Using RNNs to classify sentiment on IMDB data

In this assignment,you will train three types of RNNs: "vanilla" RNN, LSTM and GRU to predict the sentiment on IMDB reviews.

Keras provides a convenient interface to load the data and immediately encode the words into integers (based on the most common words). This will save you a lot of the drudgery that is usually involved when working with raw text.

The IMDB is data consists of 25000 training sequences and 25000 test sequences. The outcome is binary (positive/negative) and both outcomes are equally represented in both the training and the test set.

Walk through the followinng steps to prepare the data and the building of an RNN model.

```
import os

import numpy as np
import tensorflow as tf
from sklearn.datasets import load_files
import numpy as np

import tensorflow_datasets as tfds
import tensorflow as tf

import matplotlib.pyplot as plt

from tensorflow.keras import layers

# Optuna imports
import optuna
from keras.backend import clear_session


from tensorflow.keras.datasets import imdb
from tensorflow.keras.preprocessing import sequence
```

```
C:\Users\btb51\anaconda3\envs\DAAN570_tf_updated\lib\site-packages\
tqdm\auto.py:22: TqdmWarning: IProgress not found. Please update
jupyter and ipywidgets. See
https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
```

```
'''
# Old item when using the Prefetch datasets (element_spec cell)

dataset, info = tfds.load('imdb_reviews', with_info=True,
                          as_supervised=True)
train_dataset, test_dataset = dataset['train'], dataset['test']
```

```
train_dataset.element_spec
'''
```

```
"\n# Old item when using the Prefetch datasets (element_spec cell)\n\ndataset, info = tfds.load('imdb_reviews', with_info=True,\nas_supervised=True)\ntrain_dataset, test_dataset = dataset['train'],
dataset['test']\n\ntrain_dataset.element_spec\n"
```

```
'''
I am not using this dataset loader.  I am using the one found below.


dataset = tf.keras.utils.get_file(
    fname="aclImdb.tar.gz",

origin="http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz
",
    extract=True,
)

# set path to dataset
IMDB_DATADIR = os.path.join(os.path.dirname(dataset), "aclImdb")

classes = ["pos", "neg"]
train_data = load_files(
    os.path.join(IMDB_DATADIR, "train"), shuffle=True,
categories=classes
)
test_data = load_files(
    os.path.join(IMDB_DATADIR, "test"), shuffle=False,
categories=classes
)

x_train = np.array(train_data.data)
y_train = np.array(train_data.target)
x_test = np.array(test_data.data)
y_test = np.array(test_data.target)

print(x_train.shape)  # (25000,)
print(y_train.shape)  # (25000, 1)
print(x_train[0][:50])  # this film was just brilliant casting

'''
```

```
'\nI am not using this dataset loader.  I am using the one found
below.\n\n\ndataset = tf.keras.utils.get_file(\n
fname="aclImdb.tar.gz",\n
origin="http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz
",\n    extract=True,\n)\n\n# set path to dataset\nIMDB_DATADIR =
os.path.join(os.path.dirname(dataset), "aclImdb")\n\nclasses = ["pos",
"neg"]\ntrain_data = load_files(\n    os.path.join(IMDB_DATADIR,
```

```
"train"), shuffle=True, categories=classes\n)\ntest_data =
load_files(\n    os.path.join(IMDB_DATADIR, "test"), shuffle=False,
categories=classes\n)\n\nx_train = np.array(train_data.data)\ny_train
= np.array(train_data.target)\nx_test = np.array(test_data.data)\
ny_test = np.array(test_data.target)\n\nprint(x_train.shape)  #
(25000,)\nprint(y_train.shape)  # (25000, 1)\nprint(x_train[0][:50])
# this film was just brilliant casting\n\n'

# Old item when using the Prefetch datasets (element_spec cell)
# type(train_dataset)
```

1- Use the `imdb.load_data()` to load in the data

2- Specify the maximum length of a sequence to 30 words and the pick the 2000 most common words.

```
vocab_size = 2000  # number of words to consider as features
max_len = 30  # cut texts after this number of words (among top
max_features most common words)
batch_size = 32

# This data is already in integer form and does not need the
TextVectorization layer
'''
encoder = tf.keras.layers.TextVectorization(
    max_tokens=VOCAB_SIZE)

encoder.adapt(train_dataset.map(lambda text, label: text))
'''
(X_train, y_train), (X_test, y_test) =
imdb.load_data(num_words=vocab_size)

# Old item when using the Prefetch datasets (element_spec cell)

# vocab_size = 2000

# encoder = tf.keras.layers
```

3- Check that the number of sequences in train and test datasets are equal (default split):

Expected output:

- `x_train = 25000 train sequences`
- `x_test = 25000 test sequences`

```
print(len(X_train), 'train sequences')
print(len(X_test), 'test sequences')
```

```
25000 train sequences
25000 test sequences
```

4- Pad (or truncate) the sequences so that they are of the maximum length

```
X_train = sequence.pad_sequences(X_train, maxlen=max_len)
X_test = sequence.pad_sequences(X_test, maxlen=max_len)
```

5- After padding or truncating, check the dimensionality of x_train and x_test.

Expected output:

- `x_train shape: (25000, 30)`
- `x_test shape: (25000, 30)`

```
print('input_train shape:', X_train.shape)
print('input_test shape:', X_test.shape)

input_train shape: (25000, 30)
input_test shape: (25000, 30)
```

Keras layers for (Vanilla) RNNs

In this step, you will not use pre-trained word vectors, Instead you will learn an embedding as part of the the Vanilla) RNNs network Neural Network.

In the Keras API documentation, the Embedding Layer and the SimpleRNN Layer have the following syntax:

Embedding Layer

```
keras.layers.embeddings.Embedding(input_dim, output_dim,
embeddings_initializer='uniform', embeddings_regularizer=None,
activity_regularizer=None, embeddings_constraint=None,
mask_zero=False, input_length=None)
```

- This layer maps each integer into a distinct (dense) word vector of length `output_dim`.
- Can think of this as learning a word vector embedding "on the fly" rather than using an existing mapping (like GloVe)
- The `input_dim` should be the size of the vocabulary.
- The `input_length` specifies the length of the sequences that the network expects.

SimpleRNN Layer

```
keras.layers.recurrent.SimpleRNN(units, activation='tanh',
use_bias=True, kernel_initializer='glorot_uniform',
recurrent_initializer='orthogonal', bias_initializer='zeros',
kernel_regularizer=None, recurrent_regularizer=None,
bias_regularizer=None, activity_regularizer=None,
```

```
kernel_constraint=None, recurrent_constraint=None,
bias_constraint=None, dropout=0.0, recurrent_dropout=0.0)
```

- This is the basic RNN, where the output is also fed back as the "hidden state" to the next iteration.
- The parameter `units` gives the dimensionality of the output (and therefore the hidden state). Note that typically there will be another layer after the RNN mapping the (RNN) output to the network output. So we should think of this value as the desired dimensionality of the hidden state and not necessarily the desired output of the network.
- Recall that there are two sets of weights, one for the "recurrent" phase and the other for the "kernel" phase. These can be configured separately in terms of their initialization, regularization, etc.

6- Build the RNN with three layers:

- The SimpleRNN layer with 5 neurons and initialize its kernel with stddev=0.001
- The Embedding layer and initialize it by setting the word embedding dimension to 50. This means that this layer takes each integer in the sequence and embeds it in a 50-dimensional vector.
- The output layer has the sigmoid activation function.

```
from tensorflow.keras.layers import Dense

# Define the RNN
# I am not putting an encoder layer in this network.  I have seen some
that do that though

def model_RNN(vocab_size = 2000, seq_length=30):

    model = tf.keras.Sequential([
        # possition of would be text_encoder layer if used
        layers.Embedding(input_dim=vocab_size, output_dim=seq_length),
        layers.SimpleRNN(5, kernel_initializer='glorot_uniform'),
        layers.Dense(1, activation='sigmoid')
    ])

    return model
```

7- How many parameters have the embedding layer?

```
# Create the network
rnn_net = model_RNN(vocab_size=2000, seq_length=30)

rnn_net.summary()
```

```
print("By the summary, there are 60,000 parameters in the embedding
layer.")

Model: "sequential_7"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_7 (Embedding)     (None, None, 30)          60000

 simple_rnn_3 (SimpleRNN)    (None, 5)                 180

 dense_7 (Dense)             (None, 1)                 6

=================================================================
Total params: 60,186
Trainable params: 60,186
Non-trainable params: 0
_________________________________________________________________
By the summary, there are 60,000 parameters in the embedding layer.
```

