Assignment-4

Bharath Bhimireddy

Introduction: In this assignment, we investigate the use of Recurrent Neural Networks (RNNs) and Transformers on text and sequence data, with a focus on improving performance in cases with little data. Drawing inspiration from the IMDB example in Chapter 6, we made many changes to the model architecture and training settings to see how they affected prediction improvement.

Dataset: Imdb dataset. Link: http://ai.stanford.edu/~amaas/data/sentiment/

Using the conditions given in the question I have implemented the following:

Given,

Consider the IMDB example from Chapter 6. Re-run the example modifying the following:

1. Cutoff reviews after 150 words.

2. Restrict training samples to 100.

3. Validate 10,000 samples.

4. Consider only the top 10,000 words.

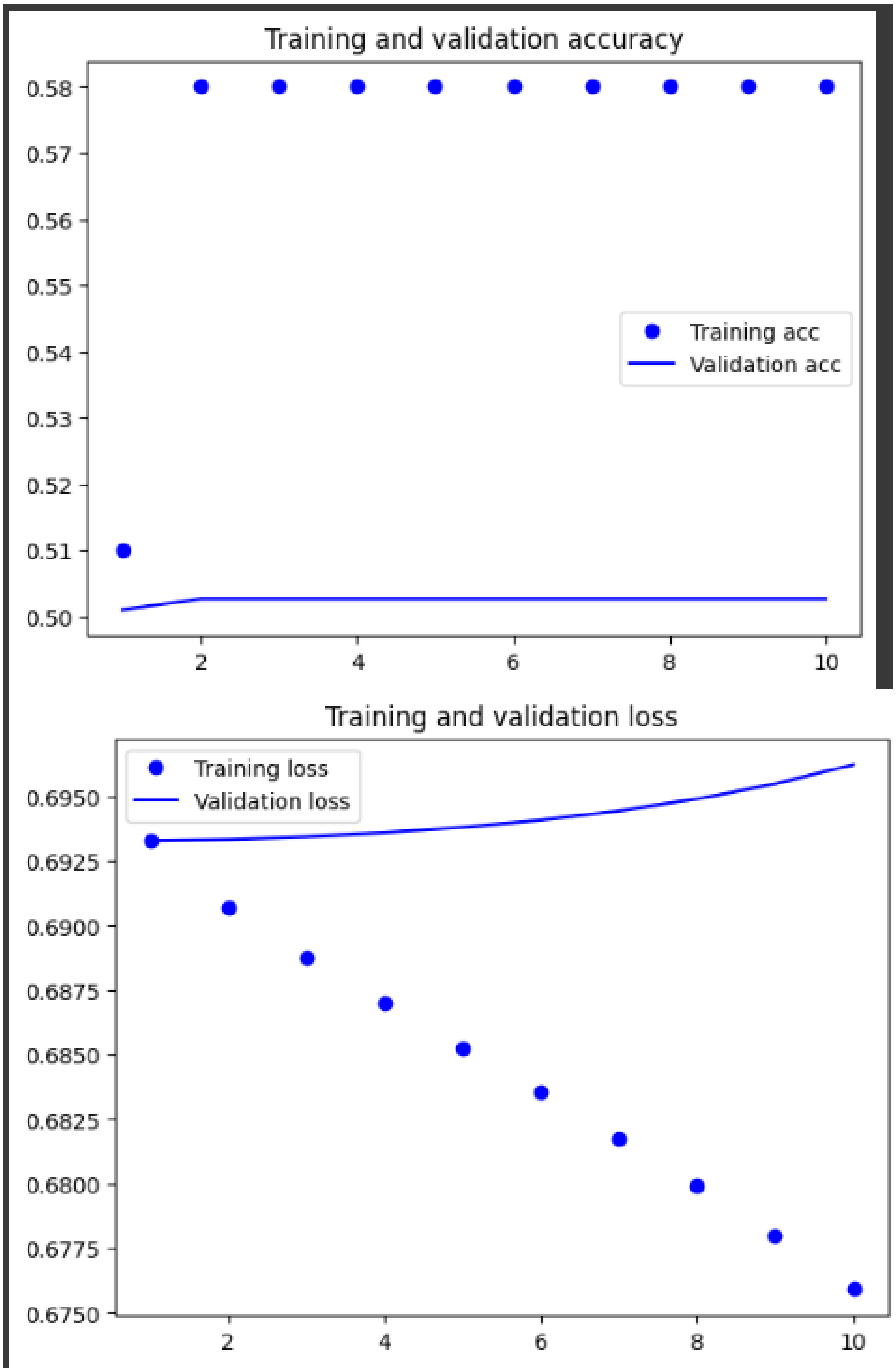
I am using an embedding layer first to run the model.

```
from keras.datasets import imdb
from keras.preprocessing import sequence
from keras.models import Sequential
from keras.layers import Embedding, LSTM, Dense

max_features = 10000  # number of words to consider as features
maxlen = 150  # cut texts after this number of words
batch_size = 32

print('Loading data...')
(input_train, y_train), (input_test, y_test) = imdb.load_data(num_words=max_features)

# Restrict training samples to 100
input_train = input_train[:100]
y_train = y_train[:100]

print(len(input_train), 'train sequences')
print(len(input_test), 'test sequences')

print('Pad sequences (samples x time)')
input_train = sequence.pad_sequences(input_train, maxlen=maxlen)
input_test = sequence.pad_sequences(input_test, maxlen=maxlen)
print('input_train shape:', input_train.shape)
print('input_test shape:', input_test.shape)

model = Sequential()
model.add(Embedding(max_features, 32))
model.add(LSTM(32))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])

# Validate on 10,000 samples
history = model.fit(input_train, y_train,
                    epochs=10,
                    batch_size=128,
                    validation_data=(input_test[:10000], y_test[:10000]))
```

