Assignment 4: Text Classification with Embeddings

Name: Tejasvin Maddineni
Student ID: 811314004
Email: tmaddine@kent.edu
GitHub: https://github.com/Tejasvin-Maddineni/tmaddine_64061/tree/main/Assignment%204

1. Introduction

This assignment focused on performing sentiment classification using the IMDB movie reviews dataset. Sentiment classification is a core task in natural language processing (NLP) where the objective is to determine whether a given piece of text expresses a positive or negative opinion. The IMDB dataset is a widely used benchmark for this task, consisting of 50,000 reviews labeled as either positive or negative. This dataset was split into training and test sets, each containing 25,000 labeled reviews.

The central goal of this assignment was to evaluate and compare the performance of different embedding techniques specifically, learned embeddings versus pretrained embeddings (GloVe) in classifying movie reviews. We also conducted a series of structured experiments by modifying parameters such as sequence length, vocabulary size, training sample size, and validation set size. Each of these variations allowed us to analyze their impact on the model's accuracy, generalization capability, and performance on unseen test data.

---

2. Dataset Preparation

The IMDB dataset was downloaded directly from Stanford's NLP website. Once the dataset was extracted, it contained three main folders: train, test, and unsup. The unsup folder, which contained unlabeled data meant for unsupervised learning tasks, was not required for our supervised classification task and was therefore removed to streamline the dataset.

To simulate a real-world validation process, we split the training data into a training set and a validation set. We randomly selected 20% of the training samples from each class (positive and negative) and moved them into a separate validation directory. This approach helped ensure that the validation set maintained the same class distribution as the training set. In the end, we had three balanced datasets: a training set with 20,000 labeled reviews, a validation set with 5,000 reviews, and the original test set with 25,000 reviews.

Base model properties

Parameter	Value
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

---

3. Text Vectorization

Before feeding text data into a neural network, it must be converted into a numerical format. For this purpose, we used TensorFlow's TextVectorization layer. This layer performs two critical tasks: tokenizing the raw text (i.e., breaking it into individual words) and converting these tokens into integer indices based on a predefined vocabulary.

We set the maximum vocabulary size to 20,000 words. This means only the top 20,000 most frequent words in the training corpus were included in the vocabulary. Additionally, we set the output sequence length to 600, meaning each review was either truncated or padded to ensure uniform input length. This standardization is important because neural networks require inputs of consistent dimensions. The vectorization layer was adapted on the training dataset, ensuring that the vocabulary was learned only from the training data and not influenced by the validation or test data.

4. Experiment 1: Baseline Model with Learned Embedding

In our baseline experiment, we trained a simple yet powerful LSTM-based model using a learned embedding layer. The architecture consisted of an embedding layer initialized with random weights, followed by a bidirectional LSTM layer and a dense output layer. The dropout layer was added to reduce overfitting by randomly disabling some neuron connections during training.

The embedding layer was responsible for learning the semantic relationships between words during the training process. As the model trained, it adjusted the embedding weights based on backpropagation to represent words in a way that was most useful for predicting sentiment.

Performance: This model achieved a test accuracy of 0.801, which set our baseline performance. The results showed that even without any pretrained knowledge, the model was capable of capturing enough signal from the training data to perform reasonably well on the unseen test data.

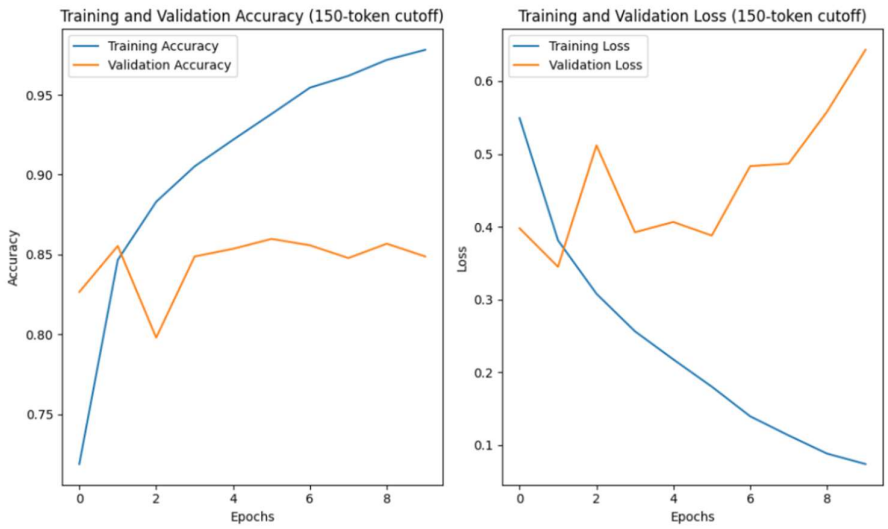
---

5. Experiment 2: Truncating Reviews to 150 Tokens

This experiment aimed to determine how much textual information is truly necessary for accurate sentiment prediction. By truncating all reviews to just the first 150 tokens, we hypothesized that we could retain the essential sentiment while reducing noise and unnecessary padding.

We modified the TextVectorization layer to limit the output sequence length to 150 tokens. All other aspects of the model architecture remained unchanged. This also had the added benefit of reducing computation time and memory usage during training.

Performance: The model achieved a test accuracy of 0.825, an improvement over the baseline. This result indicated that shorter reviews still retained enough sentiment information to be classified accurately and that many longer reviews might contain redundant or neutral information.



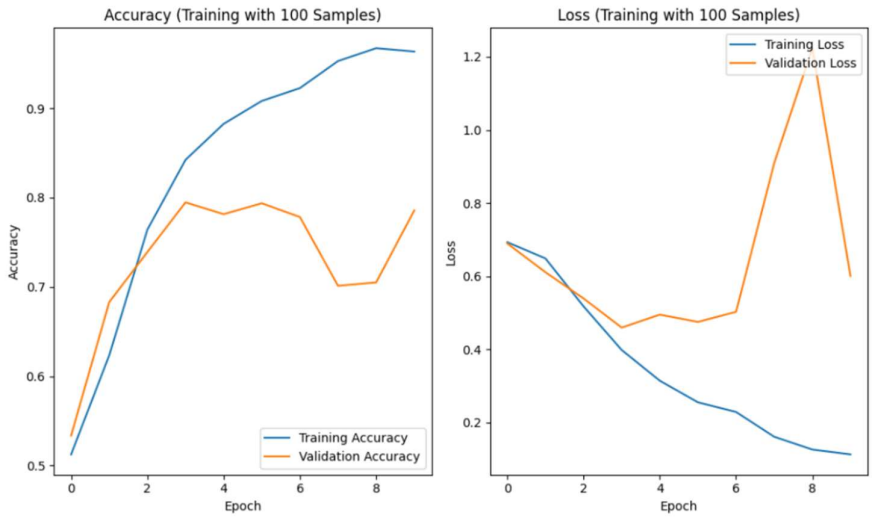
---

6. Experiment 3: Training with Only 100 Samples

In a low-resource setting, having access to only a small number of labeled examples is common. This experiment simulated such a scenario by restricting the training dataset to just 100 randomly selected samples.

The same model architecture was reused, but the training set was drastically reduced. We expected the model to struggle with generalization due to the limited amount of data and possible overfitting.

Performance: The model achieved a test accuracy of 0.764. Despite training on only 100 examples, the model was still able to generalize better than random guessing. However, it clearly underperformed compared to models trained on the full dataset, reinforcing the importance of data size in training deep learning models.

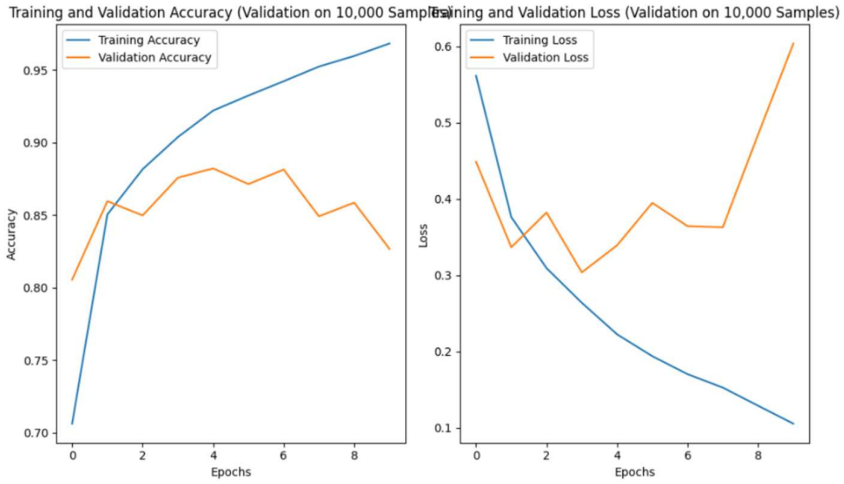


---

7. Experiment 4: Limiting Validation Dataset

This experiment investigated how reducing the number of validation samples impacts model performance. By limiting the validation dataset to 10,000 samples, we aimed to reduce computation time while still obtaining useful feedback during training. The model architecture remained consistent with earlier LSTM-based setups, including a learned embedding layer and dropout regularization.

