Advanced Machine Learning (BA-64061-001)

Assignment 4 - Text Data

Sgudise@kent.edu

GitHub: GitHub Link

A base model was created to compare the results with adjusted models to see the improvement or deterioration of the model.

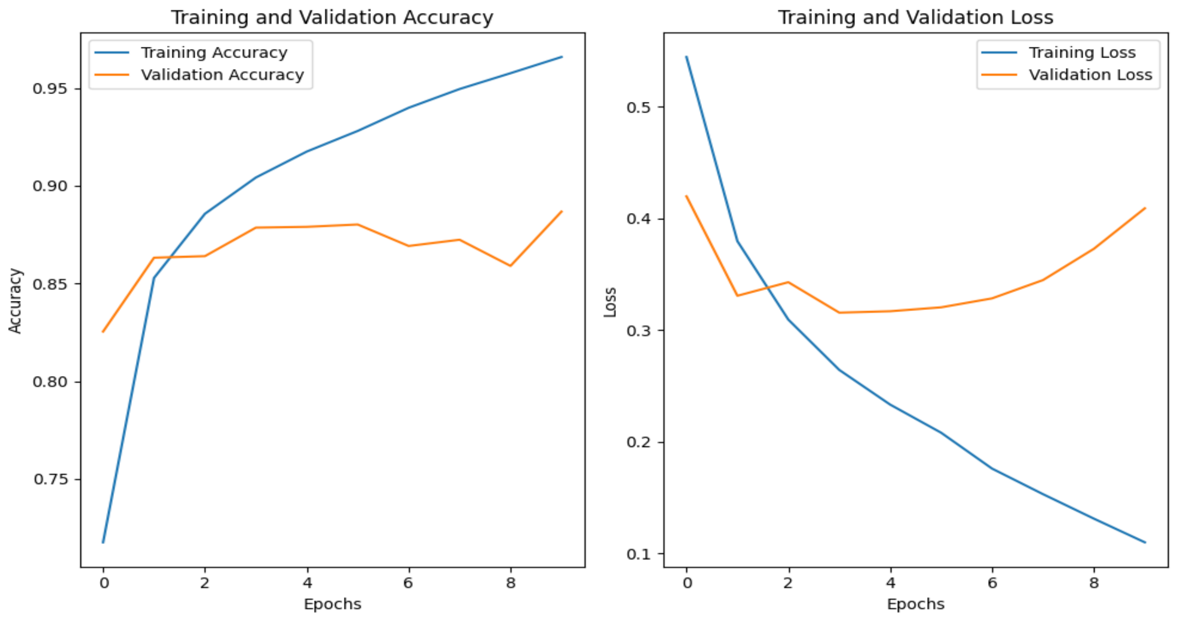
Base model properties

- Activation Function: Sigmoid (Output Layer)
- Training Size: Based on aclImdb/train, after an 80-20 split for validation (approx. 80% of original training data)
- Test Size: Based on aclImdb/test, loaded as a separate dataset
- Validation Size: 20% of training data, moved to aclImdb/val
- Embedding Layer: 128-dimensional embeddings
- RNN Layer: Bidirectional LSTM with 32 units
- Dropout: 0.5
- Output Layer: Dense layer with 1 unit and Sigmoid activation
- Max Tokens: 20,000
- Sequence Length: 600
- Batch Size: 32
- Optimizer: RMSprop
- Loss Function: Binary Crossentropy
- Callbacks: ModelCheckpoint to save the best model during training
- Epochs: 10

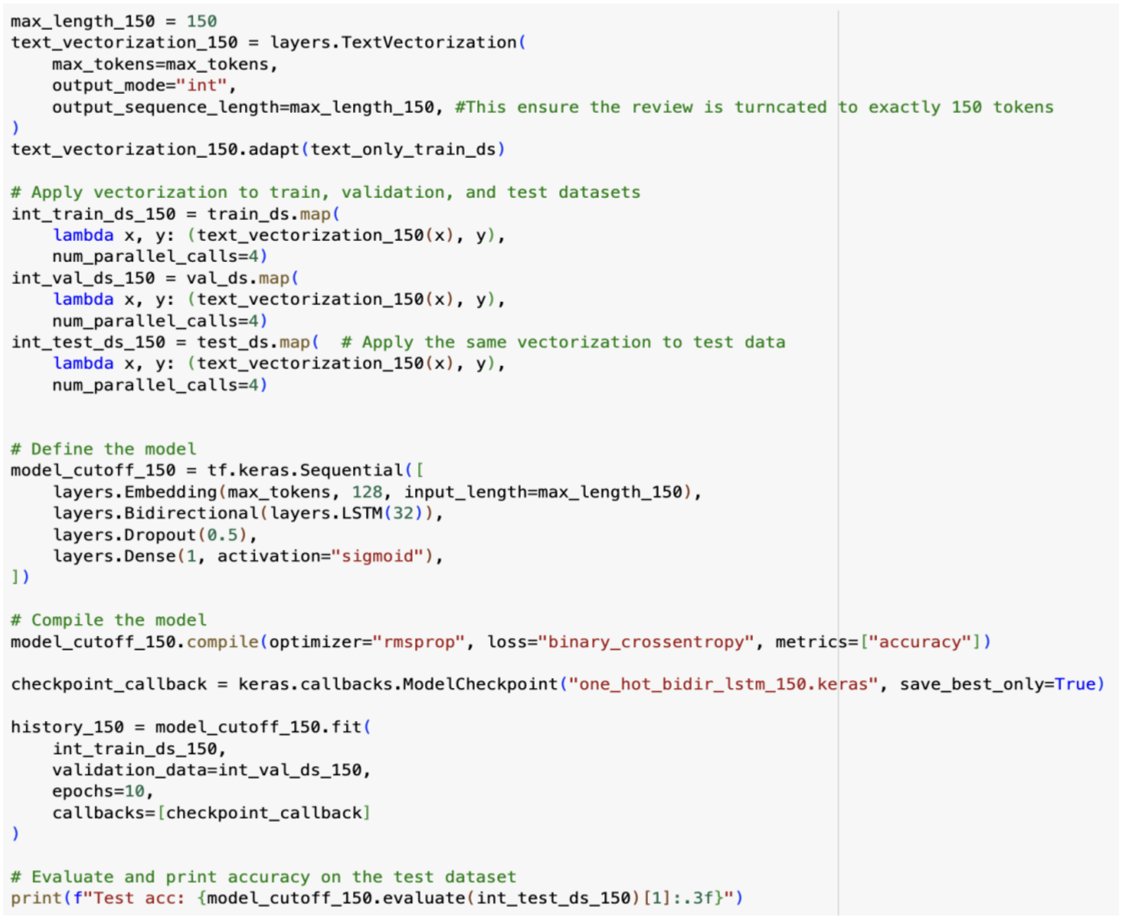
Layer Structure

- Input Layer: Input shape (None,) (integer tokens)
- Embedding Layer: Converts tokens to 128-dimensional vectors
- Bidirectional LSTM Layer: 32 LSTM units, applied bidirectionally
- Dropout Layer: Dropout with rate 0.5
- Dense Layer (Output): Single unit, Sigmoid activation

Base model Accuracy = 0.872



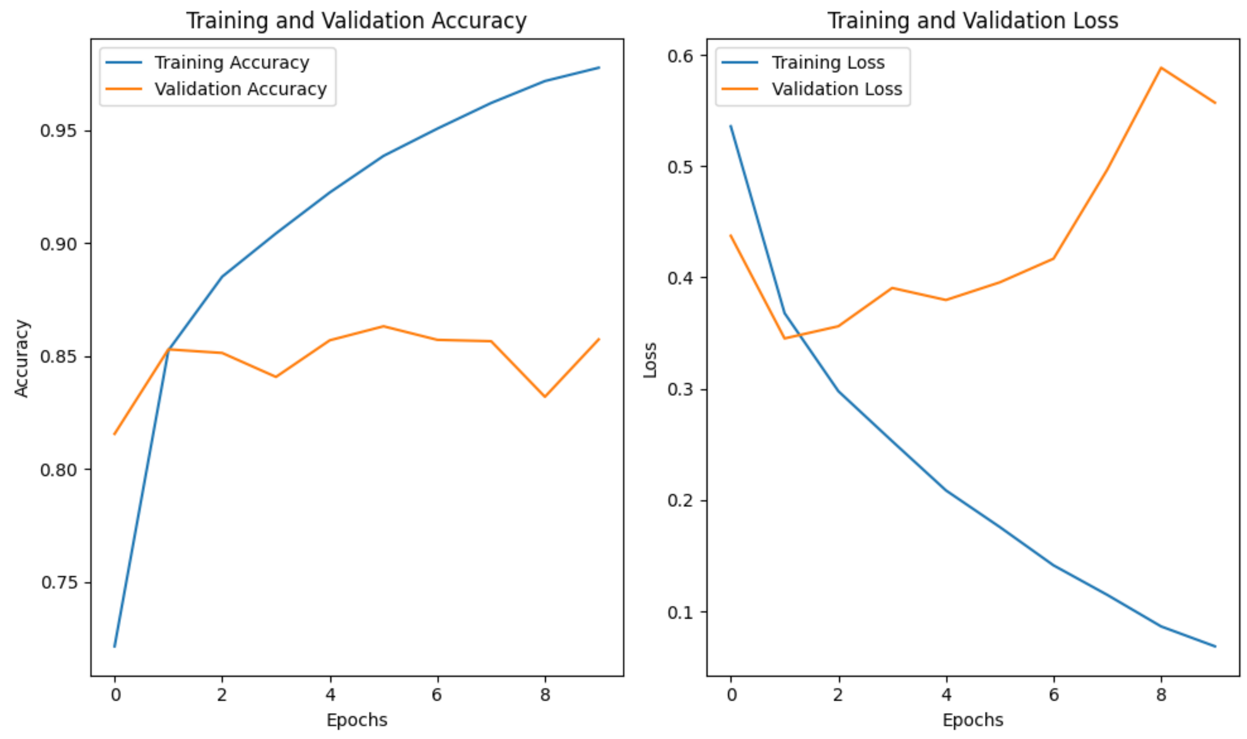
Cutoff reviews after 150 word



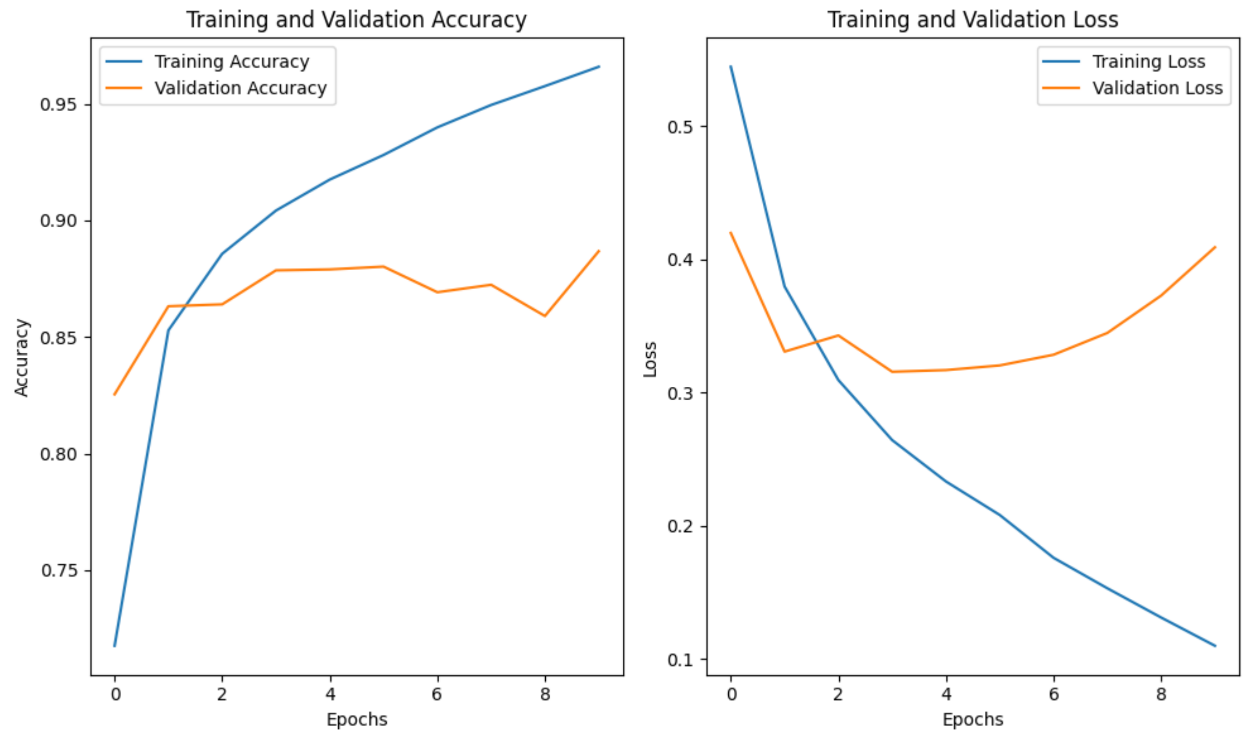
```
max_length_150 = 150
text_vectorization_150 = layers.TextVectorization(
    max_tokens=max_tokens,
    output_mode="int",
    output_sequence_length=max_length_150, #This ensure the review is turncated to exactly 150 tokens
)
text_vectorization_150.adapt(text_only_train_ds)

# Apply vectorization to train, validation, and test datasets
int_train_ds_150 = train_ds.map(
    lambda x, y: (text_vectorization_150(x), y),
    num_parallel_calls=4)
int_val_ds_150 = val_ds.map(
    lambda x, y: (text_vectorization_150(x), y),
    num_parallel_calls=4)
int_test_ds_150 = test_ds.map(  # Apply the same vectorization to test data
    lambda x, y: (text_vectorization_150(x), y),
    num_parallel_calls=4)


# Define the model
model_cutoff_150 = tf.keras.Sequential([
    layers.Embedding(max_tokens, 128, input_length=max_length_150),
    layers.Bidirectional(layers.LSTM(32)),
    layers.Dropout(0.5),
    layers.Dense(1, activation="sigmoid"),
])

# Compile the model
model_cutoff_150.compile(optimizer="rmsprop", loss="binary_crossentropy", metrics=["accuracy"])

checkpoint_callback = keras.callbacks.ModelCheckpoint("one_hot_bidir_lstm_150.keras", save_best_only=True)

history_150 = model_cutoff_150.fit(
    int_train_ds_150,
    validation_data=int_val_ds_150,
    epochs=10,
    callbacks=[checkpoint_callback]
)

# Evaluate and print accuracy on the test dataset
print(f"Test acc: {model_cutoff_150.evaluate(int_test_ds_150)[1]:.3f}")
```