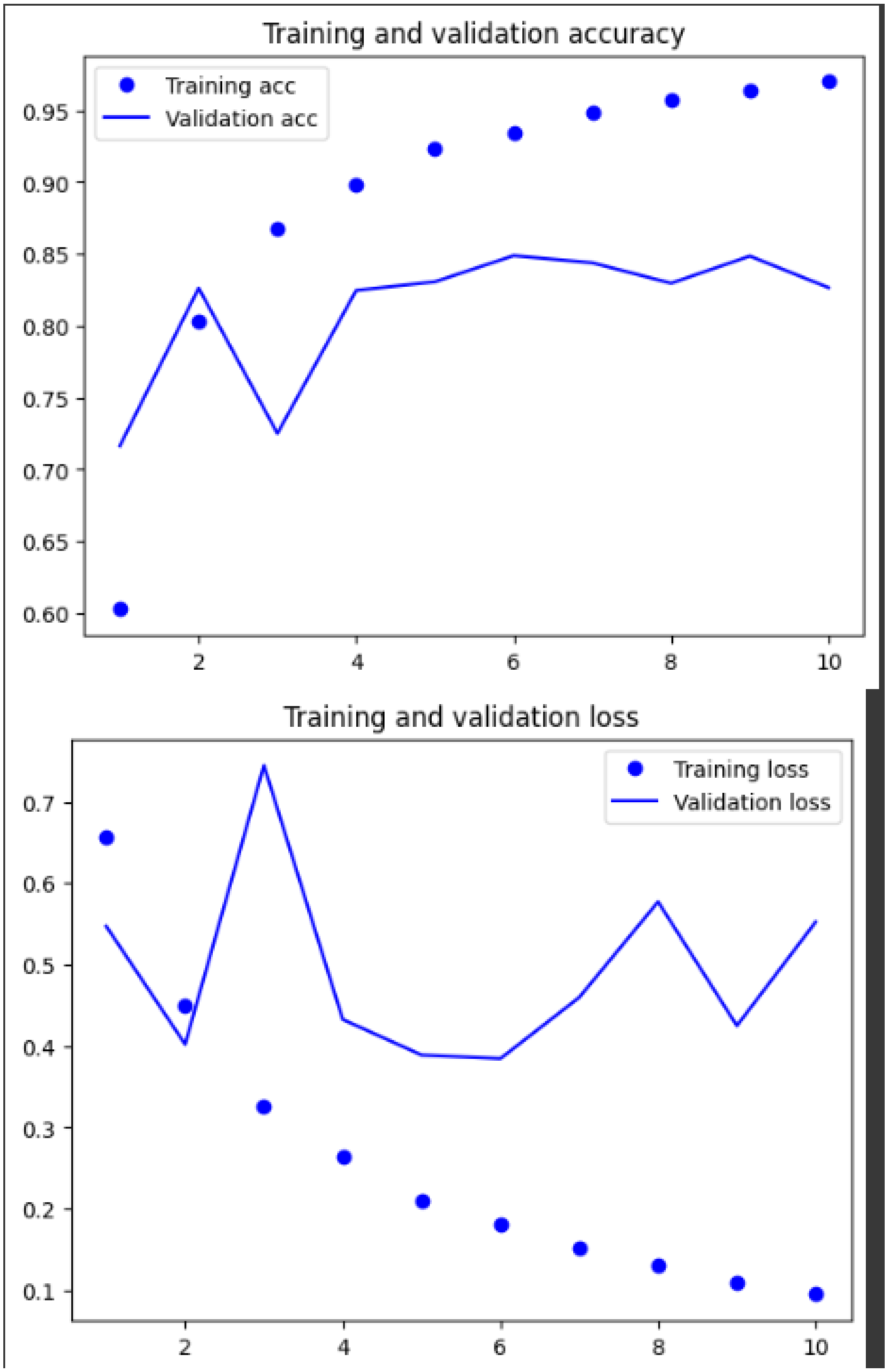
To distinguish between positive and negative movie reviews, I used an LSTM neural network. It makes use of Keras for performance assessment, text data preprocessing, and model construction and training. The code loads the 25,000 movie reviews from IMDB and preprocesses it by padding each review to be 150 words long. Then, using an embedding layer, an LSTM layer, and a dense layer, it builds a sequential model.

