RNN Model For Text and Sequence Data Analysis

Load the IMDB Data Manually from Directory

```
import zipfile
import os

# Specify the path to the zip file and the output directory
zip_file_path = "/content/archive.zip"  # Replace this with the actual path to your zip file
output_directory = "/content/IMDB text and seq"  # Replace this with the desired output path

# Create the output directory if it doesn't exist
os.makedirs(output_directory, exist_ok=True)

# Unzip the file
with zipfile.ZipFile(zip_file_path, 'r') as zip_ref:
    zip_ref.extractall(output_directory)

print("File unzipped successfully!")
```

```
File unzipped successfully!
```

Listing all the files from the unzip folder

```
import os

# Specify the path to the folder
folder_path = '/content/IMDB text and seq/aclImdb'  # Replace this with the actual path to your folder

# List all files in the directory
files = os.listdir(folder_path)

# Print the list of files
for file in files:
    print(file)
```

```
test
train
imdb.vocab
imdbEr.txt
README
```

Task 1

1. Cutoff reviews after 150 words.
2. Restrict training samples to 100.
3. Validate on 10,000 samples.
4. Consider only the top 10,000 words.

```
import os
import numpy as np

# Load reviews (you need to read them from 'train' and 'test' directories)
def load_reviews(directory, max_reviews=100, max_words=150, top_words=10000):
    all_reviews = []

    # Walk through the directory and read files
    for subdir, _, files in os.walk(directory):
        for file_name in files:
            if len(all_reviews) >= max_reviews:
                break
            file_path = os.path.join(subdir, file_name)
            with open(file_path, 'r', encoding='utf-8') as file:
                review = file.read()
                words = review.split()[:max_words]  # Limit to 150 words
                all_reviews.append(' '.join(words))

    return all_reviews

# Limit training samples to 100
train_reviews = load_reviews('/content/IMDB text and seq/aclImdb/train', max_reviews=100, max_words=150)

# Limit validation samples to 10,000
val_reviews = load_reviews('/content/IMDB text and seq/aclImdb/test', max_reviews=10000, max_words=150)

print("Sample of a processed training review:", train_reviews)
```

```
Sample of a processed training review: ['http://www.imdb.com/title/tt0453418/usercomments http://www.imdb.com/title/tt0453418/usercomments http://www.imdb.com/title/tt0453418/usercomments http://www.imdb.com/title/tt0064354/u
```

```
test_reviews = load_reviews('/content/IMDB text and seq/aclImdb/test', max_reviews=10000, max_words=150)
print("Sample of a processed test review:", test_reviews[0])
```

```
Sample of a processed test review: http://www.imdb.com/title/tt0406816/usercomments http://www.imdb.com/title/tt0406816/usercomments http://www.imdb.com/title/tt0406816/usercomments http://www.imdb.com/title/tt0406816/usercom
```

```
import os
import numpy as np
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from sklearn.model_selection import train_test_split

# Load reviews function (use your earlier code to load training and test sets)
def load_reviews(directory, max_reviews=None):
    reviews = []
    labels = []

    for label in ['pos', 'neg']:  # Assuming 'pos' and 'neg' subdirectories
        label_dir = os.path.join(directory, label)
        for file_name in os.listdir(label_dir):
            if max_reviews and len(reviews) >= max_reviews:
                break
            file_path = os.path.join(label_dir, file_name)
            with open(file_path, 'r', encoding='utf-8') as file:
                review = file.read()
                reviews.append(review)
                labels.append(1 if label == 'pos' else 0)

    return reviews, labels

# Load data
train_reviews, train_labels = load_reviews('/content/IMDB text and seq/aclImdb/train', max_reviews=100)
test_reviews, test_labels = load_reviews('/content/IMDB text and seq/aclImdb/test', max_reviews=10000)

# Tokenize and pad sequences
tokenizer = Tokenizer(num_words=10000)
tokenizer.fit_on_texts(train_reviews)

train_sequences = tokenizer.texts_to_sequences(train_reviews)
train_padded = pad_sequences(train_sequences, maxlen=150)

test_sequences = tokenizer.texts_to_sequences(test_reviews)
test_padded = pad_sequences(test_sequences, maxlen=150)
```

```
# Convert labels to numpy arrays
train_labels = np.array(train_labels)
