Ungraded Lab: Building Models for the IMDB Reviews Dataset

In this lab, you will build four models and train it on the IMDB Reviews dataset with full word encoding. These use different layers after the embedding namely `Flatten`, `LSTM`, `GRU`, and `Conv1D`. You will compare the performance and see which architecture might be best for this particular dataset. Let's begin!

Imports

You will first import common libraries that will be used throughout the exercise.

In [1]:
```
import tensorflow_datasets as tfds
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
```

Download and Prepare the Dataset

Next, you will download the `plain_text` version of the `IMDB Reviews` dataset.

In [2]:
```
# The dataset is already downloaded for you. For downloading you can use the code below.
imdb = tfds.load("imdb_reviews", as_supervised=True, data_dir="../data/", download=False)
```

In [3]:
```
# Get the train and test sets
train_dataset, test_dataset = imdb['train'], imdb['test']
```

Then, you will build the vocabulary based on the training set.

In [4]:
```
# Vectorization and Padding Parameters

VOCAB_SIZE = 10000
MAX_LENGTH = 120
PADDING_TYPE = 'pre'
TRUNC_TYPE = 'post'
```

In [5]:
```
# Instantiate the vectorization layer
vectorize_layer = tf.keras.layers.TextVectorization(max_tokens=VOCAB_SIZE)

# Get the string inputs and integer outputs of the training set
train_reviews = train_dataset.map(lambda review, label: review)

# Generate the vocabulary based only on the training set
vectorize_layer.adapt(train_reviews)

# Delete because it's no longer needed
del train_reviews
```

In Week 2, you generated the padded sequences by chaining `map()` and `apply()` methods. Here's a similar way to do that. You will just call an `apply()` then do the transformations in one preprocessing function.

In [6]:
```
def preprocessing_fn(dataset):
  '''Generates padded sequences from a tf.data.Dataset'''

  # Apply the vectorization layer to the string features
  dataset_sequences = dataset.map(
      lambda text, label: (vectorize_layer(text), label)
      )

  # Put all elements in a single ragged batch
  dataset_sequences = dataset_sequences.ragged_batch(
      batch_size=dataset_sequences.cardinality()
      )

  # Output a tensor from the single batch. Extract the sequences and labels.
  sequences, labels = dataset_sequences.get_single_element()

  # Pad the sequences
  padded_sequences = tf.keras.utils.pad_sequences(
      sequences.numpy(),
      maxlen=MAX_LENGTH,
      truncating=TRUNC_TYPE,
      padding=PADDING_TYPE
```

```
    )

  # Convert back to a tf.data.Dataset
  padded_sequences = tf.data.Dataset.from_tensor_slices(padded_sequences)
  labels = tf.data.Dataset.from_tensor_slices(labels)

  # Combine the padded sequences and labels
  dataset_vectorized = tf.data.Dataset.zip(padded_sequences, labels)

  return dataset_vectorized
```

In [7]:
```
# Preprocess the train and test data
train_dataset_vectorized = train_dataset.apply(preprocessing_fn)
test_dataset_vectorized = test_dataset.apply(preprocessing_fn)
```

View a couple of examples. You should see tuples of tensors with a padded sequence and label.