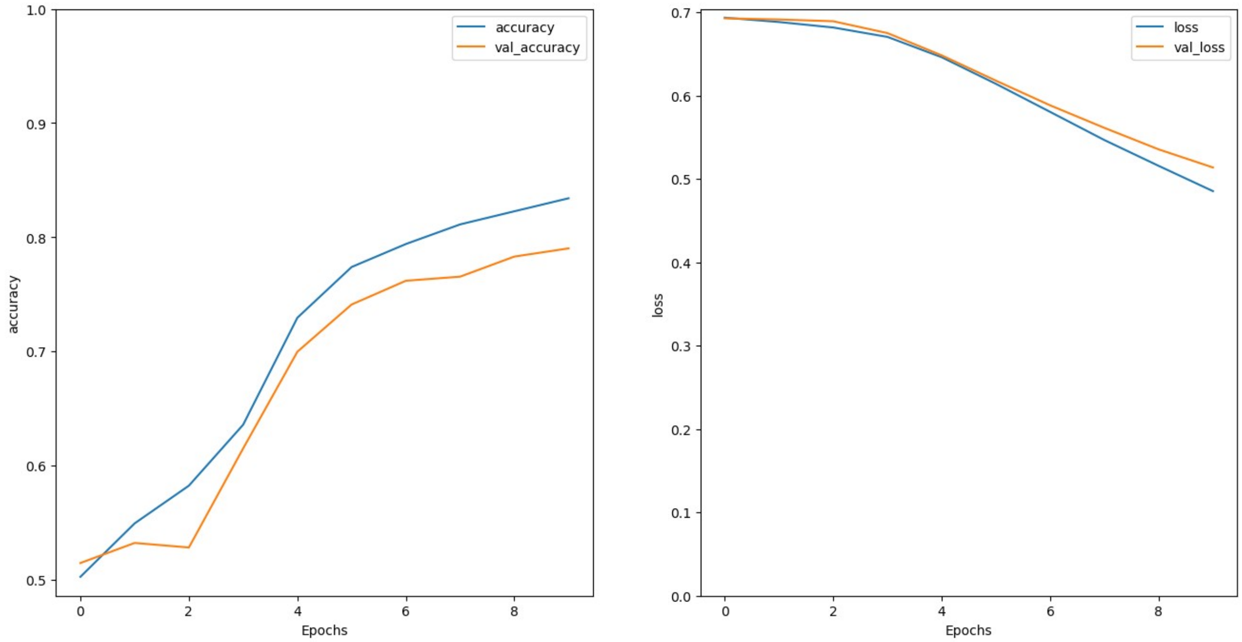
8- Train the network with the RMSprop with learning rate of .0001 and epochs=10.

```
# Compile the RNN
optimizer = tf.keras.optimizers.RMSprop(learning_rate=0.0001)

loss_fn = tf.keras.losses.BinaryCrossentropy(from_logits=False)

mets = ['accuracy']

rnn_net.compile(optimizer, loss_fn, mets)

# train the RNN

history_rnn = rnn_net.fit(X_train, y_train, epochs = 10,
batch_size=batch_size, validation_split=0.2)

Epoch 1/10
625/625 [==============================] - 86s 136ms/step - loss:
0.6935 - accuracy: 0.5023 - val_loss: 0.6927 - val_accuracy: 0.5144
Epoch 2/10
625/625 [==============================] - 85s 135ms/step - loss:
0.6883 - accuracy: 0.5491 - val_loss: 0.6913 - val_accuracy: 0.5320
Epoch 3/10
625/625 [==============================] - 84s 135ms/step - loss:
0.6817 - accuracy: 0.5822 - val_loss: 0.6892 - val_accuracy: 0.5280
Epoch 4/10
625/625 [==============================] - 84s 134ms/step - loss:
0.6704 - accuracy: 0.6356 - val_loss: 0.6750 - val_accuracy: 0.6148
Epoch 5/10
625/625 [==============================] - 85s 136ms/step - loss:
```

```
0.6460 - accuracy: 0.7294 - val_loss: 0.6483 - val_accuracy: 0.6996
Epoch 6/10
625/625 [==============================] - 84s 135ms/step - loss:
0.6141 - accuracy: 0.7739 - val_loss: 0.6179 - val_accuracy: 0.7410
Epoch 7/10
625/625 [==============================] - 84s 135ms/step - loss:
0.5804 - accuracy: 0.7940 - val_loss: 0.5883 - val_accuracy: 0.7618
Epoch 8/10
625/625 [==============================] - 84s 134ms/step - loss:
0.5466 - accuracy: 0.8112 - val_loss: 0.5613 - val_accuracy: 0.7654
Epoch 9/10
625/625 [==============================] - 84s 134ms/step - loss:
0.5157 - accuracy: 0.8227 - val_loss: 0.5355 - val_accuracy: 0.7830
Epoch 10/10
625/625 [==============================] - 85s 136ms/step - loss:
0.4854 - accuracy: 0.8341 - val_loss: 0.5138 - val_accuracy: 0.7902
```

9- Plot the loss and accuracy metrics during the training and interpret the result.

```python
# Plotting Function
def plot_graphs(history, metric):
  plt.plot(history.history[metric])
  plt.plot(history.history['val_'+metric], '')
  plt.xlabel("Epochs")
  plt.ylabel(metric)
  plt.legend([metric, 'val_'+metric])

#%%
plt.figure(figsize=(16, 8))
plt.subplot(1, 2, 1)
plot_graphs(history_rnn, 'accuracy')
plt.ylim(None, 1)
plt.subplot(1, 2, 2)
plot_graphs(history_rnn, 'loss')
plt.ylim(0, None)

(0.0, 0.7038673728704452)
```



10- Check the accuracy and the loss of your models on the test dataset.

```
test_loss, test_acc = rnn_net.evaluate(X_test, y_test)

print('Test Loss:', test_loss)
print('Test Accuracy:', test_acc)

782/782 [==============================] - 8s 10ms/step - loss: 0.7003
- accuracy: 0.5363
Test Loss: 0.7003328800201416
Test Accuracy: 0.5363199710845947
```

Hmm based on the test values it may be that we are overfitting on the training dataset... or that we don't have enough words to choose from in the vocab list in the test data.

Tuning The Vanilla RNN Network

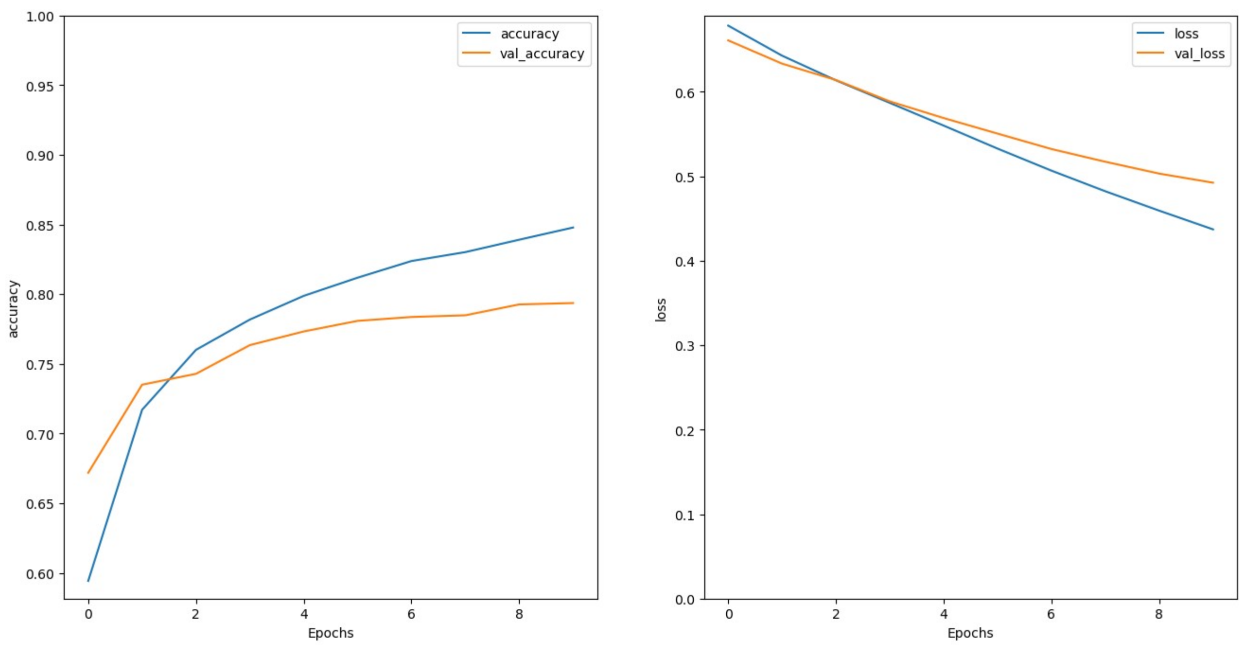
11- Prepare the data to use sequences of length 80 rather than length 30 and retrain your model. Did it improve the performance?

```
X_train_80 = sequence.pad_sequences(X_train, maxlen=80)
X_test_80 = sequence.pad_sequences(X_test, maxlen=80)
print('input_train shape:', X_train_80.shape)
print('input_test shape:', X_test_80.shape)

input_train shape: (25000, 80)
input_test shape: (25000, 80)
```

```
# Create the network
rnn_net2 = model_RNN(vocab_size=2000, seq_length=30)
rnn_net2.summary()

# Compile the RNN
optimizer = tf.keras.optimizers.RMSprop(learning_rate=0.0001)
loss_fn = tf.keras.losses.BinaryCrossentropy(from_logits=False)
mets = ['accuracy']
rnn_net2.compile(optimizer, loss_fn, mets)

# train the RNN

history_rnn_net2= rnn_net2.fit(X_train_80, y_train, epochs = 10,
batch_size=batch_size, validation_split=0.2)
```

```
Model: "sequential_8"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_8 (Embedding)     (None, None, 30)          60000

 simple_rnn_4 (SimpleRNN)    (None, 5)                 180

 dense_8 (Dense)             (None, 1)                 6

=================================================================
Total params: 60,186
Trainable params: 60,186
Non-trainable params: 0
_________________________________________________________________
Epoch 1/10
625/625 [==============================] - 88s 140ms/step - loss:
0.6784 - accuracy: 0.5942 - val_loss: 0.6609 - val_accuracy: 0.6718
Epoch 2/10
625/625 [==============================] - 87s 139ms/step - loss:
0.6428 - accuracy: 0.7170 - val_loss: 0.6333 - val_accuracy: 0.7350
Epoch 3/10
625/625 [==============================] - 87s 140ms/step - loss:
0.6136 - accuracy: 0.7600 - val_loss: 0.6139 - val_accuracy: 0.7428
Epoch 4/10
625/625 [==============================] - 86s 138ms/step - loss:
0.5869 - accuracy: 0.7818 - val_loss: 0.5886 - val_accuracy: 0.7634
Epoch 5/10
625/625 [==============================] - 87s 139ms/step - loss:
0.5600 - accuracy: 0.7987 - val_loss: 0.5689 - val_accuracy: 0.7732
Epoch 6/10
625/625 [==============================] - 87s 139ms/step - loss:
0.5327 - accuracy: 0.8118 - val_loss: 0.5505 - val_accuracy: 0.7808
Epoch 7/10
625/625 [==============================] - 87s 139ms/step - loss:
```

```
0.5067 - accuracy: 0.8238 - val_loss: 0.5322 - val_accuracy: 0.7836
Epoch 8/10
625/625 [==============================] - 87s 139ms/step - loss:
0.4822 - accuracy: 0.8302 - val_loss: 0.5172 - val_accuracy: 0.7848
Epoch 9/10
625/625 [==============================] - 87s 139ms/step - loss:
0.4593 - accuracy: 0.8390 - val_loss: 0.5032 - val_accuracy: 0.7926
Epoch 10/10
625/625 [==============================] - 87s 139ms/step - loss:
0.4371 - accuracy: 0.8478 - val_loss: 0.4924 - val_accuracy: 0.7936

def quick_plot(history):

    plt.figure(figsize=(16, 8))
    plt.subplot(1, 2, 1)
    plot_graphs(history, 'accuracy')
    plt.ylim(None, 1)
    plt.subplot(1, 2, 2)
    plot_graphs(history, 'loss')
    plt.ylim(0, None)

quick_plot(history_rnn_net2)
```