Words are numerically represented by the embedding layer, sentiment information is extracted by the LSTM layer, and reviews are categorized by the dense layer. The binary cross-entropy loss function and RMSprop optimizer are used in the model's compilation. It undergoes ten epochs of training with a batch size of 128 using the processed training data.

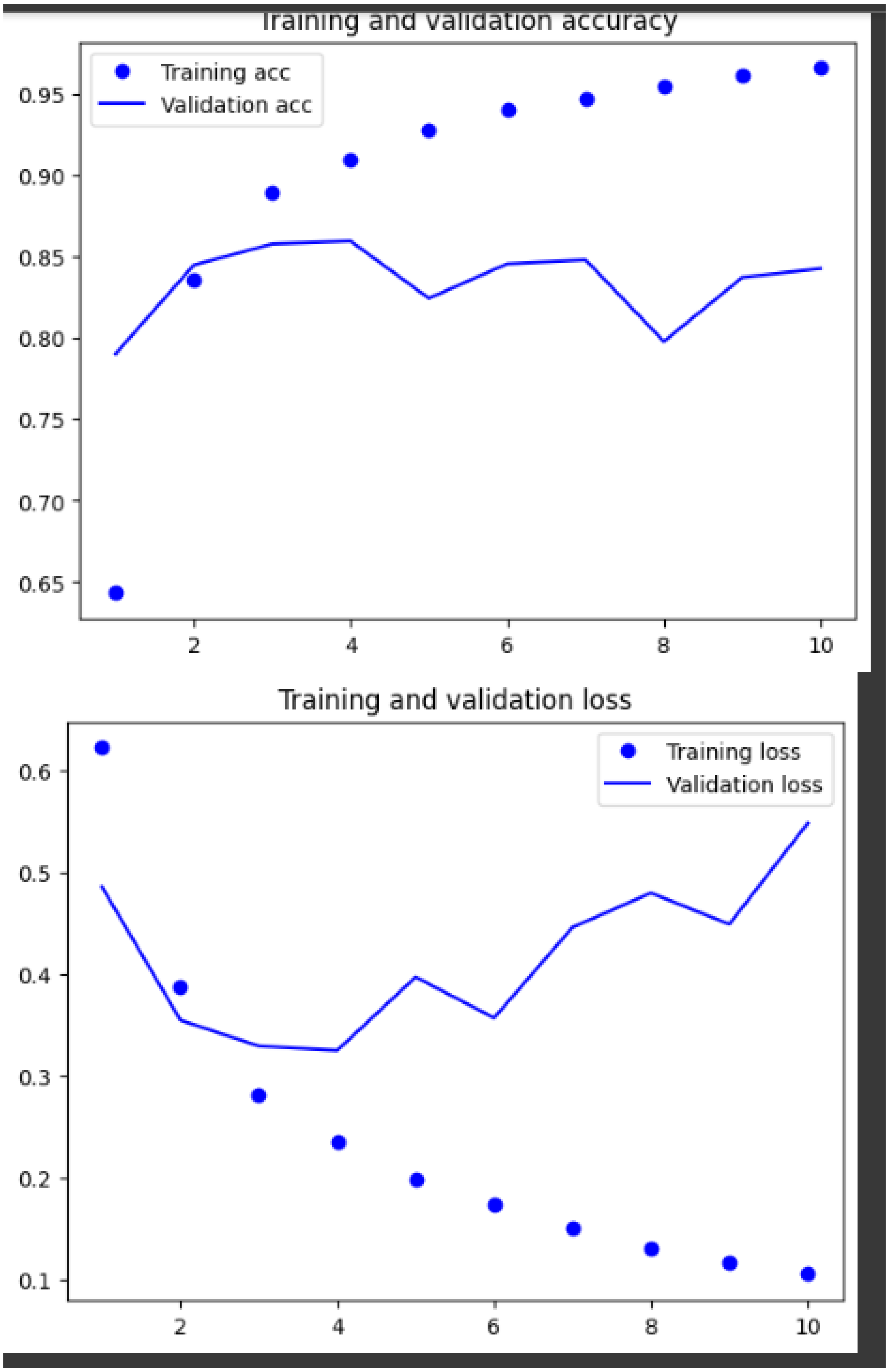
For 100 trained values, I got a validation accuracy as 50%.