test_labels = np.array(test_labels)
```

```
import os
import random
import numpy as np
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences

# Path to original train and test data
train_dir = '/content/IMDB text and seq/aclImdb/train'
test_dir = '/content/IMDB text and seq/aclImdb/test'

# Function to load IMDB data
def load_data_from_dir(dir_path):
    texts = []
    labels = []
    for label_type in ['neg', 'pos']:  # IMDB dataset contains 'neg' and 'pos' folders
        folder_path = os.path.join(dir_path, label_type)
        for filename in os.listdir(folder_path):
            with open(os.path.join(folder_path, filename), encoding='utf-8') as f:
                texts.append(f.read())
            labels.append(0 if label_type == 'neg' else 1)  # 0 for negative, 1 for positive
    return texts, labels

# Load original training data (unprocessed)
train_texts, train_labels = load_data_from_dir(train_dir)

# Check the length of training data
print(f"Original training data size: {len(train_texts)} samples")

# Randomly select 10,000 samples for validation data
random.seed(42)
val_size = 10000
val_indices = random.sample(range(len(train_texts)), val_size)

# Create validation data
val_texts = [train_texts[i] for i in val_indices]
val_labels = [train_labels[i] for i in val_indices]

# Remove the selected validation samples from the training data
train_texts = [train_texts[i] for i in range(len(train_texts)) if i not in val_indices]
train_labels = [train_labels[i] for i in range(len(train_labels)) if i not in val_indices]

# Check the size of the new train and validation sets
print(f"New training data size: {len(train_texts)} samples")
print(f"Validation data size: {len(val_texts)} samples")

# Tokenize the text data
tokenizer = Tokenizer(num_words=10000)  # Only keep the top 10,000 words
tokenizer.fit_on_texts(train_texts)

# Convert the text data to sequences
train_sequences = tokenizer.texts_to_sequences(train_texts)
val_sequences = tokenizer.texts_to_sequences(val_texts)

# Pad the sequences to have equal length
max_len = 150  # You can adjust this based on your requirements
train_padded = pad_sequences(train_sequences, maxlen=max_len)
val_padded = pad_sequences(val_sequences, maxlen=max_len)

# Print the shapes of the padded datasets
print(f"Shape of training data: {train_padded.shape}")
print(f"Shape of validation data: {val_padded.shape}")
```

```
Original training data size: 25000 samples
New training data size: 15000 samples
Validation data size: 10000 samples
Shape of training data: (15000, 150)
Shape of validation data: (10000, 150)
```

```
import numpy as np
from tensorflow.keras.preprocessing.sequence import pad_sequences

# Verify the shape of the padded data
print(f"Train data shape: {train_padded.shape}")
print(f"Validation data shape: {val_padded.shape}")
print(f"Test data shape: {test_padded.shape}")

# Ensure train_labels, val_labels, test_labels are numpy arrays and have correct shape
train_labels = np.array(train_labels)
val_labels = np.array(val_labels)
test_labels = np.array(test_labels)

print(f"Train labels shape: {train_labels.shape}")
print(f"Validation labels shape: {val_labels.shape}")
print(f"Test labels shape: {test_labels.shape}")

# If data is not a numpy array, convert it
train_padded = np.array(train_padded)
val_padded = np.array(val_padded)
test_padded = np.array(test_padded)

# If necessary, check if any additional reshaping is required
train_padded = train_padded.reshape(train_padded.shape[0], train_padded.shape[1])
val_padded = val_padded.reshape(val_padded.shape[0], val_padded.shape[1])
test_padded = test_padded.reshape(test_padded.shape[0], test_padded.shape[1])

# Now, proceed to build the RNN or LSTM model
```

```
Train data shape: (15000, 150)
Validation data shape: (10000, 150)
Test data shape: (10000, 150)
Train labels shape: (15000,)
Validation labels shape: (10000,)
Test labels shape: (10000,)
```

Building RNN Model

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, SimpleRNN, Dense, Dropout
from tensorflow.keras.optimizers import Adam

# Define RNN model
rnn_model = Sequential([
    Embedding(input_dim=10000, output_dim=128, input_length=150),  # Embedding layer
    SimpleRNN(64, activation='tanh', return_sequences=False),  # RNN layer
    Dropout(0.5),  # Dropout layer to avoid overfitting
    Dense(1, activation='sigmoid')  # Output layer for binary classification
])