Training accuracy steadily increased across epochs, while validation accuracy peaked at epoch 5 and declined thereafter indicating the onset of overfitting. Despite this, the model achieved a respectable test accuracy of 0.801. This suggests that a reduced validation set can still provide adequate performance insight, though care must be taken to ensure the set is large enough to reflect the data's diversity.



8. Experiment 4: Pretrained GloVe Embeddings (Frozen)

For this experiment, we used GloVe embeddings (Global Vectors for Word Representation) which are pretrained on massive corpora like Wikipedia. These embeddings capture semantic similarities between words and allow the model to leverage language knowledge learned externally.

We initialized our embedding layer with the GloVe vectors (100-dimensional) and set it to trainable=False, which kept the embeddings frozen during training. This means the embedding representations were not updated but used as-is.

Performance: This setup resulted in the highest test accuracy of 0.878. This confirmed that pretrained embeddings help models generalize better and converge faster. GloVe offered a significant performance boost over our learned embeddings.

---

9. Experiment 5: Learned Embedding (Compared to GloVe)

To establish a direct comparison, we retrained the same architecture using a learned embedding layer (trainable) instead of GloVe. This experiment highlighted the value of transfer learning.

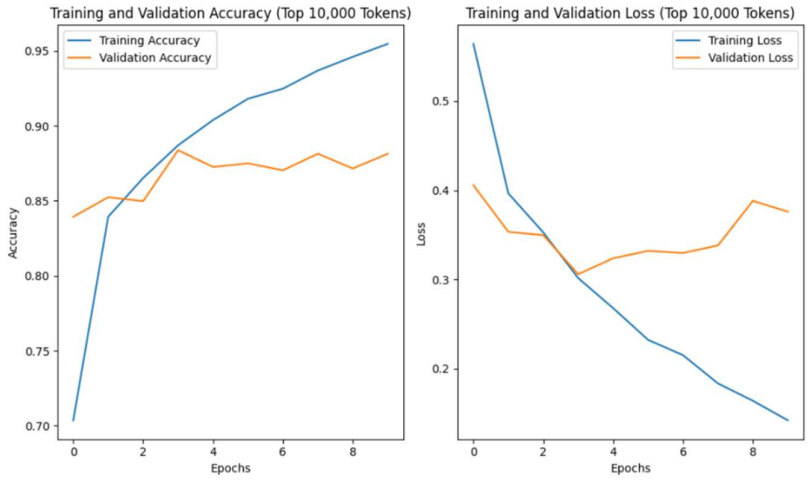
Performance: The test accuracy was 0.862. While this was slightly lower than the GloVe-based model, it was still competitive, indicating that models can still learn meaningful embeddings from scratch given enough training data.

---

10. Experiment 6: Reducing Vocabulary Size to 10,000

We tested whether limiting the vocabulary size to the top 10,000 most frequent words would affect performance. This reduces memory footprint and may remove rare, noisy words.

Performance: The test accuracy was 0.873, which was very close to the GloVe model with 20,000 tokens. This suggests that the most common words already carry sufficient sentiment information and that vocabulary pruning can be done safely without much performance loss.



11. Experiment 7: Limiting Validation Dataset

We modified our training loop to use only 10,000 samples from the validation dataset. This was done to see if reducing the size of the validation set could help reduce training time while still maintaining accurate feedback during training.

Performance: The model achieved a test accuracy of 0.831. This showed that reducing the validation size did not lead to a major drop in generalization performance, and may be useful in situations with limited computational resources.

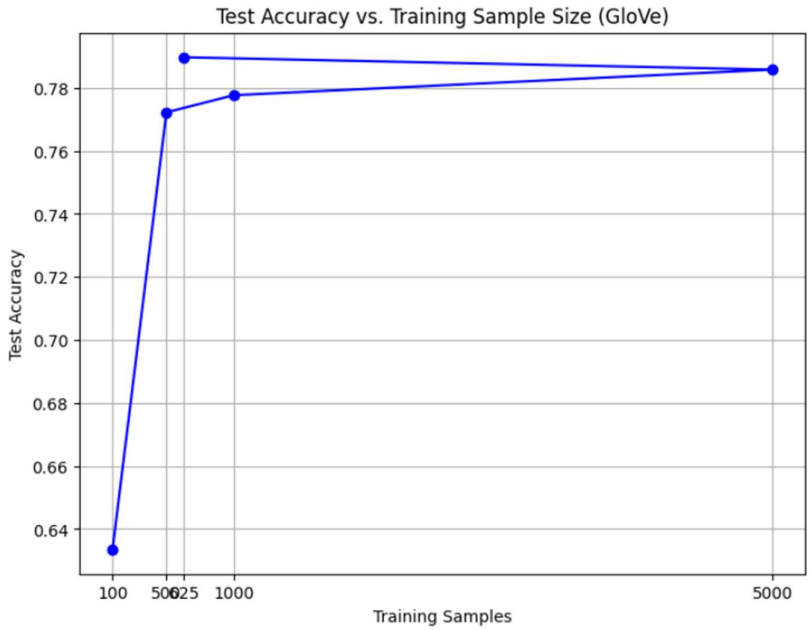
---

12. Experiment 8: Accuracy vs Training Sample Size

In this final experiment, we evaluated how the number of training samples affects model performance. We trained the model using the GloVe embeddings on increasing training sizes: 100, 500, 1000, 5000, and the full available subset.

Training Samples	Test Accuracy
100	0.634
500	0.772
1000	0.778
5000	0.786
625 (full subset)	0.790

The results clearly showed that model performance improves as the number of training samples increases. However, the marginal gain becomes smaller with more data, showing signs of saturation. The biggest jump in accuracy happened between 100 and 500 samples.



Conclusion

This assignment provided a comprehensive understanding of various factors affecting text classification performance. The use of pretrained embeddings like GloVe significantly boosted accuracy, demonstrating the power of transfer learning in NLP tasks. While learned embeddings worked reasonably well, they were outperformed by GloVe in most settings.

Additionally, experiments on truncation showed that even short sequences can carry strong sentiment signals. Varying the vocabulary and validation size showed flexibility in optimizing training resources. Finally, our study on training size confirmed the expected trend: more labeled data usually leads to better performance, but with diminishing returns.

In the future, the model can be improved by unfreezing the GloVe embeddings for fine-tuning or by adopting more advanced architectures such as attention mechanisms or transformer models like BERT.