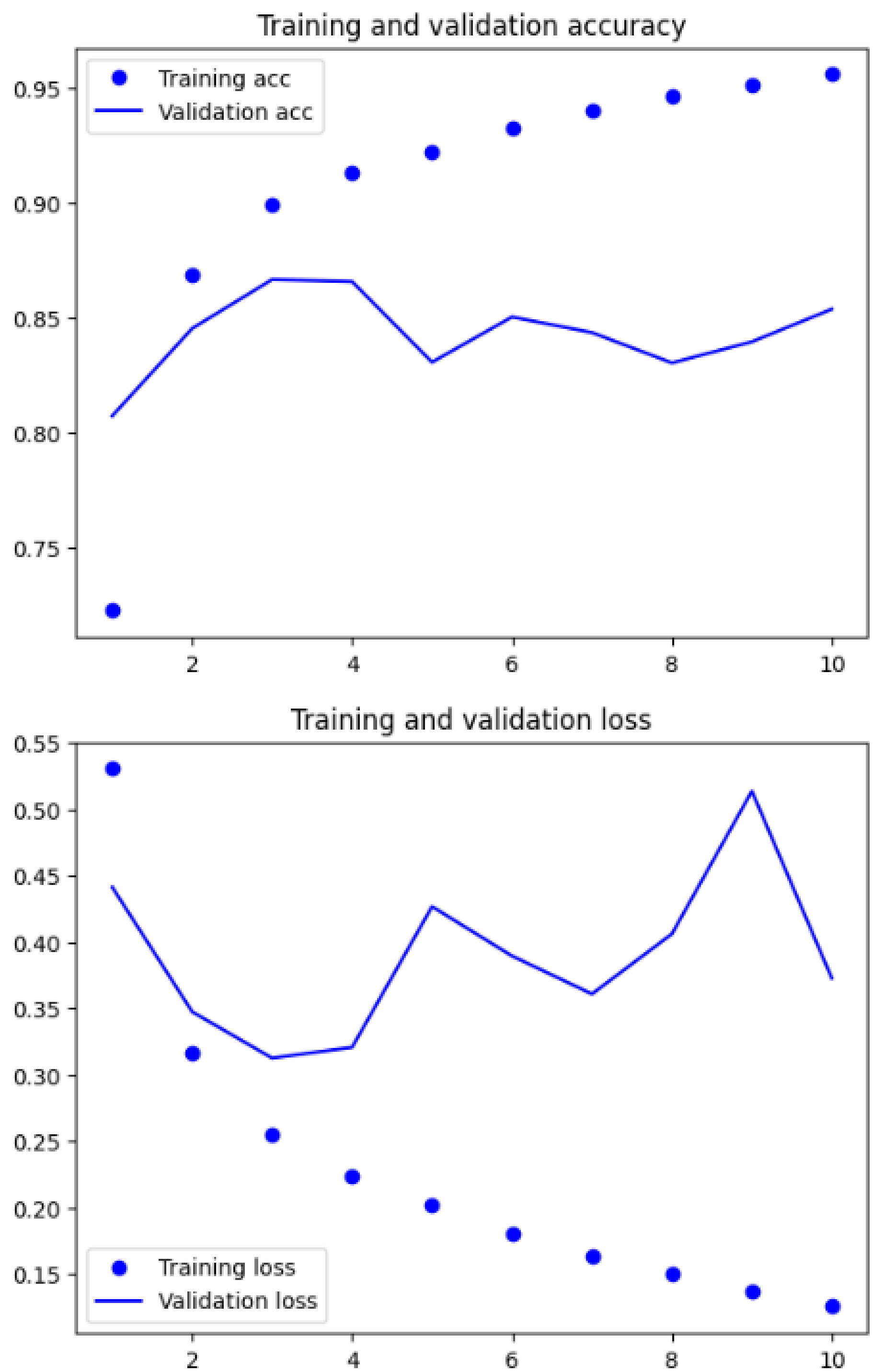
Solution: I increased the number of training words from 100 to 10000 which resulted in validation accuracy as 82.66%.