In [8]:
```
# View 2 training sequences and its labels
for example in train_dataset_vectorized.take(2):
  print(example)
  print()
```

```
(<tf.Tensor: shape=(120,), dtype=int32, numpy=
array([   0,    0,    0,    0,   11,   14,   34,  412,  384,   18,   90,
         28,    1,    8,   33, 1322, 3560,   42,  487,    1,  191,   24,
         85,  152,   19,   11,  217,  316,   28,   65,  240,  214,    8,
        489,   54,   65,   85,  112,   96,   22, 5596,   11,   93,  642,
        743,   11,   18,    7,   34,  394, 9522,  170, 2464,  408,    2,
         88, 1216,  137,   66,  144,   51,    2,    1, 7558,   66,  245,
         65, 2870,   16,    1, 2860,    1,    1, 1426, 5050,    3,   40,
          1, 1579,   17, 3560,   14,  158,   19,    4, 1216,  891, 8040,
          8,    4,   18,   12,   14, 4059,    5,   99,  146, 1241,   10,
        237,  704,   12,   48,   24,   93,   39,   11, 7339,  152,   39,
       1322,    1,   50,  398,   10,   96, 1155,  851,  141,    9],
      dtype=int32)>, <tf.Tensor: shape=(), dtype=int64, numpy=0>)

(<tf.Tensor: shape=(120,), dtype=int32, numpy=
array([   0,    0,    0,    0,    0,    0,    0,    0,   10,   26,   75,
        617,    6,  776, 2355,  299,   95,   19,   11,    7,  604,  662,
          6,    4, 2129,    5,  180,  571,   63, 1403,  107, 2410,    3,
       3905,   21,    2,    1,    3,  252,   41, 4781,    4,  169,  186,
         21,   11, 4259,   10, 1507, 2355,   80,    2,   20,   14, 1973,
          2,  114,  943,   14, 1740, 1300,  594,    3,  356,  180,  446,
          6,  596,   19,   17,   57, 1775,    5,   49,   14, 4002,   98,
         42,  134,   10,  934,   10,  194,   26, 1026,  171,    5,    2,
         20,   19,   10,  284,    2, 2065,    5,    9,    3,  279,   41,
        446,    6,  596,    5,   30,  200,    1,  201,   99,  146, 4525,
         16,  229,  329,   10,  175,  368,   11,   20,   31,   32],
      dtype=int32)>, <tf.Tensor: shape=(), dtype=int64, numpy=0>)
```

You will do the optimization and batching as usual.

In [9]:
```
SHUFFLE_BUFFER_SIZE = 1000
PREFETCH_BUFFER_SIZE = tf.data.AUTOTUNE
BATCH_SIZE = 32

# Optimize and batch the datasets for training
train_dataset_final = (train_dataset_vectorized
                       .cache()
                       .shuffle(SHUFFLE_BUFFER_SIZE)
                       .prefetch(PREFETCH_BUFFER_SIZE)
                       .batch(BATCH_SIZE)
                       )

test_dataset_final = (test_dataset_vectorized
                      .cache()
                      .prefetch(PREFETCH_BUFFER_SIZE)
                      .batch(BATCH_SIZE)
                      )
```

Plot Utility

The function below will visualize the accuracy and loss history after training.

In [10]:
```
def plot_loss_acc(history):
  '''Plots the training and validation loss and accuracy from a history object'''
  acc = history.history['accuracy']
  val_acc = history.history['val_accuracy']
```

```python
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(len(acc))

fig, ax = plt.subplots(1,2, figsize=(12, 6))
ax[0].plot(epochs, acc, 'bo', label='Training accuracy')
ax[0].plot(epochs, val_acc, 'b', label='Validation accuracy')
ax[0].set_title('Training and validation accuracy')
ax[0].set_xlabel('epochs')
ax[0].set_ylabel('accuracy')
ax[0].legend()

ax[1].plot(epochs, loss, 'bo', label='Training Loss')
ax[1].plot(epochs, val_loss, 'b', label='Validation Loss')
ax[1].set_title('Training and validation loss')
ax[1].set_xlabel('epochs')
ax[1].set_ylabel('loss')
ax[1].legend()

plt.show()
```

Model 1: Flatten

First up is simply using a `Flatten` layer after the embedding. Its main advantage is that it is very fast to train. Observe the results below.

