataset%20of,of%20word%20indexes%20(intege

This is a dataset of 25,000 movies reviews from IMDB, labeled by sentiment (positive/negative). Reviews have been preprocessed, and each review is encoded as a list of word indexes (integers). For convenience, words are indexed by overall frequency in the dataset. This allows for quick filtering operations such as: "only consider the top 10,000 most common words, but eliminate the top 20 most common words".

As a convention, "0" does not stand for a specific word, but instead is used to encode the pad token.

When you call imdb.load_data(num_words=10000), TensorFlow processes the dataset as follows:

Reserves special tokens at the beginning:

Index 0 → (Padding token)

Index 1 → (Start of sequence)

Index 2 → (Unknown word)

Index 3 → (Unused placeholder)

Shifts all actual words by +3:

The original IMDB word_index (from imdb.get_word_index()) starts at 1 for the most frequent word. But when you call imdb.load_data(), it shifts everything by +3, so the most frequent word moves to index 4 instead of 1.

Let's assume the original IMDB word_index (from imdb.get_word_index()) looks like this:

{

```
"the": 1,      # Most frequent word

"and": 2,

"a": 3,

"of": 4,

"to": 5,

...
```

}

After calling imdb.load_data(), the dataset's indices are shifted by +3:

python Copy Edit {

```
0: "<PAD>",
1: "<START>",
2: "<UNK>",
3: "<UNUSED>",
4: "the",    # Most frequent word moves from index 1 → index 4
5: "and",    # Moves from index 2 → index 5
6: "a",      # Moves from index 3 → index 6
7: "of",     # Moves from index 4 → index 7
8: "to",     # Moves from index 5 → index 8
...
```

}

[ ]:
```python
print(max_index)
```

```
9999
```

[ ]:
```python
print(x_train[0])
```

```
[    5    25   100    43   838   112    50   670 22665     9    35   480
   284     5   150     4   172   112   167 21631   336   385    39     4
   172  4536  1111    17   546    38    13   447     4   192    50    16
     6   147  2025    19    14    22     4  1920  4613   469     4    22
    71    87    12    16    43   530    38    76    15    13  1247     4
    22    17   515    17    12    16   626    18 19193     5    62   386
    12     8   316     8   106     5     4  2223  5244    16   480    66
  3785    33     4   130    12    16    38   619     5    25   124    51
    36   135    48    25  1415    33     6    22    12   215    28    77
    52     5    14   407    16    82 10311     8     4   107   117  5952
    15   256     4 31050     7  3766     5   723    36    71    43   530
   476    26   400   317    46     7     4 12118  1029    13   104    88
     4   381    15   297    98    32  2071    56    26   141     6   194
```

```
 7486    18     4   226    22    21   134   476    26   480     5   144
   30  5535    18    51    36    28   224    92    25   104     4   226
   65    16    38  1334    88    12    16   283     5    16  4472   113
  103    32    15    16  5345    19   178    32]
```

[ ]:
```
%cd /content/drive/MyDrive/Deep_Learning_With_Tensorflow
```

```
/content/drive/MyDrive/Deep_Learning_With_Tensorflow
```