Very similar to the previous model. Maybe slightly more overfitting near the end.

The overall lack of chagne could have been due to the embeddings layer not changing resulting in a similar output.

I'll test that here:

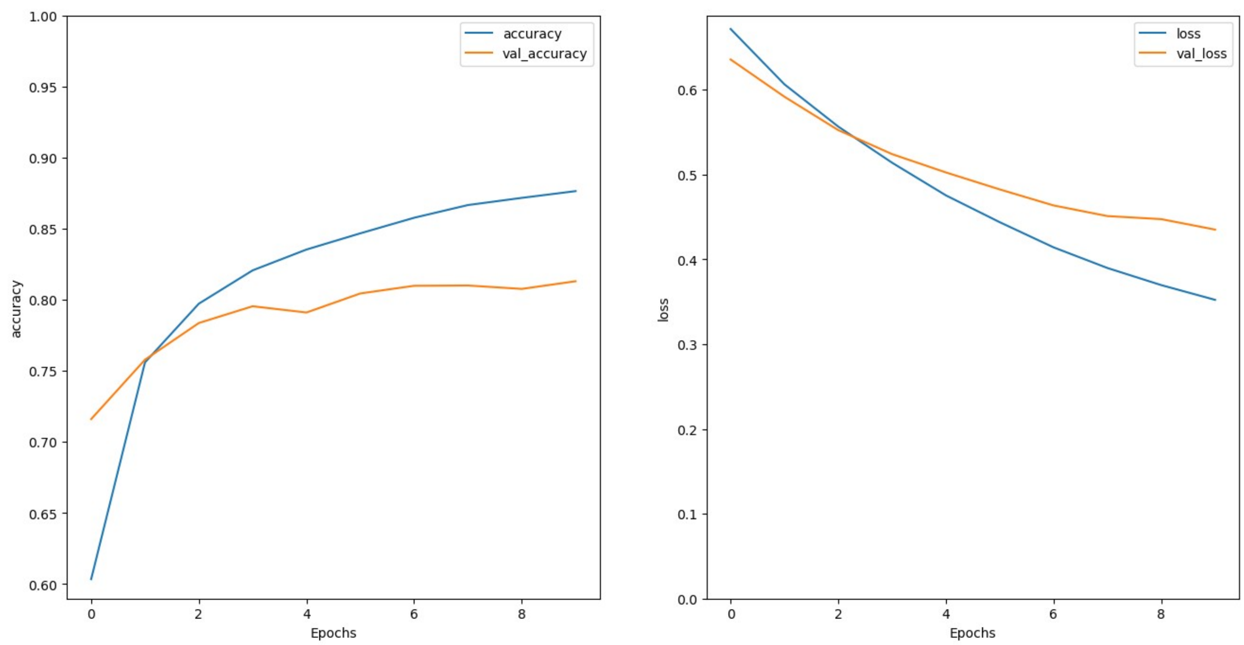
```
# Create the network
rnn_net2_80 = model_RNN(vocab_size=2000, seq_length=80)  # Note the
change of seq_length to 80
rnn_net2_80.summary()

# Compile the RNN
optimizer = tf.keras.optimizers.RMSprop(learning_rate=0.0001)
loss_fn = tf.keras.losses.BinaryCrossentropy(from_logits=False)
mets = ['accuracy']
rnn_net2_80.compile(optimizer, loss_fn, mets)

# train the RNN

history_rnn_net2_80= rnn_net2_80.fit(X_train_80, y_train, epochs = 10,
batch_size=batch_size, validation_split=0.2)
```

```
Model: "sequential_10"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_10 (Embedding)    (None, None, 80)          160000

 simple_rnn_6 (SimpleRNN)    (None, 5)                 430

 dense_10 (Dense)            (None, 1)                 6

=================================================================
Total params: 160,436
Trainable params: 160,436
Non-trainable params: 0
_________________________________________________________________
Epoch 1/10
625/625 [==============================] - 98s 155ms/step - loss:
0.6716 - accuracy: 0.6034 - val_loss: 0.6356 - val_accuracy: 0.7160
Epoch 2/10
625/625 [==============================] - 94s 150ms/step - loss:
0.6063 - accuracy: 0.7559 - val_loss: 0.5917 - val_accuracy: 0.7578
Epoch 3/10
625/625 [==============================] - 104s 167ms/step - loss:
0.5565 - accuracy: 0.7972 - val_loss: 0.5524 - val_accuracy: 0.7836
Epoch 4/10
625/625 [==============================] - 98s 157ms/step - loss:
0.5139 - accuracy: 0.8206 - val_loss: 0.5241 - val_accuracy: 0.7954
Epoch 5/10
625/625 [==============================] - 97s 155ms/step - loss:
0.4754 - accuracy: 0.8353 - val_loss: 0.5025 - val_accuracy: 0.7910
Epoch 6/10
625/625 [==============================] - 106s 169ms/step - loss:
0.4438 - accuracy: 0.8467 - val_loss: 0.4824 - val_accuracy: 0.8044
Epoch 7/10
```

```
625/625 [==============================] - 104s 167ms/step - loss:
0.4141 - accuracy: 0.8576 - val_loss: 0.4635 - val_accuracy: 0.8098
Epoch 8/10
625/625 [==============================] - 92s 147ms/step - loss:
0.3898 - accuracy: 0.8666 - val_loss: 0.4510 - val_accuracy: 0.8100
Epoch 9/10
625/625 [==============================] - 95s 153ms/step - loss:
0.3696 - accuracy: 0.8716 - val_loss: 0.4474 - val_accuracy: 0.8076
Epoch 10/10
625/625 [==============================] - 95s 152ms/step - loss:
0.3522 - accuracy: 0.8764 - val_loss: 0.4351 - val_accuracy: 0.8130
```

Note that changing the seq_length in the model created a much larger parameter space in the embedding layer. It is now at 160,000 compared to the 60,000 it was previously. The run times appear to be similar to before.

```
quick_plot(history_rnn_net2_80)
```



No substantial change is present here. We again may have some slight overfitting at later epochs.