Max length 150 model and Base model

Max length 150 Accuracy = 0.824

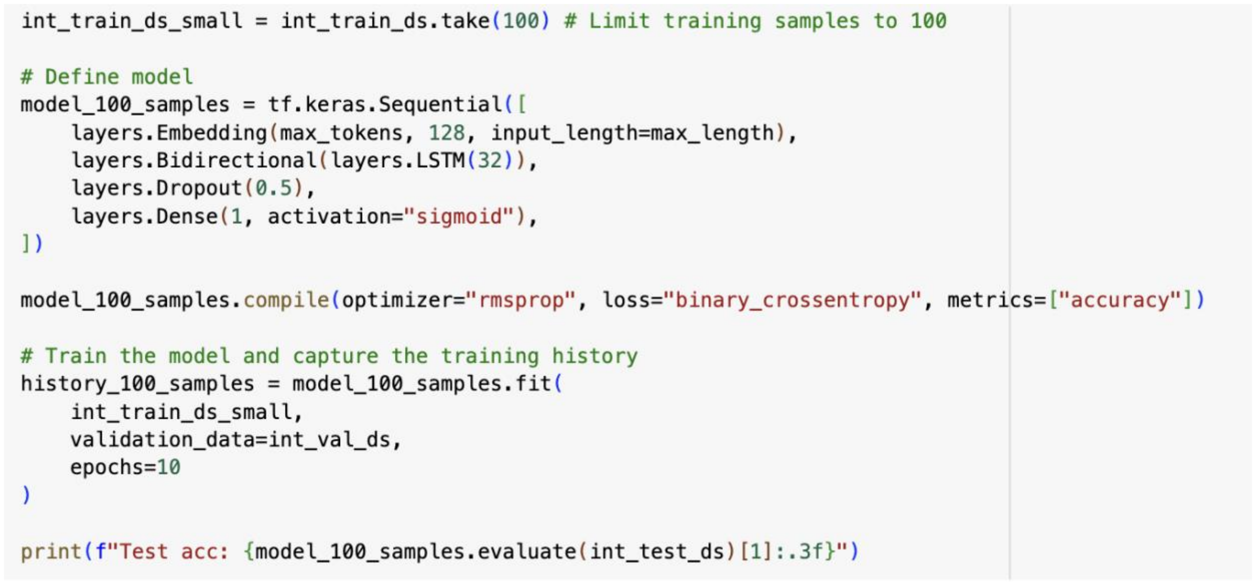


Base Model



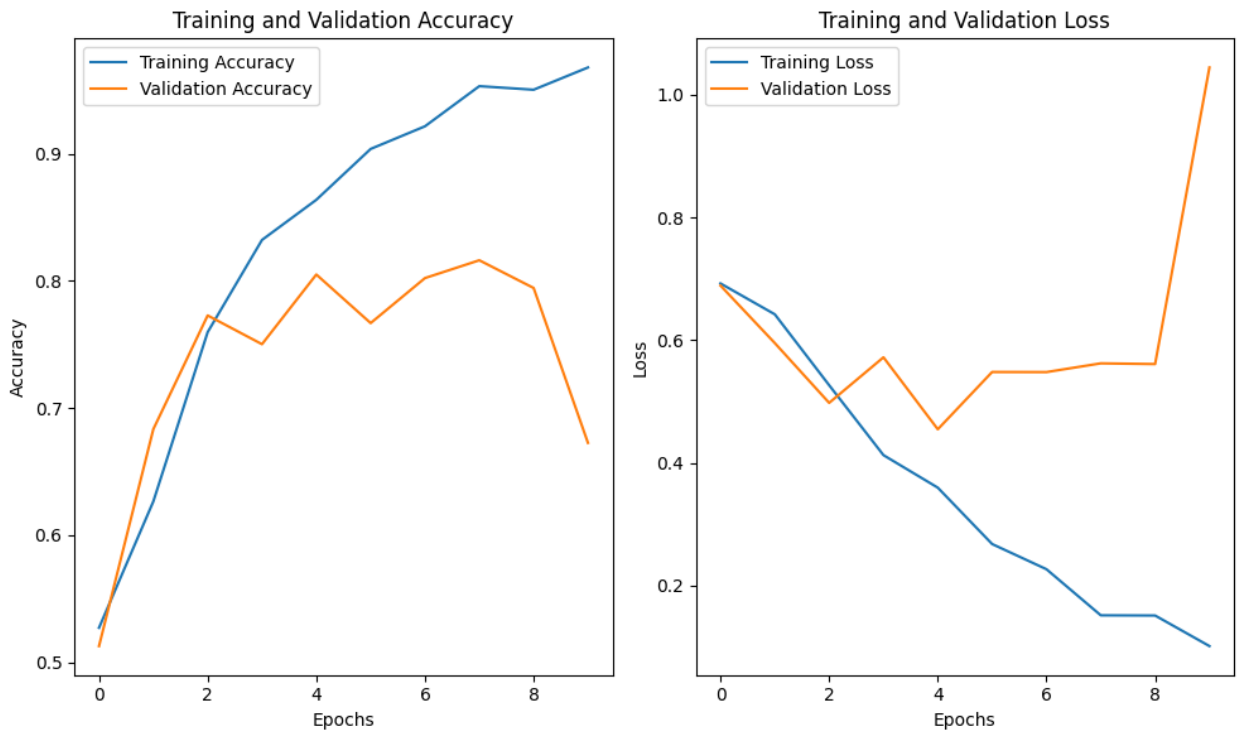
The base model outperformed the model with a maximum sequence length of 150, achieving a higher test accuracy of 0.872 compared to 0.824.

1. **Restrict training samples to 100**

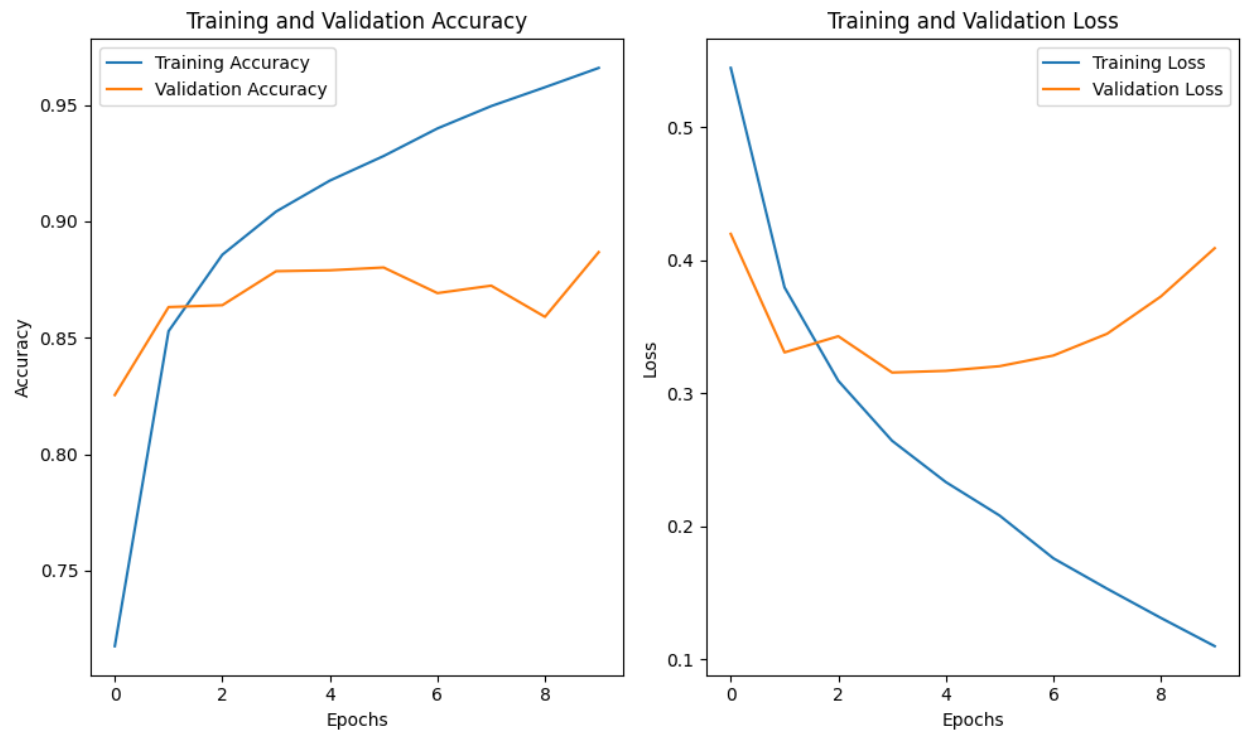


```python
int_train_ds_small = int_train_ds.take(100) # Limit training samples to 100

# Define model
model_100_samples = tf.keras.Sequential([
    layers.Embedding(max_tokens, 128, input_length=max_length),
    layers.Bidirectional(layers.LSTM(32)),
    layers.Dropout(0.5),
    layers.Dense(1, activation="sigmoid"),
])

model_100_samples.compile(optimizer="rmsprop", loss="binary_crossentropy", metrics=["accuracy"])

# Train the model and capture the training history
history_100_samples = model_100_samples.fit(
    int_train_ds_small,
    validation_data=int_val_ds,
    epochs=10
)

print(f"Test acc: {model_100_samples.evaluate(int_test_ds)[1]:.3f}")
```

Training samples to 100 Accuracy = 0.674



Base model

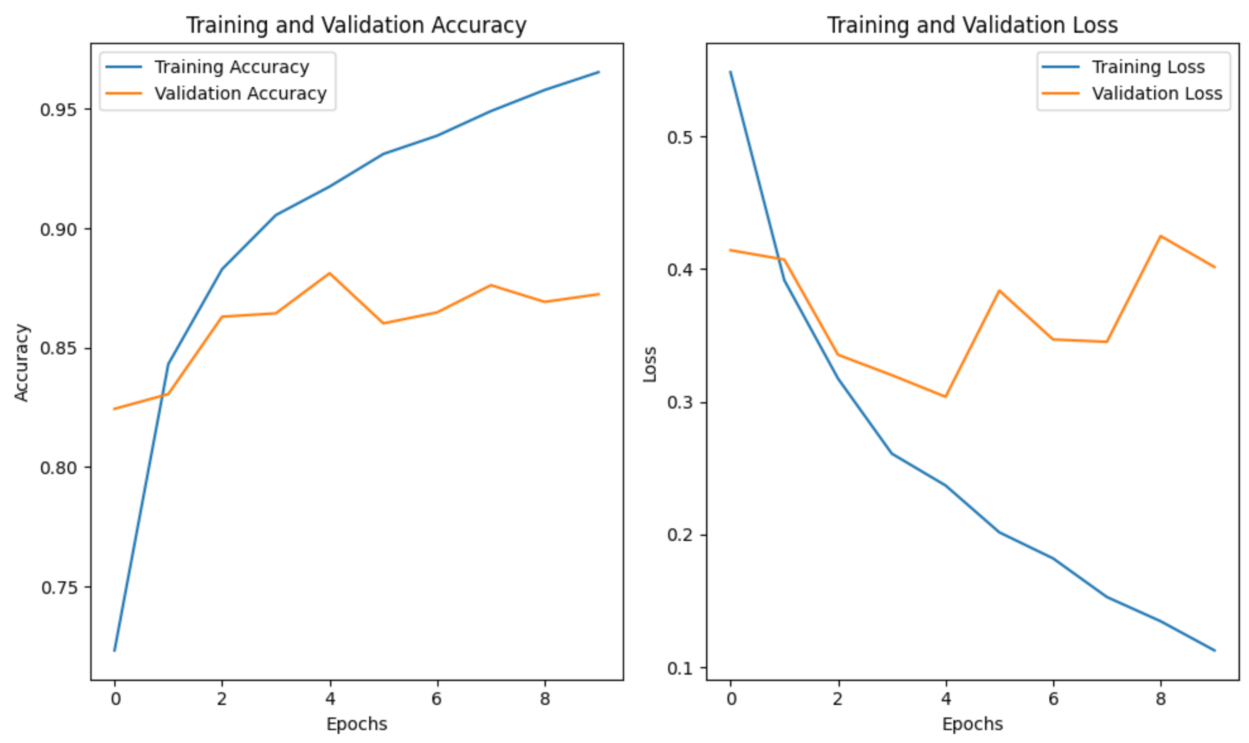


The base model achieved a test accuracy of 0.872, outperforming the model trained on only 100 samples, which had a test accuracy of 0.674.

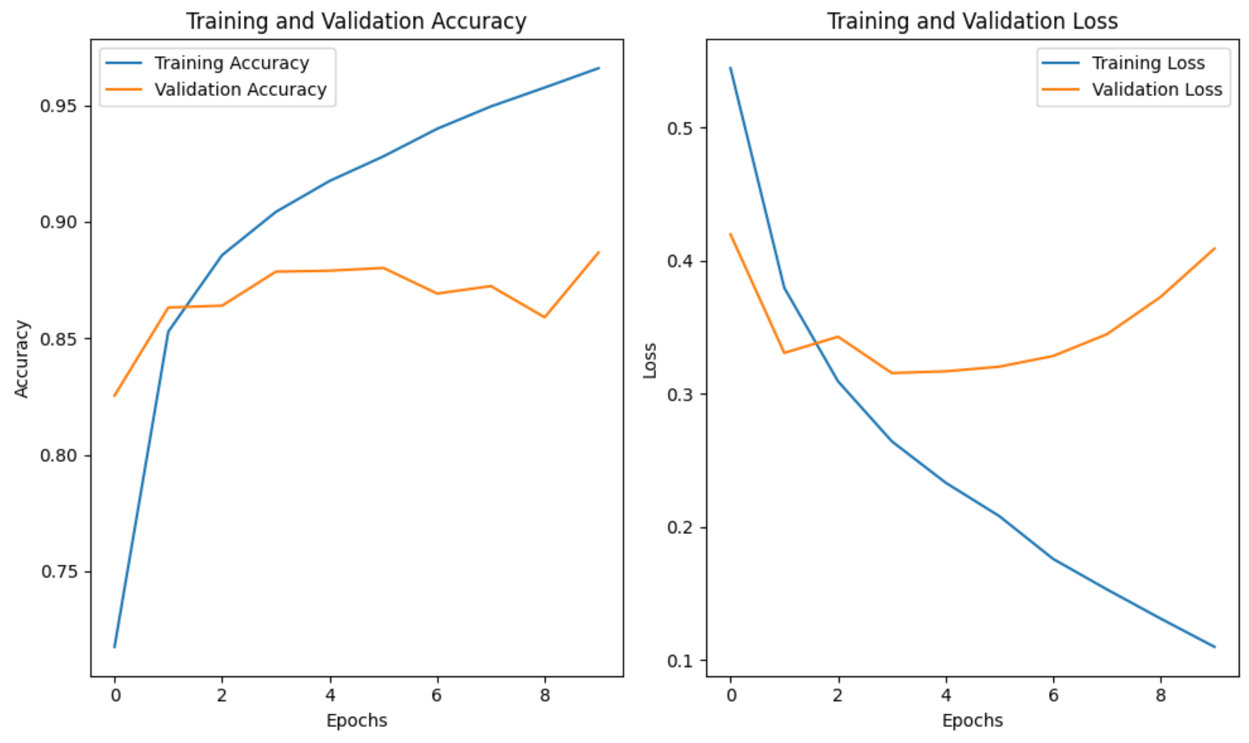
3. Validate on 10,000 samples

```
# Limit validation dataset to 10,000 samples
int_val_ds_10k = int_val_ds.take(10000)

# Define model
model_validate_10k = tf.keras.Sequential([
    layers.Embedding(max_tokens, 128, input_length=max_length),
    layers.Bidirectional(layers.LSTM(32)),
    layers.Dropout(0.5),
    layers.Dense(1, activation="sigmoid"),
])

# Compile the model
model_validate_10k.compile(optimizer="rmsprop", loss="binary_crossentropy", metrics=["accuracy"])

history_validate_10k = model_validate_10k.fit(
    int_train_ds,
    validation_data=int_val_ds_10k,
    epochs=10
)

# Evaluate and print accuracy on the test dataset
print(f"Test acc: {model_validate_10k.evaluate(int_test_ds)[1]:.3f}")
```