In [11]:

```python
# Parameters
EMBEDDING_DIM = 16
DENSE_DIM = 6

# Model Definition with a Flatten layer
model_flatten = tf.keras.Sequential([
    tf.keras.Input(shape=(MAX_LENGTH,)),
    tf.keras.layers.Embedding(input_dim=VOCAB_SIZE, output_dim=EMBEDDING_DIM),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(DENSE_DIM, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

# Set the training parameters
model_flatten.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accuracy'])

# Print the model summary
model_flatten.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 120, 16)	160,000
flatten (Flatten)	(None, 1920)	0
dense (Dense)	(None, 6)	11,526
dense_1 (Dense)	(None, 1)	7

Total params: 171,533 (670.05 KB)

Trainable params: 171,533 (670.05 KB)

Non-trainable params: 0 (0.00 B)

In [12]:

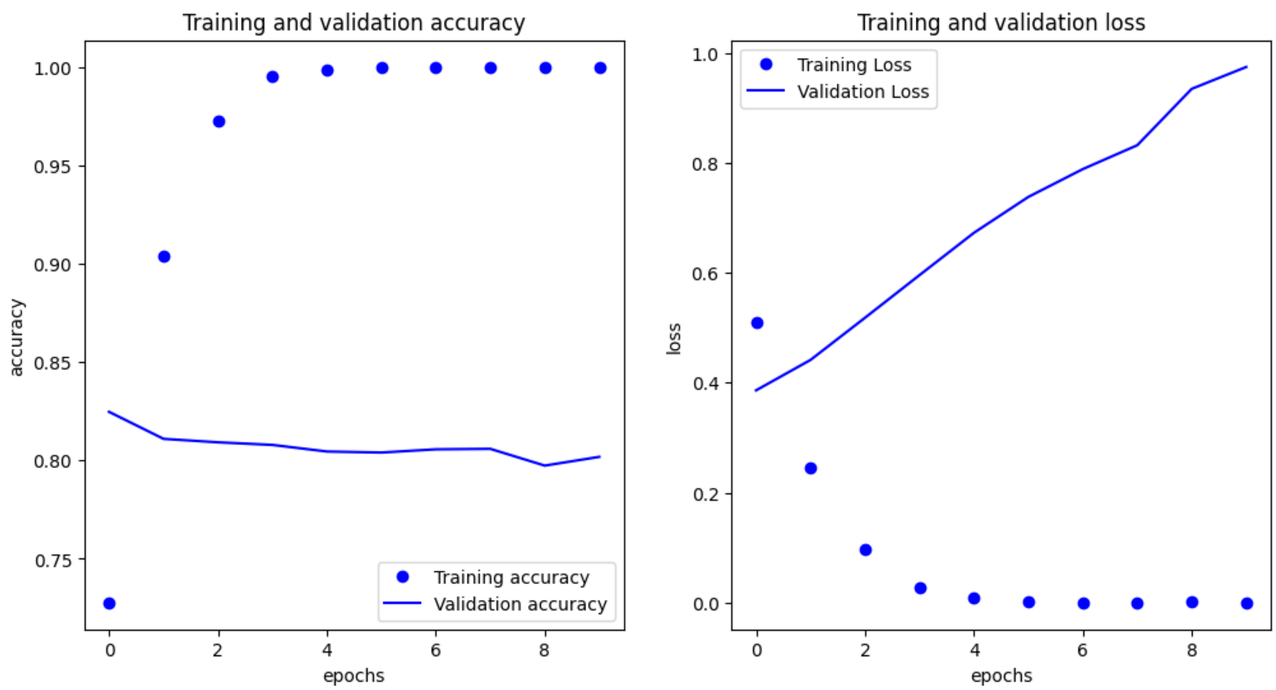
```python
NUM_EPOCHS = 10

# Train the model
history_flatten = model_flatten.fit(train_dataset_final, epochs=NUM_EPOCHS, validation_data=(test_dataset_final))
```

```
Epoch 1/10
WARNING: All log messages before absl::InitializeLog() is called are written to STDERR
I0000 00:00:1745359944.982301    5153 service.cc:145] XLA service 0x70a6a452f1c0 initialized for platform CUDA (this does not gua
rantee that XLA will be used). Devices:
I0000 00:00:1745359944.982352    5153 service.cc:153]   StreamExecutor device (0): NVIDIA A10G, Compute Capability 8.6
180/782 ━━━━━━━━━━━━━━━━━━━━ 0s 848us/step - accuracy: 0.5190 - loss: 0.6926
I0000 00:00:1745359947.174889    5153 device_compiler.h:188] Compiled cluster using XLA!  This line is logged at most once for th
e lifetime of the process.
```