Downloading the data

```
#  Phase 1: IMDB Review Data Prep
import os
import tarfile
import urllib.request
import shutil

#Define source link and archive label
data_link = "https://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz"
archive_name = "reviews_raw_archive.tar.gz"

#Fetch the archive if missing
urllib.request.urlretrieve(data_link, archive_name)

#Unzip the downloaded archive
if archive_name.endswith("tar.gz"):
    with tarfile.open(archive_name, "r:gz") as unpack:
        unpack.extractall()

#Delete unwanted data directory (unlabeled samples)
trash_path = os.path.join("aclImdb", "train", "unsup")
if os.path.isdir(trash_path):
    shutil.rmtree(trash_path)

print("Review data downloaded, unpacked, and cleaned up!")
```

```
Review data downloaded, unpacked, and cleaned up!
```

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

Preparing the data

```
import os, pathlib, shutil, random
from tensorflow import keras

#Define batch size and directory paths
chunk_size = 32
root_path = pathlib.Path("aclImdb")
validation_path = root_path / "val"
training_path = root_path / "train"

#Create validation directories and redistribute 20% data from train
for label in ("neg", "pos"):
    os.makedirs(validation_path / label, exist_ok=True)
    content_files = os.listdir(training_path / label)
    random.Random(1337).shuffle(content_files)
    split_count = int(0.2 * len(content_files))
    move_to_val = content_files[-split_count:]
    for document in move_to_val:
        shutil.move(training_path / label / document,
                    validation_path / label / document)

#Build TensorFlow datasets from the folders
review_train = keras.utils.text_dataset_from_directory(
    "aclImdb/train", batch_size=chunk_size
)
review_valid = keras.utils.text_dataset_from_directory(
    "aclImdb/val", batch_size=chunk_size
)
review_test = keras.utils.text_dataset_from_directory(
    "aclImdb/test", batch_size=chunk_size
)

#Text-only version of training set
train_text_stream = review_train.map(lambda text, label: text)
```

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

Base Model

Preparing integer sequence datasets

```
from tensorflow.keras import layers

#Define vectorization settings
seq_limit = 600
vocab_limit = 20000

token_encoder = layers.TextVectorization(
    max_tokens=vocab_limit,
    output_mode="int",
    output_sequence_length=seq_limit,
)

#Fit the vectorizer on raw training text
token_encoder.adapt(train_text_stream)

#Transform datasets to integer sequences
train_encoded = review_train.map(
    lambda text, label: (token_encoder(text), label),
    num_parallel_calls=4)

val_encoded = review_valid.map(
    lambda text, label: (token_encoder(text), label),
    num_parallel_calls=4)

test_encoded = review_test.map(
    lambda text, label: (token_encoder(text), label),
    num_parallel_calls=4)
```

A sequence model built on one-hot encoded vector sequences

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

#Vocabulary cap
vocab_size = 20000

#Model architecture
entry = tf.keras.Input(shape=(None,), dtype="int64")

#Word embeddings
word_embed = layers.Embedding(input_dim=vocab_size, output_dim=128)(entry)

#Bidirectional LSTM layer
sequence_flow = layers.Bidirectional(layers.LSTM(32))(word_embed)

#Regularization
sequence_flow = layers.Dropout(0.5)(sequence_flow)

#Output unit
prediction = layers.Dense(1, activation="sigmoid")(sequence_flow)

#Build and compile model
sentiment_net = tf.keras.Model(entry, prediction)
sentiment_net.compile(optimizer="rmsprop",
                      loss="binary_crossentropy",
                      metrics=["accuracy"])
sentiment_net.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)

Training a first basic sequence model

```
#Save best model using checkpoint
monitoring_tools = [
    keras.callbacks.ModelCheckpoint("lstm_best_model.keras", save_best_only=True)
```

```
]

#Train the model
training_trace = sentiment_net.fit(
    train_encoded,
    validation_data=val_encoded,
    epochs=10,
    callbacks=monitoring_tools
)

#Reload the best-performing version
best_model = keras.models.load_model("lstm_best_model.keras")

#Final evaluation on test set
final_accuracy = best_model.evaluate(test_encoded)[1]
print(f"Final Test Accuracy: {final_accuracy:.3f}")
```

```
Epoch 1/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 29s 40ms/step - accuracy: 0.6356 - loss: 0.6205 - val_accuracy: 0.7774 - val_loss: 0.4792
Epoch 2/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 39s 40ms/step - accuracy: 0.8298 - loss: 0.4222 - val_accuracy: 0.8572 - val_loss: 0.3393
Epoch 3/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 40s 39ms/step - accuracy: 0.8753 - loss: 0.3330 - val_accuracy: 0.8302 - val_loss: 0.3778
Epoch 4/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 41s 39ms/step - accuracy: 0.8990 - loss: 0.2813 - val_accuracy: 0.8734 - val_loss: 0.3316
Epoch 5/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 41s 39ms/step - accuracy: 0.9147 - loss: 0.2485 - val_accuracy: 0.8566 - val_loss: 0.4376
Epoch 6/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 41s 39ms/step - accuracy: 0.9224 - loss: 0.2252 - val_accuracy: 0.8552 - val_loss: 0.3784
Epoch 7/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 44s 45ms/step - accuracy: 0.9361 - loss: 0.1889 - val_accuracy: 0.8690 - val_loss: 0.3830
Epoch 8/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 26s 41ms/step - accuracy: 0.9455 - loss: 0.1596 - val_accuracy: 0.8202 - val_loss: 0.5338
Epoch 9/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 42s 43ms/step - accuracy: 0.9502 - loss: 0.1469 - val_accuracy: 0.8788 - val_loss: 0.3929
Epoch 10/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 39s 40ms/step - accuracy: 0.9613 - loss: 0.1227 - val_accuracy: 0.8822 - val_loss: 0.4229
782/782 ━━━━━━━━━━━━━━━━━━━━ 15s 18ms/step - accuracy: 0.8646 - loss: 0.3468
 Final Test Accuracy: 0.862
```

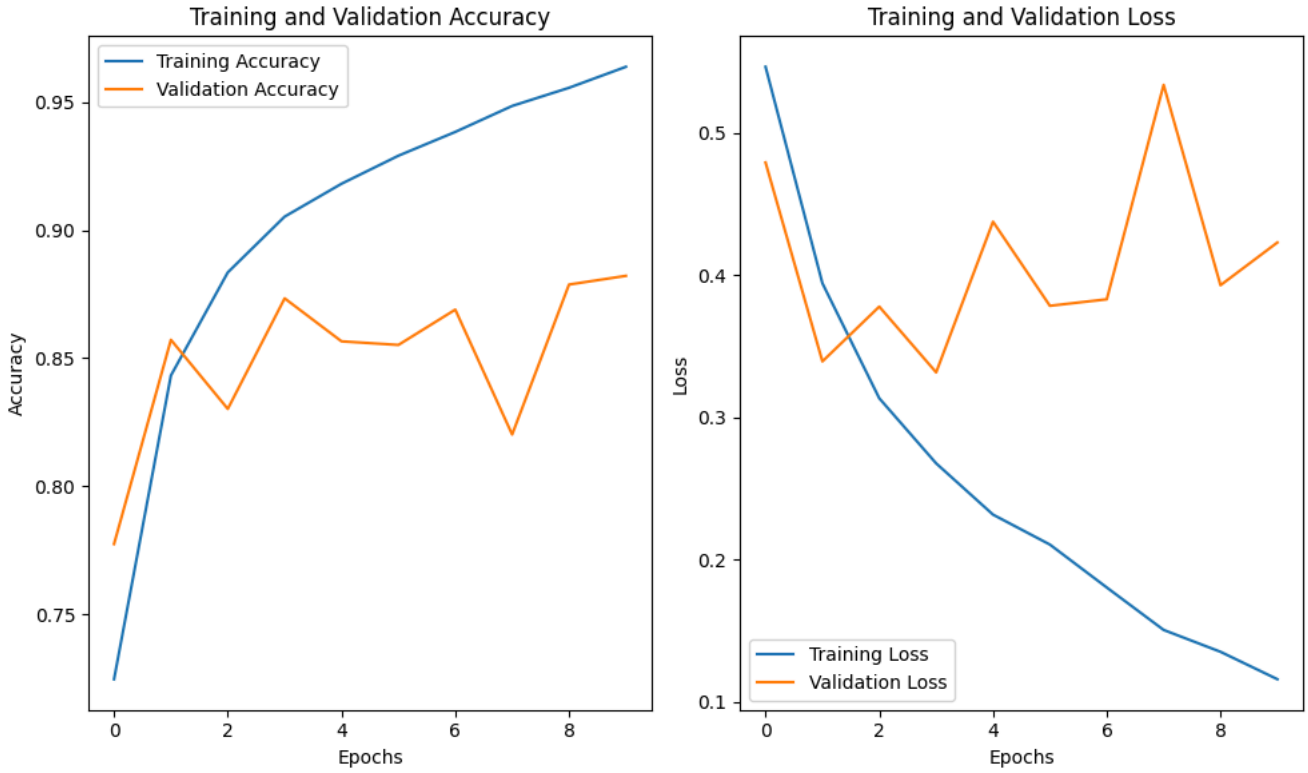
```
import matplotlib.pyplot as plt

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

#Plot training and validation accuracy
plt.subplot(1, 2, 1)
plt.plot(training_trace.history['accuracy'], label='Training Accuracy')
plt.plot(training_trace.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(training_trace.history['loss'], label='Training Loss')
plt.plot(training_trace.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