10,000 Validation samples = 0.846

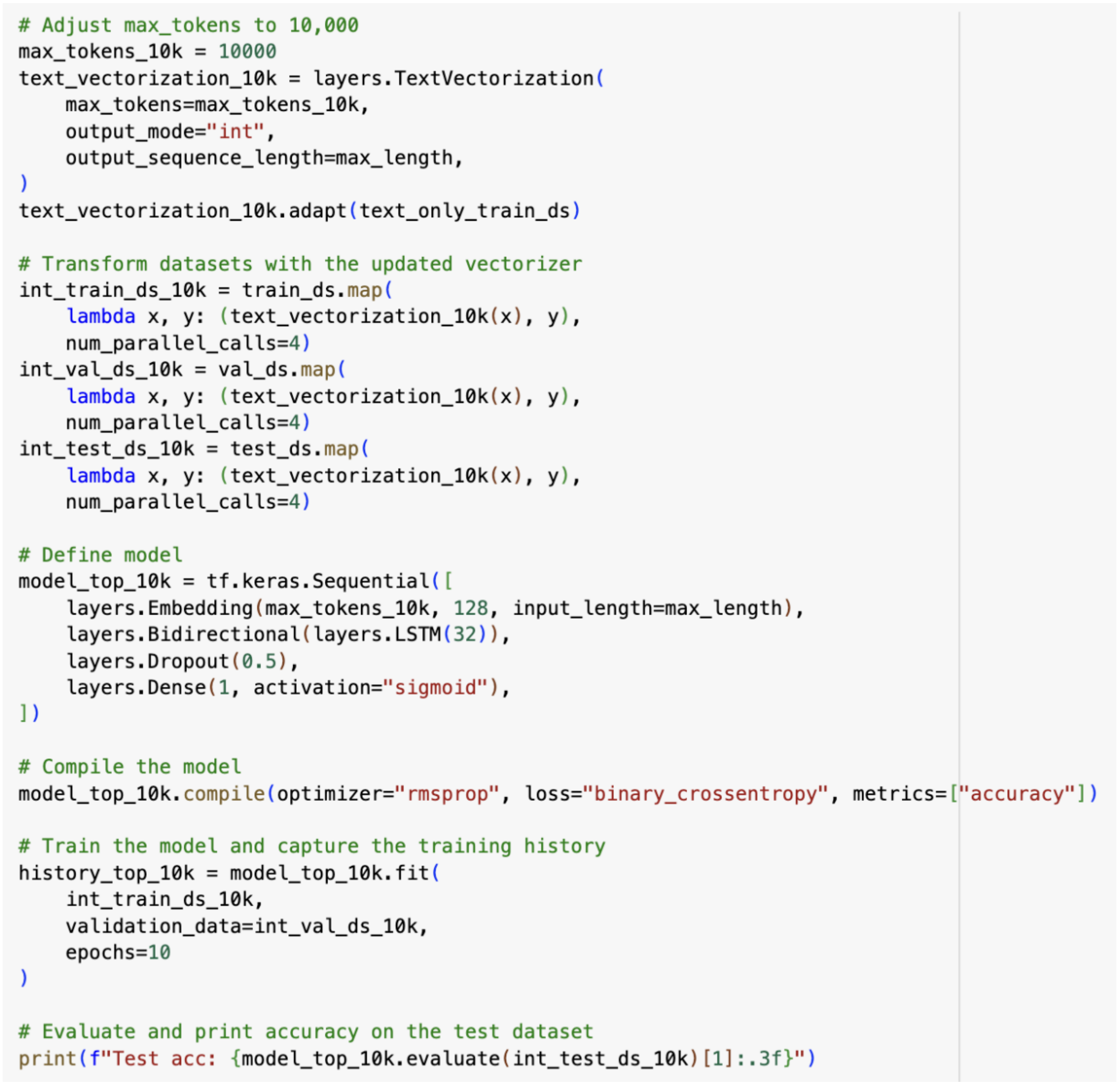


Base model



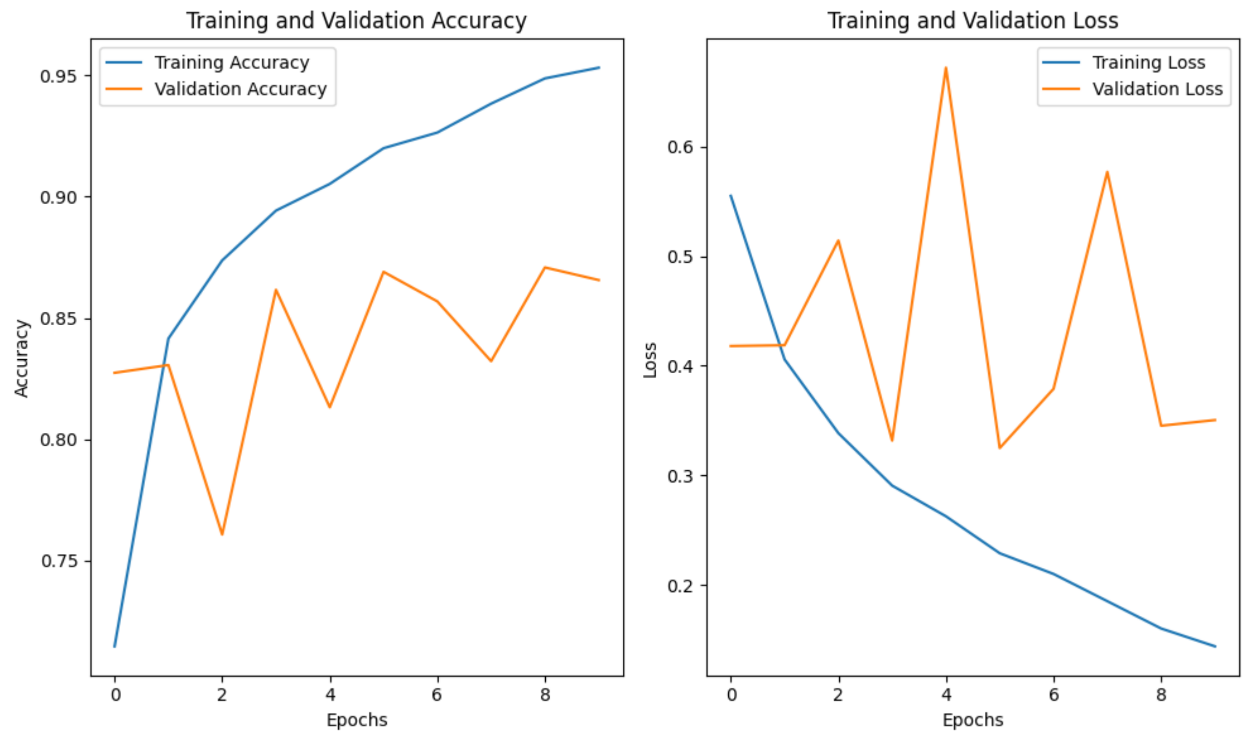
The base model performed better, achieving a test accuracy of 0.872 compared to 0.846 for the model with 10,000 validation samples.

4. Consider only the top 10,000 words

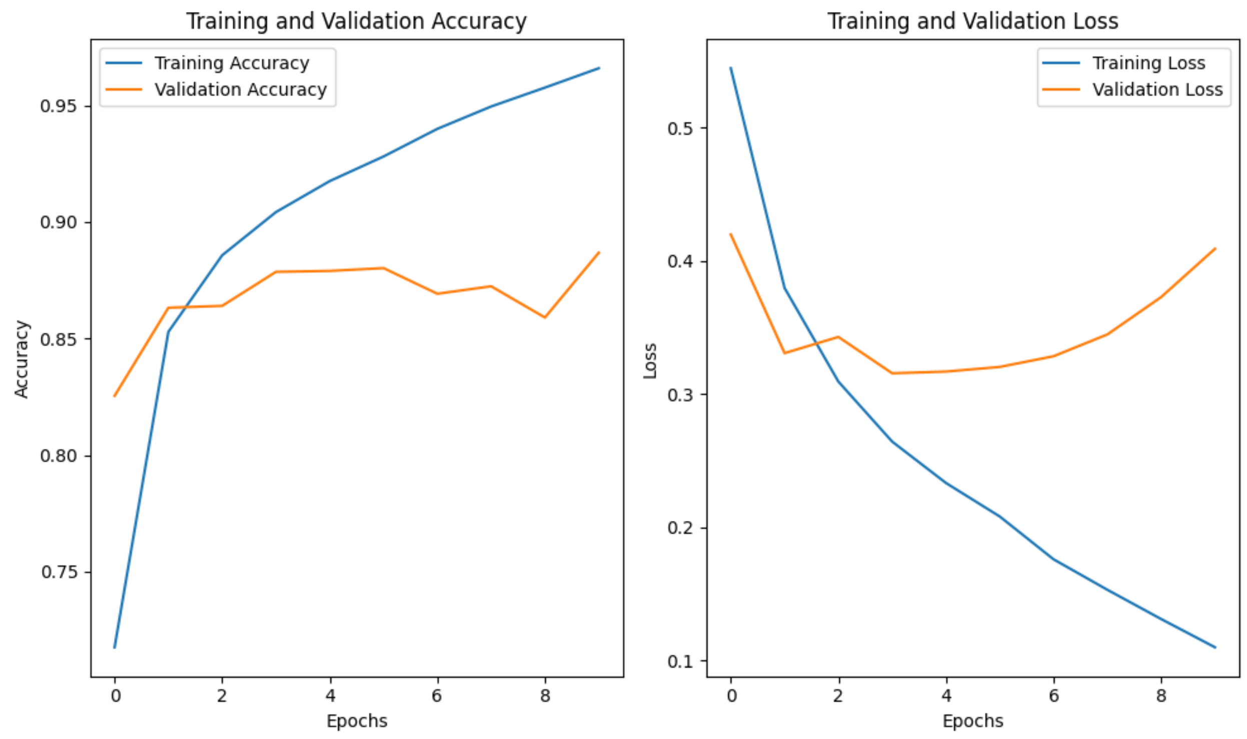


```python
# Adjust max_tokens to 10,000
max_tokens_10k = 10000
text_vectorization_10k = layers.TextVectorization(
    max_tokens=max_tokens_10k,
    output_mode="int",
    output_sequence_length=max_length,
)
text_vectorization_10k.adapt(text_only_train_ds)

# Transform datasets with the updated vectorizer
int_train_ds_10k = train_ds.map(
    lambda x, y: (text_vectorization_10k(x), y),
    num_parallel_calls=4)
int_val_ds_10k = val_ds.map(
    lambda x, y: (text_vectorization_10k(x), y),
    num_parallel_calls=4)
int_test_ds_10k = test_ds.map(
    lambda x, y: (text_vectorization_10k(x), y),
    num_parallel_calls=4)

# Define model
model_top_10k = tf.keras.Sequential([
    layers.Embedding(max_tokens_10k, 128, input_length=max_length),
    layers.Bidirectional(layers.LSTM(32)),
    layers.Dropout(0.5),
    layers.Dense(1, activation="sigmoid"),
])

# Compile the model
model_top_10k.compile(optimizer="rmsprop", loss="binary_crossentropy", metrics=["accuracy"])

# Train the model and capture the training history
history_top_10k = model_top_10k.fit(
    int_train_ds_10k,
    validation_data=int_val_ds_10k,
    epochs=10
)

# Evaluate and print accuracy on the test dataset
print(f"Test acc: {model_top_10k.evaluate(int_test_ds_10k)[1]:.3f}")
```

Model with top 10000 = 0.856



Base Model

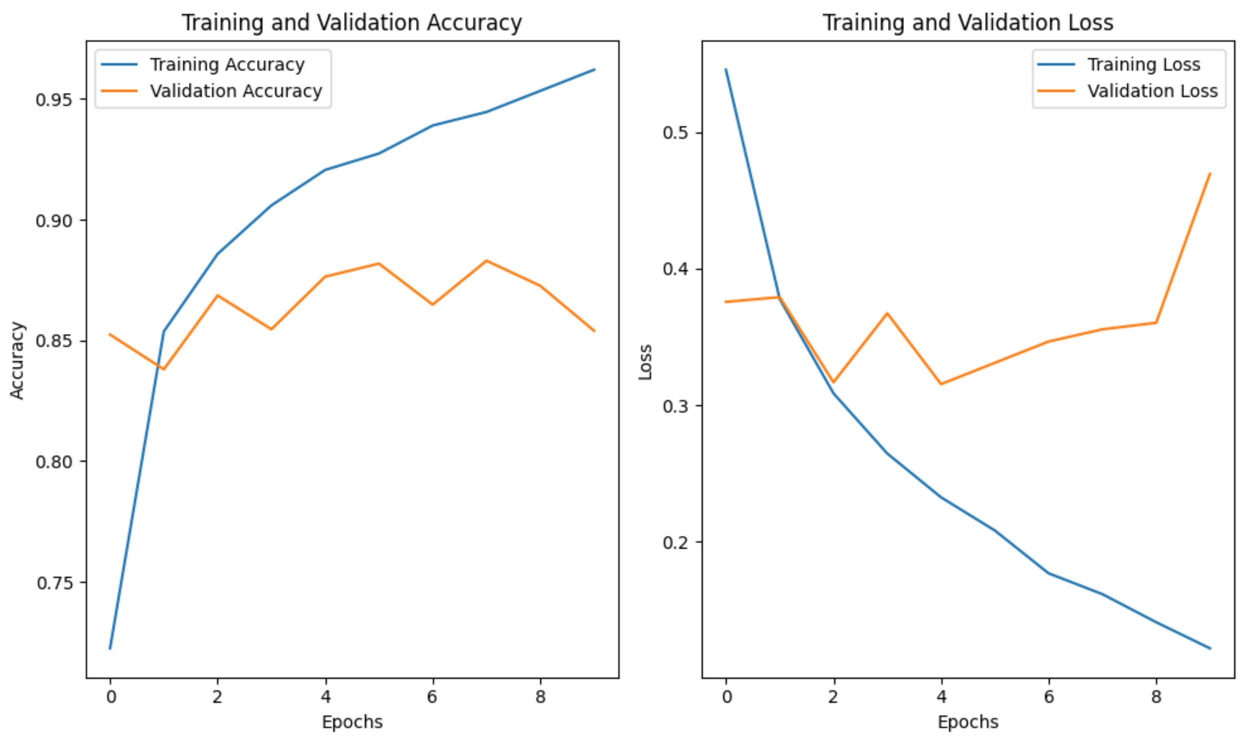


The base model performed better, achieving a test accuracy of 0.872 compared to 0.856 for the model with the top 10,000 words.

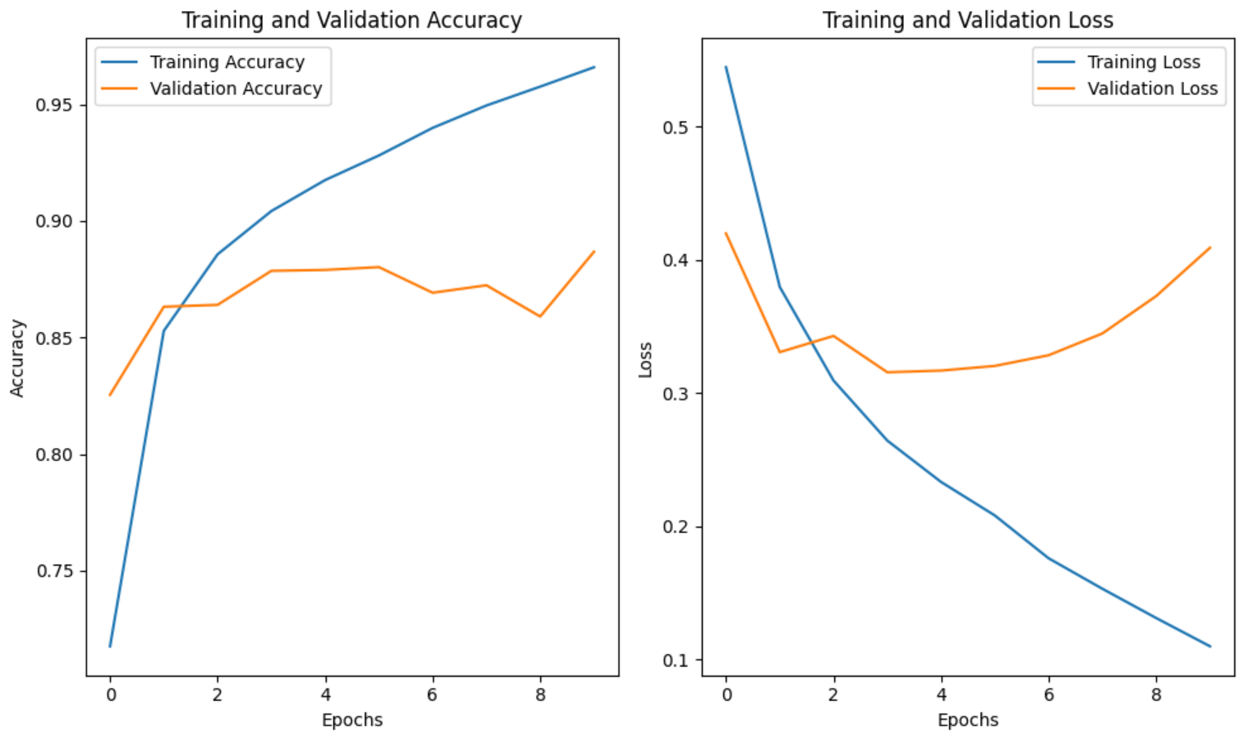
5. Before the layers. Bidirectional layer, consider

a) an embedding layer

```
# Define the model with the learned embedding layer
inputs = tf.keras.Input(shape=(None,), dtype="int64")
x = embedding_layer_trained(inputs)  # Trained embedding
x = layers.Bidirectional(layers.LSTM(32))(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)

model_trained = tf.keras.Model(inputs, outputs)
model_trained.compile(optimizer="rmsprop", loss="binary_crossentropy", metrics=["accuracy"])

# Train the model and capture the training history
history_trained = model_trained.fit(int_train_ds, validation_data=int_val_ds, epochs=10)

# Evaluate and print accuracy on the test dataset
print(f"Test acc (Trained Embedding): {model_trained.evaluate(int_test_ds)[1]:.3f}")
```



Base model

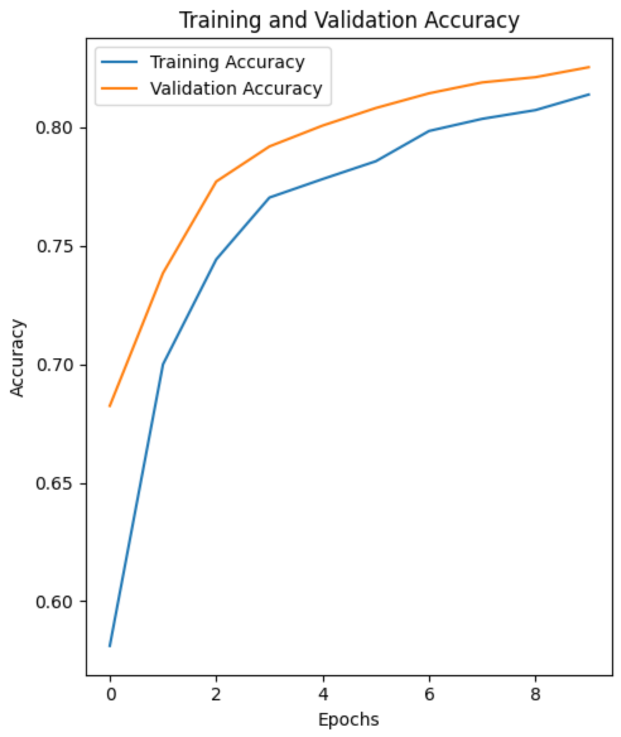


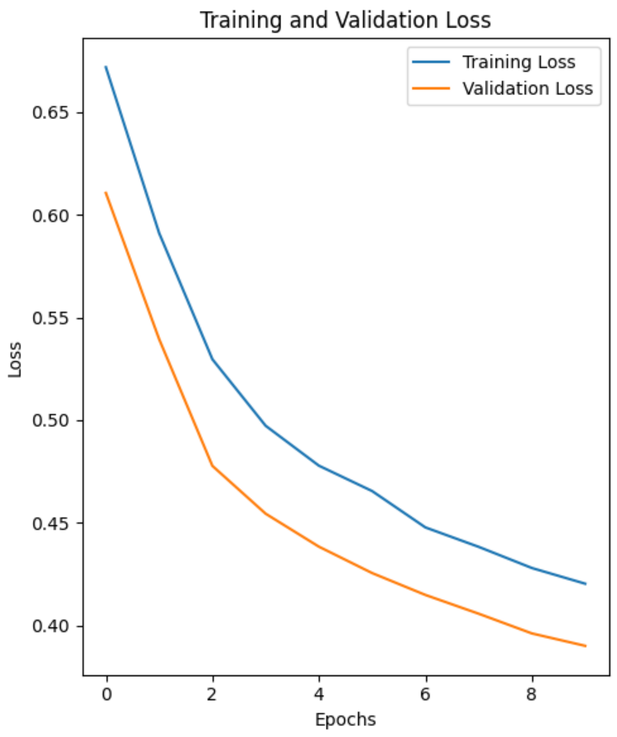
The base model outperformed the model with an embedding layer, achieving a higher test accuracy of 0.872 compared to 0.847.

b) A pretrained word embedding.