12- Try different values of the maximum length of a sequence ("max_features") [known as vocab_length here]. Can you improve the performance?

```
# Build an Optuna Study to look at various sequence legnths
(vocab_lengths)

def objective_vocab_lengths(trial):
```

```
    clear_session()

    # Set the testing items for optuna
    vocab_size = trial.suggest_categorical('vocab_len', [500, 1000,
3000])  # number of words to consider as features

    max_len = 30  # cut texts after this number of words (among top
max_features most common words)
    batch_size = 32

    (X_train, y_train), (X_test, y_test) =
imdb.load_data(num_words=vocab_size)

    X_train = sequence.pad_sequences(X_train, maxlen=max_len)
    X_test = sequence.pad_sequences(X_test, maxlen=max_len)

    seq_length = 30

    model = tf.keras.Sequential([
        # possition of would be text_encoder layer if used
        layers.Embedding(input_dim=vocab_size, output_dim=seq_length),
# NOte vocab_size is being optimized here
        layers.SimpleRNN(5, kernel_initializer='glorot_uniform'),
        layers.Dense(1, activation='sigmoid')
    ])


    # Compile the RNN
    optimizer = tf.keras.optimizers.RMSprop(learning_rate=0.0001)
    loss_fn = tf.keras.losses.BinaryCrossentropy(from_logits=False)
    mets = ['accuracy']
    model.compile(optimizer, loss_fn, mets)


    # Shortened epochs to reduce time and based on prior runs that is
a fair sample size to get
    # an idea of what is going on
    history = model.fit(X_train, y_train, epochs = 5,
batch_size=batch_size, validation_split=0.2)

    eval_score = model.evaluate(X_test, y_test)

    return eval_score[1]



study_vocab = optuna.create_study(direction='maximize')

study_vocab.optimize(objective_vocab_lengths, n_trials = 6,
timeout=3600, gc_after_trial=True)
```

```
[I 2023-07-15 18:58:11,647] A new study created in memory with name:
no-name-475e4786-b49f-4d9a-ab1e-f095e1909c02

Epoch 1/5
625/625 [==============================] - 40s 63ms/step - loss:
0.6921 - accuracy: 0.5175 - val_loss: 0.6915 - val_accuracy: 0.5184
Epoch 2/5
625/625 [==============================] - 39s 62ms/step - loss:
0.6825 - accuracy: 0.5847 - val_loss: 0.6774 - val_accuracy: 0.6090
Epoch 3/5
625/625 [==============================] - 39s 62ms/step - loss:
0.6632 - accuracy: 0.6449 - val_loss: 0.6604 - val_accuracy: 0.6434
Epoch 4/5
625/625 [==============================] - 39s 62ms/step - loss:
0.6436 - accuracy: 0.6664 - val_loss: 0.6448 - val_accuracy: 0.6558
Epoch 5/5
625/625 [==============================] - 39s 62ms/step - loss:
0.6242 - accuracy: 0.6866 - val_loss: 0.6306 - val_accuracy: 0.6670
782/782 [==============================] - 8s 10ms/step - loss: 0.6270
- accuracy: 0.6741

[I 2023-07-15 19:01:39,335] Trial 0 finished with value:
0.6741200089454651 and parameters: {'vocab_len': 500}. Best is trial 0
with value: 0.6741200089454651.

Epoch 1/5
625/625 [==============================] - 41s 64ms/step - loss:
0.6896 - accuracy: 0.5379 - val_loss: 0.6790 - val_accuracy: 0.6050
Epoch 2/5
625/625 [==============================] - 39s 63ms/step - loss:
0.6614 - accuracy: 0.6455 - val_loss: 0.6526 - val_accuracy: 0.6554
Epoch 3/5
625/625 [==============================] - 36s 57ms/step - loss:
0.6346 - accuracy: 0.6805 - val_loss: 0.6324 - val_accuracy: 0.6734
Epoch 4/5
625/625 [==============================] - 37s 60ms/step - loss:
0.6095 - accuracy: 0.7063 - val_loss: 0.6131 - val_accuracy: 0.6904
Epoch 5/5
625/625 [==============================] - 36s 57ms/step - loss:
0.5867 - accuracy: 0.7229 - val_loss: 0.5964 - val_accuracy: 0.6984
782/782 [==============================] - 7s 8ms/step - loss: 0.5918
- accuracy: 0.7058

[I 2023-07-15 19:04:59,932] Trial 1 finished with value:
0.7057600021362305 and parameters: {'vocab_len': 1000}. Best is trial
1 with value: 0.7057600021362305.

Epoch 1/5
625/625 [==============================] - 37s 58ms/step - loss:
0.6845 - accuracy: 0.5519 - val_loss: 0.6611 - val_accuracy: 0.6254
```

```
Epoch 2/5
625/625 [==============================] - 36s 58ms/step - loss:
0.6405 - accuracy: 0.6482 - val_loss: 0.6339 - val_accuracy: 0.6636
Epoch 3/5
625/625 [==============================] - 36s 57ms/step - loss:
0.6174 - accuracy: 0.6784 - val_loss: 0.6191 - val_accuracy: 0.6754
Epoch 4/5
625/625 [==============================] - 36s 57ms/step - loss:
0.6001 - accuracy: 0.6967 - val_loss: 0.6080 - val_accuracy: 0.6870
Epoch 5/5
625/625 [==============================] - 36s 57ms/step - loss:
0.5856 - accuracy: 0.7091 - val_loss: 0.5984 - val_accuracy: 0.6922
782/782 [==============================] - 6s 8ms/step - loss: 0.5947
- accuracy: 0.6955

[I 2023-07-15 19:08:10,650] Trial 2 finished with value:
0.6955199837684631 and parameters: {'vocab_len': 500}. Best is trial 1
with value: 0.7057600021362305.

Epoch 1/5
625/625 [==============================] - 36s 56ms/step - loss:
0.6904 - accuracy: 0.5304 - val_loss: 0.6808 - val_accuracy: 0.5822
Epoch 2/5
625/625 [==============================] - 34s 55ms/step - loss:
0.6579 - accuracy: 0.6516 - val_loss: 0.6436 - val_accuracy: 0.6696
Epoch 3/5
625/625 [==============================] - 34s 54ms/step - loss:
0.6202 - accuracy: 0.7006 - val_loss: 0.6174 - val_accuracy: 0.6916
Epoch 4/5
625/625 [==============================] - 34s 54ms/step - loss:
0.5910 - accuracy: 0.7272 - val_loss: 0.5958 - val_accuracy: 0.7122
Epoch 5/5
625/625 [==============================] - 34s 54ms/step - loss:
0.5660 - accuracy: 0.7453 - val_loss: 0.5775 - val_accuracy: 0.7200
782/782 [==============================] - 6s 8ms/step - loss: 0.5736
- accuracy: 0.7244

[I 2023-07-15 19:11:12,617] Trial 3 finished with value:
0.724399983882904 and parameters: {'vocab_len': 1000}. Best is trial 3
with value: 0.724399983882904.

Epoch 1/5
625/625 [==============================] - 35s 54ms/step - loss:
0.6448 - accuracy: 0.6244 - val_loss: 0.6002 - val_accuracy: 0.6798
Epoch 2/5
625/625 [==============================] - 34s 54ms/step - loss:
0.5613 - accuracy: 0.7160 - val_loss: 0.5574 - val_accuracy: 0.7126
Epoch 3/5
625/625 [==============================] - 34s 54ms/step - loss:
0.5225 - accuracy: 0.7430 - val_loss: 0.5348 - val_accuracy: 0.7284
```

```
Epoch 4/5
625/625 [==============================] - 34s 54ms/step - loss:
0.5014 - accuracy: 0.7603 - val_loss: 0.5256 - val_accuracy: 0.7348
Epoch 5/5
625/625 [==============================] - 34s 54ms/step - loss:
0.4873 - accuracy: 0.7685 - val_loss: 0.5166 - val_accuracy: 0.7446
782/782 [==============================] - 6s 8ms/step - loss: 0.5069
- accuracy: 0.7515

[I 2023-07-15 19:14:12,254] Trial 4 finished with value:
0.7515199780464172 and parameters: {'vocab_len': 1000}. Best is trial
4 with value: 0.7515199780464172.

Epoch 1/5
625/625 [==============================] - 35s 55ms/step - loss:
0.6959 - accuracy: 0.5083 - val_loss: 0.6910 - val_accuracy: 0.5248
Epoch 2/5
625/625 [==============================] - 34s 55ms/step - loss:
0.6851 - accuracy: 0.5486 - val_loss: 0.6871 - val_accuracy: 0.5448
Epoch 3/5
625/625 [==============================] - 35s 57ms/step - loss:
0.6760 - accuracy: 0.5832 - val_loss: 0.6843 - val_accuracy: 0.5470
Epoch 4/5
625/625 [==============================] - 34s 55ms/step - loss:
0.6666 - accuracy: 0.6081 - val_loss: 0.6825 - val_accuracy: 0.5428
Epoch 5/5
625/625 [==============================] - 35s 55ms/step - loss:
0.6569 - accuracy: 0.6239 - val_loss: 0.6817 - val_accuracy: 0.5420
782/782 [==============================] - 6s 8ms/step - loss: 0.6858
- accuracy: 0.5460

[I 2023-07-15 19:17:15,951] Trial 5 finished with value:
0.5460399985313416 and parameters: {'vocab_len': 1000}. Best is trial
4 with value: 0.7515199780464172.
```

```
# Print outputs of study

print("You completed {} trials.".format(len(study_vocab.trials)))

print("Best run:")
best = study_vocab.best_trial
print("    Accuracy Value: {}".format(best.value))

# Now for the parameters
print("    Parameters: ")
for key, value in best.params.items():    # go through the dictionary
    print("        {}: {}".format(key,value))
```

```
You completed 6 trials.
Best run:
    Accuracy Value: 0.7515199780464172
```

```
    Parameters:
        vocab_len: 1000
```

From what I have trialed here, it looks as if a lower vocab length is more likely to produce worse results though it still has the potential to fit well depending on the dataset. From other tests that I had copied over I saw that a larger vocab length has a better chance to perform well: You completed 20 trials. Best run: Accuracy Value: 0.7510799765586853 Parameters: vocab_len: 3000

The later GRU and LSTM tests also seem to show that a longer vocab length seems to increase the accuracy. This makes sense as you have more words to pull from.