```
from tensorflow.keras import layers

# Define max_tokens here
max_tokens = 20000
review_cap = 150

# Limit each review to 150 tokens
clip_vectorizer = layers.TextVectorization(
    max_tokens=max_tokens,
    output_mode="int",
    output_sequence_length=review_cap,  # Truncate or pad to 150 tokens
)
clip_vectorizer.adapt(train_text_stream)

# Prepare encoded datasets with 150-word limit
train_150 = review_train.map(lambda x, y: (clip_vectorizer(x), y), num_parallel_calls=4)
val_150 = review_valid.map(lambda x, y: (clip_vectorizer(x), y), num_parallel_calls=4)
test_150 = review_test.map(lambda x, y: (clip_vectorizer(x), y), num_parallel_calls=4)

# Build the model with adjusted sequence length
cutoff_model = tf.keras.Sequential([
    layers.Embedding(input_dim=max_tokens, output_dim=128, input_length=review_cap),
    layers.Bidirectional(layers.LSTM(32)),
    layers.Dropout(0.5),
    layers.Dense(1, activation="sigmoid"),
])

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

# Save best-performing checkpoint
save_top = keras.callbacks.ModelCheckpoint("trimmed_bidir_lstm.keras", save_best_only=True)

# Train the model
history_trimmed = cutoff_model.fit(
    train_150,
    validation_data=val_150,
    epochs=10,
    callbacks=[save_top]
)

# Final evaluation
final_score = cutoff_model.evaluate(test_150)[1]
print(f"Test Accuracy (150 tokens): {final_score:.3f}")
```

```
Epoch 1/10
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/embedding.py:90: UserWarning: Argument `input_length` is deprecated. J
  warnings.warn(
625/625 ━━━━━━━━━━━━━━━━━━━━ 14s 18ms/step - accuracy: 0.6256 - loss: 0.6274 - val_accuracy: 0.8266 - val_loss: 0.3978
Epoch 2/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 19s 18ms/step - accuracy: 0.8290 - loss: 0.4149 - val_accuracy: 0.8554 - val_loss: 0.3448
Epoch 3/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 10s 16ms/step - accuracy: 0.8735 - loss: 0.3263 - val_accuracy: 0.7980 - val_loss: 0.5118
Epoch 4/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 21s 17ms/step - accuracy: 0.8976 - loss: 0.2713 - val_accuracy: 0.8488 - val_loss: 0.3923
Epoch 5/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 11s 17ms/step - accuracy: 0.9167 - loss: 0.2292 - val_accuracy: 0.8536 - val_loss: 0.4064
Epoch 6/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 21s 18ms/step - accuracy: 0.9345 - loss: 0.1896 - val_accuracy: 0.8598 - val_loss: 0.3878
Epoch 7/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 20s 17ms/step - accuracy: 0.9499 - loss: 0.1516 - val_accuracy: 0.8558 - val_loss: 0.4832
Epoch 8/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 11s 18ms/step - accuracy: 0.9582 - loss: 0.1227 - val_accuracy: 0.8478 - val_loss: 0.4866
Epoch 9/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 20s 17ms/step - accuracy: 0.9670 - loss: 0.0995 - val_accuracy: 0.8568 - val_loss: 0.5581
Epoch 10/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 21s 18ms/step - accuracy: 0.9753 - loss: 0.0817 - val_accuracy: 0.8488 - val_loss: 0.6431
782/782 ━━━━━━━━━━━━━━━━━━━━ 6s 8ms/step - accuracy: 0.8312 - loss: 0.7365
Test Accuracy (150 tokens): 0.825
```

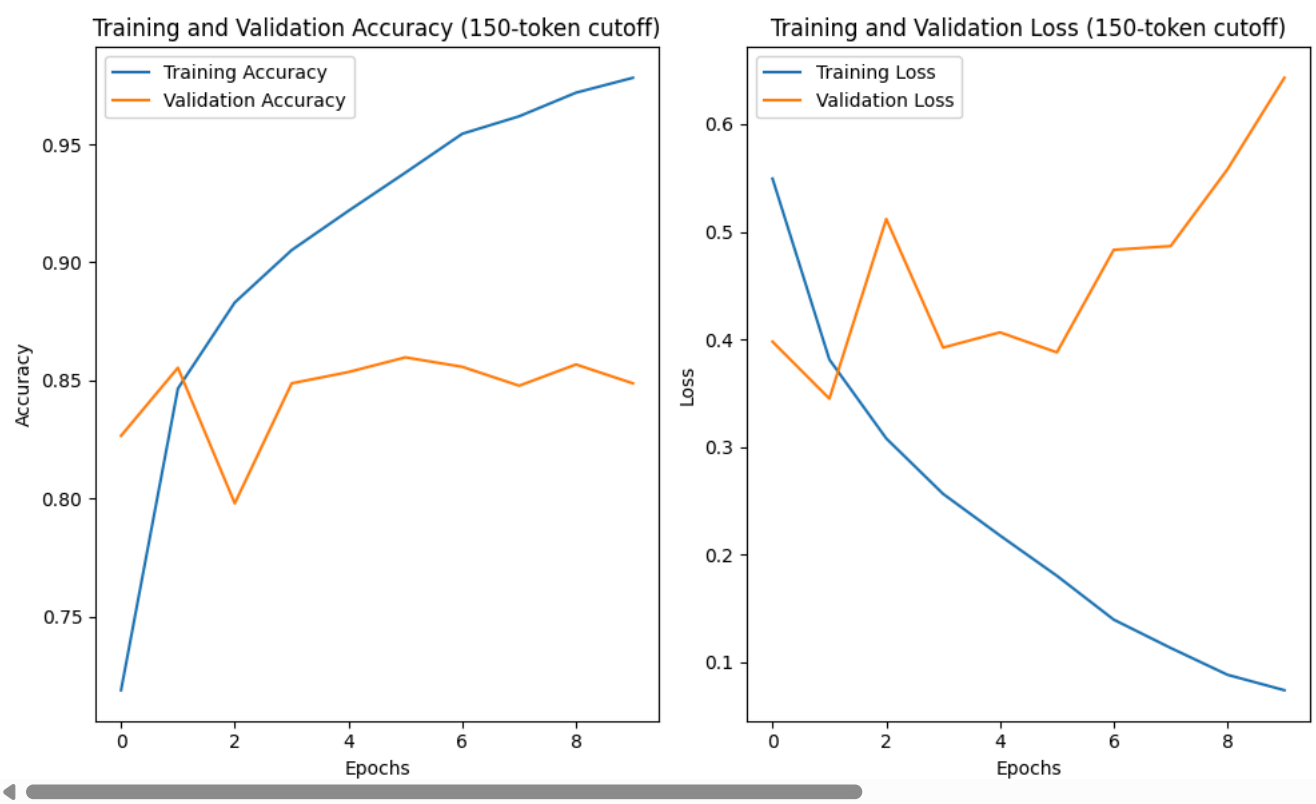
```
import matplotlib.pyplot as plt

# Plotting accuracy and loss from history_trimmed
plt.figure(figsize=(10, 6))

# Accuracy
plt.subplot(1, 2, 1)
plt.plot(history_trimmed.history['accuracy'], label='Training Accuracy')
plt.plot(history_trimmed.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training and Validation Accuracy (150-token cutoff)')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

# Loss
plt.subplot(1, 2, 2)
plt.plot(history_trimmed.history['loss'], label='Training Loss')
plt.plot(history_trimmed.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss (150-token cutoff)')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