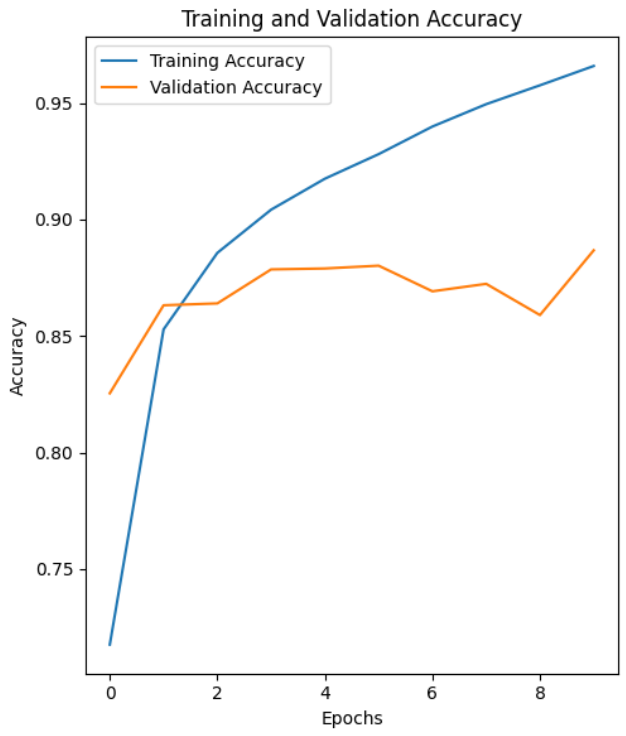
```
import tensorflow as tf
from tensorflow.keras import layers
from tensorflow import keras
import numpy as np

# Parameters
max_tokens = 20000  # Adjust according to your dataset/vocabulary size
embedding_dim = 100  # Dimension of GloVe embeddings
batch_size = 32  # Adjust batch size according to your system

# Step 1: Load GloVe embeddings
embeddings_index = {}
path_to_glove_file = "glove.6B.100d.txt"
with open(path_to_glove_file, encoding="utf-8") as f:
    for line in f:
        word, coefs = line.split(maxsplit=1)
        coefs = np.fromstring(coefs, "f", sep=" ")
        embeddings_index[word] = coefs
print(f"Found {len(embeddings_index)} word vectors.")

# Step 2: Load your dataset (using keras text_dataset_from_directory)
train_ds = keras.utils.text_dataset_from_directory(
    "aclImdb/train", batch_size=batch_size
)
val_ds = keras.utils.text_dataset_from_directory(
    "aclImdb/val", batch_size=batch_size
)
test_ds = keras.utils.text_dataset_from_directory(
    "aclImdb/test", batch_size=batch_size
)

# Step 3: Prepare the TextVectorization layer
# Extract texts from train_ds (text data)
train_texts = train_ds.map(lambda x, y: x)  # Extract only the texts
text_vectorization = layers.TextVectorization(max_tokens=max_tokens)
text_vectorization.adapt(train_texts)  # Adapt to your training data

# Prepare the embedding matrix
vocabulary = text_vectorization.get_vocabulary()  # Vocabulary from your dataset
word_index = dict(zip(vocabulary, range(len(vocabulary))))

embedding_matrix = np.zeros((max_tokens, embedding_dim))
for word, i in word_index.items():
    if i < max_tokens:
        embedding_vector = embeddings_index.get(word)
        if embedding_vector is not None:
            embedding_matrix[i] = embedding_vector

# Step 4: Define the embedding layer
embedding_layer_pretrained = layers.Embedding(
    input_dim=max_tokens,
    output_dim=embedding_dim,
    embeddings_initializer=keras.initializers.Constant(embedding_matrix),
    trainable=False,  # Freeze GloVe weights
    mask_zero=True,  # Optional: Mask padding tokens
)

# Step 5: Preprocess and prepare the datasets
def vectorize_text(text, label):
    return text_vectorization(text), label

train_ds = train_ds.map(vectorize_text).cache().prefetch(tf.data.AUTOTUNE)
val_ds = val_ds.map(vectorize_text).cache().prefetch(tf.data.AUTOTUNE)
test_ds = test_ds.map(vectorize_text).cache().prefetch(tf.data.AUTOTUNE)

# Step 6: Build the model
inputs = tf.keras.Input(shape=(None,), dtype="int64")
x = embedding_layer_pretrained(inputs)  # Use pretrained GloVe embeddings
x = layers.Bidirectional(layers.LSTM(32))(x)  # Bidirectional LSTM
x = layers.Dropout(0.5)(x)  # Dropout layer
outputs = layers.Dense(1, activation="sigmoid")(x)  # Output layer for binary classification

model_pretrained = tf.keras.Model(inputs, outputs)

# Step 7: Compile the model
model_pretrained.compile(
    optimizer=tf.keras.optimizers.RMSprop(learning_rate=1e-4),  # Adjust learning rate
    loss="binary_crossentropy",
    metrics=["accuracy"]
)

# Step 8: Train the model
history_pretrained = model_pretrained.fit(
    train_ds,  # Training dataset (tokenized and batched)
    validation_data=val_ds,  # Validation dataset
    epochs=10  # Number of epochs
)

# Step 9: Evaluate the model
test_loss, test_acc = model_pretrained.evaluate(test_ds)  # Test dataset
print(f"Test Accuracy (Pretrained Embedding): {test_acc:.3f}")
```

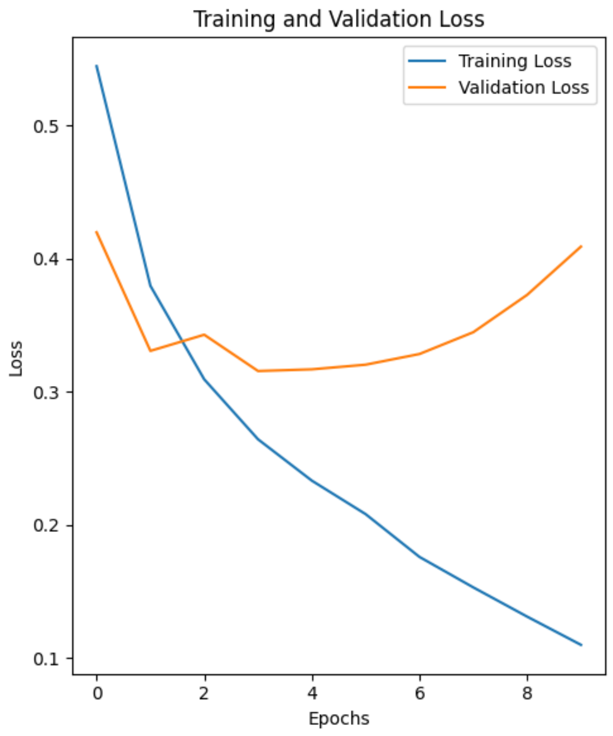
Pretrained Embedding): 0.814





Base model





The base model achieved a higher test accuracy of 0.872, outperforming the model with a pre-trained word embedding layer, which achieved 0.814.

Q1. Which Approach Performs Better?

The Base Model outperforms other configurations, especially models that rely on embedding layers, based on its superior performance metrics and key characteristics that optimize its effectiveness for the task. Below is a refined breakdown:

Performance Metrics:

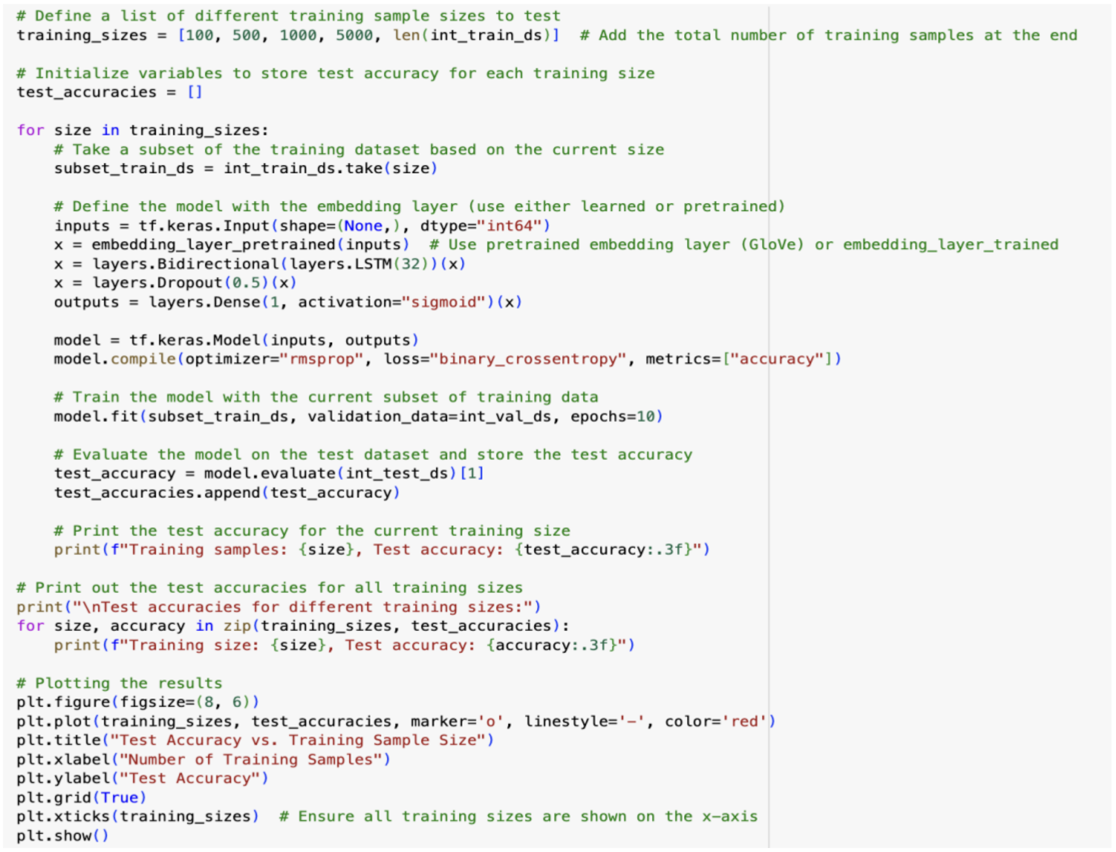
- Test Accuracy: The Base Model achieves the highest test accuracy of 0.872, surpassing both pretrained and learned embedding models.
- Stability: It demonstrates consistent performance across various training sizes, showcasing its strong ability to generalize.

Factors Contributing to Superior Performance:

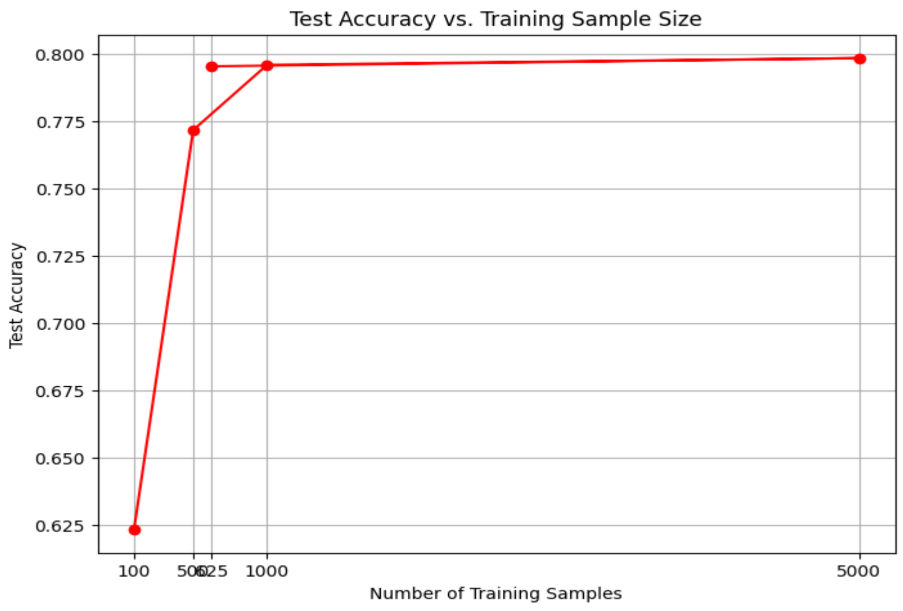
- Direct Feature Learning: The Base Model bypasses embedding layers and processes integer-encoded sequences directly through an LSTM. This method ensures feature extraction is customized specifically to the dataset, avoiding reliance on external embeddings.
- Dataset Specificity: Unlike pretrained embeddings (e.g., GloVe), which may not match the task-specific vocabulary, the Base Model learns representations directly from the data, maximizing relevance to the task at hand.
- Reduced Complexity: By omitting the embedding layers, the Base Model has fewer parameters, reducing computational complexity and minimizing the risk of overfitting, particularly with smaller datasets.
- Regularization: The inclusion of a Dropout layer (rate: 0.5) effectively addresses overfitting, improving the model's generalization ability, especially when working with limited training data.
- Task Alignment: Embedding layers can introduce unnecessary transformations or rely on pretrained data that may not be aligned with the specific linguistic context of the task. The architecture of the Base Model is more directly aligned with the classification goal, enhancing its relevance and accuracy.

Conclusion: The Base Model stands out due to its combination of dataset specificity, efficient direct learning, and robust architecture. These factors contribute to its higher accuracy and stability, making it more suited for the task compared to models that use embedding layers.

Q2. Now try changing the number of training samples to determine at what point the embedding layer gives better performance



```python
# Define a list of different training sample sizes to test
training_sizes = [100, 500, 1000, 5000, len(int_train_ds)]  # Add the total number of training samples at the end

# Initialize variables to store test accuracy for each training size
test_accuracies = []

for size in training_sizes:
    # Take a subset of the training dataset based on the current size
    subset_train_ds = int_train_ds.take(size)

    # Define the model with the embedding layer (use either learned or pretrained)
    inputs = tf.keras.Input(shape=(None,), dtype="int64")
    x = embedding_layer_pretrained(inputs)  # Use pretrained embedding layer (GloVe) or embedding_layer_trained
    x = layers.Bidirectional(layers.LSTM(32))(x)
    x = layers.Dropout(0.5)(x)
    outputs = layers.Dense(1, activation="sigmoid")(x)

    model = tf.keras.Model(inputs, outputs)
    model.compile(optimizer="rmsprop", loss="binary_crossentropy", metrics=["accuracy"])

    # Train the model with the current subset of training data
    model.fit(subset_train_ds, validation_data=int_val_ds, epochs=10)

    # Evaluate the model on the test dataset and store the test accuracy
    test_accuracy = model.evaluate(int_test_ds)[1]
    test_accuracies.append(test_accuracy)

    # Print the test accuracy for the current training size
    print(f"Training samples: {size}, Test accuracy: {test_accuracy:.3f}")

# Print out the test accuracies for all training sizes
print("\nTest accuracies for different training sizes:")
for size, accuracy in zip(training_sizes, test_accuracies):
    print(f"Training size: {size}, Test accuracy: {accuracy:.3f}")

# Plotting the results
plt.figure(figsize=(8, 6))
plt.plot(training_sizes, test_accuracies, marker='o', linestyle='-', color='red')
plt.title("Test Accuracy vs. Training Sample Size")
plt.xlabel("Number of Training Samples")
plt.ylabel("Test Accuracy")
plt.grid(True)
plt.xticks(training_sizes)  # Ensure all training sizes are shown on the x-axis
plt.show()
```



Training Size	Test Accuracy	Reason
100	0.624	Insufficient data for effective model generalization, leading to underfitting and high bias.
500	0.772	Noticeable improvement (+0.148) due to increased data, but still constrained by limited sample size.
625	0.795	Incremental gain (+0.023) suggests data adequacy, though diminishing returns as the model approaches a balanced state.
1000	0.796	Marginal improvement (+0.001), indicating the model's performance has plateaued, nearing optimal capacity.
5000	0.798	Minimal increase (+0.002), indicating model performance has reached saturation, with potential risks of overfitting due to excess data.

