Ungraded Lab: Single Layer LSTM

So far in this course, you've been using mostly basic dense layers and embeddings to build your models. It detects how the combination of words (or subwords) in the input text determines the output class. In the labs this week, you will look at other layers you can use to build your models. Most of these will deal with *Recurrent Neural Networks*, a kind of model that takes the ordering of inputs into account. This makes it suitable for different applications such as parts-of-speech tagging, music composition, language translation, and the like. For example, you may want your model to differentiate sentiments even if the words used in two sentences are the same:

```
1: My friends do like the movie but I don't. --> negative review
2: My friends don't like the movie but I do. --> positive review
```

The first layer you will be looking at is the *LSTM (Long Short-Term Memory)*. In a nutshell, it computes the state of a current timestep and passes it on to the next timesteps where this state is also updated. The process repeats until the final timestep where the output computation is affected by all previous states. Not only that, it can be configured to be bidirectional so you can get the relationship of later words to earlier ones. If you want to go in-depth of how these processes work, you can look at the Sequence Models course of the Deep Learning Specialization. For this lab, you can take advantage of Tensorflow's APIs that implements the complexities of these layers for you. This makes it easy to just plug it in to your model. Let's see how to do that in the next sections below.

Imports

Start by installing and importing the required packages.

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

Load the dataset

You will load the IMDB Reviews dataset via Tensorflow Datasets as you've done last week:

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)
```

Then, you will separate the reviews and labels.

In [3]:
```
# Extract the train reviews and labels
train_reviews = imdb['train'].map(lambda review, label: review)
train_labels = imdb['train'].map(lambda review, label: label)

# Extract the test reviews and labels
test_reviews = imdb['test'].map(lambda review, label: review)
test_labels = imdb['test'].map(lambda review, label: label)
```

Prepare the dataset

You will use subword tokenization in this lab. We'll provide the vocabulary text file already so you won't need to generate it yourself.

In [4]:
```
# Download the subword vocabulary (not needed in Coursera)
# !wget -nc https://storage.googleapis.com/tensorflow-1-public/course3/imdb_vocab_subwords.txt
```

You can just pass this directly to the `WordPieceTokenizer` class to instantiate the tokenizer.

In [5]:
```
# Initialize the subword tokenizer
subword_tokenizer = keras_nlp.tokenizers.WordPieceTokenizer(
    vocabulary='./imdb_vocab_subwords.txt'
)
```

You can then get the train and test splits and generate padded sequences.

Note: To make the training go faster in this lab, you will increase the batch size that Laurence used in the lecture. In particular, you will use `256` *and this takes roughly a minute to train per epoch. In the video, Laurence used* `16` *which takes around 4 minutes per epoch.*

In [6]:
```
# Data pipeline and padding parameters
SHUFFLE_BUFFER_SIZE = 10000
PREFETCH_BUFFER_SIZE = tf.data.AUTOTUNE
BATCH_SIZE = 256
```

```
PADDING_TYPE = 'pre'
TRUNC_TYPE = 'post'
```

In [7]:
```
def padding_func(sequences):
  '''Generates padded sequences from a tf.data.Dataset'''

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

  # Output a tensor from the single batch
  sequences = sequences.get_single_element()

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

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

  return padded_sequences
```

In [8]:
```
# Generate integer sequences using the subword tokenizer
train_sequences_subword = train_reviews.map(lambda review: subword_tokenizer.tokenize(review)).apply(padding_func)
test_sequences_subword = test_reviews.map(lambda review: subword_tokenizer.tokenize(review)).apply(padding_func)

# Combine the integer sequence and labels
train_dataset_vectorized = tf.data.Dataset.zip(train_sequences_subword,train_labels)
test_dataset_vectorized = tf.data.Dataset.zip(test_sequences_subword,test_labels)

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

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

Build and compile the model

Now you will build the model. You will simply swap the `Flatten` or `GlobalAveragePooling1D` from before with an `LSTM` layer. Moreover, you will nest it inside a Biderectional layer so the passing of the sequence information goes both forwards and backwards. These additional computations will naturally make the training go slower than the models you built last week. You should take this into account when using RNNs in your own applications.