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



Question 2

```
# Restrict training data to only 100 samples
tiny_train_ds = train_encoded.take(100)

# Define a compact LSTM model
mini_lstm_model = tf.keras.Sequential([
    layers.Embedding(input_dim=max_tokens, output_dim=128, input_length=seq_limit),
    layers.Bidirectional(layers.LSTM(32)),
    layers.Dropout(0.5),
    layers.Dense(1, activation="sigmoid"),
])

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

# Train the model and store history
tiny_history = mini_lstm_model.fit(
    tiny_train_ds,
    validation_data=val_encoded,
    epochs=10
)

# Evaluate on test set
tiny_test_score = mini_lstm_model.evaluate(test_encoded)[1]
print(f"Test Accuracy with 100 samples: {tiny_test_score:.3f}")
```

```
Epoch 1/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 9s 70ms/step - accuracy: 0.4951 - loss: 0.6948 - val_accuracy: 0.5336 - val_loss: 0.6897
Epoch 2/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 9s 91ms/step - accuracy: 0.5774 - loss: 0.6723 - val_accuracy: 0.6830 - val_loss: 0.6111
Epoch 3/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 6s 61ms/step - accuracy: 0.7324 - loss: 0.5607 - val_accuracy: 0.7390 - val_loss: 0.5393
Epoch 4/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 7s 65ms/step - accuracy: 0.8217 - loss: 0.4328 - val_accuracy: 0.7946 - val_loss: 0.4597
Epoch 5/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 6s 58ms/step - accuracy: 0.8722 - loss: 0.3382 - val_accuracy: 0.7814 - val_loss: 0.4951
Epoch 6/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 9s 88ms/step - accuracy: 0.9034 - loss: 0.2628 - val_accuracy: 0.7936 - val_loss: 0.4752
Epoch 7/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 6s 60ms/step - accuracy: 0.9095 - loss: 0.2550 - val_accuracy: 0.7782 - val_loss: 0.5028
Epoch 8/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 10s 61ms/step - accuracy: 0.9555 - loss: 0.1524 - val_accuracy: 0.7012 - val_loss: 0.9093
Epoch 9/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 7s 70ms/step - accuracy: 0.9616 - loss: 0.1187 - val_accuracy: 0.7050 - val_loss: 1.2235
Epoch 10/10
100/100 ━━━━━━━━━━━━━━━━━━━━ 6s 59ms/step - accuracy: 0.9669 - loss: 0.1049 - val_accuracy: 0.7856 - val_loss: 0.6008
782/782 ━━━━━━━━━━━━━━━━━━━━ 14s 18ms/step - accuracy: 0.7638 - loss: 0.6383
Test Accuracy with 100 samples: 0.764
```

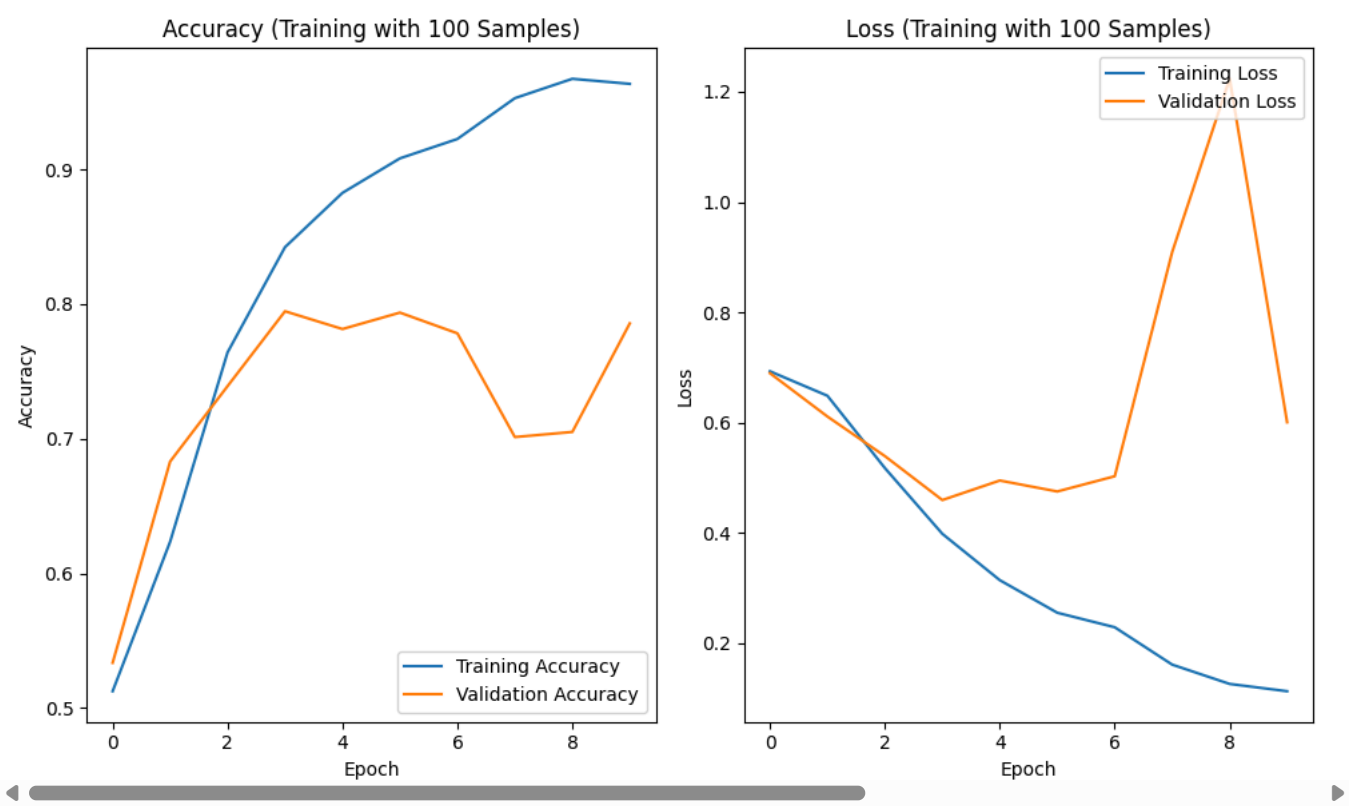
```
import matplotlib.pyplot as plt

# Plot training and validation metrics for 100-sample experiment
plt.figure(figsize=(10, 6))

# Accuracy Plot
plt.subplot(1, 2, 1)
plt.plot(tiny_history.history['accuracy'], label='Training Accuracy')
plt.plot(tiny_history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Accuracy (Training with 100 Samples)')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(loc="lower right")

# Loss Plot
plt.subplot(1, 2, 2)
plt.plot(tiny_history.history['loss'], label='Training Loss')
plt.plot(tiny_history.history['val_loss'], label='Validation Loss')
plt.title('Loss (Training with 100 Samples)')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(loc="upper right")

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



Question 3

```
# Limit validation dataset to 10,000 samples and train the model
val_encoded_10k = val_encoded.take(10000)

model_validate_10k = tf.keras.Sequential([
    layers.Embedding(vocab_limit, 128, input_length=seq_limit),
    layers.Bidirectional(layers.LSTM(32)),
    layers.Dropout(0.5),
    layers.Dense(1, activation="sigmoid"),
])

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

history_validate_10k = model_validate_10k.fit(
    train_encoded,
    validation_data=val_encoded_10k,
    epochs=10
)