13- Try smaller and larger sizes of the RNN hidden dimension. How does it affect the model performance? How does it affect the run time?

```
# Build an Optuna Study to look at various sequence legnths
(vocab_lengths)

def objective_hidden_dim(trial):

    clear_session()


    vocab_size = 2000  # number of words to consider as features

    max_len = 30  # cut texts after this number of words (among top
max_features most common words)
    batch_size = 32

    (X_train, y_train), (X_test, y_test) =
imdb.load_data(num_words=vocab_size)

    X_train = sequence.pad_sequences(X_train, maxlen=max_len)
    X_test = sequence.pad_sequences(X_test, maxlen=max_len)

    seq_length = 30

    # Create the trial parameter for the number of hidden layers
    n_hidden = trial.suggest_categorical("n_hidden",
[1,2,3,4,5,6,7,8,9,10])

    model = tf.keras.Sequential([
        # possition of would be text_encoder layer if used
        layers.Embedding(input_dim=vocab_size, output_dim=seq_length),
# NOte vocab_size is being optimized here
        layers.SimpleRNN(n_hidden,
kernel_initializer='glorot_uniform'),
        layers.Dense(1, activation='sigmoid')
    ])
```

```
    # Compile the RNN
    optimizer = tf.keras.optimizers.RMSprop(learning_rate=0.0001)
    loss_fn = tf.keras.losses.BinaryCrossentropy(from_logits=False)
    mets = ['accuracy']
    model.compile(optimizer, loss_fn, mets)


    # Shortened epochs to reduce time and based on prior runs that is
a fair sample size to get
    # an idea of what is going on
    history = model.fit(X_train, y_train, epochs = 5,
batch_size=batch_size, validation_split=0.2)

    eval_score = model.evaluate(X_test, y_test)

    return eval_score[1]

study_hiddendim = optuna.create_study(direction='maximize')

study_hiddendim.optimize(objective_hidden_dim, n_trials = 20,
timeout=7200, gc_after_trial=True)
```

Some of the output to minimze pdf length:

Epoch 1/5 625/625 [==============================] - 48s 76ms/step - loss: 0.6936 - accuracy: 0.5096 - val_loss: 0.6896 - val_accuracy: 0.5328 Epoch 2/5 625/625 [==============================] - 44s 70ms/step - loss: 0.6842 - accuracy: 0.5669 - val_loss: 0.6833 - val_accuracy: 0.5682 Epoch 3/5 625/625 [==============================] - 51s 82ms/step - loss: 0.6598 - accuracy: 0.6356 - val_loss: 0.6413 - val_accuracy: 0.6566 Epoch 4/5 625/625 [==============================] - 50s 79ms/step - loss: 0.6045 - accuracy: 0.6974 - val_loss: 0.5989 - val_accuracy: 0.6850 Epoch 5/5 625/625 [==============================] - 52s 83ms/step - loss: 0.5546 - accuracy: 0.7359 - val_loss: 0.5682 - val_accuracy: 0.7118 782/782 [==============================] - 12s 16ms/step - loss: 0.5644 - accuracy: 0.7160

[I 2023-07-16 12:34:19,105] Trial 11 finished with value: 0.7160000205039978 and parameters: {'n_hidden': 10}. Best is trial 8 with value: 0.7307599782943726.

[I 2023-07-16 12:57:15,772] Trial 16 finished with value: 0.7482399940490723 and parameters: {'n_hidden': 1}. Best is trial 16 with value: 0.7482399940490723.

Epoch 1/5 625/625 [==============================] - 38s 59ms/step - loss: 0.6940 - accuracy: 0.5160 - val_loss: 0.6917 - val_accuracy: 0.5190 Epoch 2/5 625/625 [==============================] - 36s 58ms/step - loss: 0.6845 - accuracy: 0.5515 - val_loss: 0.6896 - val_accuracy: 0.5192 Epoch 3/5 625/625 [==============================] - 36s 58ms/step - loss: 0.6775 - accuracy: 0.5757 - val_loss: 0.6889 - val_accuracy: 0.5260 Epoch 4/5 625/625

[==============================] - 50s 80ms/step - loss: 0.6713 - accuracy: 0.5903 - val_loss: 0.6895 - val_accuracy: 0.5296 Epoch 5/5 625/625 [==============================] - 48s 77ms/step - loss: 0.6658 - accuracy: 0.6042 - val_loss: 0.6908 - val_accuracy: 0.5240 782/782 [==============================] - 14s 18ms/step - loss: 0.6902 - accuracy: 0.5346

[I 2023-07-16 13:01:01,997] Trial 17 finished with value: 0.5345600247383118 and parameters: {'n_hidden': 1}. Best is trial 16 with value: 0.7482399940490723.

```
# Print outputs of study

print("You completed {} trials.".format(len(study_hiddendim.trials)))

print("Best run:")
best = study_hiddendim.best_trial
print("    Accuracy Value: {}".format(best.value))

# Now for the parameters
print("    Parameters: ")
for key, value in best.params.items():    # go through the dictionary
    print("        {}: {}".format(key,value))
```

```
You completed 20 trials.
Best run:
    Accuracy Value: 0.7482399940490723
    Parameters:
        n_hidden: 1
```

It seemed that in all cases, the amount of time to execute was around 40-60 seconds with only a minor change from having 1 hidden layer to 10 hidden layers. I wonder if this is correct or due to a hardware bottleneck?

Train LSTM and GRU networks

14- Build LSTM and GRU networks and compare their performance (accuracy and execution time) with the SimpleRNN. What is your conclusion?

GRU Network

```
# Reload the dataset to ensure I didn't change it in the past cells

vocab_size = 2000  # number of words to consider as features
max_len = 30  # cut texts after this number of words (among top
max_features most common words)
batch_size = 32

(X_train, y_train), (X_test, y_test) =
imdb.load_data(num_words=vocab_size)
X_train = sequence.pad_sequences(X_train, maxlen=max_len)
X_test = sequence.pad_sequences(X_test, maxlen=max_len)
```

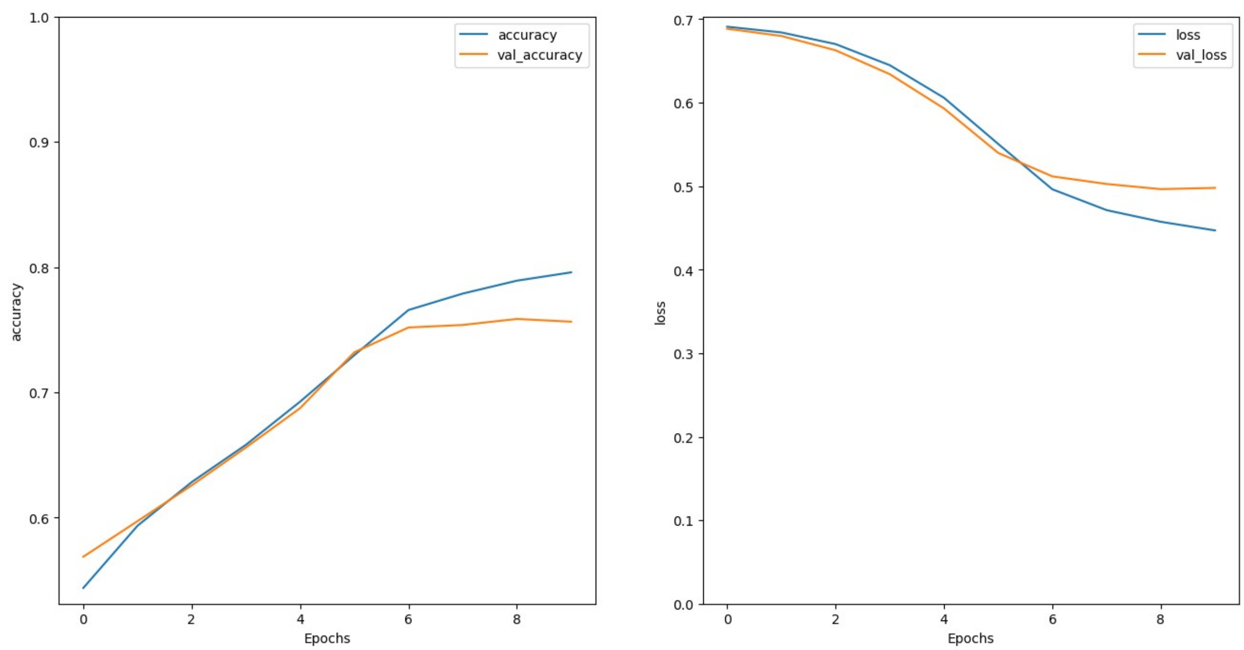
```
# Now things match the first SimpleRNN run

# You can either build this as
    # gru_cell = layers.GRUCell
    # gru_layer = layers.RNN(gru_cell)

# or as
        # layers.GRU()


def GRU_RNN(vocab_size=2000, seq_length=30, hidden_layers=5):

    model = tf.keras.Sequential([
        layers.Embedding(input_dim=vocab_size, output_dim=seq_length),
        layers.GRU(units=hidden_layers),
        layers.Dense(1, activation='sigmoid')
    ])

    return model

gru_rnn = GRU_RNN(vocab_size=2000, seq_length=30, hidden_layers=5)

# Compile the RNN
optimizer = tf.keras.optimizers.RMSprop(learning_rate=0.0001)
loss_fn = tf.keras.losses.BinaryCrossentropy(from_logits=False)
mets = ['accuracy']
gru_rnn.compile(optimizer, loss_fn, mets)


# Shortened epochs to reduce time and based on prior runs that is a
fair sample size to get
# an idea of what is going on
history_gru = gru_rnn.fit(X_train, y_train, epochs = 10,
batch_size=batch_size, validation_split=0.2)
```

```
Epoch 1/10
625/625 [==============================] - 8s 9ms/step - loss: 0.6909
- accuracy: 0.5439 - val_loss: 0.6885 - val_accuracy: 0.5688
Epoch 2/10
625/625 [==============================] - 5s 8ms/step - loss: 0.6840
- accuracy: 0.5936 - val_loss: 0.6798 - val_accuracy: 0.5972
Epoch 3/10
625/625 [==============================] - 5s 8ms/step - loss: 0.6701
- accuracy: 0.6284 - val_loss: 0.6627 - val_accuracy: 0.6258
Epoch 4/10
625/625 [==============================] - 5s 9ms/step - loss: 0.6448
- accuracy: 0.6582 - val_loss: 0.6341 - val_accuracy: 0.6560
Epoch 5/10
625/625 [==============================] - 5s 8ms/step - loss: 0.6058
```

```
- accuracy: 0.6926 - val_loss: 0.5930 - val_accuracy: 0.6874
Epoch 6/10
625/625 [==============================] - 5s 8ms/step - loss: 0.5506
- accuracy: 0.7297 - val_loss: 0.5398 - val_accuracy: 0.7320
Epoch 7/10
625/625 [==============================] - 5s 8ms/step - loss: 0.4962
- accuracy: 0.7657 - val_loss: 0.5117 - val_accuracy: 0.7518
Epoch 8/10
625/625 [==============================] - 5s 8ms/step - loss: 0.4713
- accuracy: 0.7789 - val_loss: 0.5026 - val_accuracy: 0.7538
Epoch 9/10
625/625 [==============================] - 5s 8ms/step - loss: 0.4574
- accuracy: 0.7891 - val_loss: 0.4964 - val_accuracy: 0.7586
Epoch 10/10
625/625 [==============================] - 5s 9ms/step - loss: 0.4470
- accuracy: 0.7958 - val_loss: 0.4979 - val_accuracy: 0.7564
```

