```
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras import layers, models

from sklearn.preprocessing import LabelEncoder
import pickle
import numpy as np
import pandas as pd

np.random.seed(1234)
```

```
df_names = ['Twitter_ID', 'Subject_Matter', 'Sentiment', 'Text']
df = pd.read_csv('drive/MyDrive/Colab Notebooks/TextClassification2/twitter_sentiment/twitter_training.csv', header=0, encoding='latin-1'
df = df.drop(columns=['Twitter_ID', 'Subject_Matter'], axis=1)
df = df[df.Sentiment != 'Irrelevant']
df = df[df.Sentiment != 'Neutral']
df['Sentiment'].replace(['Positive', 'Negative'], [0, 1], inplace=True)
df = df.dropna()
df.Text = df.Text.astype(str)
print(df.head())
```

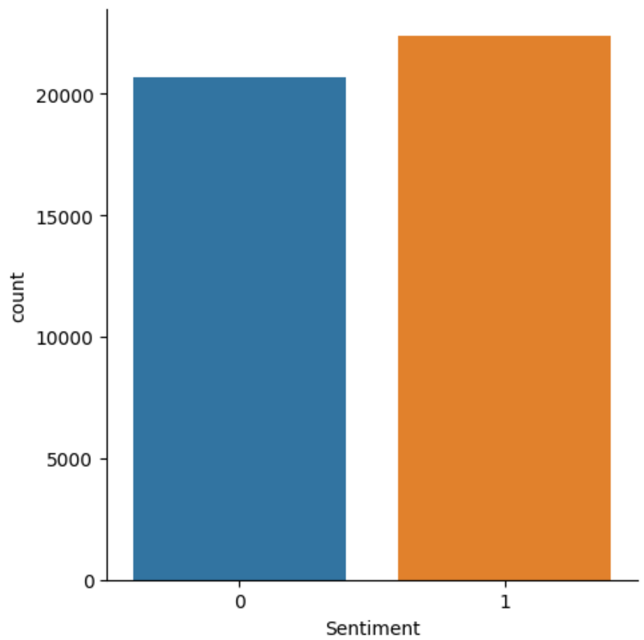
```
   Sentiment                                               Text
0          0  I am coming to the borders and I will kill you...
1          0  im getting on borderlands and i will kill you ...
2          0  im coming on borderlands and i will murder you...
3          0  im getting on borderlands 2 and i will murder ...
4          0  im getting into borderlands and i can murder y...
```

My dataset was obtained on Kaggle. The dataset is primarily sourced from Twitter and seeks to predict the sentiment of tweets from various users on topics related to, on first glance, video games. The target values in the original dataset are: Positive, Negative, Neutral, and Irrelevant.

However, to make model training simpler, I chose to remove data instances where that contain the target values of neutral and irrelevant. This leaves room for a binary classification of positive or negative.
Furthermore, I determined it was best to remove fields such as topic and twitter_id (most likely representing the twitter user that made the comment).
Additionally, I also removed data instances that had a NULL or NaN value in any fields.

```
import seaborn as sb
sb.catplot(x='Sentiment', kind='count', data=df)
```

```
<seaborn.axisgrid.FacetGrid at 0x7f0b711230d0>
```



As you can see, the sentiment is evenly distributed. This may or may not have relevance to the results of our training.

```
i = np.random.rand(len(df)) < 0.8
train = df[i]
test·=·df[~i]
print("train data size: ", train.shape)
print("test data size: ", test.shape)
```

```
train data size:  (34364, 2)
test data size:  (8648, 2)
```

```
num_labels = 2
vocab_size = 25000
batch_size = 100
```