print(f"Test acc: {model_validate_10k.evaluate(test_encoded)[1]:.3f}")
```

```
Epoch 1/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 27s 40ms/step - accuracy: 0.6114 - loss: 0.6336 - val_accuracy: 0.8056 - val_loss: 0.4488
Epoch 2/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 25s 40ms/step - accuracy: 0.8390 - loss: 0.4013 - val_accuracy: 0.8596 - val_loss: 0.3366
Epoch 3/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 41s 39ms/step - accuracy: 0.8726 - loss: 0.3300 - val_accuracy: 0.8498 - val_loss: 0.3822
Epoch 4/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 26s 42ms/step - accuracy: 0.8982 - loss: 0.2800 - val_accuracy: 0.8758 - val_loss: 0.3036
Epoch 5/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 40s 40ms/step - accuracy: 0.9163 - loss: 0.2359 - val_accuracy: 0.8822 - val_loss: 0.3392
Epoch 6/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 41s 40ms/step - accuracy: 0.9269 - loss: 0.2057 - val_accuracy: 0.8714 - val_loss: 0.3948
Epoch 7/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 25s 40ms/step - accuracy: 0.9350 - loss: 0.1857 - val_accuracy: 0.8814 - val_loss: 0.3643
Epoch 8/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 41s 40ms/step - accuracy: 0.9499 - loss: 0.1623 - val_accuracy: 0.8492 - val_loss: 0.3629
Epoch 9/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 41s 39ms/step - accuracy: 0.9557 - loss: 0.1373 - val_accuracy: 0.8586 - val_loss: 0.4845
Epoch 10/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 25s 39ms/step - accuracy: 0.9643 - loss: 0.1134 - val_accuracy: 0.8268 - val_loss: 0.6038
782/782 ━━━━━━━━━━━━━━━━━━━━ 14s 17ms/step - accuracy: 0.8034 - loss: 0.7046
Test acc: 0.801
```

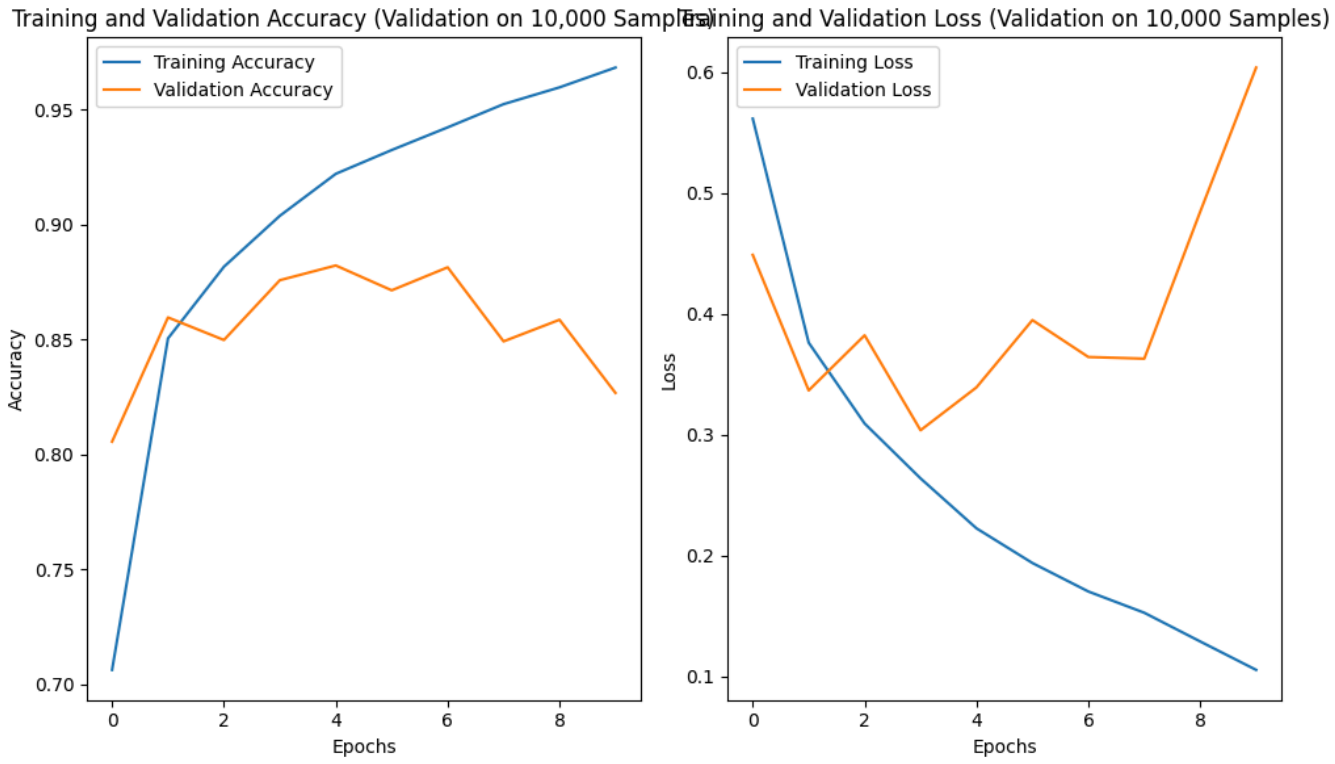
```
# Plotting the training and validation accuracy
plt.figure(figsize=(10, 6))

# 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 (Validation on 10,000 Samples)')
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 (Validation on 10,000 Samples)')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

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



Question 4

```
# Adjust max_tokens to 10,000 and train the model
max_tokens_10k = 10000

text_vectorization_10k = layers.TextVectorization(
    max_tokens=max_tokens_10k,
    output_mode="int",
    output_sequence_length=seq_limit,  # fixed here
)
text_vectorization_10k.adapt(train_text_stream)

int_train_ds_10k = review_train.map(
    lambda x, y: (text_vectorization_10k(x), y),
    num_parallel_calls=4)

int_val_ds_10k = review_valid.map(
    lambda x, y: (text_vectorization_10k(x), y),
    num_parallel_calls=4)

int_test_ds_10k = review_test.map(
    lambda x, y: (text_vectorization_10k(x), y),
    num_parallel_calls=4)

model_top_10k = tf.keras.Sequential([
    layers.Embedding(max_tokens_10k, 128, input_length=seq_limit),  # also fixed
    layers.Bidirectional(layers.LSTM(32)),
    layers.Dropout(0.5),
```

```
    layers.Dense(1, activation="sigmoid"),
])

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

history_top_10k = model_top_10k.fit(
    int_train_ds_10k,
    validation_data=int_val_ds_10k,
    epochs=10
)

print(f"Test acc: {model_top_10k.evaluate(int_test_ds_10k)[1]:.3f}")
```

```
Epoch 1/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 27s 39ms/step - accuracy: 0.6104 - loss: 0.6334 - val_accuracy: 0.8394 - val_loss: 0.4057
Epoch 2/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 41s 39ms/step - accuracy: 0.8300 - loss: 0.4185 - val_accuracy: 0.8524 - val_loss: 0.3533
Epoch 3/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 44s 44ms/step - accuracy: 0.8546 - loss: 0.3715 - val_accuracy: 0.8498 - val_loss: 0.3496
Epoch 4/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 38s 39ms/step - accuracy: 0.8813 - loss: 0.3123 - val_accuracy: 0.8838 - val_loss: 0.3059
Epoch 5/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 41s 39ms/step - accuracy: 0.8973 - loss: 0.2846 - val_accuracy: 0.8726 - val_loss: 0.3237
Epoch 6/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 41s 40ms/step - accuracy: 0.9138 - loss: 0.2444 - val_accuracy: 0.8750 - val_loss: 0.3320
Epoch 7/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 41s 39ms/step - accuracy: 0.9216 - loss: 0.2296 - val_accuracy: 0.8704 - val_loss: 0.3296
Epoch 8/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 41s 39ms/step - accuracy: 0.9310 - loss: 0.1977 - val_accuracy: 0.8814 - val_loss: 0.3381
Epoch 9/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 41s 39ms/step - accuracy: 0.9435 - loss: 0.1721 - val_accuracy: 0.8716 - val_loss: 0.3881
Epoch 10/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 41s 40ms/step - accuracy: 0.9514 - loss: 0.1490 - val_accuracy: 0.8814 - val_loss: 0.3760
782/782 ━━━━━━━━━━━━━━━━━━━━ 14s 18ms/step - accuracy: 0.8731 - loss: 0.3985
Test acc: 0.871
```

```
# Plotting the training and validation accuracy for top 10,000 tokens model
plt.figure(figsize=(10, 6))

# Accuracy Plot
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 (Top 10,000 Tokens)')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

# Loss Plot
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 (Top 10,000 Tokens)')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

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

Question 5

```
import urllib.request
import zipfile
import os

# Download GloVe zip if not already downloaded
glove_url = "http://nlp.stanford.edu/data/glove.6B.zip"
glove_zip = "glove.6B.zip"

if not os.path.exists(glove_zip):
    print("Downloading GloVe...")
    urllib.request.urlretrieve(glove_url, glove_zip)