CHEESE AND CRACKERS BATMAN! Those epochs ran very, very quickly. They were about 7x faster than the SimpleRNN with similar results.

```
quick_plot(history_gru)
```

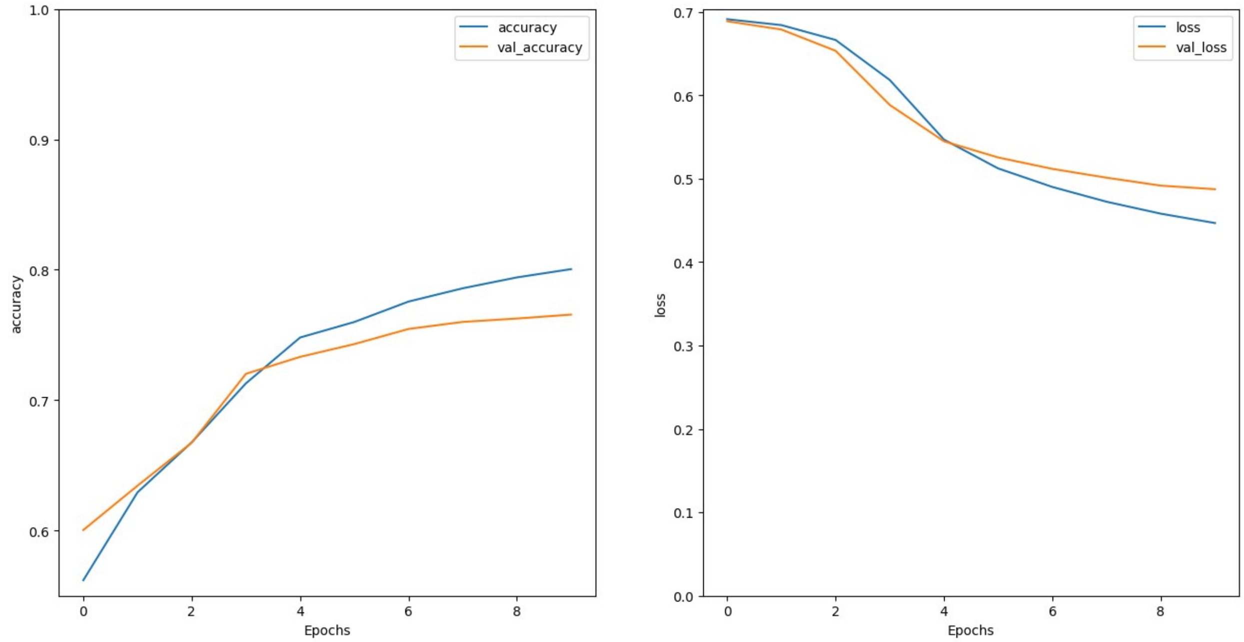


The accuracy and loss are similar to the SimpleRNNs with the slight edge (maybe) going to the GRU given that the val_loss and train_loss stay together longer which would imply less overfitting.

LSTM Network

```
# Reload the dataset to ensure I didn't change it in the past cells

vocab_size = 2000  # number of words to consider as features
max_len = 30  # cut texts after this number of words (among top
max_features most common words)
batch_size = 32

(X_train, y_train), (X_test, y_test) =
imdb.load_data(num_words=vocab_size)
X_train = sequence.pad_sequences(X_train, maxlen=max_len)
X_test = sequence.pad_sequences(X_test, maxlen=max_len)

# Now things match the first SimpleRNN run

# You can either build this as
    # gru_cell = layers.GRUCell
    # gru_layer = layers.RNN(gru_cell)

# or as
        # layers.GRU()


def LSTM_RNN(vocab_size=2000, seq_length=30, hidden_layers=5):

    model = tf.keras.Sequential([
        layers.Embedding(input_dim=vocab_size, output_dim=seq_length),
        layers.LSTM(units=hidden_layers),   # the only change here
from the GRU is this line to LSTM
        layers.Dense(1, activation='sigmoid')
    ])

    return model

lstm_rnn = LSTM_RNN(vocab_size=2000, seq_length=30, hidden_layers=5)

# Compile the RNN
optimizer = tf.keras.optimizers.RMSprop(learning_rate=0.0001)
loss_fn = tf.keras.losses.BinaryCrossentropy(from_logits=False)
mets = ['accuracy']
lstm_rnn.compile(optimizer, loss_fn, mets)


# Shortened epochs to reduce time and based on prior runs that is a
fair sample size to get
# an idea of what is going on
history_lstm = lstm_rnn.fit(X_train, y_train, epochs = 10,
batch_size=batch_size, validation_split=0.2)
```

```
Epoch 1/10
625/625 [==============================] - 7s 9ms/step - loss: 0.6912
- accuracy: 0.5619 - val_loss: 0.6888 - val_accuracy: 0.6004
Epoch 2/10
625/625 [==============================] - 5s 9ms/step - loss: 0.6841
- accuracy: 0.6293 - val_loss: 0.6789 - val_accuracy: 0.6344
Epoch 3/10
625/625 [==============================] - 5s 9ms/step - loss: 0.6663
- accuracy: 0.6676 - val_loss: 0.6533 - val_accuracy: 0.6674
Epoch 4/10
625/625 [==============================] - 5s 8ms/step - loss: 0.6183
- accuracy: 0.7129 - val_loss: 0.5884 - val_accuracy: 0.7202
Epoch 5/10
625/625 [==============================] - 5s 9ms/step - loss: 0.5469
- accuracy: 0.7480 - val_loss: 0.5449 - val_accuracy: 0.7332
Epoch 6/10
625/625 [==============================] - 5s 8ms/step - loss: 0.5123
- accuracy: 0.7599 - val_loss: 0.5254 - val_accuracy: 0.7430
Epoch 7/10
625/625 [==============================] - 5s 8ms/step - loss: 0.4901
- accuracy: 0.7756 - val_loss: 0.5118 - val_accuracy: 0.7546
Epoch 8/10
625/625 [==============================] - 5s 9ms/step - loss: 0.4724
- accuracy: 0.7858 - val_loss: 0.5012 - val_accuracy: 0.7600
Epoch 9/10
625/625 [==============================] - 5s 8ms/step - loss: 0.4581
- accuracy: 0.7941 - val_loss: 0.4917 - val_accuracy: 0.7626
Epoch 10/10
625/625 [==============================] - 5s 8ms/step - loss: 0.4468
- accuracy: 0.8005 - val_loss: 0.4874 - val_accuracy: 0.7656

quick_plot(history_lstm)
```



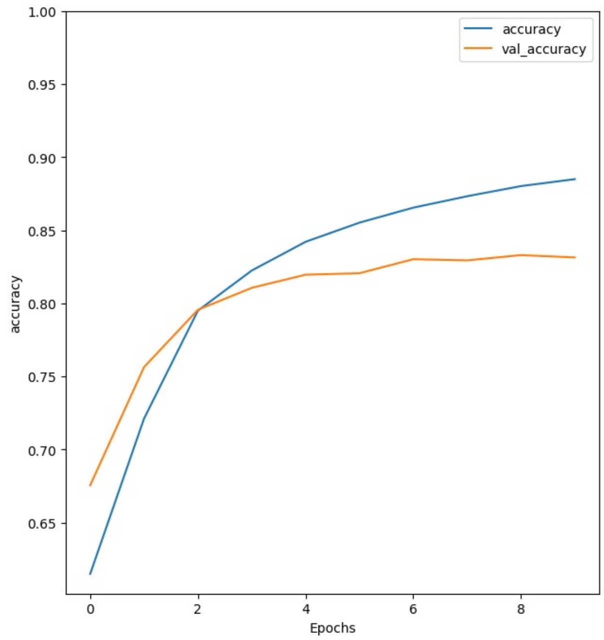
This LSTM has very similar results to the GRU with just a smidge longer execution time compared to the GRU. I am currious if I can use a large vocab here and provide some interesting results.