```
782/782 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.6220 - loss: 0.6191 - val_accuracy: 0.8247 - val_loss: 0.3862
Epoch 2/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8783 - loss: 0.2946 - val_accuracy: 0.8110 - val_loss: 0.4416
Epoch 3/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9559 - loss: 0.1364 - val_accuracy: 0.8092 - val_loss: 0.5182
Epoch 4/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9927 - loss: 0.0390 - val_accuracy: 0.8079 - val_loss: 0.5957
Epoch 5/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9980 - loss: 0.0131 - val_accuracy: 0.8045 - val_loss: 0.6726
Epoch 6/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9996 - loss: 0.0037 - val_accuracy: 0.8040 - val_loss: 0.7379
Epoch 7/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 1.0000 - loss: 0.0011 - val_accuracy: 0.8056 - val_loss: 0.7885
Epoch 8/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 1.0000 - loss: 5.3765e-04 - val_accuracy: 0.8059 - val_loss: 0.8319
Epoch 9/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9998 - loss: 9.7721e-04 - val_accuracy: 0.7974 - val_loss: 0.9344
Epoch 10/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 1.0000 - loss: 6.9019e-04 - val_accuracy: 0.8018 - val_loss: 0.9741
```

In [13]:

```
# Plot the accuracy and loss history
plot_loss_acc(history_flatten)
```



LSTM

Next, you will use an LSTM. This is slower to train but useful in applications where the order of the tokens is important.

In [14]:

```
# Parameters
EMBEDDING_DIM = 16
LSTM_DIM = 32
DENSE_DIM = 6

# Model Definition with LSTM
model_lstm = tf.keras.Sequential([
    tf.keras.Input(shape=(MAX_LENGTH,)),
    tf.keras.layers.Embedding(input_dim=VOCAB_SIZE, output_dim=EMBEDDING_DIM),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(LSTM_DIM)),
    tf.keras.layers.Dense(DENSE_DIM, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

# Set the training parameters
model_lstm.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accuracy'])

# Print the model summary
model_lstm.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
embedding_1 (Embedding)	(None, 120, 16)	160,000
bidirectional (Bidirectional)	(None, 64)	12,544
dense_2 (Dense)	(None, 6)	390
dense_3 (Dense)	(None, 1)	7

Total params: 172,941 (675.55 KB)

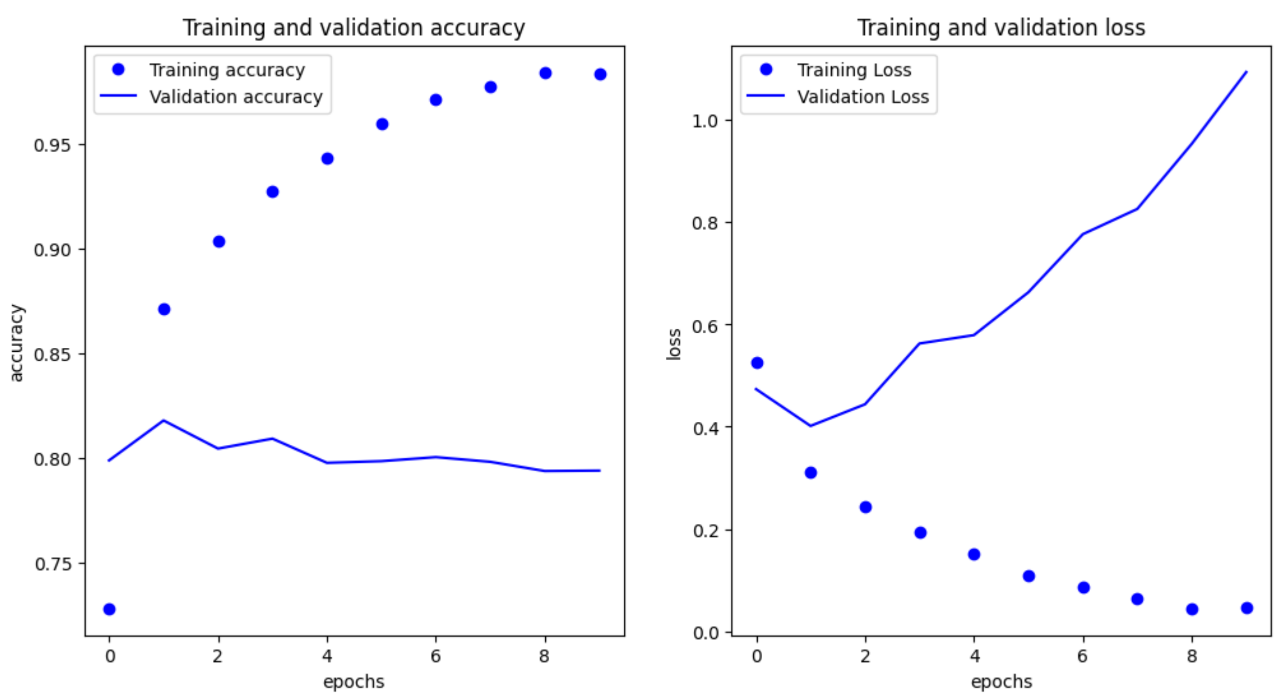
Trainable params: 172,941 (675.55 KB)