If I increased the number to 15000, I got validation accuracy as 84.26%.



If I increased the number to 25000, I got validation accuracy as 85.39%.



Increasing sample size will reduce the overfitting problem and increase the accuracy of the model.

Using a pre-trained Embedding:

```python
import os

imdb_dir = '/content/aclImdb'
train_dir = os.path.join(imdb_dir, 'train')

labels = []
texts = []

for label_type in ['neg', 'pos']:
    dir_name = os.path.join(train_dir, label_type)
    for fname in os.listdir(dir_name):
        if fname[-4:] == '.txt':
            f = open(os.path.join(dir_name, fname))
            texts.append(f.read())
            f.close()
            if label_type == 'neg':
                labels.append(0)
            else:
                labels.append(1)
```

[ ]
```python
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
import numpy as np

maxlen = 150  # Cutoff reviews after 150 words
training_samples = 100  # Restrict training samples to 100
validation_samples = 10000  # Validate on 10,000 samples
max_words = 10000  # Consider only the top 10,000 words

tokenizer = Tokenizer(num_words=max_words)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)

word_index = tokenizer.word_index
print('Found %s unique tokens.' % len(word_index))

data = pad_sequences(sequences, maxlen=maxlen)

labels = np.asarray(labels)
print('Shape of data tensor:', data.shape)
print('Shape of label tensor:', labels.shape)

# Split the data into a training set and a validation set
# But first, shuffle the data, since we started from data
# where samples are ordered (all negative first, then all positive).
indices = np.arange(data.shape[0])
np.random.shuffle(indices)
data = data[indices]
labels = labels[indices]

x_train = data[:training_samples]
y_train = labels[:training_samples]
x_val = data[training_samples: training_samples + validation_samples]
y_val = labels[training_samples: training_samples + validation_samples]
```

Using a pretrained word embedding

[ ]
```python
glove_dir = '/content/drive/MyDrive/glove.6B'

embeddings_index = {}
f = open(os.path.join(glove_dir, 'glove.6B.100d.txt'))
for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
f.close()

print('Found %s word vectors.' % len(embeddings_index))
```

```
Found 400000 word vectors.
```

```python
embedding_dim = 100

embedding_matrix = np.zeros((max_words, embedding_dim))
for word, i in word_index.items():
    embedding_vector = embeddings_index.get(word)
    if i < max_words:
        if embedding_vector is not None:
            # Words not found in embedding index will be all-zeros.
            embedding_matrix[i] = embedding_vector
```