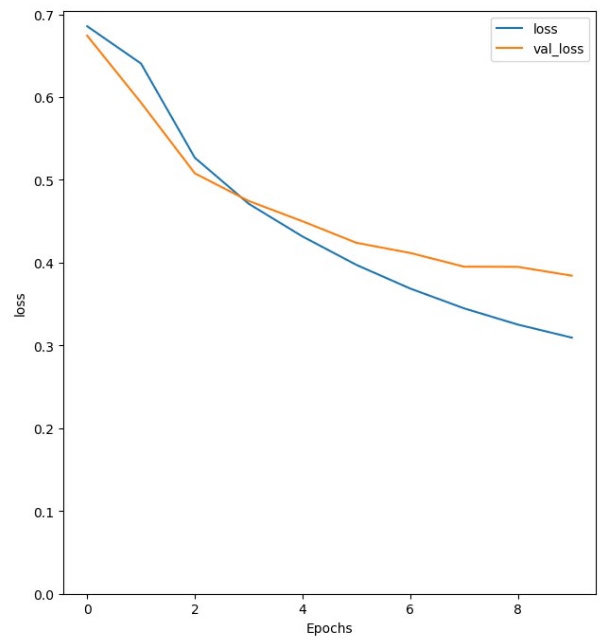
```
# Reload the dataset to ensure I didn't change it in the past cells

vocab_size = 5000  # number of words to consider as features
max_len = 80  # cut texts after this number of words (among top
max_features most common words)
batch_size = 32

(X_train, y_train), (X_test, y_test) =
imdb.load_data(num_words=vocab_size)
X_train = sequence.pad_sequences(X_train, maxlen=max_len)
X_test = sequence.pad_sequences(X_test, maxlen=max_len)

# THIS DATA IS DIFFERENT FROM THE INITIAL SimpleRNN RUN

lstm_rnn_big = LSTM_RNN(vocab_size=5000, seq_length=80,
hidden_layers=5)

# Compile the RNN
optimizer = tf.keras.optimizers.RMSprop(learning_rate=0.0001)
loss_fn = tf.keras.losses.BinaryCrossentropy(from_logits=False)
mets = ['accuracy']
lstm_rnn_big.compile(optimizer, loss_fn, mets)


# Shortened epochs to reduce time and based on prior runs that is a
fair sample size to get
```

```python
# an idea of what is going on
history_lstm_big = lstm_rnn_big.fit(X_train, y_train, epochs = 10,
batch_size=batch_size, validation_split=0.2)
```

```
Epoch 1/10
625/625 [==============================] - 8s 10ms/step - loss: 0.6856
- accuracy: 0.6150 - val_loss: 0.6742 - val_accuracy: 0.6756
Epoch 2/10
625/625 [==============================] - 6s 9ms/step - loss: 0.6405
- accuracy: 0.7212 - val_loss: 0.5933 - val_accuracy: 0.7564
Epoch 3/10
625/625 [==============================] - 6s 10ms/step - loss: 0.5267
- accuracy: 0.7950 - val_loss: 0.5079 - val_accuracy: 0.7956
Epoch 4/10
625/625 [==============================] - 6s 9ms/step - loss: 0.4712
- accuracy: 0.8224 - val_loss: 0.4745 - val_accuracy: 0.8106
Epoch 5/10
625/625 [==============================] - 6s 9ms/step - loss: 0.4316
- accuracy: 0.8421 - val_loss: 0.4500 - val_accuracy: 0.8196
Epoch 6/10
625/625 [==============================] - 6s 10ms/step - loss: 0.3973
- accuracy: 0.8552 - val_loss: 0.4240 - val_accuracy: 0.8206
Epoch 7/10
625/625 [==============================] - 6s 9ms/step - loss: 0.3687
- accuracy: 0.8655 - val_loss: 0.4117 - val_accuracy: 0.8302
Epoch 8/10
625/625 [==============================] - 6s 10ms/step - loss: 0.3448
- accuracy: 0.8733 - val_loss: 0.3952 - val_accuracy: 0.8294
Epoch 9/10
625/625 [==============================] - 6s 9ms/step - loss: 0.3252
- accuracy: 0.8802 - val_loss: 0.3949 - val_accuracy: 0.8330
Epoch 10/10
625/625 [==============================] - 6s 9ms/step - loss: 0.3095
- accuracy: 0.8849 - val_loss: 0.3843 - val_accuracy: 0.8314
```

```python
quick_plot(history_lstm_big)
```

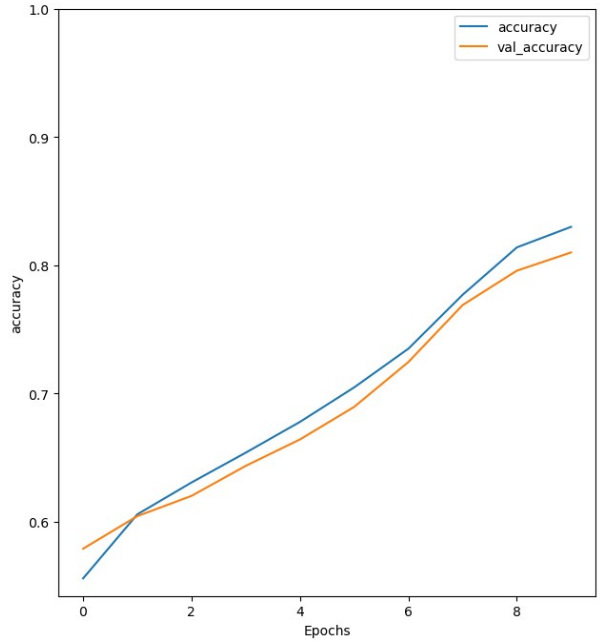


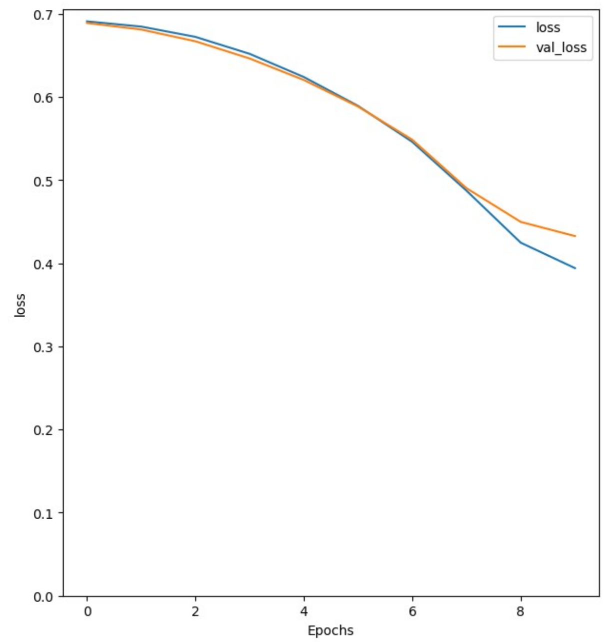


Interesting. With a larger vocab list and large sequence size, I was able to improve accuracy with what looks to be some slight overfitting. But the execution time minimally changed. How does the GRU stack up to this? Let's find out.

```
# Reload the dataset to ensure I didn't change it in the past cells

vocab_size = 5000  # number of words to consider as features
max_len = 80  # cut texts after this number of words (among top
max_features most common words)
batch_size = 32

(X_train, y_train), (X_test, y_test) =
imdb.load_data(num_words=vocab_size)
X_train = sequence.pad_sequences(X_train, maxlen=max_len)
X_test = sequence.pad_sequences(X_test, maxlen=max_len)

# THIS DATA IS DIFFERENT FROM THE INITIAL SimpleRNN RUN

gru_rnn_big = GRU_RNN(vocab_size=5000, seq_length=50, hidden_layers=5)

# Compile the RNN
optimizer = tf.keras.optimizers.RMSprop(learning_rate=0.0001)
loss_fn = tf.keras.losses.BinaryCrossentropy(from_logits=False)
mets = ['accuracy']
gru_rnn_big.compile(optimizer, loss_fn, mets)


# Shortened epochs to reduce time and based on prior runs that is a
fair sample size to get
# an idea of what is going on
```

```
history_gru_big = gru_rnn_big.fit(X_train, y_train, epochs = 10,
batch_size=batch_size, validation_split=0.2)

Epoch 1/10
625/625 [==============================] - 8s 10ms/step - loss: 0.6911
- accuracy: 0.5559 - val_loss: 0.6892 - val_accuracy: 0.5790
Epoch 2/10
625/625 [==============================] - 6s 10ms/step - loss: 0.6848
- accuracy: 0.6058 - val_loss: 0.6813 - val_accuracy: 0.6044
Epoch 3/10
625/625 [==============================] - 6s 10ms/step - loss: 0.6724
- accuracy: 0.6306 - val_loss: 0.6672 - val_accuracy: 0.6202
Epoch 4/10
625/625 [==============================] - 6s 9ms/step - loss: 0.6520
- accuracy: 0.6539 - val_loss: 0.6464 - val_accuracy: 0.6436
Epoch 5/10
625/625 [==============================] - 6s 10ms/step - loss: 0.6241
- accuracy: 0.6779 - val_loss: 0.6205 - val_accuracy: 0.6642
Epoch 6/10
625/625 [==============================] - 6s 9ms/step - loss: 0.5894
- accuracy: 0.7048 - val_loss: 0.5886 - val_accuracy: 0.6896
Epoch 7/10
625/625 [==============================] - 6s 9ms/step - loss: 0.5461
- accuracy: 0.7350 - val_loss: 0.5487 - val_accuracy: 0.7246
Epoch 8/10
625/625 [==============================] - 6s 10ms/step - loss: 0.4874
- accuracy: 0.7770 - val_loss: 0.4904 - val_accuracy: 0.7690
Epoch 9/10
625/625 [==============================] - 6s 9ms/step - loss: 0.4248
- accuracy: 0.8140 - val_loss: 0.4498 - val_accuracy: 0.7958
Epoch 10/10
625/625 [==============================] - 6s 10ms/step - loss: 0.3942
- accuracy: 0.8299 - val_loss: 0.4328 - val_accuracy: 0.8100

quick_plot(history_gru_big)
```





Slightly less accuracy but less overfitting in comparision. This isn't surprising as the GRU has fewer gates compared to the LSTM.