1. Small Training Sizes (100 & 500)

- **Reason for Lower Accuracy**: With only 100 and 500 training samples, the model lacks sufficient data to capture meaningful patterns, leading to underfitting. This results in lower test accuracy: 0.624 for 100 samples and 0.772 for 500 samples.
- **Performance Gain (500 → 625)**: Increasing the training size from 500 to 625 samples improves test accuracy from 0.772 to 0.795, a modest gain of +0.023. This improvement is due to the additional data providing more diversity, but the gain is limited due to the small increase in sample size.

2. Moderate Training Size (1000)

- **Reason for Higher Accuracy**: At 1000 samples, the model can capture a broader range of patterns, leading to improved generalization and a peak test accuracy of 0.796. This indicates that the dataset is large enough for the model to learn more complex relationships within the data.
- **Optimal Training Size**: The accuracy plateaus at 1000 samples, suggesting that this is the optimal dataset size for the model. At this point, the model has enough data to generalize effectively, and further increases in training size yield minimal improvement.

3. Large Training Size (5000)

- **Diminishing Returns**: Increasing the training size to 5000 samples results in a very small accuracy increase from 0.796 to 0.798 (+0.002). This suggests that the model's learning has saturated, and additional data introduces redundancy rather than new information.

- **Potential Overfitting**: The negligible increase in accuracy, coupled with the possibility of overfitting, indicates that the model may become overly tailored to the training data and less capable of generalizing to unseen data.

4. Specific Case: Training Size of 625

- **Unexpected Accuracy Boost**: At 625 samples, the accuracy (0.795) is very close to the peak accuracy at 1000 samples (0.796). This could be due to the 625-sample dataset containing a sufficiently representative subset of the data, allowing the model to generalize almost as effectively as with 1000 samples. Random variance in the test set may also play a role in this similarity.

5. Why Test Accuracy Plateaus

- **Saturation**: Once the model has been trained on a sufficiently large portion of the data (around 1000 samples), adding more samples yields diminishing returns. The model has already learned the majority of relevant patterns and further increases in training size contribute less to improving generalization. This results in a plateau in test accuracy as the model reaches a state of saturation.

Limitations:

- **Model Complexity**: If the model architecture is overly simplistic, it may fail to fully leverage the additional data, limiting its potential.
- **Data Quality**: Training with redundant or noisy data may not lead to significant improvements in performance, as it doesn't add valuable information.

Downloading the data

```
# Step 1: Download the dataset
import os
import tarfile
import urllib.request

# Download the dataset
url = "https://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz"
file_name = "aclImdb_v1.tar.gz"
urllib.request.urlretrieve(url, file_name)

# Step 2: Extract the tar.gz file
if file_name.endswith("tar.gz"):
    with tarfile.open(file_name, "r:gz") as tar:
        tar.extractall()

# Step 3: Remove the 'unsup' directory (Windows-compatible)
unsup_dir = "aclImdb/train/unsup"

# Check if directory exists before removing it
if os.path.exists(unsup_dir):
    import shutil
    shutil.rmtree(unsup_dir)

print("Dataset downloaded, extracted, and 'unsup' directory removed.")
```

```
Dataset downloaded, extracted, and 'unsup' directory removed.
```

```
!curl -O https://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz
!tar -xf aclImdb_v1.tar.gz
!rm -r aclImdb/train/unsup
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 80.2M  100 80.2M    0     0  29.9M      0  0:00:02  0:00:02 --:--:-- 29.9M
```

Preparing the data

```
import os, pathlib, shutil, random
from tensorflow import keras
batch_size = 32
base_dir = pathlib.Path("aclImdb")
val_dir = base_dir / "val"
train_dir = base_dir / "train"
for category in ("neg", "pos"):
    os.makedirs(val_dir / category, exist_ok=True)
    files = os.listdir(train_dir / category)
    random.Random(1337).shuffle(files)
    num_val_samples = int(0.2 * len(files))
    val_files = files[-num_val_samples:]
    for fname in val_files:
        shutil.move(train_dir / category / fname,
                    val_dir / category / fname)

train_ds = keras.utils.text_dataset_from_directory(
    "aclImdb/train", batch_size=batch_size
)
val_ds = keras.utils.text_dataset_from_directory(
    "aclImdb/val", batch_size=batch_size
)
test_ds = keras.utils.text_dataset_from_directory(
    "aclImdb/test", batch_size=batch_size
)
text_only_train_ds = train_ds.map(lambda x, y: x)
```

```
Found 20000 files belonging to 2 classes.
Found 5000 files belonging to 2 classes.
Found 25000 files belonging to 2 classes.
```

Preparing integer sequence datasets

```
from tensorflow.keras import layers

max_length = 600
max_tokens = 20000
text_vectorization = layers.TextVectorization(
    max_tokens=max_tokens,
    output_mode="int",
    output_sequence_length=max_length,
)
text_vectorization.adapt(text_only_train_ds)

int_train_ds = train_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)
int_val_ds = val_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)
int_test_ds = test_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)
```

A sequence model built on one-hot encoded vector sequences

```
import tensorflow as tf
from tensorflow.keras import layers, models

# Define max_tokens
max_tokens = 20000

# Define model
inputs = tf.keras.Input(shape=(None,), dtype="int64")
# Embed each integer token using an Embedding layer
# This will give you a 3D tensor (batch_size, sequence_length, embedding_dim)
embedded = layers.Embedding(input_dim=max_tokens, output_dim=128)(inputs)
# Now the output of the embedding layer is 3D, suitable for the LSTM
x = layers.Bidirectional(layers.LSTM(32))(embedded)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)

model = tf.keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
model.summary()
```

Model: "functional"

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, None)	0
embedding (Embedding)	(None, None, 128)	2,560,000
bidirectional (Bidirectional)	(None, 64)	41,216
dropout (Dropout)	(None, 64)	0
dense (Dense)	(None, 1)	65

```
Total params: 2,601,281 (9.92 MB)
Trainable params: 2,601,281 (9.92 MB)
Non-trainable params: 0 (0.00 B)
```

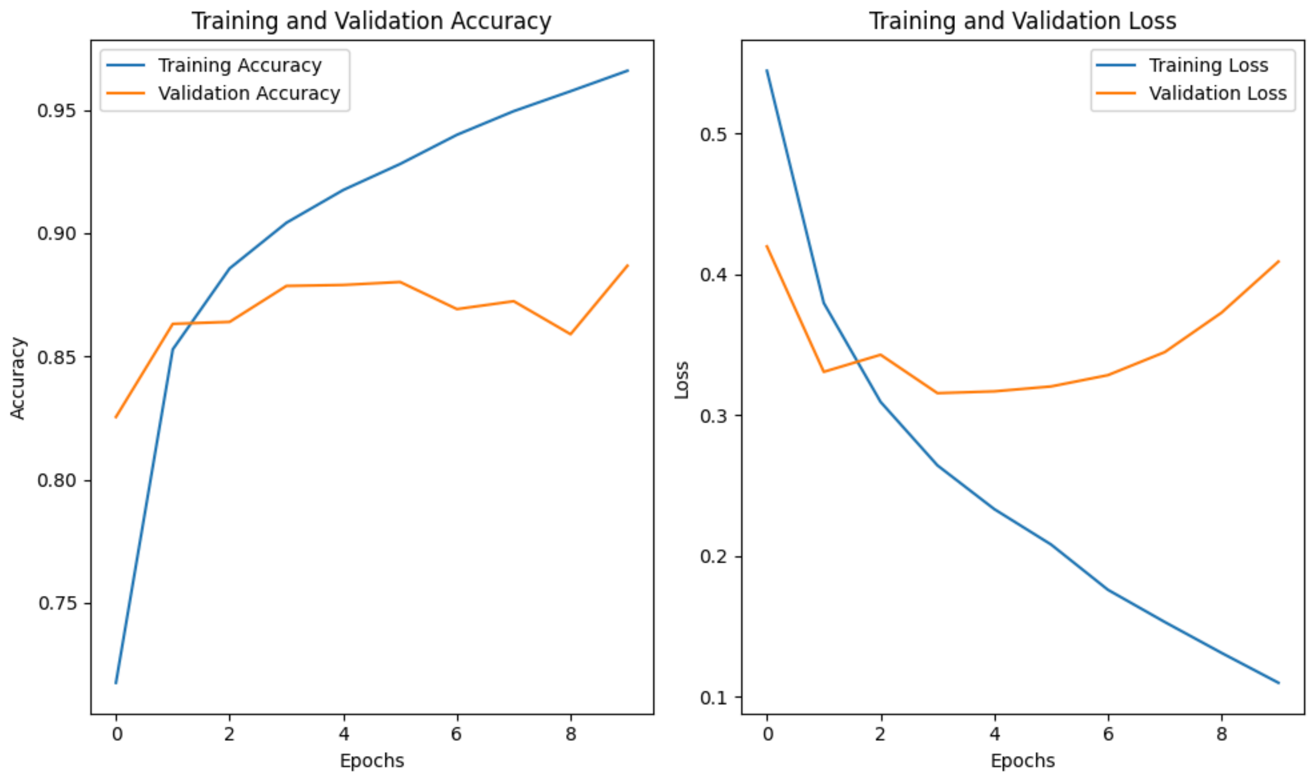
Training a first basic sequence model

```
callbacks = [
    keras.callbacks.ModelCheckpoint("one_hot_bidir_lstm.keras",
                                    save_best_only=True)
]
History = model.fit(int_train_ds, validation_data=int_val_ds, epochs=10, callbacks=callbacks)
model = keras.models.load_model("one_hot_bidir_lstm.keras")
print(f"Test acc: {model.evaluate(int_test_ds)[1]:.3f}")
```

```
Epoch 1/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 31s 43ms/step - accuracy: 0.6196 - loss: 0.6246 - val_accuracy: 0.8254 - val_loss: 0.4197
Epoch 2/10
```

```
625/625 ━━━━━━━━━━━━━━━━━━━━ 27s 44ms/step - accuracy: 0.8378 - loss: 0.4022 - val_accuracy: 0.8632 - val_loss: 0.3307
Epoch 3/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 42s 45ms/step - accuracy: 0.8774 - loss: 0.3233 - val_accuracy: 0.8640 - val_loss: 0.3428
Epoch 4/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 26s 42ms/step - accuracy: 0.8986 - loss: 0.2794 - val_accuracy: 0.8786 - val_loss: 0.3155
Epoch 5/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 26s 42ms/step - accuracy: 0.9142 - loss: 0.2406 - val_accuracy: 0.8790 - val_loss: 0.3168
Epoch 6/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 40s 41ms/step - accuracy: 0.9233 - loss: 0.2160 - val_accuracy: 0.8802 - val_loss: 0.3203
Epoch 7/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 41s 41ms/step - accuracy: 0.9360 - loss: 0.1845 - val_accuracy: 0.8692 - val_loss: 0.3283
Epoch 8/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 28s 45ms/step - accuracy: 0.9457 - loss: 0.1624 - val_accuracy: 0.8724 - val_loss: 0.3447
Epoch 9/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 39s 41ms/step - accuracy: 0.9544 - loss: 0.1402 - val_accuracy: 0.8590 - val_loss: 0.3727
Epoch 10/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 44s 46ms/step - accuracy: 0.9647 - loss: 0.1145 - val_accuracy: 0.8868 - val_loss: 0.4090
782/782 ━━━━━━━━━━━━━━━━━━━━ 15s 18ms/step - accuracy: 0.8702 - loss: 0.3323
Test acc: 0.872
```

```
import matplotlib.pyplot as plt
# Plotting the training and validation accuracy
plt.figure(figsize=(10, 6))

# Plot training and validation accuracy
plt.subplot(1, 2, 1)
plt.plot(History.history['accuracy'], label='Training Accuracy')
plt.plot(History.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

# Plot training and validation loss
plt.subplot(1, 2, 2)
plt.plot(History.history['loss'], label='Training Loss')
plt.plot(History.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.tight_layout()
plt.show()
```

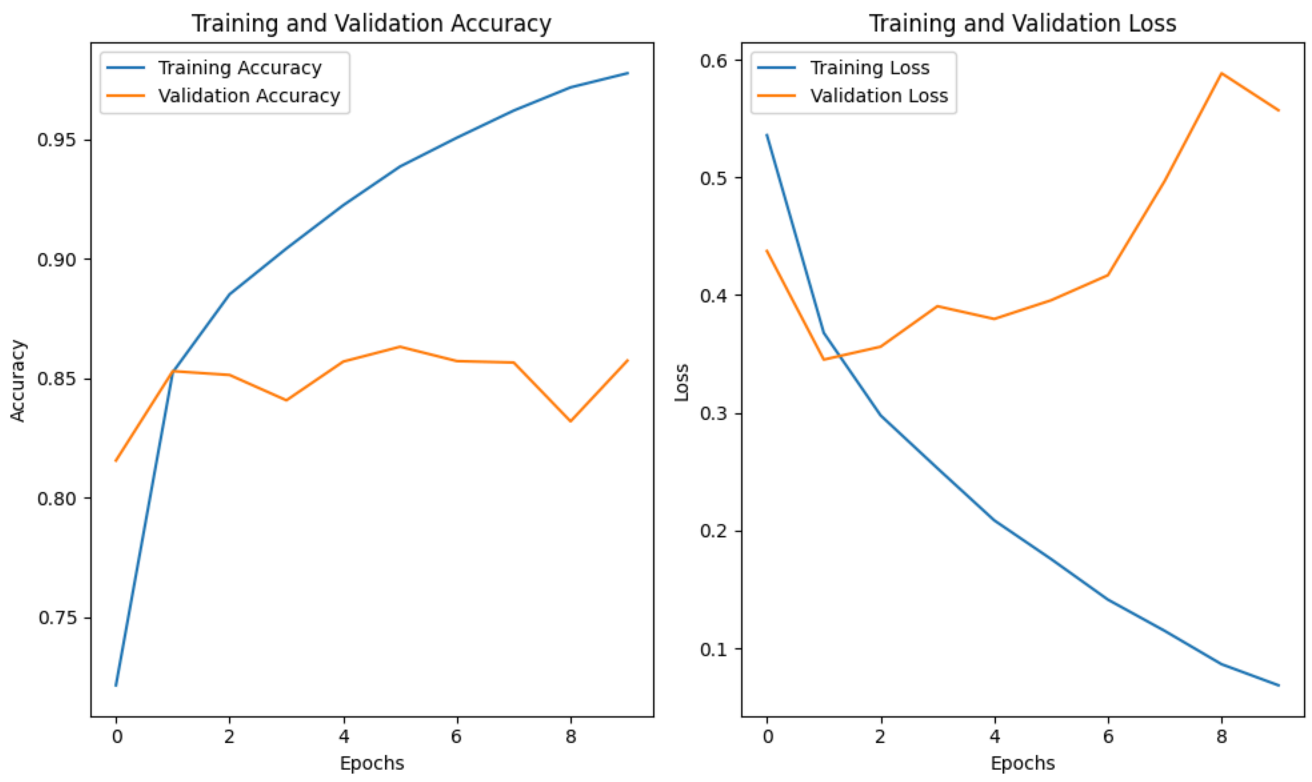


Question 1

```
max_length_150 = 150
text_vectorization_150 = layers.TextVectorization(
    max_tokens=max_tokens,
    output_mode="int",
    output_sequence_length=max_length_150, #This ensure the review is turncated to exactly 150 tokens
)
text_vectorization_150.adapt(text_only_train_ds)

# Apply vectorization to train, validation, and test datasets
int_train_ds_150 = train_ds.map(
    lambda x, y: (text_vectorization_150(x), y),
    num_parallel_calls=4)
int_val_ds_150 = val_ds.map(
    lambda x, y: (text_vectorization_150(x), y),
    num_parallel_calls=4)
int_test_ds_150 = test_ds.map(  # Apply the same vectorization to test data
    lambda x, y: (text_vectorization_150(x), y),
    num_parallel_calls=4)


# Define the model
model_cutoff_150 = tf.keras.Sequential([
    layers.Embedding(max_tokens, 128, input_length=max_length_150),
    layers.Bidirectional(layers.LSTM(32)),
    layers.Dropout(0.5),
    layers.Dense(1, activation="sigmoid"),
])

# Compile the model
model_cutoff_150.compile(optimizer="rmsprop", loss="binary_crossentropy", metrics=["accuracy"])

checkpoint_callback = keras.callbacks.ModelCheckpoint("one_hot_bidir_lstm_150.keras", save_best_only=True)

history_150 = model_cutoff_150.fit(
    int_train_ds_150,
    validation_data=int_val_ds_150,
    epochs=10,
    callbacks=[checkpoint_callback]
)

# Evaluate and print accuracy on the test dataset
print(f"Test acc: {model_cutoff_150.evaluate(int_test_ds_150)[1]:.3f}")
```

```
Epoch 1/10
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/embedding.py:90: UserWarning: Argument `input_length` is depre
  warnings.warn(
625/625 ━━━━━━━━━━━━━━━━━━━━ 15s 19ms/step - accuracy: 0.6218 - loss: 0.6185 - val_accuracy: 0.8156 - val_loss: 0.4375
Epoch 2/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 11s 18ms/step - accuracy: 0.8401 - loss: 0.3920 - val_accuracy: 0.8530 - val_loss: 0.3452
Epoch 3/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 20s 18ms/step - accuracy: 0.8760 - loss: 0.3149 - val_accuracy: 0.8514 - val_loss: 0.3562
Epoch 4/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 11s 18ms/step - accuracy: 0.8987 - loss: 0.2673 - val_accuracy: 0.8408 - val_loss: 0.3906
Epoch 5/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 11s 18ms/step - accuracy: 0.9162 - loss: 0.2241 - val_accuracy: 0.8570 - val_loss: 0.3798
Epoch 6/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 11s 17ms/step - accuracy: 0.9342 - loss: 0.1877 - val_accuracy: 0.8632 - val_loss: 0.3956
Epoch 7/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 21s 18ms/step - accuracy: 0.9464 - loss: 0.1504 - val_accuracy: 0.8572 - val_loss: 0.4169
Epoch 8/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 20s 18ms/step - accuracy: 0.9562 - loss: 0.1270 - val_accuracy: 0.8566 - val_loss: 0.4971
Epoch 9/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 21s 18ms/step - accuracy: 0.9680 - loss: 0.0972 - val_accuracy: 0.8320 - val_loss: 0.5885
Epoch 10/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 21s 19ms/step - accuracy: 0.9754 - loss: 0.0766 - val_accuracy: 0.8574 - val_loss: 0.5571
782/782 ━━━━━━━━━━━━━━━━━━━━ 6s 8ms/step - accuracy: 0.8276 - loss: 0.6728
Test acc: 0.824
```

```
# Plotting the training and validation accuracy
plt.figure(figsize=(10, 6))

# Plot training and validation accuracy
plt.subplot(1, 2, 1)
plt.plot(history_150.history['accuracy'], label='Training Accuracy')
plt.plot(history_150.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

# Plot training and validation loss
plt.subplot(1, 2, 2)
plt.plot(history_150.history['loss'], label='Training Loss')
plt.plot(history_150.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.tight_layout()
plt.show()
```