We start with the Sequential model

```
tokenizer = Tokenizer(num_words=vocab_size)
tokenizer.fit_on_texts(train.Text)

x_train = tokenizer.texts_to_matrix(train.Text, mode='tfidf')
x_test = tokenizer.texts_to_matrix(test.Text, mode='tfidf')

encoder = LabelEncoder()
encoder.fit(train.Sentiment)
y_train = encoder.transform(train.Sentiment)
y_test = encoder.transform(test.Sentiment)


"""
from keras.utils import to_categorical
y_train = to_categorical(y_train, 3)
y_test = to_categorical(y_test, 3)
"""
```

```
'\nfrom keras.utils import to_categorical\ny_train = to_categorical(y_train, 3)\ny_test = to_categoric
al(y_test, 3)\n'
```

```
print("train shapes:", x_train.shape, y_train.shape)
print("test shapes:", x_test.shape, y_test.shape)
print("test first five labels:", y_test[:5])
```

```
train shapes: (34444, 25000) (34444,)
test shapes: (8568, 25000) (8568,)
test first five labels: [0 0 0 1 0]
```

```
model = models.Sequential()
model.add(layers.Dense(16, input_dim=vocab_size, kernel_initializer='normal', activation='relu'))
model.add(layers.Dense(1, input_dim=vocab_size, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

history = model.fit(x_train, y_train,
                    batch_size=batch_size,
                    epochs=10,
                    verbose=1,
                    validation_split=0.3)
```

```
Epoch 1/10
242/242 [==============================] - 10s 37ms/step - loss: 0.4529 - accuracy: 0.7834 - val_loss: 0.5152 - val_accuracy: 0.763
Epoch 2/10
242/242 [==============================] - 6s 25ms/step - loss: 0.1602 - accuracy: 0.9425 - val_loss: 0.6095 - val_accuracy: 0.7527
Epoch 3/10
242/242 [==============================] - 8s 31ms/step - loss: 0.0902 - accuracy: 0.9674 - val_loss: 0.6870 - val_accuracy: 0.7574
Epoch 4/10
242/242 [==============================] - 6s 26ms/step - loss: 0.0651 - accuracy: 0.9752 - val_loss: 0.7762 - val_accuracy: 0.7518
Epoch 5/10
242/242 [==============================] - 7s 27ms/step - loss: 0.0527 - accuracy: 0.9784 - val_loss: 0.8548 - val_accuracy: 0.7515
Epoch 6/10
242/242 [==============================] - 6s 26ms/step - loss: 0.0456 - accuracy: 0.9803 - val_loss: 0.9297 - val_accuracy: 0.7473
Epoch 7/10
242/242 [==============================] - 6s 25ms/step - loss: 0.0408 - accuracy: 0.9818 - val_loss: 0.9983 - val_accuracy: 0.7480
Epoch 8/10
242/242 [==============================] - 8s 31ms/step - loss: 0.0375 - accuracy: 0.9830 - val_loss: 1.0774 - val_accuracy: 0.7437
Epoch 9/10
242/242 [==============================] - 6s 26ms/step - loss: 0.0350 - accuracy: 0.9841 - val_loss: 1.1316 - val_accuracy: 0.7442
Epoch 10/10
242/242 [==============================] - 8s 33ms/step - loss: 0.0329 - accuracy: 0.9840 - val_loss: 1.2146 - val_accuracy: 0.7385
```

```
# evaluate
score = model.evaluate(x_test, y_test, batch_size=batch_size, verbose=1)
print('Accuracy: ', score[1])
```

```
86/86 [==============================] - 1s 10ms/step - loss: 0.4720 - accuracy: 0.8853
Accuracy:  0.8852707743644714
```

```
losses_and_metrics = model.evaluate(x_test, y_test, batch_size=128)
losses_and_metrics
```

```
67/67 [==============================] - 1s 8ms/step - loss: 0.4720 - accuracy: 0.8853
[0.47197115421295166, 0.8852707743644714]
```

The accuracy results are quite good considering that the valuation accuracy merely trends at 70%.

```
classes = model.predict(x_test, batch_size=128)
classes[:5]
```

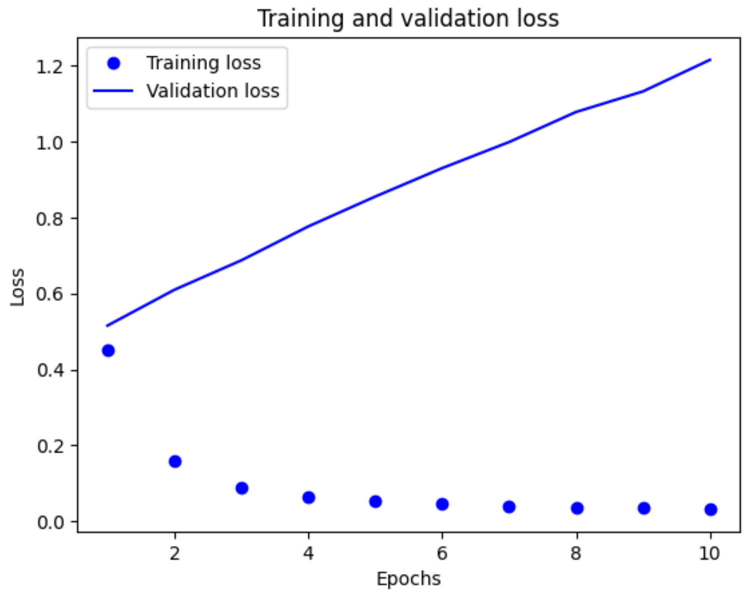
```
67/67 [==============================] - 1s 12ms/step
array([[3.3187138e-07],
       [6.6364482e-05],
       [2.8622677e-03],
       [9.9999863e-01],
       [1.2087857e-13]], dtype=float32)
```

```
# plot the training and validation loss
import matplotlib.pyplot as plt

loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(loss)+1)

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```



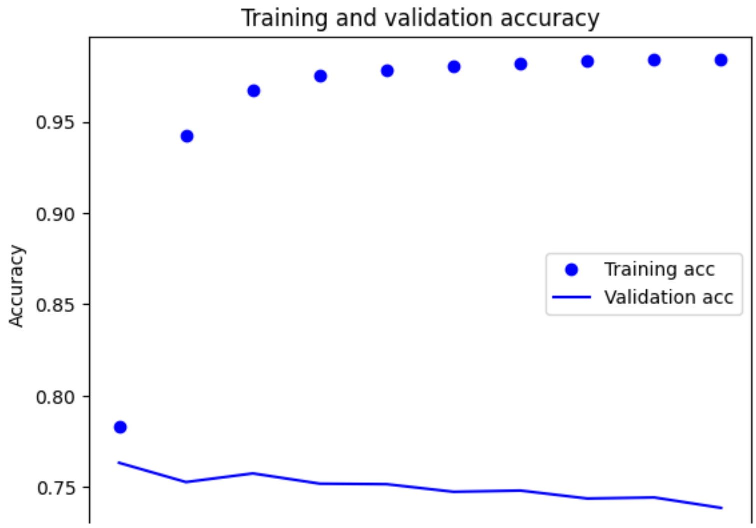