Non-trainable params: 0 (0.00 B)

In [15]:
```
NUM_EPOCHS = 10

# Train the model
history_lstm = model_lstm.fit(train_dataset_final, epochs=NUM_EPOCHS, validation_data=test_dataset_final)
```

```
Epoch 1/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 13s 13ms/step - accuracy: 0.6294 - loss: 0.6236 - val_accuracy: 0.7990 - val_loss: 0.4735
Epoch 2/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 10s 13ms/step - accuracy: 0.8586 - loss: 0.3385 - val_accuracy: 0.8181 - val_loss: 0.4015
Epoch 3/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 10s 13ms/step - accuracy: 0.8964 - loss: 0.2591 - val_accuracy: 0.8046 - val_loss: 0.4438
Epoch 4/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 10s 13ms/step - accuracy: 0.9223 - loss: 0.2070 - val_accuracy: 0.8094 - val_loss: 0.5628
Epoch 5/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 10s 13ms/step - accuracy: 0.9374 - loss: 0.1633 - val_accuracy: 0.7978 - val_loss: 0.5791
Epoch 6/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 10s 13ms/step - accuracy: 0.9558 - loss: 0.1177 - val_accuracy: 0.7986 - val_loss: 0.6630
Epoch 7/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 10s 13ms/step - accuracy: 0.9693 - loss: 0.0895 - val_accuracy: 0.8006 - val_loss: 0.7766
Epoch 8/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 10s 13ms/step - accuracy: 0.9769 - loss: 0.0677 - val_accuracy: 0.7983 - val_loss: 0.8260
Epoch 9/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 10s 13ms/step - accuracy: 0.9837 - loss: 0.0463 - val_accuracy: 0.7939 - val_loss: 0.9534
Epoch 10/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 10s 13ms/step - accuracy: 0.9827 - loss: 0.0511 - val_accuracy: 0.7941 - val_loss: 1.0937
```

In [16]:
```
# Plot the accuracy and loss history
plot_loss_acc(history_lstm)
```



GRU

The *Gated Recurrent Unit* or GRU is usually referred to as a simpler version of the LSTM. It can be used in applications where the sequence is important but you want faster results and can sacrifice some accuracy. You will notice in the model summary that it is a bit smaller than the LSTM and it also trains faster by a few seconds.