# Compile the model
rnn_model.compile(optimizer=Adam(), loss='binary_crossentropy', metrics=['accuracy'])

# Summary of the model architecture
rnn_model.summary()

# Train the model
rnn_history = rnn_model.fit(
    train_padded, train_labels,
    epochs=10,  # You can adjust this based on your dataset size and requirements
```

```
    batch_size=32,
    validation_data=(val_padded, val_labels),  # Validation set
    verbose=2  # Display the training progress
)
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/embedding.py:90: UserWarning: Argument `input_length` is deprecated. Just remove it.
  warnings.warn(
Model: "sequential"
```

Layer (type)	Output Shape	Param #
embedding (Embedding)	?	0 (unbuilt)
simple_rnn (SimpleRNN)	?	0 (unbuilt)
dropout (Dropout)	?	0
dense (Dense)	?	0 (unbuilt)

```
 Total params: 0 (0.00 B)
 Trainable params: 0 (0.00 B)
 Non-trainable params: 0 (0.00 B)
Epoch 1/10
469/469 - 30s - 64ms/step - accuracy: 0.6262 - loss: 0.6314 - val_accuracy: 0.7120 - val_loss: 0.5642
Epoch 2/10
469/469 - 47s - 100ms/step - accuracy: 0.8213 - loss: 0.4146 - val_accuracy: 0.7781 - val_loss: 0.4846
Epoch 3/10
469/469 - 35s - 76ms/step - accuracy: 0.9009 - loss: 0.2547 - val_accuracy: 0.7551 - val_loss: 0.5753
Epoch 4/10
469/469 - 42s - 89ms/step - accuracy: 0.9211 - loss: 0.2012 - val_accuracy: 0.7781 - val_loss: 0.6010
Epoch 5/10
469/469 - 42s - 89ms/step - accuracy: 0.9739 - loss: 0.0781 - val_accuracy: 0.7562 - val_loss: 0.7625
Epoch 6/10
469/469 - 40s - 86ms/step - accuracy: 0.9815 - loss: 0.0558 - val_accuracy: 0.6091 - val_loss: 1.2439
Epoch 7/10
469/469 - 39s - 83ms/step - accuracy: 0.9723 - loss: 0.0740 - val_accuracy: 0.7620 - val_loss: 0.8681
Epoch 8/10
469/469 - 43s - 91ms/step - accuracy: 0.9929 - loss: 0.0233 - val_accuracy: 0.7376 - val_loss: 1.0509
Epoch 9/10
469/469 - 30s - 64ms/step - accuracy: 0.9975 - loss: 0.0103 - val_accuracy: 0.7415 - val_loss: 1.1535
Epoch 10/10
469/469 - 38s - 82ms/step - accuracy: 0.9988 - loss: 0.0054 - val_accuracy: 0.7617 - val_loss: 1.1751
```

Using LSTM Layers to Improve the performance of the model

```
from tensorflow.keras.layers import LSTM

# Define LSTM model
lstm_model = Sequential([
    Embedding(input_dim=10000, output_dim=128, input_length=150),  # Embedding layer
    LSTM(64, activation='tanh', return_sequences=False),  # LSTM layer
    Dropout(0.5),  # Dropout layer to avoid overfitting
    Dense(1, activation='sigmoid')  # Output layer for binary classification
])

# Compile the model
lstm_model.compile(optimizer=Adam(), loss='binary_crossentropy', metrics=['accuracy'])

# Summary of the model architecture
lstm_model.summary()

# Train the model
lstm_history = lstm_model.fit(
    train_padded, train_labels,
    epochs=10,  # You can adjust this based on your dataset size and requirements
    batch_size=32,
    validation_data=(val_padded, val_labels),  # Validation set
    verbose=2  # Display the training progress
)
```