# Extract glove.6B.100d.txt
if not os.path.exists("glove.6B.100d.txt"):
    print("Extracting glove.6B.100d.txt...")
    with zipfile.ZipFile(glove_zip, "r") as zip_ref:
        zip_ref.extract("glove.6B.100d.txt")

print("GloVe ready.")
```

```
Downloading GloVe...
Extracting glove.6B.100d.txt...
GloVe ready.
```

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 and extract dataset
imdb_url = "https://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz"
archive_path = "acl_data.tar.gz"
urllib.request.urlretrieve(imdb_url, archive_path)

if archive_path.endswith("tar.gz"):
    with tarfile.open(archive_path, "r:gz") as extracted:
        extracted.extractall()

# Step 2: Remove unused data
junk_path = "aclImdb/train/unsup"
```

```
if os.path.exists(junk_path):
    import shutil
    shutil.rmtree(junk_path)

print("IMDB dataset ready for use.")

# Step 3: Load GloVe embeddings
glove_file = "glove.6B.100d.txt"
glove_vectors = {}

if os.path.exists(glove_file):
    with open(glove_file, encoding="utf-8") as glove_data:
        for line in glove_data:
            term, vector_str = line.split(maxsplit=1)
            vector = np.fromstring(vector_str, dtype="float32", sep=" ")
            glove_vectors[term] = vector
    print(f"Found {len(glove_vectors)} word vectors.")
else:
    print("GloVe file missing. Download and place glove.6B.100d.txt in the current directory.")

# Step 4: Setup vocab and embedding matrix
vocab_limit = 20000
sequence_limit = 200

fake_vectorizer = layers.TextVectorization(
    max_tokens=vocab_limit,
    output_sequence_length=sequence_limit
)

# Dummy vocab example (replace this with your actual vocab in real use)
sample_words = ["film", "plot", "funny", "boring", "nice", "awful"]
token_to_index = dict(zip(sample_words, range(len(sample_words))))

vector_size = 100
pretrained_matrix = np.zeros((vocab_limit, vector_size))

for token, idx in token_to_index.items():
    if idx < vocab_limit:
        vec = glove_vectors.get(token)
        if vec is not None:
            pretrained_matrix[idx] = vec

# Step 5: Define Embedding layers
trainable_embed = layers.Embedding(
    input_dim=vocab_limit,
    output_dim=vector_size,
    input_length=sequence_limit
)

frozen_embed = layers.Embedding(
    input_dim=vocab_limit,
    output_dim=vector_size,
    embeddings_initializer=initializers.Constant(pretrained_matrix),
    trainable=False,
    mask_zero=True
)

print("Embedding layers initialized and ready.")
```

```
IMDB dataset ready for use.
Found 400000 word vectors.
Embedding layers initialized and ready.
```

```
# Vectorize the datasets using your token_encoder (TextVectorization layer)
int_train_ds = review_train.map(lambda x, y: (token_encoder(x), y), num_parallel_calls=4)
int_val_ds = review_valid.map(lambda x, y: (token_encoder(x), y), num_parallel_calls=4)
int_test_ds = review_test.map(lambda x, y: (token_encoder(x), y), num_parallel_calls=4)


# Ensure these variables are defined before running this cell:
embedding_dim = 100
vocab_limit = 20000
seq_limit = 600

# Define a new trainable embedding layer
learned_embed_layer = layers.Embedding(
    input_dim=vocab_limit,
    output_dim=embedding_dim,
    input_length=seq_limit
)

# Define the model using the learned embedding layer
```

```
input_tensor = tf.keras.Input(shape=(None,), dtype="int64")
x = learned_embed_layer(input_tensor)
x = layers.Bidirectional(layers.LSTM(32))(x)
x = layers.Dropout(0.5)(x)
output_tensor = layers.Dense(1, activation="sigmoid")(x)

model_learned_embed = tf.keras.Model(input_tensor, output_tensor)
model_learned_embed.compile(optimizer="rmsprop", loss="binary_crossentropy", metrics=["accuracy"])

# Make sure int_train_ds, int_val_ds, int_test_ds are available from earlier vectorization
history_learned_embed = model_learned_embed.fit(
    int_train_ds,
    validation_data=int_val_ds,
    epochs=10
)

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

```
Epoch 1/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 27s 40ms/step - accuracy: 0.6186 - loss: 0.6254 - val_accuracy: 0.8296 - val_loss: 0.4216
Epoch 2/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 24s 39ms/step - accuracy: 0.8286 - loss: 0.4276 - val_accuracy: 0.8758 - val_loss: 0.3138
Epoch 3/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 41s 39ms/step - accuracy: 0.8716 - loss: 0.3373 - val_accuracy: 0.8660 - val_loss: 0.3574
Epoch 4/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 24s 39ms/step - accuracy: 0.8975 - loss: 0.2896 - val_accuracy: 0.8700 - val_loss: 0.3326
Epoch 5/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 41s 39ms/step - accuracy: 0.9149 - loss: 0.2483 - val_accuracy: 0.8432 - val_loss: 0.3424
Epoch 6/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 41s 39ms/step - accuracy: 0.9263 - loss: 0.2198 - val_accuracy: 0.8722 - val_loss: 0.4240
Epoch 7/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 41s 39ms/step - accuracy: 0.9360 - loss: 0.1921 - val_accuracy: 0.8742 - val_loss: 0.3244
Epoch 8/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 41s 39ms/step - accuracy: 0.9493 - loss: 0.1579 - val_accuracy: 0.8774 - val_loss: 0.3638
Epoch 9/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 41s 39ms/step - accuracy: 0.9536 - loss: 0.1498 - val_accuracy: 0.8856 - val_loss: 0.4005
Epoch 10/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 41s 39ms/step - accuracy: 0.9625 - loss: 0.1215 - val_accuracy: 0.8824 - val_loss: 0.3912
782/782 ━━━━━━━━━━━━━━━━━━━━ 14s 17ms/step - accuracy: 0.8648 - loss: 0.4452
Test acc (Trained Embedding): 0.862
```

```
# Plotting the training and validation accuracy/loss for the learned embedding model
plt.figure(figsize=(10, 6))

# Accuracy plot
plt.subplot(1, 2, 1)
plt.plot(history_learned_embed.history['accuracy'], label='Training Accuracy')
plt.plot(history_learned_embed.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training and Validation Accuracy (Trained Embedding)')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

# Loss plot
plt.subplot(1, 2, 2)
plt.plot(history_learned_embed.history['loss'], label='Training Loss')
plt.plot(history_learned_embed.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss (Trained Embedding)')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

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


```
import requests
import zipfile
import os
import numpy as np

# Step 1: Download GloVe zip file
glove_url = "http://nlp.stanford.edu/data/glove.6B.zip"
glove_zip = "glove.6B.zip"
glove_txt = "glove.6B.100d.txt"

if not os.path.exists(glove_txt):
    print("Downloading GloVe embeddings...")
    response = requests.get(glove_url)
    with open(glove_zip, "wb") as file:
        file.write(response.content)

    print("Extracting GloVe files...")
    with zipfile.ZipFile(glove_zip, 'r') as zip_ref:
        zip_ref.extractall(".")

    os.remove(glove_zip)  # Optional cleanup
    print("GloVe download and extraction complete.")
else:
    print("GloVe file already exists.")

# Step 2: Load glove.6B.100d.txt into a dictionary
embeddings_index = {}
try:
    print("Loading GloVe word vectors into dictionary...")
    with open(glove_txt, 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.")
except FileNotFoundError:
    print(f"File not found: {glove_txt}")
except Exception as e:
    print(f"Error while loading GloVe: {e}")
```

```
GloVe file already exists.
Loading GloVe word vectors into dictionary...
Found 400000 word vectors.
```