In [17]:
```
# Parameters
EMBEDDING_DIM = 16
GRU_DIM = 32
DENSE_DIM = 6

# Model Definition with GRU
model_gru = tf.keras.Sequential([
    tf.keras.Input(shape=(MAX_LENGTH,)),
    tf.keras.layers.Embedding(input_dim=VOCAB_SIZE, output_dim=EMBEDDING_DIM),
    tf.keras.layers.Bidirectional(tf.keras.layers.GRU(GRU_DIM)),
    tf.keras.layers.Dense(DENSE_DIM, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

# Set the training parameters
model_gru.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accuracy'])

# Print the model summary
model_gru.summary()
```

Model: "sequential_2"

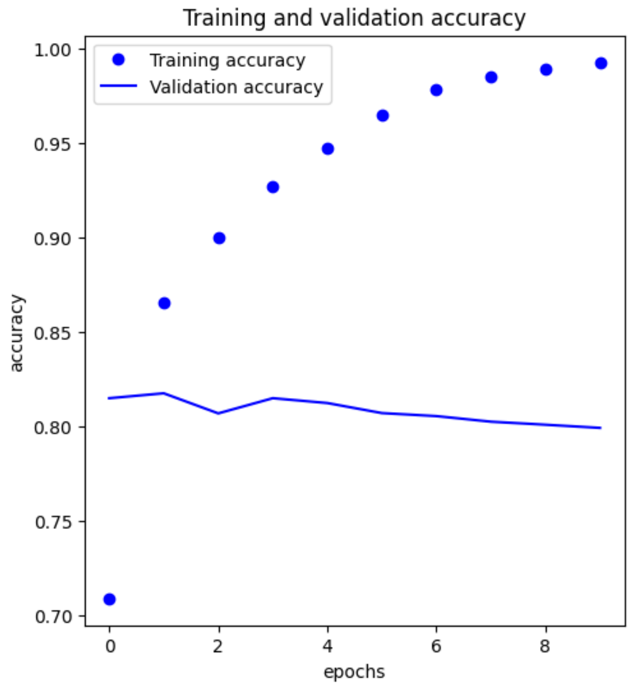
Layer (type)	Output Shape	Param #
embedding_2 (Embedding)	(None, 120, 16)	160,000
bidirectional_1 (Bidirectional)	(None, 64)	9,600
dense_4 (Dense)	(None, 6)	390
dense_5 (Dense)	(None, 1)	7

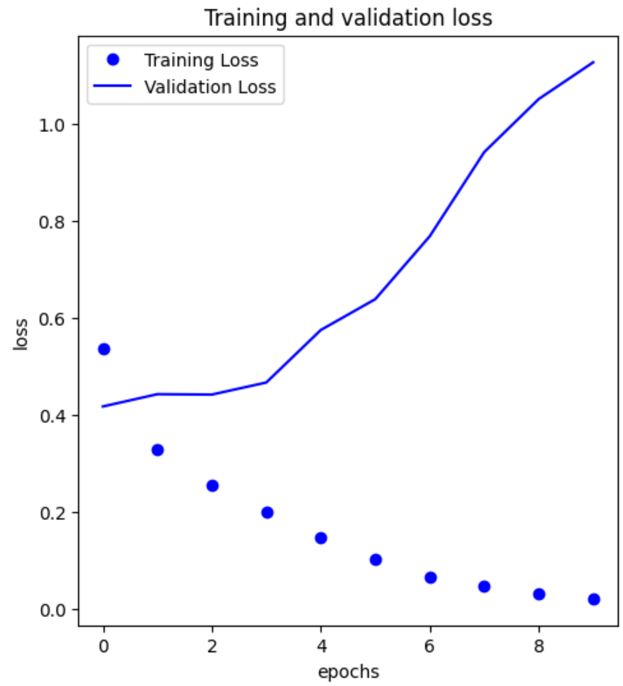
Total params: 169,997 (664.05 KB)
Trainable params: 169,997 (664.05 KB)
Non-trainable params: 0 (0.00 B)