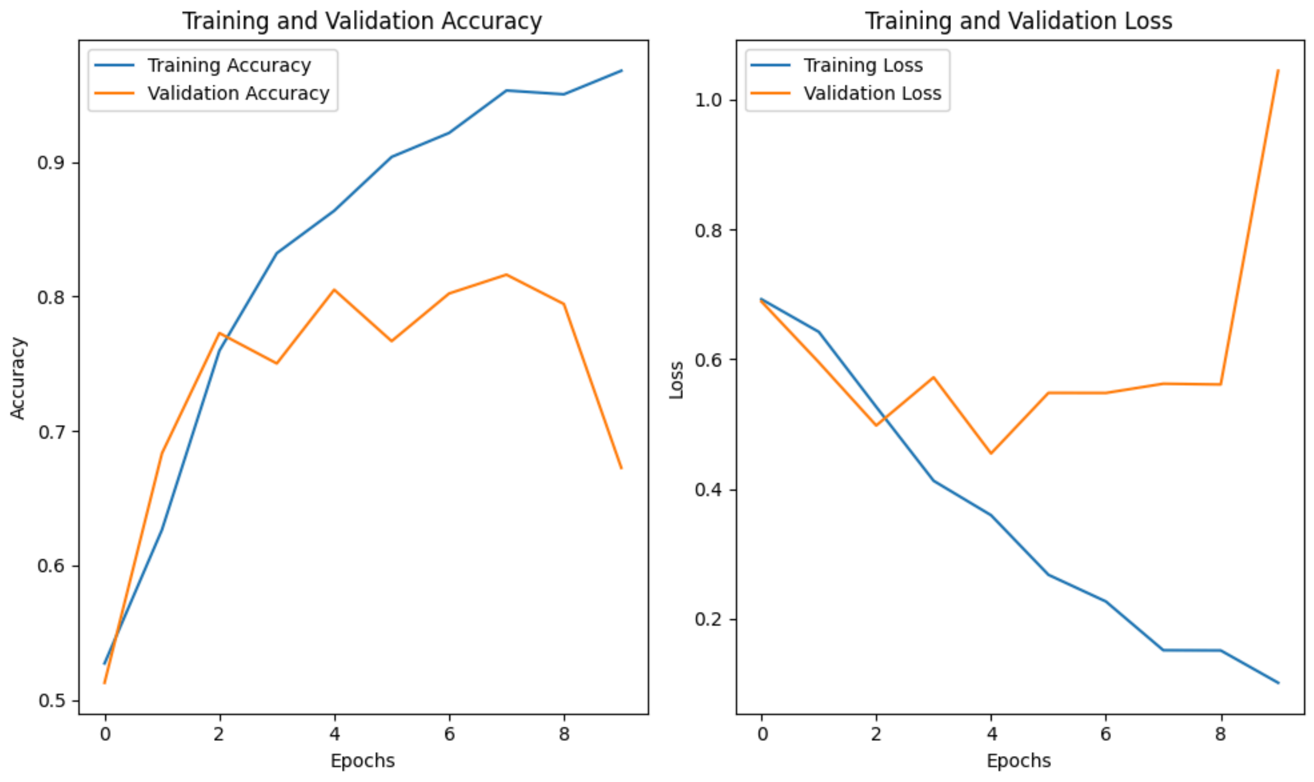
Question 2

```
int_train_ds_small = int_train_ds.take(100) # Limit training samples to 100

# Define model
model_100_samples = tf.keras.Sequential([
    layers.Embedding(max_tokens, 128, input_length=max_length),
    layers.Bidirectional(layers.LSTM(32)),
    layers.Dropout(0.5),
    layers.Dense(1, activation="sigmoid"),
])

model_100_samples.compile(optimizer="rmsprop", loss="binary_crossentropy", metrics=["accuracy"])

# Train the model and capture the training history
history_100_samples = model_100_samples.fit(
    int_train_ds_small,
    validation_data=int_val_ds,
    epochs=10
)

print(f"Test acc: {model_100_samples.evaluate(int_test_ds)[1]:.3f}")
```

```
Epoch 1/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 9s 64ms/step - accuracy: 0.5092 - loss: 0.6937 - val_accuracy: 0.5128 - val_loss: 0.6890
Epoch 2/10
```

```
100/100 ━━━━━━━━━━━━━━━━━━━━ 7s 68ms/step - accuracy: 0.5737 - loss: 0.6711 - val_accuracy: 0.6834 - val_loss: 0.5954
Epoch 3/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 9s 86ms/step - accuracy: 0.7445 - loss: 0.5503 - val_accuracy: 0.7728 - val_loss: 0.4980
Epoch 4/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 6s 62ms/step - accuracy: 0.8103 - loss: 0.4382 - val_accuracy: 0.7502 - val_loss: 0.5722
Epoch 5/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 9s 88ms/step - accuracy: 0.8594 - loss: 0.3698 - val_accuracy: 0.8050 - val_loss: 0.4549
Epoch 6/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 9s 88ms/step - accuracy: 0.9058 - loss: 0.2615 - val_accuracy: 0.7668 - val_loss: 0.5483
Epoch 7/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.9155 - loss: 0.2306 - val_accuracy: 0.8022 - val_loss: 0.5481
Epoch 8/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 9s 94ms/step - accuracy: 0.9611 - loss: 0.1415 - val_accuracy: 0.8162 - val_loss: 0.5623
Epoch 9/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 7s 68ms/step - accuracy: 0.9540 - loss: 0.1377 - val_accuracy: 0.7944 - val_loss: 0.5612
Epoch 10/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 6s 61ms/step - accuracy: 0.9775 - loss: 0.0726 - val_accuracy: 0.6726 - val_loss: 1.0444
782/782 ━━━━━━━━━━━━━━━━━━━━ 14s 18ms/step - accuracy: 0.6652 - loss: 1.0621
Test acc: 0.674
```

```python
# Plotting the training and validation accuracy
plt.figure(figsize=(10, 6))

# Plot training and validation accuracy
plt.subplot(1, 2, 1)
plt.plot(history_100_samples.history['accuracy'], label='Training Accuracy')
plt.plot(history_100_samples.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

# Plot training and validation loss
plt.subplot(1, 2, 2)
plt.plot(history_100_samples.history['loss'], label='Training Loss')
plt.plot(history_100_samples.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.tight_layout()
plt.show()
```

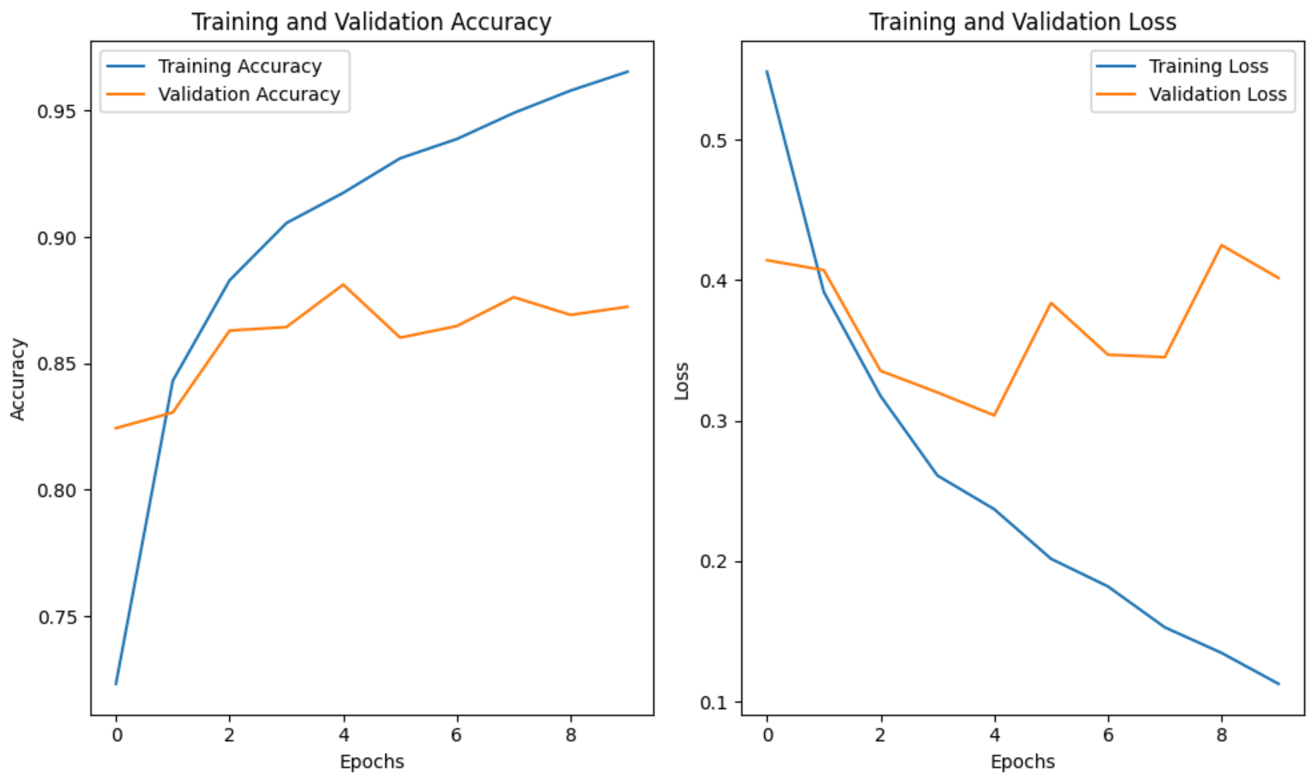


Question 3

```
# Limit validation dataset to 10,000 samples
int_val_ds_10k = int_val_ds.take(10000)

# Define model
model_validate_10k = tf.keras.Sequential([
    layers.Embedding(max_tokens, 128, input_length=max_length),
    layers.Bidirectional(layers.LSTM(32)),
    layers.Dropout(0.5),
    layers.Dense(1, activation="sigmoid"),
])

# Compile the model
model_validate_10k.compile(optimizer="rmsprop", loss="binary_crossentropy", metrics=["accuracy"])

history_validate_10k = model_validate_10k.fit(
    int_train_ds,
    validation_data=int_val_ds_10k,
    epochs=10
)

# Evaluate and print accuracy on the test dataset
print(f"Test acc: {model_validate_10k.evaluate(int_test_ds)[1]:.3f}")
```

```
Epoch 1/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 29s 42ms/step - accuracy: 0.6348 - loss: 0.6222 - val_accuracy: 0.8244 - val_loss: 0.4143
Epoch 2/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 26s 41ms/step - accuracy: 0.8263 - loss: 0.4159 - val_accuracy: 0.8306 - val_loss: 0.4072
Epoch 3/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 41s 41ms/step - accuracy: 0.8783 - loss: 0.3284 - val_accuracy: 0.8630 - val_loss: 0.3355
Epoch 4/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 26s 41ms/step - accuracy: 0.8993 - loss: 0.2760 - val_accuracy: 0.8644 - val_loss: 0.3202
Epoch 5/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 41s 41ms/step - accuracy: 0.9138 - loss: 0.2448 - val_accuracy: 0.8812 - val_loss: 0.3038
Epoch 6/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 42s 42ms/step - accuracy: 0.9267 - loss: 0.2158 - val_accuracy: 0.8602 - val_loss: 0.3839
Epoch 7/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 26s 41ms/step - accuracy: 0.9332 - loss: 0.1958 - val_accuracy: 0.8648 - val_loss: 0.3470
Epoch 8/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 26s 41ms/step - accuracy: 0.9485 - loss: 0.1570 - val_accuracy: 0.8762 - val_loss: 0.3453
Epoch 9/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 41s 41ms/step - accuracy: 0.9566 - loss: 0.1377 - val_accuracy: 0.8692 - val_loss: 0.4250
Epoch 10/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 41s 41ms/step - accuracy: 0.9642 - loss: 0.1177 - val_accuracy: 0.8724 - val_loss: 0.4017
782/782 ━━━━━━━━━━━━━━━━━━━━ 14s 18ms/step - accuracy: 0.8462 - loss: 0.4867
Test acc: 0.846
```

```
# Plotting the training and validation accuracy
plt.figure(figsize=(10, 6))

# Plot training and validation accuracy
plt.subplot(1, 2, 1)
plt.plot(history_validate_10k.history['accuracy'], label='Training Accuracy')
plt.plot(history_validate_10k.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

# Plot training and validation loss
plt.subplot(1, 2, 2)
plt.plot(history_validate_10k.history['loss'], label='Training Loss')
plt.plot(history_validate_10k.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.tight_layout()
plt.show()
```