[ ]
```python
from keras.models import Sequential
from keras.layers import Embedding, Flatten, Dense

model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=maxlen))
model.add(Flatten())
model.add(Dense(32, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.summary()
```

```
model.layers[0].set_weights([embedding_matrix])
model.layers[0].trainable = False
```

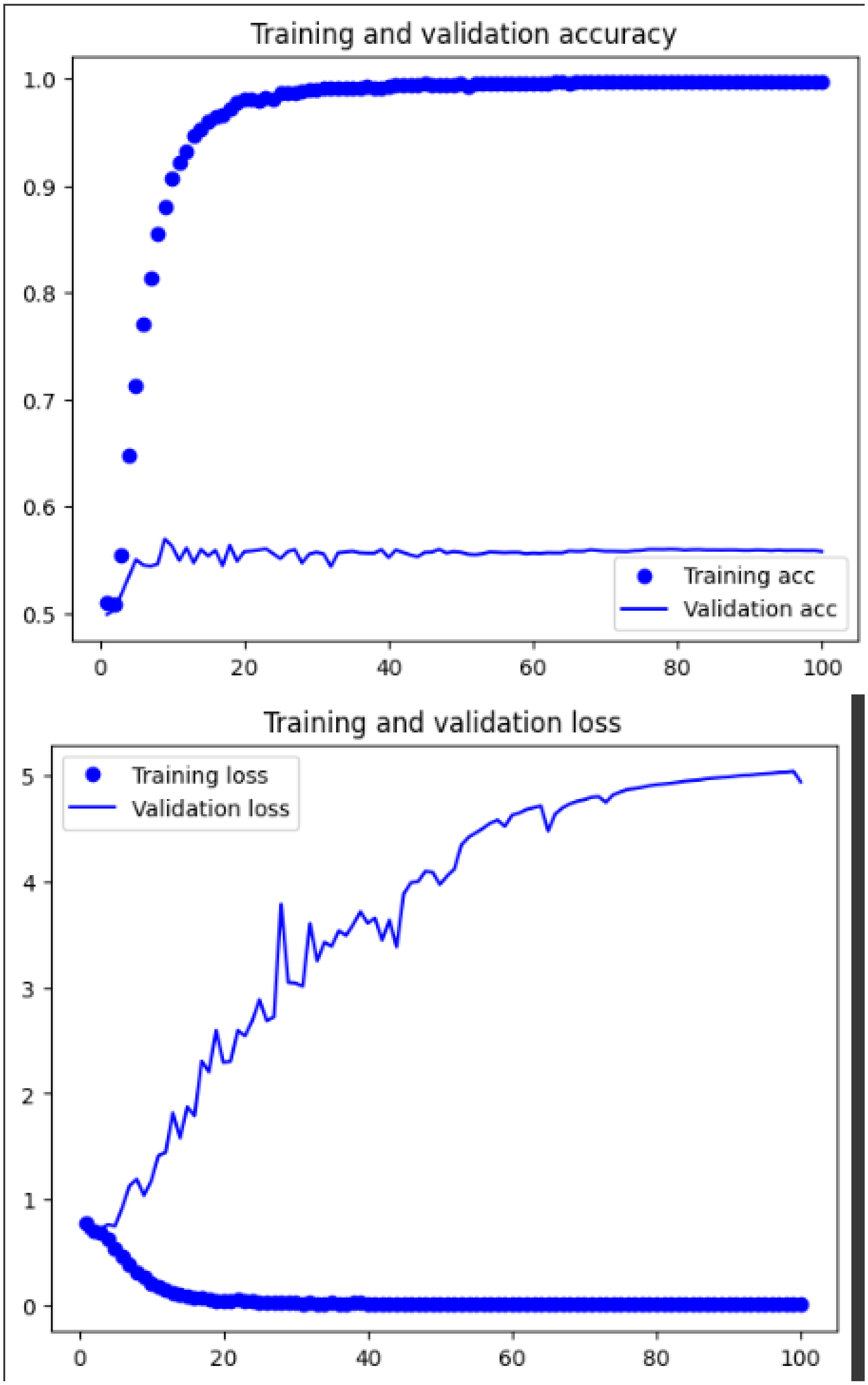
```
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])
history = model.fit(x_train, y_train,
                    epochs=10,
                    batch_size=32,
                    validation_data=(x_val, y_val))
model.save_weights('pre_trained_glove_model.h5')
```

```
Epoch 1/10
4/4 [==============================] - 2s 262ms/step - loss: 2.9468 - acc: 0.5900 - val_loss: 0.8126 - val_acc: 0.5019
Epoch 2/10
4/4 [==============================] - 1s 201ms/step - loss: 0.4552 - acc: 0.7500 - val_loss: 2.4037 - val_acc: 0.5019
Epoch 3/10
4/4 [==============================] - 1s 219ms/step - loss: 0.8799 - acc: 0.5500 - val_loss: 0.7569 - val_acc: 0.5299
Epoch 4/10
4/4 [==============================] - 1s 218ms/step - loss: 0.1453 - acc: 1.0000 - val_loss: 1.0091 - val_acc: 0.5030
Epoch 5/10
4/4 [==============================] - 1s 204ms/step - loss: 0.1186 - acc: 1.0000 - val_loss: 1.2388 - val_acc: 0.5037
Epoch 6/10
4/4 [==============================] - 1s 199ms/step - loss: 0.0927 - acc: 1.0000 - val_loss: 0.7093 - val_acc: 0.5542
Epoch 7/10
4/4 [==============================] - 1s 202ms/step - loss: 0.0496 - acc: 1.0000 - val_loss: 0.7231 - val_acc: 0.5524
Epoch 8/10
4/4 [==============================] - 1s 226ms/step - loss: 0.0561 - acc: 1.0000 - val_loss: 1.5030 - val_acc: 0.5038
Epoch 9/10
4/4 [==============================] - 1s 290ms/step - loss: 0.0276 - acc: 1.0000 - val_loss: 1.3909 - val_acc: 0.5059
Epoch 10/10
4/4 [==============================] - 1s 299ms/step - loss: 0.0161 - acc: 1.0000 - val_loss: 1.0039 - val_acc: 0.5178
```