```
Model: "sequential_1"
```

Layer (type)	Output Shape	Param #
embedding_1 (Embedding)	?	0 (unbuilt)
lstm (LSTM)	?	0 (unbuilt)
dropout_1 (Dropout)	?	0
dense_1 (Dense)	?	0 (unbuilt)

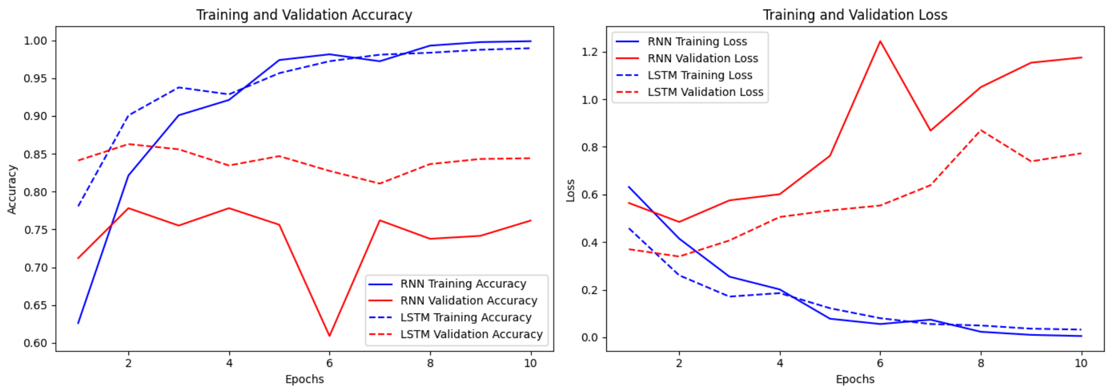
```
 Total params: 0 (0.00 B)
 Trainable params: 0 (0.00 B)
 Non-trainable params: 0 (0.00 B)
Epoch 1/10
469/469 - 62s - 133ms/step - accuracy: 0.7806 - loss: 0.4579 - val_accuracy: 0.8411 - val_loss: 0.3699
Epoch 2/10
469/469 - 78s - 166ms/step - accuracy: 0.9007 - loss: 0.2605 - val_accuracy: 0.8628 - val_loss: 0.3394
Epoch 3/10
469/469 - 84s - 180ms/step - accuracy: 0.9377 - loss: 0.1710 - val_accuracy: 0.8559 - val_loss: 0.4069
Epoch 4/10
469/469 - 56s - 120ms/step - accuracy: 0.9285 - loss: 0.1857 - val_accuracy: 0.8345 - val_loss: 0.5055
Epoch 5/10
469/469 - 83s - 176ms/step - accuracy: 0.9565 - loss: 0.1222 - val_accuracy: 0.8469 - val_loss: 0.5331
Epoch 6/10
469/469 - 81s - 172ms/step - accuracy: 0.9723 - loss: 0.0802 - val_accuracy: 0.8273 - val_loss: 0.5536
Epoch 7/10
469/469 - 79s - 168ms/step - accuracy: 0.9809 - loss: 0.0558 - val_accuracy: 0.8106 - val_loss: 0.6389
Epoch 8/10
469/469 - 84s - 180ms/step - accuracy: 0.9835 - loss: 0.0497 - val_accuracy: 0.8364 - val_loss: 0.8701
Epoch 9/10
469/469 - 82s - 175ms/step - accuracy: 0.9875 - loss: 0.0363 - val_accuracy: 0.8431 - val_loss: 0.7390
Epoch 10/10
469/469 - 81s - 172ms/step - accuracy: 0.9895 - loss: 0.0326 - val_accuracy: 0.8441 - val_loss: 0.7729
```

```
# Evaluate RNN model on test data
rnn_test_loss, rnn_test_accuracy = rnn_model.evaluate(test_padded, test_labels, verbose=2)
print(f"RNN Test Accuracy: {rnn_test_accuracy*100:.2f}%")

# Evaluate LSTM model on test data
lstm_test_loss, lstm_test_accuracy = lstm_model.evaluate(test_padded, test_labels, verbose=2)
print(f"LSTM Test Accuracy: {lstm_test_accuracy*100:.2f}%")
```

```
313/313 - 5s - 14ms/step - accuracy: 0.5279 - loss: 2.6242
RNN Test Accuracy: 52.79%
313/313 - 7s - 22ms/step - accuracy: 0.5211 - loss: 2.4262
LSTM Test Accuracy: 52.11%
```

```
import matplotlib.pyplot as plt

# Function to plot training and validation accuracy and loss for comparison
def plot_history_comparison(rnn_history, lstm_history):
    # Extract accuracy and loss data from the histories
    rnn_acc = rnn_history.history['accuracy']
    rnn_val_acc = rnn_history.history['val_accuracy']
    rnn_loss = rnn_history.history['loss']
    rnn_val_loss = rnn_history.history['val_loss']

    lstm_acc = lstm_history.history['accuracy']
    lstm_val_acc = lstm_history.history['val_accuracy']
    lstm_loss = lstm_history.history['loss']
    lstm_val_loss = lstm_history.history['val_loss']

    epochs = range(1, len(rnn_acc) + 1)