[ ]:
```
import numpy as np
from tensorflow.keras.datasets import imdb
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Flatten, Dense
from tensorflow.keras.preprocessing.sequence import pad_sequences

# Load IMDB dataset (restrict vocabulary to 10,000 most frequent words)
num_words = 10000  # Vocabulary size limit
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=num_words)

# Pad sequences to a fixed length
max_length = 200
x_train = pad_sequences(x_train, maxlen=max_length)
x_test = pad_sequences(x_test, maxlen=max_length)

# Define vocabulary size correctly
vocab_size = num_words  # Fix vocab size

# Build a simple neural network with an embedding layer
embedding_dim = 50
model = Sequential([
    # Embedding(input_dim=vocab_size, output_dim=embedding_dim,␣
 ↪input_length=max_length),

    Embedding(input_dim=vocab_size,␣
 ↪output_dim=embedding_dim,input_shape=(max_length,)),
    Flatten(),
    Dense(1, activation='sigmoid')
])

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])

# Display the model summary
model.summary()

# Train the model
```

```python
model.fit(x_train, y_train, epochs=3, batch_size=32)  # Train for a small number of epochs for demonstration

# Step 9: Evaluate Model
loss, accuracy = model.evaluate(x_test, y_test)
print(f"Test Accuracy: {accuracy * 100:.2f}%")



# Extract the learned word embeddings
embedding_layer = model.layers[0]
weights = embedding_layer.get_weights()[0]  # Shape: (vocab_size, embedding_dim)

# Load IMDB word index and **correctly align indices**
imdb_word_index = imdb.get_word_index()




#print(imdb_word_index.items())
# Reconstruct word index **as per load_data() conventions**
word_index = {word: (index + 3) for word, index in imdb_word_index.items()}  # Shift indices by 3
word_index["<PAD>"] = 0
word_index["<START>"] = 1
word_index["<UNK>"] = 2
word_index["<UNUSED>"] = 3


# Reverse lookup dictionary for saving embeddings
reverse_word_index = {i: word for word, i in word_index.items()}

print("\n Printing reverse word index first few items \n")

print(reverse_word_index.get(0), " ", reverse_word_index.get(1), " ", reverse_word_index.get(2),"  ",reverse_word_index.get(3))

# Save the learned word embeddings correctly
with open("word_embeddings.txt", "w", encoding="utf-8") as file:
    for i in range(1, vocab_size):  # Skip padding index (0)
        word = reverse_word_index.get(i, "<UNK>")  # Use <UNK> for missing words
        embedding = " ".join(map(str, weights[i]))  # Convert embedding to space-separated string
        file.write(f"{word} {embedding}\n")

print("Word embeddings saved to word_embeddings.txt")
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/embedding.py:93:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When
using Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
  super().__init__(**kwargs)
```

Model: "sequential_4"

```
 Layer (type)                          Output Shape                      ␣
 ↪Param #

 embedding_4 (Embedding)               (None, 200, 50)                   ␣
 ↪500,000

 flatten_4 (Flatten)                   (None, 10000)                         ␣
 ↪  0

 dense_4 (Dense)                       (None, 1)                          ␣
 ↪10,001
```

Total params: 510,001 (1.95 MB)

Trainable params: 510,001 (1.95 MB)

Non-trainable params: 0 (0.00 B)

```
Epoch 1/3
782/782              3s 3ms/step -
accuracy: 0.6785 - loss: 0.5684
Epoch 2/3
782/782              2s 2ms/step -
accuracy: 0.9240 - loss: 0.2080
Epoch 3/3
782/782              3s 2ms/step -
accuracy: 0.9810 - loss: 0.0879
782/782              2s 2ms/step -
accuracy: 0.8600 - loss: 0.3418
Test Accuracy: 86.26%

 Printing reverse word index first few items

<PAD>   <START>   <UNK>    <UNUSED>
Word embeddings saved to word_embeddings.txt
```