In [9]:
```
# Model Parameters
EMBEDDING_DIM = 64
LSTM_DIM = 64
DENSE_DIM = 64

# Build the model
model = tf.keras.Sequential([
    tf.keras.Input(shape=(None,)),
    tf.keras.layers.Embedding(subword_tokenizer.vocabulary_size(), 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')
])

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

```
Model: "sequential"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━┓
┃ Layer (type)                       ┃ Output Shape             ┃       Param #  ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━┩
│ embedding (Embedding)              │ (None, None, 64)         │       488,640  │
├────────────────────────────────────┼──────────────────────────┼────────────────┤
│ bidirectional (Bidirectional)      │ (None, 128)              │        66,048  │
├────────────────────────────────────┼──────────────────────────┼────────────────┤
│ dense (Dense)                      │ (None, 64)               │         8,256  │
├────────────────────────────────────┼──────────────────────────┼────────────────┤
│ dense_1 (Dense)                    │ (None, 1)                │            65  │
└────────────────────────────────────┴──────────────────────────┴────────────────┘

 Total params: 563,009 (2.15 MB)
```

```
Trainable params: 563,009 (2.15 MB)
Non-trainable params: 0 (0.00 B)
```

In [10]:
```
# Set the training parameters
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

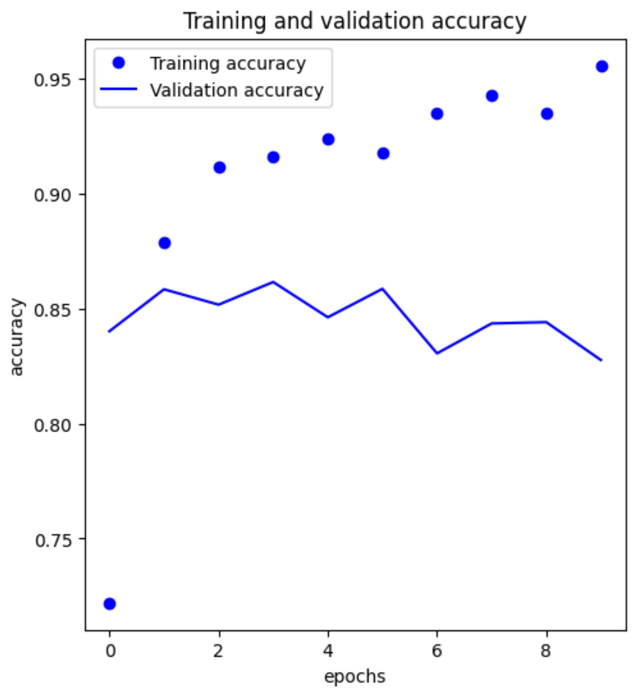
Train the model

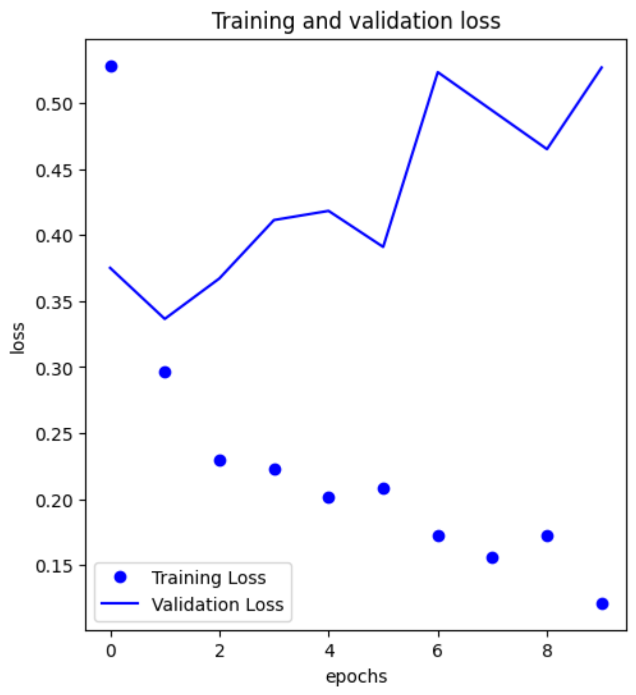
Now you can start training. Using the default parameters above, you should reach around 95% training accuracy and 84% validation accuracy. You can visualize the results using the same plot utilities. See if you can still improve on this by modifying the hyperparameters or by training with more epochs.