    # Plot accuracy
```

```
    plt.figure(figsize=(14, 5))

    plt.subplot(1, 2, 1)
    plt.plot(epochs, rnn_acc, 'b-', label='RNN Training Accuracy')
    plt.plot(epochs, rnn_val_acc, 'r-', label='RNN Validation Accuracy')
    plt.plot(epochs, lstm_acc, 'b--', label='LSTM Training Accuracy')
    plt.plot(epochs, lstm_val_acc, 'r--', label='LSTM Validation Accuracy')
    plt.title('Training and Validation Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend()

    # Plot loss
    plt.subplot(1, 2, 2)
    plt.plot(epochs, rnn_loss, 'b-', label='RNN Training Loss')
    plt.plot(epochs, rnn_val_loss, 'r-', label='RNN Validation Loss')
    plt.plot(epochs, lstm_loss, 'b--', label='LSTM Training Loss')
    plt.plot(epochs, lstm_val_loss, 'r--', label='LSTM Validation Loss')
    plt.title('Training and Validation Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()

    plt.tight_layout()
    plt.show()

# Call the function with the histories of both models
plot_history_comparison(rnn_history, lstm_history)
```



Task

Pretrained Embedded layers and Word Embedded Layers

```
# Function to load GloVe embeddings
def load_glove_embeddings(filepath, word_index, embedding_dim=100):
    embeddings_index = {}
    with open(filepath, encoding='utf-8') as f:
        for line in f:
            values = line.split()
            word = values[0]
            coefs = np.asarray(values[1:], dtype='float32')
            embeddings_index[word] = coefs

    # Create an embedding matrix for words in our vocabulary
    embedding_matrix = np.zeros((len(word_index) + 1, embedding_dim))
    for word, i in word_index.items():
        embedding_vector = embeddings_index.get(word)
        if embedding_vector is not None:
            embedding_matrix[i] = embedding_vector  # Words not found in embedding index will be all zeros

    return embedding_matrix

# Load GloVe embeddings and create an embedding matrix
embedding_dim = 100
glove_file_path = 'glove.6B.100d.txt'  # Replace with your GloVe file path
word_index = tokenizer.word_index  # a tokenizer defined
```

```
# Model 1: Simple Embedding Layer
max_sequence_length = 150
simple_model = Sequential([
    Embedding(input_dim=10000, output_dim=128, input_length=max_sequence_length),
    LSTM(64),
    Dense(1, activation='sigmoid')
])
simple_model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
simple_history = simple_model.fit(train_padded, train_labels, epochs=5, validation_data=(val_padded, val_labels), batch_size=32)
```

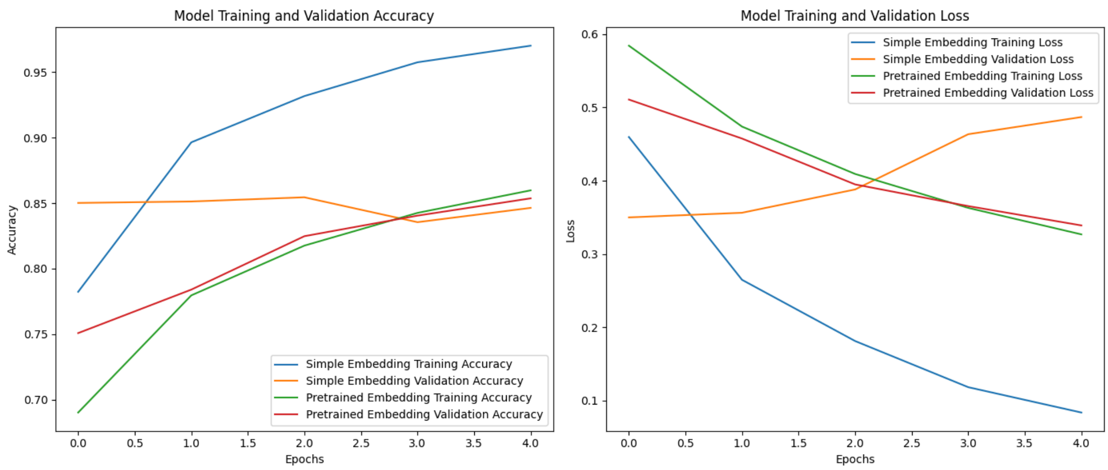
```
Epoch 1/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 65s 132ms/step - accuracy: 0.7096 - loss: 0.5509 - val_accuracy: 0.8502 - val_loss: 0.3500
Epoch 2/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 83s 135ms/step - accuracy: 0.8974 - loss: 0.2674 - val_accuracy: 0.8513 - val_loss: 0.3564
Epoch 3/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 64s 136ms/step - accuracy: 0.9347 - loss: 0.1744 - val_accuracy: 0.8545 - val_loss: 0.3882
Epoch 4/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 83s 138ms/step - accuracy: 0.9657 - loss: 0.1001 - val_accuracy: 0.8355 - val_loss: 0.4635
Epoch 5/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 80s 134ms/step - accuracy: 0.9748 - loss: 0.0735 - val_accuracy: 0.8464 - val_loss: 0.4869
```