In [18]:
```
NUM_EPOCHS = 10

# Train the model
history_gru = model_gru.fit(train_dataset_final, epochs=NUM_EPOCHS, validation_data=(test_dataset_final))
```

```
Epoch 1/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 11s 13ms/step - accuracy: 0.6004 - loss: 0.6286 - val_accuracy: 0.8150 - val_loss: 0.4164
Epoch 2/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 10s 13ms/step - accuracy: 0.8467 - loss: 0.3650 - val_accuracy: 0.8176 - val_loss: 0.4418
Epoch 3/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 10s 12ms/step - accuracy: 0.8897 - loss: 0.2760 - val_accuracy: 0.8070 - val_loss: 0.4410
Epoch 4/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 10s 12ms/step - accuracy: 0.9182 - loss: 0.2180 - val_accuracy: 0.8150 - val_loss: 0.4659
Epoch 5/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 10s 13ms/step - accuracy: 0.9401 - loss: 0.1651 - val_accuracy: 0.8124 - val_loss: 0.5741
Epoch 6/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 10s 12ms/step - accuracy: 0.9618 - loss: 0.1125 - val_accuracy: 0.8071 - val_loss: 0.6377
Epoch 7/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 10s 12ms/step - accuracy: 0.9764 - loss: 0.0733 - val_accuracy: 0.8055 - val_loss: 0.7675
Epoch 8/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 10s 12ms/step - accuracy: 0.9843 - loss: 0.0492 - val_accuracy: 0.8026 - val_loss: 0.9410
Epoch 9/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 10s 12ms/step - accuracy: 0.9886 - loss: 0.0331 - val_accuracy: 0.8010 - val_loss: 1.0504
Epoch 10/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 10s 12ms/step - accuracy: 0.9924 - loss: 0.0213 - val_accuracy: 0.7993 - val_loss: 1.1262
```

In [19]:
```
# Plot the accuracy and loss history
plot_loss_acc(history_gru)
```





Convolution

Lastly, you will use a convolution layer to extract features from your dataset. You will append a GlobalAveragePooling1D layer to reduce the results before passing it on to the dense layers. Like the model with `Flatten`, this also trains much faster than the ones using RNN layers like `LSTM` and `GRU`.

In [20]:
```
# Parameters
EMBEDDING_DIM = 16
FILTERS = 128
KERNEL_SIZE = 5
DENSE_DIM = 6

# Model Definition with Conv1D
model_conv = tf.keras.Sequential([
    tf.keras.Input(shape=(MAX_LENGTH,)),
    tf.keras.layers.Embedding(input_dim=VOCAB_SIZE, output_dim=EMBEDDING_DIM),
    tf.keras.layers.Conv1D(FILTERS, KERNEL_SIZE, activation='relu'),
    tf.keras.layers.GlobalAveragePooling1D(),
    tf.keras.layers.Dense(DENSE_DIM, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

# Set the training parameters
model_conv.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accuracy'])

# Print the model summary
model_conv.summary()
```

Model: "sequential_3"