[ ]:
```python
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Dense, Flatten
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.datasets import imdb

# Load GloVe embeddings (50 dimensions)
glove_path = "glove.6B.50d.txt"  # Update path if necessary
embedding_dim = 50  # GloVe dimension
max_features = 10000  # Vocabulary size (top words from IMDB)
max_len = 200  # Maximum sequence length

# Step 1: Load IMDB dataset
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=max_features)

# Step 2: Load IMDB word index and adjust special tokens
imdb_word_index = imdb.get_word_index()

# Shift indices to match IMDB's encoding (0: PAD, 1: START, 2: UNK)
word_index = {word: (index + 3) for word, index in imdb_word_index.items()}
word_index["<PAD>"] = 0
word_index["<START>"] = 1
word_index["<UNK>"] = 2
word_index["<UNUSED>"] = 3

# Create reverse lookup dictionary
inverse_word_index = {i: word for word, i in word_index.items()}

# Step 3: Load GloVe embeddings
glove_embeddings = {}
with open(glove_path, "r", encoding="utf-8") as f:
    for line in f:
        values = line.strip().split()
        word = values[0]  # First entry is the word
        vector = np.asarray(values[1:], dtype='float32')  # Remaining are␣
↪vector values
        glove_embeddings[word] = vector

# Step 4: Create embedding matrix (mapping IMDB words to GloVe vectors)
embedding_matrix = np.zeros((max_features, embedding_dim))

for word, i in word_index.items():
    if i < max_features:
        embedding_vector = glove_embeddings.get(word.lower())  #␣
↪Case-insensitive match
        if embedding_vector is not None:
```

```python
            embedding_matrix[i] = embedding_vector  # Assign GloVe vector

# Step 5: Convert text to padded sequences
x_train_seq = pad_sequences(x_train, maxlen=max_len)
x_test_seq = pad_sequences(x_test, maxlen=max_len)

# Step 6: Define Model with Pre-Trained Embedding Layer
model = Sequential([
    Embedding(input_dim=max_features, output_dim=embedding_dim, weights=[embedding_matrix], trainable=True),
    Flatten(),
    Dense(10, activation='relu'),
    Dense(1, activation='sigmoid')
])

# Step 7: Compile and Train Model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
model.fit(x_train_seq, y_train, epochs=5, batch_size=32, validation_data=(x_test_seq, y_test))

# Step 8: Save Updated Embeddings Correctly
updated_embeddings = model.layers[0].get_weights()[0]

with open("updated_glove_embeddings.txt", "w", encoding="utf-8") as f:
    for i in range(1, max_features):  # Skip index 0 (padding)
        word = inverse_word_index.get(i, "<UNK>")
        vector_str = " ".join(map(str, updated_embeddings[i]))
        f.write(f"{word} {vector_str}\n")

print("Updated GloVe embeddings saved to updated_glove_embeddings.txt")

# Step 9: Evaluate Model
loss, accuracy = model.evaluate(x_test_seq, y_test)
print(f"Test Accuracy: {accuracy * 100:.2f}%")
```

```
Epoch 1/5
782/782              8s 7ms/step -
accuracy: 0.5654 - loss: 0.6727 - val_accuracy: 0.7897 - val_loss: 0.5083
Epoch 2/5
782/782              8s 5ms/step -
accuracy: 0.8525 - loss: 0.4240 - val_accuracy: 0.8345 - val_loss: 0.4172
Epoch 3/5
782/782              3s 4ms/step -
accuracy: 0.9176 - loss: 0.2818 - val_accuracy: 0.8437 - val_loss: 0.3968
Epoch 4/5
782/782              6s 5ms/step -
```

```
accuracy: 0.9519 - loss: 0.1912 - val_accuracy: 0.8369 - val_loss: 0.4144
Epoch 5/5
782/782              5s 6ms/step -
accuracy: 0.9686 - loss: 0.1379 - val_accuracy: 0.8407 - val_loss: 0.4615
Updated GloVe embeddings saved to updated_glove_embeddings.txt
782/782              2s 2ms/step -
accuracy: 0.8410 - loss: 0.4547
Test Accuracy: 84.07%
```

[ ]:
```
for i in range(5):  # Check first 5 rows
    word = reverse_word_index.get(i, "<UNK>")
    print(f"Index {i}: {word}, Embedding: {weights[i][:5]}")
```

```
Index 0: <PAD>, Embedding: [ 0.00251354 -0.00647312 -0.01036933  0.02173389
-0.00247443]
Index 1: <START>, Embedding: [-0.02398156  0.03206484  0.12111472 -0.04793818
-0.12537038]
Index 2: <UNK>, Embedding: [-0.03338226  0.01270346 -0.03153225 -0.04895811
0.09749824]
Index 3: <UNUSED>, Embedding: [-0.02883574  0.04383698  0.01707685  0.01299843
-0.04851248]
Index 4: the, Embedding: [ 0.10981566  0.06363965  0.05287238  0.0558015
-0.01431246]
```