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

```
# Parameters
vocab_limit = 20000
embedding_dim = 100
seq_limit = 600
batch_size = 32

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

# Step 2: Load dataset
review_train = keras.utils.text_dataset_from_directory("aclImdb/train", batch_size=batch_size)
review_valid = keras.utils.text_dataset_from_directory("aclImdb/val", batch_size=batch_size)
review_test = keras.utils.text_dataset_from_directory("aclImdb/test", batch_size=batch_size)

# Step 3: Vectorization layer
train_text_stream = review_train.map(lambda text, label: text)
token_encoder = layers.TextVectorization(max_tokens=vocab_limit, output_sequence_length=seq_limit)
token_encoder.adapt(train_text_stream)

# Step 4: Create embedding matrix from GloVe
vocab = token_encoder.get_vocabulary()
word_index = dict(zip(vocab, range(len(vocab))))

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

# Step 5: Define the frozen embedding layer
frozen_embed = layers.Embedding(
    input_dim=vocab_limit,
    output_dim=embedding_dim,
    embeddings_initializer=keras.initializers.Constant(embedding_matrix),
    trainable=False,
    mask_zero=True
)

# Step 6: Vectorize the datasets
def vectorize_data(text, label):
    return token_encoder(text), label

encoded_train = review_train.map(vectorize_data).cache().prefetch(tf.data.AUTOTUNE)
encoded_valid = review_valid.map(vectorize_data).cache().prefetch(tf.data.AUTOTUNE)
encoded_test = review_test.map(vectorize_data).cache().prefetch(tf.data.AUTOTUNE)

# Step 7: Build the model using GloVe embeddings
glove_input = tf.keras.Input(shape=(None,), dtype="int64")
x = frozen_embed(glove_input)
x = layers.Bidirectional(layers.LSTM(32))(x)
x = layers.Dropout(0.5)(x)
glove_output = layers.Dense(1, activation="sigmoid")(x)

glove_embed_model = tf.keras.Model(glove_input, glove_output)

# Step 8: Compile and train
glove_embed_model.compile(optimizer="rmsprop", loss="binary_crossentropy", metrics=["accuracy"])

glove_history = glove_embed_model.fit(
    encoded_train,
    validation_data=encoded_valid,
    epochs=10
)

# Step 9: Evaluate
glove_test_score = glove_embed_model.evaluate(encoded_test)[1]
print(f" Test Accuracy (GloVe Pretrained): {glove_test_score:.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 ━━━━━━━━━━━━━━━━━━━━ 31s 37ms/step - accuracy: 0.6298 - loss: 0.6273 - val_accuracy: 0.7944 - val_loss: 0.4517
Epoch 2/10
```

```
782/782 ━━━━━━━━━━━━━━━━━━━━ 27s 34ms/step - accuracy: 0.7919 - loss: 0.4578 - val_accuracy: 0.8270 - val_loss: 0.3845
Epoch 3/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 27s 34ms/step - accuracy: 0.8212 - loss: 0.3999 - val_accuracy: 0.8252 - val_loss: 0.3771
Epoch 4/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 42s 35ms/step - accuracy: 0.8431 - loss: 0.3652 - val_accuracy: 0.8732 - val_loss: 0.3074
Epoch 5/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 40s 34ms/step - accuracy: 0.8561 - loss: 0.3377 - val_accuracy: 0.8862 - val_loss: 0.2798
Epoch 6/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 27s 34ms/step - accuracy: 0.8669 - loss: 0.3129 - val_accuracy: 0.8872 - val_loss: 0.2772
Epoch 7/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 27s 34ms/step - accuracy: 0.8783 - loss: 0.2909 - val_accuracy: 0.9002 - val_loss: 0.2518
Epoch 8/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 26s 33ms/step - accuracy: 0.8852 - loss: 0.2803 - val_accuracy: 0.9032 - val_loss: 0.2411
Epoch 9/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 26s 33ms/step - accuracy: 0.8939 - loss: 0.2651 - val_accuracy: 0.9142 - val_loss: 0.2230
Epoch 10/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 41s 33ms/step - accuracy: 0.9006 - loss: 0.2496 - val_accuracy: 0.9174 - val_loss: 0.2188
782/782 ━━━━━━━━━━━━━━━━━━━━ 13s 17ms/step - accuracy: 0.8769 - loss: 0.2861
 Test Accuracy (GloVe Pretrained): 0.878
```

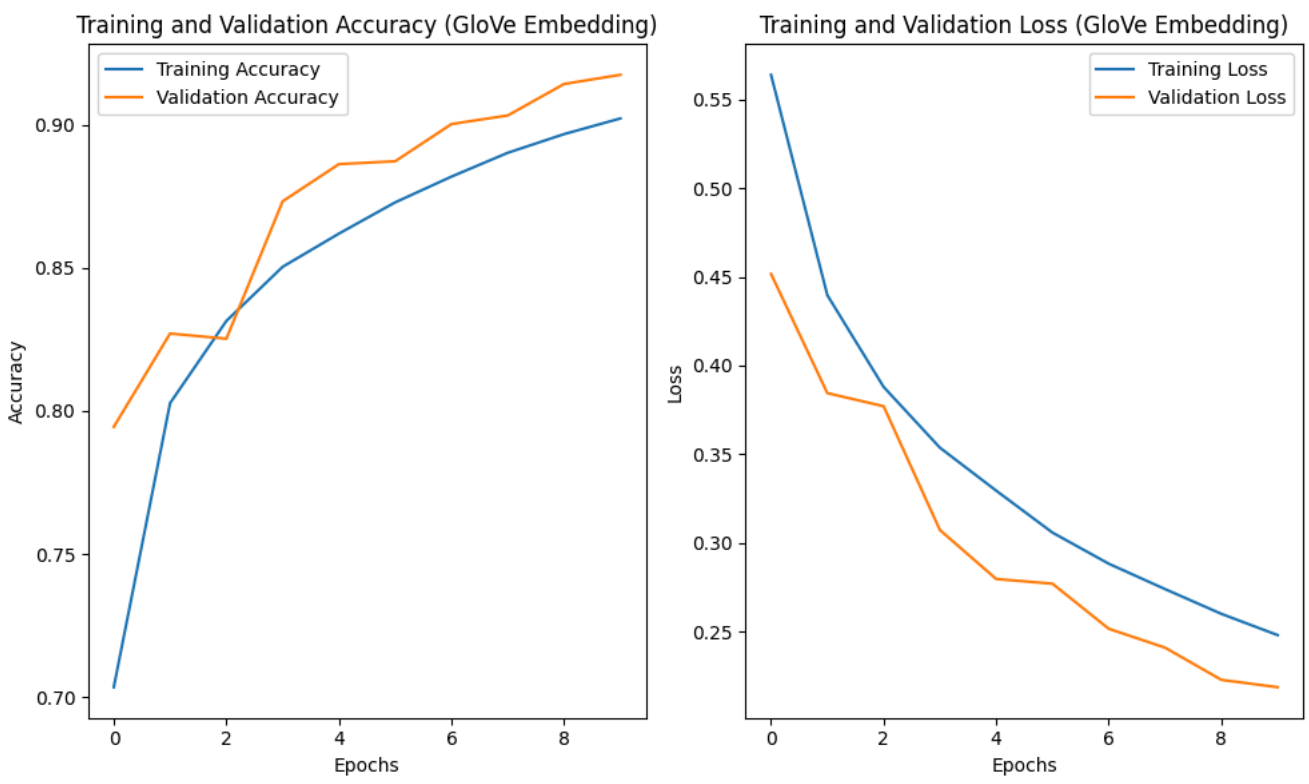
```
import matplotlib.pyplot as plt

# Plotting accuracy and loss for GloVe-based model
plt.figure(figsize=(10, 6))

# Accuracy plot
plt.subplot(1, 2, 1)
plt.plot(glove_history.history['accuracy'], label='Training Accuracy')
plt.plot(glove_history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training and Validation Accuracy (GloVe Embedding)')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

# Loss plot
plt.subplot(1, 2, 2)
plt.plot(glove_history.history['loss'], label='Training Loss')
plt.plot(glove_history.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss (GloVe Embedding)')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

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




```
import matplotlib.pyplot as plt

# Define different training sample sizes
sample_sizes = [100, 500, 1000, 5000, len(list(int_train_ds))]
```

```
# Store test accuracies
test_acc_vs_size = []

for size in sample_sizes:
    # Limit the training dataset
    partial_train_ds = int_train_ds.take(size)

    # Define model with frozen GloVe embedding
    sample_input = tf.keras.Input(shape=(None,), dtype="int64")
    x = frozen_embed(sample_input)
    x = layers.Bidirectional(layers.LSTM(32))(x)
    x = layers.Dropout(0.5)(x)
    sample_output = layers.Dense(1, activation="sigmoid")(x)

    sample_model = tf.keras.Model(sample_input, sample_output)
    sample_model.compile(optimizer="rmsprop", loss="binary_crossentropy", metrics=["accuracy"])

    # Train on subset
    print(f"\n Training with {size} samples...")
    sample_model.fit(partial_train_ds, validation_data=int_val_ds, epochs=10, verbose=0)

    # Evaluate
    acc = sample_model.evaluate(int_test_ds, verbose=0)[1]
    test_acc_vs_size.append(acc)
    print(f"Training Size: {size}, Test Accuracy: {acc:.3f}")

# Print final results
print("\n Final Test Accuracies:")
for s, a in zip(sample_sizes, test_acc_vs_size):
    print(f"Training Size: {s}, Test Accuracy: {a:.3f}")

# Plot
plt.figure(figsize=(8, 6))
plt.plot(sample_sizes, test_acc_vs_size, marker='o', linestyle='-', color='blue')
plt.title("Test Accuracy vs. Training Sample Size (GloVe)")
plt.xlabel("Training Samples")
plt.ylabel("Test Accuracy")
plt.grid(True)
plt.xticks(sample_sizes)
plt.show()
```