In [11]:
```
NUM_EPOCHS = 10

history = model.fit(train_dataset_final, epochs=NUM_EPOCHS, validation_data=test_dataset_final)
```

```
Epoch 1/10
98/98 ━━━━━━━━━━━━━━━━━━━━ 39s 355ms/step - accuracy: 0.6278 - loss: 0.6273 - val_accuracy: 0.8401 - val_loss: 0.3751
Epoch 2/10
98/98 ━━━━━━━━━━━━━━━━━━━━ 34s 350ms/step - accuracy: 0.8608 - loss: 0.3301 - val_accuracy: 0.8583 - val_loss: 0.3365
Epoch 3/10
98/98 ━━━━━━━━━━━━━━━━━━━━ 34s 349ms/step - accuracy: 0.9042 - loss: 0.2451 - val_accuracy: 0.8516 - val_loss: 0.3671
Epoch 4/10
98/98 ━━━━━━━━━━━━━━━━━━━━ 34s 349ms/step - accuracy: 0.9198 - loss: 0.2164 - val_accuracy: 0.8615 - val_loss: 0.4114
Epoch 5/10
98/98 ━━━━━━━━━━━━━━━━━━━━ 34s 349ms/step - accuracy: 0.9168 - loss: 0.2150 - val_accuracy: 0.8462 - val_loss: 0.4183
Epoch 6/10
98/98 ━━━━━━━━━━━━━━━━━━━━ 34s 350ms/step - accuracy: 0.9251 - loss: 0.1912 - val_accuracy: 0.8585 - val_loss: 0.3910
Epoch 7/10
98/98 ━━━━━━━━━━━━━━━━━━━━ 34s 349ms/step - accuracy: 0.9285 - loss: 0.1863 - val_accuracy: 0.8305 - val_loss: 0.5233
Epoch 8/10
98/98 ━━━━━━━━━━━━━━━━━━━━ 34s 348ms/step - accuracy: 0.9354 - loss: 0.1716 - val_accuracy: 0.8435 - val_loss: 0.4942
Epoch 9/10
98/98 ━━━━━━━━━━━━━━━━━━━━ 34s 349ms/step - accuracy: 0.9478 - loss: 0.1437 - val_accuracy: 0.8440 - val_loss: 0.4650
Epoch 10/10
98/98 ━━━━━━━━━━━━━━━━━━━━ 34s 349ms/step - accuracy: 0.9545 - loss: 0.1255 - val_accuracy: 0.8276 - val_loss: 0.5268
```

In [12]:
```
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']
  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()

plot_loss_acc(history)
```





Wrap Up

In this lab, you got a first look at using LSTM layers to build Recurrent Neural Networks. You only used a single LSTM layer but this can be stacked as well to build deeper networks. You will see how to do that in the next lab. Before doing so, run the cell below to free up resources for the next lab. You might see a pop-up about restarting the kernel afterwards. You can safely ignore it and just press `Ok` . You can then close this lab, then go back to the classroom. See you there!

```
In [13]: # 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[13]: {'status': 'ok', 'restart': False}
```