```
import numpy as np

# Define the function to load GloVe embeddings
def load_glove_embeddings(filepath, word_index, embedding_dim=100):
    # Load GloVe embeddings from the file
    embeddings_index = {}
    with open(filepath, 'r', encoding='utf-8') as f:
        for line in f:
            values = line.split()
            word = values[0]
            coefficients = np.asarray(values[1:], dtype='float32')
            embeddings_index[word] = coefficients

    # Prepare the embedding matrix
    embedding_matrix = np.zeros((len(word_index) + 1, embedding_dim))
    for word, i in word_index.items():
        embedding_vector = embeddings_index.get(word)
        if embedding_vector is not None:
            embedding_matrix[i] = embedding_vector

    return embedding_matrix

# Example usage:
```

```
filepath = '/content/glove.6B.100d.txt'  # Replace with the correct path to your GloVe file
embedding_matrix = load_glove_embeddings(filepath, word_index)


# Define the pretrained model using the embedding matrix
pretrained_model = Sequential([
    Embedding(input_dim=len(word_index) + 1,
              output_dim=embedding_dim,
              weights=[embedding_matrix],
              input_length=max_sequence_length,
              trainable=False),
    LSTM(64),
    Dense(1, activation='sigmoid')
])

pretrained_model.compile(optimizer='adam',
                         loss='binary_crossentropy',
                         metrics=['accuracy'])
pretrained_history = pretrained_model.fit(train_padded,
                                          train_labels,
                                          epochs=5,
                                          validation_data=(val_padded, val_labels),
                                          batch_size=32)
```

```
Epoch 1/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 58s 118ms/step - accuracy: 0.6238 - loss: 0.6329 - val_accuracy: 0.7508 - val_loss: 0.5108
Epoch 2/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 46s 98ms/step - accuracy: 0.7662 - loss: 0.4970 - val_accuracy: 0.7840 - val_loss: 0.4576
Epoch 3/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 79s 92ms/step - accuracy: 0.8103 - loss: 0.4180 - val_accuracy: 0.8248 - val_loss: 0.3950
Epoch 4/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 82s 93ms/step - accuracy: 0.8439 - loss: 0.3632 - val_accuracy: 0.8405 - val_loss: 0.3655
Epoch 5/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 82s 92ms/step - accuracy: 0.8567 - loss: 0.3336 - val_accuracy: 0.8537 - val_loss: 0.3391
```

```
import matplotlib.pyplot as plt

# Plot function to compare histories
def plot_model_comparison(history1, history2, model1_name='Model 1', model2_name='Model 2'):
    # Plot accuracy
    plt.figure(figsize=(14, 6))

    # Accuracy plot
    plt.subplot(1, 2, 1)
    plt.plot(history1.history['accuracy'], label=f'{model1_name} Training Accuracy')
    plt.plot(history1.history['val_accuracy'], label=f'{model1_name} Validation Accuracy')
    plt.plot(history2.history['accuracy'], label=f'{model2_name} Training Accuracy')
    plt.plot(history2.history['val_accuracy'], label=f'{model2_name} Validation Accuracy')
    plt.title('Model Training and Validation Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend()