This code uses a deep learning model and a pre-trained GloVe word embedding to categorize movie reviews as either positive or negative. The IMDB dataset is loaded first, and the text data is preprocessed using padding and tokenization. It then builds a sequential model that consists of an embedding layer, a flattening layer, a dense layer activated by ReLU, and a final dense layer activated by sigmoid. The pre-trained GloVe word embeddings are used to initialize the model's weights, and the embedding layer is configured to be non-trainable during training. After that, the model is compiled using accuracy metrics, a binary cross-entropy loss function, and an RMSprop optimizer. Ultimately, the model undergoes ten epochs of training on the training data before being assessed on the validation data.

When I ran this model on the Imdb dataset I got a test accuracy as 50.78%.

Solution: Using 100 trained samples I got an accuracy of 50.78%, this is a clear case of overfitting. To solve this problem, I increased the size of the trained samples from 100 to 10000 which resulted in better accuracy of 56.89%.




```
model.load_weights('pre_trained_glove_model.h5')
model.evaluate(x_test, y_test)
```

```
782/782 [==============================] - 2s 2ms/step - loss: 4.9163 - acc: 0.5685
[4.916293621063232, 0.5685200095176697]
```

Summary of results:

S No.	Method	Hidden layers	Training size	Training accuracy	Validation Accuracy
1	Embedding Layer	32	100	58	50
2	Embedding Layer	32	10000	97	82.66
3	Embedding Layer	32	25000	95.61	85.39
4	Pre-trained		100	100	55.38
5	Pre-trained		10000	99.66	55.79

In conclusion, our findings show that for tasks involving limited data and text classification on the IMDB dataset, using an embedding layer outperforms pre-trained word embeddings. Furthermore, as the amount of training data increases, the benefits of the embedding layer become more apparent. This emphasizes the significance of customizing model topologies to the specific qualities and restrictions of the dataset under consideration.