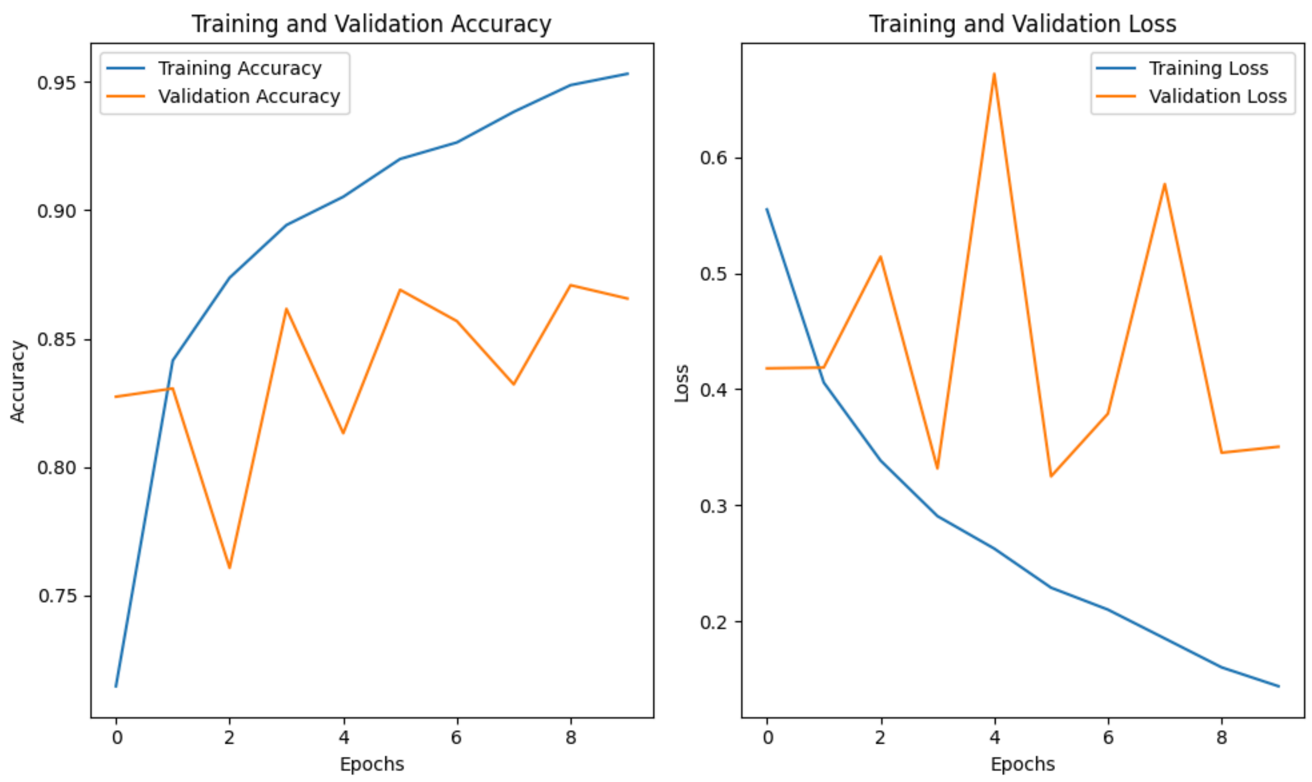


Question 4

```
# Adjust max_tokens to 10,000
max_tokens_10k = 10000
text_vectorization_10k = layers.TextVectorization(
    max_tokens=max_tokens_10k,
    output_mode="int",
    output_sequence_length=max_length,
)
text_vectorization_10k.adapt(text_only_train_ds)

# Transform datasets with the updated vectorizer
int_train_ds_10k = train_ds.map(
    lambda x, y: (text_vectorization_10k(x), y),
    num_parallel_calls=4)
int_val_ds_10k = val_ds.map(
    lambda x, y: (text_vectorization_10k(x), y),
    num_parallel_calls=4)
int_test_ds_10k = test_ds.map(
    lambda x, y: (text_vectorization_10k(x), y),
    num_parallel_calls=4)

# Define model
model_top_10k = tf.keras.Sequential([
    layers.Embedding(max_tokens_10k, 128, input_length=max_length),
    layers.Bidirectional(layers.LSTM(32)),
    layers.Dropout(0.5),
    layers.Dense(1, activation="sigmoid"),
])

# Compile the model
model_top_10k.compile(optimizer="rmsprop", loss="binary_crossentropy", metrics=["accuracy"])

# Train the model and capture the training history
history_top_10k = model_top_10k.fit(
    int_train_ds_10k,
    validation_data=int_val_ds_10k,
    epochs=10
)

# Evaluate and print accuracy on the test dataset
print(f"Test acc: {model_top_10k.evaluate(int_test_ds_10k)[1]:.3f}")
```

```
Epoch 1/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 30s 45ms/step - accuracy: 0.6208 - loss: 0.6252 - val_accuracy: 0.8274 - val_loss: 0.4181
Epoch 2/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 25s 41ms/step - accuracy: 0.8323 - loss: 0.4186 - val_accuracy: 0.8306 - val_loss: 0.4188
Epoch 3/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 26s 42ms/step - accuracy: 0.8672 - loss: 0.3482 - val_accuracy: 0.7608 - val_loss: 0.5143
Epoch 4/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 40s 41ms/step - accuracy: 0.8854 - loss: 0.3044 - val_accuracy: 0.8616 - val_loss: 0.3318
Epoch 5/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 43s 45ms/step - accuracy: 0.9019 - loss: 0.2704 - val_accuracy: 0.8132 - val_loss: 0.6718
Epoch 6/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 39s 41ms/step - accuracy: 0.9182 - loss: 0.2361 - val_accuracy: 0.8690 - val_loss: 0.3250
Epoch 7/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 44s 45ms/step - accuracy: 0.9219 - loss: 0.2129 - val_accuracy: 0.8568 - val_loss: 0.3790
Epoch 8/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 41s 45ms/step - accuracy: 0.9363 - loss: 0.1908 - val_accuracy: 0.8322 - val_loss: 0.5769
Epoch 9/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 38s 41ms/step - accuracy: 0.9460 - loss: 0.1672 - val_accuracy: 0.8708 - val_loss: 0.3454
Epoch 10/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 41s 41ms/step - accuracy: 0.9511 - loss: 0.1528 - val_accuracy: 0.8656 - val_loss: 0.3505
782/782 ━━━━━━━━━━━━━━━━━━━━ 15s 19ms/step - accuracy: 0.8586 - loss: 0.3761
Test acc: 0.856
```

```
# Plotting the training and validation accuracy
plt.figure(figsize=(10, 6))

# Plot training and validation accuracy
plt.subplot(1, 2, 1)
plt.plot(history_top_10k.history['accuracy'], label='Training Accuracy')
plt.plot(history_top_10k.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

# Plot training and validation loss
plt.subplot(1, 2, 2)
plt.plot(history_top_10k.history['loss'], label='Training Loss')
plt.plot(history_top_10k.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.tight_layout()
plt.show()
```



Question 5

Preparing pre trained model

```
import os
import tarfile
import urllib.request
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras import initializers

# Step 1: Download the dataset
url = "https://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz"
file_name = "aclImdb_v1.tar.gz"
urllib.request.urlretrieve(url, file_name)

# Step 2: Extract the tar.gz file
if file_name.endswith("tar.gz"):
    with tarfile.open(file_name, "r:gz") as tar:
        tar.extractall()

# Step 3: Remove the 'unsup' directory (Windows-compatible)
unsup_dir = "aclImdb/train/unsup"
if os.path.exists(unsup_dir):
    import shutil
    shutil.rmtree(unsup_dir)

print("Dataset downloaded, extracted, and 'unsup' directory removed.")

# Step 4: Load GloVe embeddings
path_to_glove_file = "glove.6B.100d.txt"
embeddings_index = {}
if os.path.exists(path_to_glove_file):
    with open(path_to_glove_file, encoding="utf-8") as f:
        for line in f:
            word, coefs = line.split(maxsplit=1)
            coefs = np.fromstring(coefs, "f", sep=" ")
            embeddings_index[word] = coefs
    print(f"Found {len(embeddings_index)} word vectors.")
else:
    print("GloVe embeddings file not found. Please download it and place it in the working directory.")
```

```
# Step 5: Prepare vocabulary and embedding matrix
# Example TextVectorization layer
max_tokens = 20000  # Maximum number of words in the vocabulary
max_length = 200  # Maximum length of input sequences
text_vectorization = layers.TextVectorization(max_tokens=max_tokens, output_sequence_length=max_length)

# Simulating a vocabulary (replace with your actual vocabulary from `text_vectorization`)
vocabulary = ["the", "and", "movie", "good", "bad", "great"]  # Example
word_index = dict(zip(vocabulary, range(len(vocabulary))))

# Initialize embedding matrix
embedding_dim = 100  # Must match the GloVe file dimension
embedding_matrix = np.zeros((max_tokens, embedding_dim))
for word, i in word_index.items():
    if i < max_tokens:
        embedding_vector = embeddings_index.get(word)
        if embedding_vector is not None:
            embedding_matrix[i] = embedding_vector

# Step 6: Define embedding layers
# a) Embedding layer with learned embeddings
embedding_layer_trained = layers.Embedding(
    max_tokens,
    embedding_dim,
    input_length=max_length
)

# b) Pretrained embedding layer using GloVe (frozen)
embedding_layer_pretrained = layers.Embedding(
    max_tokens,
    embedding_dim,
    embeddings_initializer=initializers.Constant(embedding_matrix),
    trainable=False,  # Freeze embeddings
    mask_zero=True
)

print("Embedding layers defined.")
```

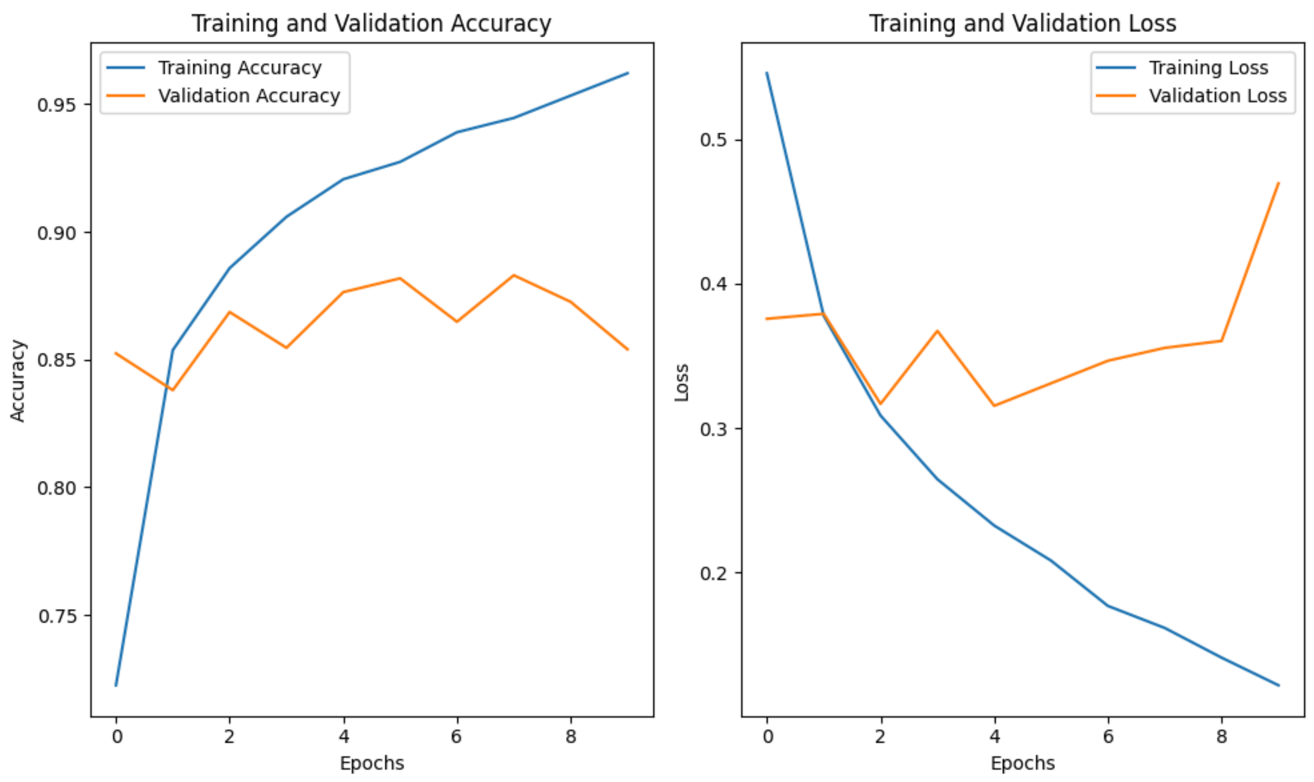
```
Dataset downloaded, extracted, and 'unsup' directory removed.
GloVe embeddings file not found. Please download it and place it in the working directory.
Embedding layers defined.
```

```
# Define the model with the learned embedding layer
inputs = tf.keras.Input(shape=(None,), dtype="int64")
x = embedding_layer_trained(inputs)  # Trained embedding
x = layers.Bidirectional(layers.LSTM(32))(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)

model_trained = tf.keras.Model(inputs, outputs)
model_trained.compile(optimizer="rmsprop", loss="binary_crossentropy", metrics=["accuracy"])

# Train the model and capture the training history
history_trained = model_trained.fit(int_train_ds, validation_data=int_val_ds, epochs=10)

# Evaluate and print accuracy on the test dataset
print(f"Test acc (Trained Embedding): {model_trained.evaluate(int_test_ds)[1]:.3f}")
```

```
Epoch 1/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 30s 45ms/step - accuracy: 0.6278 - loss: 0.6228 - val_accuracy: 0.8524 - val_loss: 0.3757
Epoch 2/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 25s 40ms/step - accuracy: 0.8431 - loss: 0.4032 - val_accuracy: 0.8380 - val_loss: 0.3791
Epoch 3/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 41s 40ms/step - accuracy: 0.8789 - loss: 0.3222 - val_accuracy: 0.8686 - val_loss: 0.3168
Epoch 4/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 25s 40ms/step - accuracy: 0.8991 - loss: 0.2825 - val_accuracy: 0.8546 - val_loss: 0.3673
Epoch 5/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 25s 40ms/step - accuracy: 0.9164 - loss: 0.2399 - val_accuracy: 0.8764 - val_loss: 0.3155
Epoch 6/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 25s 40ms/step - accuracy: 0.9240 - loss: 0.2215 - val_accuracy: 0.8818 - val_loss: 0.3310
Epoch 7/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 43s 44ms/step - accuracy: 0.9335 - loss: 0.1926 - val_accuracy: 0.8648 - val_loss: 0.3466
Epoch 8/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 39s 40ms/step - accuracy: 0.9351 - loss: 0.1794 - val_accuracy: 0.8830 - val_loss: 0.3555
Epoch 9/10
```

```
625/625 ━━━━━━━━━━━━━━━━━━━━ 41s 40ms/step - accuracy: 0.9514 - loss: 0.1496 - val_accuracy: 0.8726 - val_loss: 0.3604
Epoch 10/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 41s 40ms/step - accuracy: 0.9587 - loss: 0.1340 - val_accuracy: 0.8540 - val_loss: 0.4693
782/782 ━━━━━━━━━━━━━━━━━━━━ 14s 18ms/step - accuracy: 0.8451 - loss: 0.4961
Test acc (Trained Embedding): 0.847
```

```
# Plotting the training and validation accuracy
plt.figure(figsize=(10, 6))

# Plot training and validation accuracy
plt.subplot(1, 2, 1)
plt.plot(history_trained.history['accuracy'], label='Training Accuracy')
plt.plot(history_trained.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

# Plot training and validation loss
plt.subplot(1, 2, 2)
plt.plot(history_trained.history['loss'], label='Training Loss')
plt.plot(history_trained.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.tight_layout()
plt.show()
```



```
import requests
import zipfile
import os

# Download the file
url = "http://nlp.stanford.edu/data/glove.6B.zip"
output_path = "glove.6B.zip"

print("Downloading Glove embeddings...")
response = requests.get(url)
with open(output_path, "wb") as file:
    file.write(response.content)

# Extract the zip file
```

```
print("Extracting the zip file...")
with zipfile.ZipFile(output_path, 'r') as zip_ref:
    zip_ref.extractall(".")

# Cleanup: Optional
os.remove(output_path)
print("Download and extraction complete.")
```

```
Downloading Glove embeddings...
Extracting the zip file...
Download and extraction complete.
```

```
import numpy as np

# Path to the GloVe embeddings file
path_to_glove_file = "glove.6B.100d.txt"  # Update the path if needed

# Dictionary to store word embeddings
embeddings_index = {}

try:
    print("Loading GloVe embeddings...")
    with open(path_to_glove_file, encoding="utf-8") as f:
        for line in f:
            # Split each line into the word and its coefficients
            word, coefs = line.split(maxsplit=1)
            coefs = np.fromstring(coefs, "f", sep=" ")  # Convert coefficients to numpy array
            embeddings_index[word] = coefs  # Add to dictionary

    print(f"Found {len(embeddings_index)} word vectors.")
except FileNotFoundError:
    print(f"Error: File not found at {path_to_glove_file}. Please check the file path.")
except Exception as e:
    print(f"An error occurred: {e}")
```

```
Loading GloVe embeddings...
Found 400000 word vectors.
```