    # Loss plot
    plt.subplot(1, 2, 2)
    plt.plot(history1.history['loss'], label=f'{model1_name} Training Loss')
    plt.plot(history1.history['val_loss'], label=f'{model1_name} Validation Loss')
    plt.plot(history2.history['loss'], label=f'{model2_name} Training Loss')
    plt.plot(history2.history['val_loss'], label=f'{model2_name} Validation Loss')
    plt.title('Model Training and Validation Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()

    plt.tight_layout()
    plt.show()

# Assume simple_history and pretrained_history are the histories obtained from model training
plot_model_comparison(simple_history,
                      pretrained_history, 'Simple Embedding', 'Pretrained Embedding')
```




```
# Evaluate the simple embedding layer model
simple_test_loss, simple_test_accuracy = simple_model.evaluate(test_padded, test_labels, verbose=2)
print(f"Simple Embedding Model Test Accuracy: {simple_test_accuracy:.2%}")

# Evaluate the GloVe embedding layer model
pretrained_test_loss, pretrained_test_accuracy = pretrained_model.evaluate(test_padded, test_labels, verbose=2)
print(f"GloVe Embedding Model Test Accuracy: {pretrained_test_accuracy:.2%}")
```

```
313/313 - 8s - 27ms/step - accuracy: 0.5263 - loss: 1.4460
Simple Embedding Model Test Accuracy: 52.63%
313/313 - 8s - 25ms/step - accuracy: 0.5625 - loss: 0.9715
GloVe Embedding Model Test Accuracy: 56.25%
```

Task

Trying Different Number of Samples like 1000, 5000, 10000

```
import matplotlib.pyplot as plt
# Define a function to train and evaluate models with different training sample sizes
def train_and_evaluate_models(sample_sizes, train_padded, train_labels, val_padded, val_labels, embedding_matrix, word_index, max_sequence_length):
    embedding_model_accuracies = []
    pretrained_model_accuracies = []

    for sample_size in sample_sizes:
        print(f"\nTraining with {sample_size} samples...")

        # Subset training data
        train_subset = train_padded[:sample_size]
        train_labels_subset = train_labels[:sample_size]
```

```
        # Model 1: Simple Embedding Layer Model
        simple_model = Sequential([
            Embedding(input_dim=len(word_index) + 1, output_dim=100, input_length=max_sequence_length),
            LSTM(64),
            Dense(1, activation='sigmoid')
        ])
        simple_model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
        simple_model.fit(train_subset, train_labels_subset, epochs=5, validation_data=(val_padded, val_labels), batch_size=32, verbose=0)
        _, simple_val_acc = simple_model.evaluate(val_padded, val_labels, verbose=0)
        embedding_model_accuracies.append(simple_val_acc)

        # Model 2: Pretrained GloVe Embedding Model
        pretrained_model = Sequential([
            Embedding(input_dim=len(word_index) + 1, output_dim=100, weights=[embedding_matrix], input_length=max_sequence_length, trainable=False),
            LSTM(64),
            Dense(1, activation='sigmoid')
        ])
        pretrained_model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
        pretrained_model.fit(train_subset, train_labels_subset, epochs=5, validation_data=(val_padded, val_labels), batch_size=32, verbose=0)
        _, pretrained_val_acc = pretrained_model.evaluate(val_padded, val_labels, verbose=0)
        pretrained_model_accuracies.append(pretrained_val_acc)

    # Plot the results
    plt.figure(figsize=(12, 6))
    plt.plot(sample_sizes, embedding_model_accuracies, label='Embedding Layer Model', marker='o')
    plt.plot(sample_sizes, pretrained_model_accuracies, label='Pretrained GloVe Model', marker='o')
    plt.title('Comparison of Model Performance with Varying Training Sample Sizes')
    plt.xlabel('Number of Training Samples')
    plt.ylabel('Validation Accuracy')
    plt.legend()
    plt.grid(True)
    plt.show()

# Define sample sizes to test (adjust based on your data)
sample_sizes = [1000, 5000, 10000, len(train_padded)]

# Run the comparison
train_and_evaluate_models(sample_sizes, train_padded, train_labels, val_padded, val_labels, embedding_matrix, word_index, max_sequence_length)
```

```
Training with 1000 samples...

Training with 5000 samples...

Training with 10000 samples...

Training with 15000 samples...
```