Questions that Remain

Throughout the training (.fit) steps, my GPU (1070) was being used but only at 4-6% utilization. For the CNN tasks in Assignment #2 (zoo) it was around 45% utilization. Was the low utilization due to the vectorization of the dataset or are RNNs / GRUs / LSTMs just that much less intensive than CNNs?

The first setup I tried using a different loading method

You may have noticed early on some data loading that was commented out. I was going through this in a slightly different approach at first which I'll show below.

There are some areas in here that I had questions on especially with the data loading and TextVectorization layer

```
# Data loading from tfds

dataset, info = tfds.load('imdb_reviews', with_info=True,
                          as_supervised=True)
train_dataset, test_dataset = dataset['train'], dataset['test']

train_dataset.element_spec
```

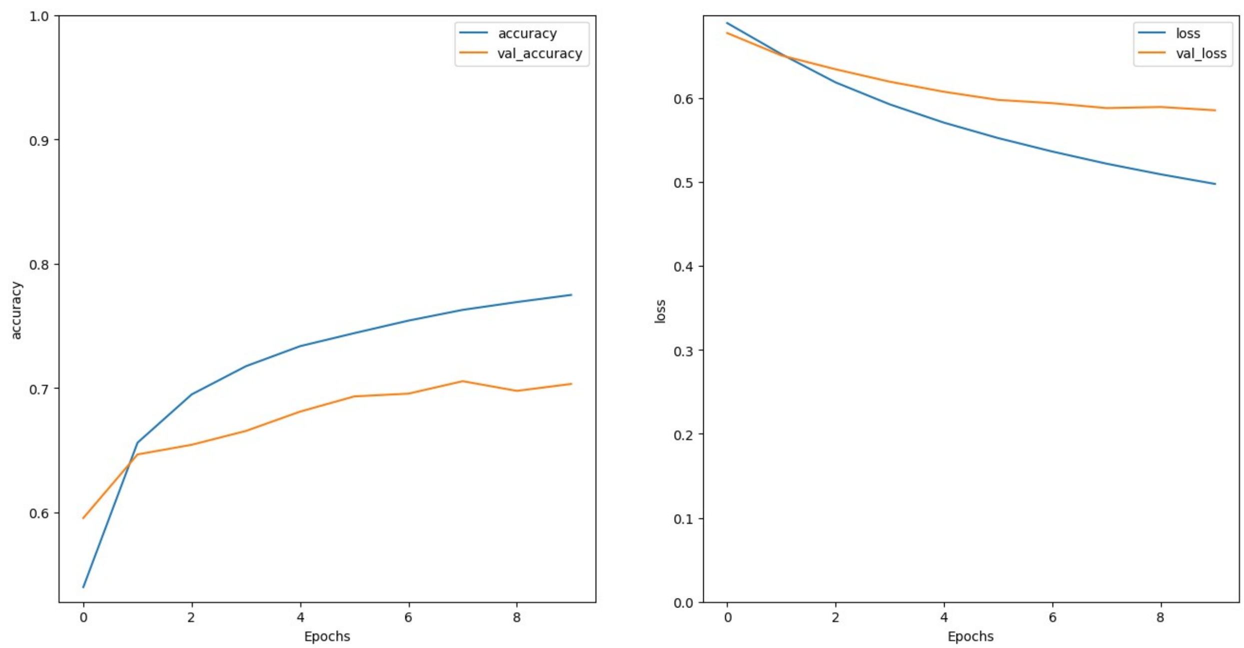
```
(TensorSpec(shape=(), dtype=tf.string, name=None),
 TensorSpec(shape=(), dtype=tf.int64, name=None))
```

```
# Set up AUTOTUNING for the data


#QUESION:
# I'm not 100% certain as to what the buffer size is doing but I was
under the impression for the
# shuffle I want the buffer to be the size of the dataset.  However,
when the buffer was initially
# at 10,000 I ran into errors so I reduced it and was successful.  So
what is this doing?

BUFFER_SIZE = 4000
BATCH_SIZE =  30 #(text, label) pairs


# This allows preprocessing to happen on the CPU asynchronously

# QUESTION:  CAN I PUT THIS ANYWHERE ONCE THE DATASET IS
CREATED?.......[end of next cell]
train_dataset_30 =
train_dataset.shuffle(BUFFER_SIZE).batch(BATCH_SIZE).prefetch(tf.data.
AUTOTUNE)
test_dataset_30 =
test_dataset.batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE)

# The train dataset need to be vectorized into integer Tensor

vocab_size = 2000
max_length = 30

# The TextVectorization layer takes care of setting the vocabulary
size
# via max_tokens.  The output_sequence either pads or truncates to the

# max length prescribed.  The Standardize sets all words to lowercase
and
# removes punctuation.

text_encoder = layers.TextVectorization(max_tokens=vocab_size,

output_sequence_length=max_length,

standardize='lower_and_strip_punctuation'

                                        )

# The .adapt function call applies the text encoder across the
elements
```

```
text_encoder.adapt(train_dataset.map(lambda text, label: text))

# ...... OR DOES THE AUTOTUNING NEED TO GO HERE AFTER THE ENCODER? AS
IN:
    # https://keras.io/guides/preprocessing_layers/

# The SimpleRNN model
def model_RNN_old(encoder, seq_length:int):

    model = tf.keras.Sequential([
        encoder,  # QUESTION:  Was I correct to place this encoder
layer here?  or is this out of palce?
        layers.Embedding(input_dim=vocab_size, output_dim=seq_length),
        layers.SimpleRNN(5, kernel_initializer='glorot_uniform'),
        layers.Dense(1, activation='sigmoid')
    ])

    return model

# Build the model section
test_rnn_net = model_RNN_old(encoder=text_encoder, seq_length=30)

optimizer = tf.keras.optimizers.RMSprop(learning_rate=0.0001)
loss_fn = tf.keras.losses.BinaryCrossentropy(from_logits=False)
mets = ['accuracy']

test_rnn_net.compile(optimizer, loss_fn, mets)

# Fit the model
history_test_rnn = test_rnn_net.fit(train_dataset_30,
                          epochs=10,
                          validation_data=test_dataset_30,
                          validation_steps=30)
```

```
Epoch 1/10
834/834 [==============================] - 86s 100ms/step - loss:
0.6892 - accuracy: 0.5400 - val_loss: 0.6773 - val_accuracy: 0.5956
Epoch 2/10
834/834 [==============================] - 89s 107ms/step - loss:
0.6522 - accuracy: 0.6561 - val_loss: 0.6506 - val_accuracy: 0.6467
Epoch 3/10
834/834 [==============================] - 92s 110ms/step - loss:
0.6185 - accuracy: 0.6950 - val_loss: 0.6342 - val_accuracy: 0.6544
Epoch 4/10
834/834 [==============================] - 86s 102ms/step - loss:
0.5924 - accuracy: 0.7176 - val_loss: 0.6194 - val_accuracy: 0.6656
Epoch 5/10
834/834 [==============================] - 77s 93ms/step - loss:
0.5706 - accuracy: 0.7337 - val_loss: 0.6074 - val_accuracy: 0.6811
Epoch 6/10
```

```
834/834 [==============================] - 83s 99ms/step - loss:
0.5522 - accuracy: 0.7442 - val_loss: 0.5977 - val_accuracy: 0.6933
Epoch 7/10
834/834 [==============================] - 82s 98ms/step - loss:
0.5362 - accuracy: 0.7543 - val_loss: 0.5938 - val_accuracy: 0.6956
Epoch 8/10
834/834 [==============================] - 83s 100ms/step - loss:
0.5217 - accuracy: 0.7629 - val_loss: 0.5879 - val_accuracy: 0.7056
Epoch 9/10
834/834 [==============================] - 67s 81ms/step - loss:
0.5090 - accuracy: 0.7692 - val_loss: 0.5892 - val_accuracy: 0.6978
Epoch 10/10
834/834 [==============================] - 69s 83ms/step - loss:
0.4976 - accuracy: 0.7749 - val_loss: 0.5853 - val_accuracy: 0.7033

quick_plot(history_test_rnn)
```



And this looks very similar to the RNN method I used at the start of this so I'm assuming both ways are working the same. Am I correct in this assumption?