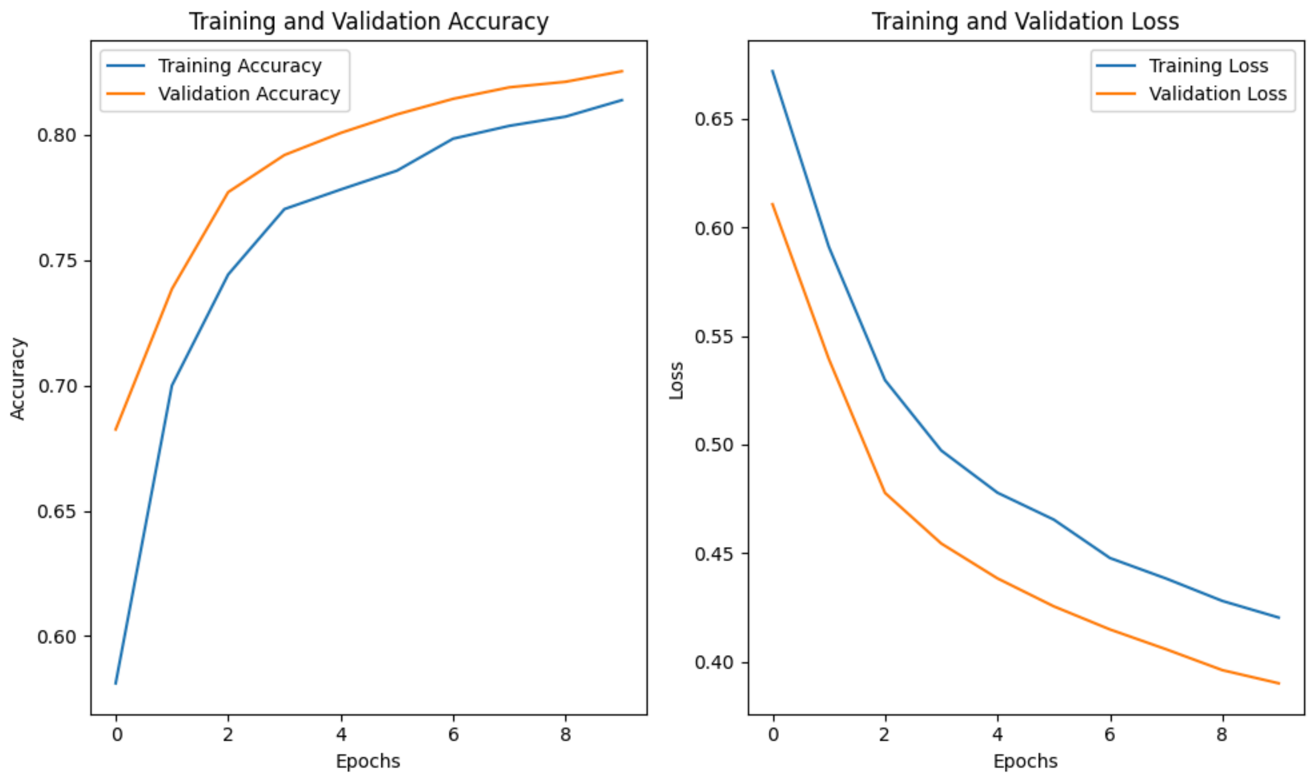
```
import tensorflow as tf
from tensorflow.keras import layers
from tensorflow import keras
import numpy as np

# Parameters
max_tokens = 20000  # Adjust according to your dataset/vocabulary size
embedding_dim = 100  # Dimension of GloVe embeddings
batch_size = 32  # Adjust batch size according to your system

# Step 1: Load GloVe embeddings
embeddings_index = {}
path_to_glove_file = "glove.6B.100d.txt"
with open(path_to_glove_file, encoding="utf-8") as f:
    for line in f:
        word, coefs = line.split(maxsplit=1)
        coefs = np.fromstring(coefs, "f", sep=" ")
        embeddings_index[word] = coefs
print(f"Found {len(embeddings_index)} word vectors.")

# Step 2: Load your dataset (using keras text_dataset_from_directory)
train_ds = keras.utils.text_dataset_from_directory(
    "aclImdb/train", batch_size=batch_size
)
val_ds = keras.utils.text_dataset_from_directory(
    "aclImdb/val", batch_size=batch_size
)
test_ds = keras.utils.text_dataset_from_directory(
    "aclImdb/test", batch_size=batch_size
)

# Step 3: Prepare the TextVectorization layer
# Extract texts from train_ds (text data)
train_texts = train_ds.map(lambda x, y: x)  # Extract only the texts
text_vectorization = layers.TextVectorization(max_tokens=max_tokens)
text_vectorization.adapt(train_texts)  # Adapt to your training data
```

```
# Prepare the embedding matrix
vocabulary = text_vectorization.get_vocabulary()  # Vocabulary from your dataset
word_index = dict(zip(vocabulary, range(len(vocabulary))))

embedding_matrix = np.zeros((max_tokens, embedding_dim))
for word, i in word_index.items():
    if i < max_tokens:
        embedding_vector = embeddings_index.get(word)
        if embedding_vector is not None:
            embedding_matrix[i] = embedding_vector

# Step 4: Define the embedding layer
embedding_layer_pretrained = layers.Embedding(
    input_dim=max_tokens,
    output_dim=embedding_dim,
    embeddings_initializer=keras.initializers.Constant(embedding_matrix),
    trainable=False,  # Freeze GloVe weights
    mask_zero=True,  # Optional: Mask padding tokens
)

# Step 5: Preprocess and prepare the datasets
def vectorize_text(text, label):
    return text_vectorization(text), label

train_ds = train_ds.map(vectorize_text).cache().prefetch(tf.data.AUTOTUNE)
val_ds = val_ds.map(vectorize_text).cache().prefetch(tf.data.AUTOTUNE)
test_ds = test_ds.map(vectorize_text).cache().prefetch(tf.data.AUTOTUNE)

# Step 6: Build the model
inputs = tf.keras.Input(shape=(None,), dtype="int64")
x = embedding_layer_pretrained(inputs)  # Use pretrained GloVe embeddings
x = layers.Bidirectional(layers.LSTM(32))(x)  # Bidirectional LSTM
x = layers.Dropout(0.5)(x)  # Dropout layer
outputs = layers.Dense(1, activation="sigmoid")(x)  # Output layer for binary classification

model_pretrained = tf.keras.Model(inputs, outputs)

# Step 7: Compile the model
model_pretrained.compile(
    optimizer=tf.keras.optimizers.RMSprop(learning_rate=1e-4),  # Adjust learning rate
    loss="binary_crossentropy",
    metrics=["accuracy"]
)

# Step 8: Train the model
history_pretrained = model_pretrained.fit(
    train_ds,  # Training dataset (tokenized and batched)
    validation_data=val_ds,  # Validation dataset
    epochs=10  # Number of epochs
)

# Step 9: Evaluate the model
test_loss, test_acc = model_pretrained.evaluate(test_ds)  # Test dataset
print(f"Test Accuracy (Pretrained Embedding): {test_acc:.3f}")
```

```
Found 400000 word vectors.
Found 25000 files belonging to 2 classes.
Found 5000 files belonging to 2 classes.
Found 25000 files belonging to 2 classes.
Epoch 1/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 41s 50ms/step - accuracy: 0.5381 - loss: 0.6944 - val_accuracy: 0.6824 - val_loss: 0.6105
Epoch 2/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 35s 44ms/step - accuracy: 0.6840 - loss: 0.6077 - val_accuracy: 0.7384 - val_loss: 0.5394
Epoch 3/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 37s 47ms/step - accuracy: 0.7304 - loss: 0.5501 - val_accuracy: 0.7770 - val_loss: 0.4778
Epoch 4/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 37s 48ms/step - accuracy: 0.7637 - loss: 0.5087 - val_accuracy: 0.7918 - val_loss: 0.4545
Epoch 5/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 39s 46ms/step - accuracy: 0.7707 - loss: 0.4873 - val_accuracy: 0.8006 - val_loss: 0.4385
Epoch 6/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 34s 44ms/step - accuracy: 0.7793 - loss: 0.4751 - val_accuracy: 0.8080 - val_loss: 0.4256
Epoch 7/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 36s 46ms/step - accuracy: 0.7923 - loss: 0.4576 - val_accuracy: 0.8142 - val_loss: 0.4150
Epoch 8/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 35s 45ms/step - accuracy: 0.7980 - loss: 0.4470 - val_accuracy: 0.8188 - val_loss: 0.4059
Epoch 9/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 35s 45ms/step - accuracy: 0.8023 - loss: 0.4365 - val_accuracy: 0.8210 - val_loss: 0.3963
```

```
Epoch 10/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 39s 50ms/step - accuracy: 0.8089 - loss: 0.4287 - val_accuracy: 0.8252 - val_loss: 0.3902
782/782 ━━━━━━━━━━━━━━━━━━━━ 17s 22ms/step - accuracy: 0.8129 - loss: 0.4138
Test Accuracy (Pretrained Embedding): 0.814
```

```python
import matplotlib.pyplot as plt

# Plotting the training and validation accuracy
plt.figure(figsize=(10, 6))

# Plot training and validation accuracy
plt.subplot(1, 2, 1)
plt.plot(history_pretrained.history['accuracy'], label='Training Accuracy')
plt.plot(history_pretrained.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

# Plot training and validation loss
plt.subplot(1, 2, 2)
plt.plot(history_pretrained.history['loss'], label='Training Loss')
plt.plot(history_pretrained.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.tight_layout()
plt.show()
```



Increasing Values

```python
# Define a list of different training sample sizes to test
training_sizes = [100, 500, 1000, 5000, len(int_train_ds)]  # Add the total number of training samples at the end

# Initialize variables to store test accuracy for each training size
test_accuracies = []

for size in training_sizes:
    # Take a subset of the training dataset based on the current size
    subset_train_ds = int_train_ds.take(size)
```

```
        # Define the model with the embedding layer (use either learned or pretrained)
        inputs = tf.keras.Input(shape=(None,), dtype="int64")
        x = embedding_layer_pretrained(inputs)  # Use pretrained embedding layer (GloVe) or embedding_layer_trained
        x = layers.Bidirectional(layers.LSTM(32))(x)
        x = layers.Dropout(0.5)(x)
        outputs = layers.Dense(1, activation="sigmoid")(x)

        model = tf.keras.Model(inputs, outputs)
        model.compile(optimizer="rmsprop", loss="binary_crossentropy", metrics=["accuracy"])

        # Train the model with the current subset of training data
        model.fit(subset_train_ds, validation_data=int_val_ds, epochs=10)

        # Evaluate the model on the test dataset and store the test accuracy
        test_accuracy = model.evaluate(int_test_ds)[1]
        test_accuracies.append(test_accuracy)

        # Print the test accuracy for the current training size
        print(f"Training samples: {size}, Test accuracy: {test_accuracy:.3f}")

    # Print out the test accuracies for all training sizes
    print("\nTest accuracies for different training sizes:")
    for size, accuracy in zip(training_sizes, test_accuracies):
        print(f"Training size: {size}, Test accuracy: {accuracy:.3f}")

    # Plotting the results
    plt.figure(figsize=(8, 6))
    plt.plot(training_sizes, test_accuracies, marker='o', linestyle='-', color='red')
    plt.title("Test Accuracy vs. Training Sample Size")
    plt.xlabel("Number of Training Samples")
    plt.ylabel("Test Accuracy")
    plt.grid(True)
    plt.xticks(training_sizes)  # Ensure all training sizes are shown on the x-axis
    plt.show()
```

```
Epoch 1/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 9s 77ms/step - accuracy: 0.5037 - loss: 0.7074 - val_accuracy: 0.5214 - val_loss: 0.6885
Epoch 2/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 6s 59ms/step - accuracy: 0.5400 - loss: 0.6846 - val_accuracy: 0.5612 - val_loss: 0.6788
Epoch 3/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 7s 70ms/step - accuracy: 0.5720 - loss: 0.6725 - val_accuracy: 0.5858 - val_loss: 0.6722
Epoch 4/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 6s 58ms/step - accuracy: 0.6078 - loss: 0.6566 - val_accuracy: 0.5334 - val_loss: 0.7125
Epoch 5/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 7s 74ms/step - accuracy: 0.6231 - loss: 0.6483 - val_accuracy: 0.6378 - val_loss: 0.6352
Epoch 6/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 6s 58ms/step - accuracy: 0.6212 - loss: 0.6408 - val_accuracy: 0.6406 - val_loss: 0.6353
Epoch 7/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 7s 70ms/step - accuracy: 0.6437 - loss: 0.6186 - val_accuracy: 0.6264 - val_loss: 0.6420
Epoch 8/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 6s 57ms/step - accuracy: 0.6765 - loss: 0.5946 - val_accuracy: 0.6506 - val_loss: 0.6234
Epoch 9/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 7s 71ms/step - accuracy: 0.6867 - loss: 0.5895 - val_accuracy: 0.6642 - val_loss: 0.6106
Epoch 10/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 6s 58ms/step - accuracy: 0.7119 - loss: 0.5715 - val_accuracy: 0.6236 - val_loss: 0.6679
782/782 ━━━━━━━━━━━━━━━━━━━━ 15s 19ms/step - accuracy: 0.6180 - loss: 0.6757
Training samples: 100, Test accuracy: 0.624
Epoch 1/10
500/500 ━━━━━━━━━━━━━━━━━━━━ 23s 43ms/step - accuracy: 0.5401 - loss: 0.6919 - val_accuracy: 0.6290 - val_loss: 0.6486
Epoch 2/10
500/500 ━━━━━━━━━━━━━━━━━━━━ 20s 40ms/step - accuracy: 0.6188 - loss: 0.6509 - val_accuracy: 0.5654 - val_loss: 0.6891
Epoch 3/10
500/500 ━━━━━━━━━━━━━━━━━━━━ 21s 42ms/step - accuracy: 0.6519 - loss: 0.6246 - val_accuracy: 0.7002 - val_loss: 0.5748
Epoch 4/10
500/500 ━━━━━━━━━━━━━━━━━━━━ 40s 40ms/step - accuracy: 0.6860 - loss: 0.5912 - val_accuracy: 0.7224 - val_loss: 0.5460
Epoch 5/10
500/500 ━━━━━━━━━━━━━━━━━━━━ 22s 44ms/step - accuracy: 0.7116 - loss: 0.5607 - val_accuracy: 0.6670 - val_loss: 0.6190
Epoch 6/10
500/500 ━━━━━━━━━━━━━━━━━━━━ 42s 47ms/step - accuracy: 0.7255 - loss: 0.5394 - val_accuracy: 0.7440 - val_loss: 0.5149
Epoch 7/10
500/500 ━━━━━━━━━━━━━━━━━━━━ 20s 41ms/step - accuracy: 0.7486 - loss: 0.5089 - val_accuracy: 0.7670 - val_loss: 0.4827
Epoch 8/10
500/500 ━━━━━━━━━━━━━━━━━━━━ 21s 42ms/step - accuracy: 0.7672 - loss: 0.4807 - val_accuracy: 0.7716 - val_loss: 0.4763
Epoch 9/10
500/500 ━━━━━━━━━━━━━━━━━━━━ 21s 42ms/step - accuracy: 0.7840 - loss: 0.4548 - val_accuracy: 0.7846 - val_loss: 0.4536
Epoch 10/10
500/500 ━━━━━━━━━━━━━━━━━━━━ 43s 45ms/step - accuracy: 0.7920 - loss: 0.4391 - val_accuracy: 0.7718 - val_loss: 0.4822
782/782 ━━━━━━━━━━━━━━━━━━━━ 14s 18ms/step - accuracy: 0.7738 - loss: 0.4822
Training samples: 500, Test accuracy: 0.772
Epoch 1/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 27s 40ms/step - accuracy: 0.5331 - loss: 0.6939 - val_accuracy: 0.6470 - val_loss: 0.6376
Epoch 2/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 25s 40ms/step - accuracy: 0.6268 - loss: 0.6433 - val_accuracy: 0.6644 - val_loss: 0.6078
Epoch 3/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 27s 43ms/step - accuracy: 0.6724 - loss: 0.6030 - val_accuracy: 0.7068 - val_loss: 0.5672
Epoch 4/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 25s 40ms/step - accuracy: 0.7065 - loss: 0.5660 - val_accuracy: 0.7272 - val_loss: 0.5447
Epoch 5/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 25s 40ms/step - accuracy: 0.7311 - loss: 0.5331 - val_accuracy: 0.7518 - val_loss: 0.5076
Epoch 6/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 41s 40ms/step - accuracy: 0.7550 - loss: 0.5012 - val_accuracy: 0.7728 - val_loss: 0.4764
Epoch 7/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 27s 43ms/step - accuracy: 0.7742 - loss: 0.4702 - val_accuracy: 0.7736 - val_loss: 0.4617
Epoch 8/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 27s 42ms/step - accuracy: 0.7863 - loss: 0.4455 - val_accuracy: 0.7902 - val_loss: 0.4429
Epoch 9/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 24s 39ms/step - accuracy: 0.8043 - loss: 0.4238 - val_accuracy: 0.7940 - val_loss: 0.4328
Epoch 10/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 43s 43ms/step - accuracy: 0.8136 - loss: 0.4094 - val_accuracy: 0.7928 - val_loss: 0.4289
782/782 ━━━━━━━━━━━━━━━━━━━━ 15s 19ms/step - accuracy: 0.7958 - loss: 0.4278
Training samples: 1000, Test accuracy: 0.796
Epoch 1/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 27s 41ms/step - accuracy: 0.5464 - loss: 0.6892 - val_accuracy: 0.6080 - val_loss: 0.6510
Epoch 2/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 25s 40ms/step - accuracy: 0.6339 - loss: 0.6375 - val_accuracy: 0.6684 - val_loss: 0.6061
Epoch 3/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 27s 43ms/step - accuracy: 0.6817 - loss: 0.5958 - val_accuracy: 0.7176 - val_loss: 0.5576
Epoch 4/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 25s 41ms/step - accuracy: 0.7061 - loss: 0.5599 - val_accuracy: 0.7406 - val_loss: 0.5233
Epoch 5/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 25s 40ms/step - accuracy: 0.7335 - loss: 0.5301 - val_accuracy: 0.7538 - val_loss: 0.5009
Epoch 6/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 43s 43ms/step - accuracy: 0.7505 - loss: 0.5016 - val_accuracy: 0.7676 - val_loss: 0.4808
```