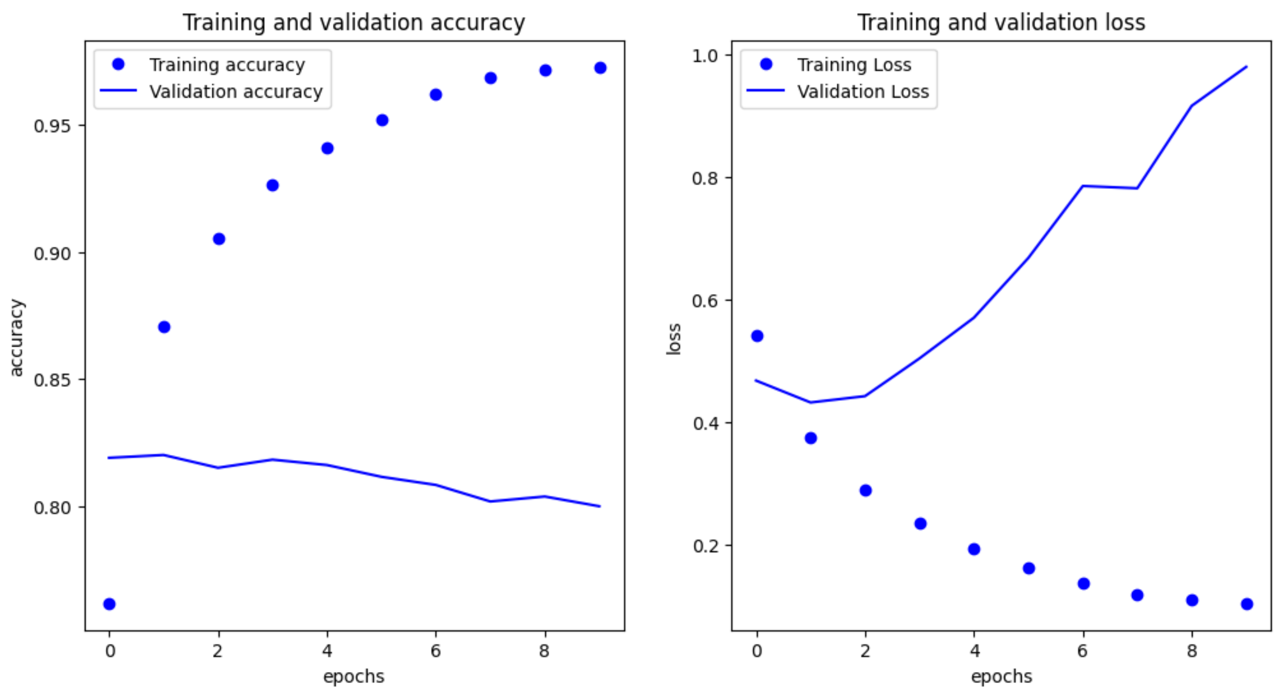
Layer (type)	Output Shape	Param #
embedding_3 (Embedding)	(None, 120, 16)	160,000
conv1d (Conv1D)	(None, 116, 128)	10,368
global_average_pooling1d (GlobalAveragePooling1D)	(None, 128)	0
dense_6 (Dense)	(None, 6)	774
dense_7 (Dense)	(None, 1)	7

Total params: 171,149 (668.55 KB)
Trainable params: 171,149 (668.55 KB)
Non-trainable params: 0 (0.00 B)

In [21]:
```python
NUM_EPOCHS = 10

# Train the model
history_conv = model_conv.fit(train_dataset_final, epochs=NUM_EPOCHS, validation_data=(test_dataset_final))
```

```
Epoch 1/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.6696 - loss: 0.6132 - val_accuracy: 0.8189 - val_loss: 0.4681
Epoch 2/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8679 - loss: 0.3945 - val_accuracy: 0.8200 - val_loss: 0.4324
Epoch 3/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9028 - loss: 0.3015 - val_accuracy: 0.8150 - val_loss: 0.4428
Epoch 4/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9245 - loss: 0.2412 - val_accuracy: 0.8182 - val_loss: 0.5042
Epoch 5/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9402 - loss: 0.1974 - val_accuracy: 0.8161 - val_loss: 0.5706
Epoch 6/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9505 - loss: 0.1687 - val_accuracy: 0.8114 - val_loss: 0.6681
Epoch 7/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9625 - loss: 0.1380 - val_accuracy: 0.8082 - val_loss: 0.7854
Epoch 8/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9697 - loss: 0.1182 - val_accuracy: 0.8017 - val_loss: 0.7817
Epoch 9/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9737 - loss: 0.1058 - val_accuracy: 0.8037 - val_loss: 0.9161
Epoch 10/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9745 - loss: 0.1001 - val_accuracy: 0.7998 - val_loss: 0.9796
```

In [22]:
```python
# Plot the accuracy and loss history
plot_loss_acc(history_conv)
```



Wrap Up

Now that you've seen the results for each model, can you make a recommendation on what works best for this dataset? Do you still get the same results if you tweak some hyperparameters like the vocabulary size? Try tweaking some of the values some more so you can get more insight on what model performs best.

Run the cell below to free up resources for the next lab

In [23]:
```python
# Shutdown the kernel to free up resources.
# Note: You can expect a pop-up when you run this cell. You can safely ignore that and just press `Ok`.

from IPython import get_ipython

k = get_ipython().kernel

k.do_shutdown(restart=False)
```

Out[23]: `{'status': 'ok', 'restart': False}`