```
# plot the training and validation accuracy

plt.clf()   # clear

acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.show()
```



The graphs appear consistent with the graphs demonstrated in class. In my case, they are noticeably smoother as there are not a lot of outliers.

```
train.Text.shape
```

```
(34364,)
```

```
# Change the tokenized text to a sequence
# Change the 'sentiment' part of the data to numpy array
tokenizer = Tokenizer(num_words=vocab_size)
tokenizer.fit_on_texts(train.Text)
train_data = tokenizer.texts_to_sequences(train.Text)
tokenizer.fit_on_texts(test.Text)
test_data = tokenizer.texts_to_sequences(test.Text)
```

```
train_labels = train.Sentiment.to_numpy()
print(train_labels)
print(train_labels.shape)
test_labels = test.Sentiment.to_numpy()
print(test_labels)
print(test_labels.shape)
```

```
[0 0 0 ... 0 0 0]
(34364,)
[0 0 1 ... 1 1 0]
(8648,)
```

```
train_data = preprocessing.sequence.pad_sequences(train_data, maxlen=500)
test_data = preprocessing.sequence.pad_sequences(test_data, maxlen=500)
```

```
print(train_data.shape)
print(test_data.shape)
```

```
(34364, 500)
(8648, 500)
```

RNN Architecture:

```
model = models.Sequential()
model.add(layers.Embedding(vocab_size, 32))
model.add(layers.SimpleRNN(32))
model.add(layers.Dense(1, activation='sigmoid'))
```

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

```
history = model.fit(train_data,
                    train_labels,
                    epochs=10,
                    batch_size=128,
                    validation_split=0.2)
```

```
Epoch 1/10
215/215 [==============================] - 47s 213ms/step - loss: 0.5569 - accuracy: 0.7132 - val_loss: 0.5512 - val_accuracy: 0.72
Epoch 2/10
215/215 [==============================] - 48s 223ms/step - loss: 0.2889 - accuracy: 0.8841 - val_loss: 0.5160 - val_accuracy: 0.76
Epoch 3/10
215/215 [==============================] - 52s 241ms/step - loss: 0.1760 - accuracy: 0.9308 - val_loss: 0.5928 - val_accuracy: 0.73
Epoch 4/10
215/215 [==============================] - 45s 212ms/step - loss: 0.1220 - accuracy: 0.9519 - val_loss: 0.6424 - val_accuracy: 0.75
Epoch 5/10
215/215 [==============================] - 46s 214ms/step - loss: 0.2385 - accuracy: 0.9148 - val_loss: 0.6313 - val_accuracy: 0.76
Epoch 6/10
215/215 [==============================] - 46s 212ms/step - loss: 0.0763 - accuracy: 0.9690 - val_loss: 0.9120 - val_accuracy: 0.71
Epoch 7/10
215/215 [==============================] - 46s 213ms/step - loss: 0.1086 - accuracy: 0.9567 - val_loss: 0.7787 - val_accuracy: 0.74
Epoch 8/10
215/215 [==============================] - 46s 212ms/step - loss: 0.0928 - accuracy: 0.9643 - val_loss: 0.8511 - val_accuracy: 0.75
Epoch 9/10
215/215 [==============================] - 46s 214ms/step - loss: 0.1219 - accuracy: 0.9545 - val_loss: 0.7790 - val_accuracy: 0.72
Epoch 10/10
215/215 [==============================] - 46s 212ms/step - loss: 0.1823 - accuracy: 0.9393 - val_loss: 1.1142 - val_accuracy: 0.65
```

```
print(test_data.shape)
```

```
(8648, 500)
```

```
from sklearn.metrics import classification_report

pred = model.predict(test_data)
pred = [1.0 if p>= 0.5 else 0.0 for p in pred]
print(classification_report(test_labels, pred))
```

```
271/271 [==============================] - 10s 35ms/step
              precision    recall  f1-score   support

           0       0.45      0.11      0.17      4144
           1       0.52      0.88      0.65      4504

    accuracy                           0.51      8648
   macro avg       0.49      0.49      0.41      8648
weighted avg       0.49      0.51      0.42      8648
```

Unfortunately the RNN architecture does not seem to be able to capture the patterns and thus the accuracy is very low.

LSTM RNN Architecture:

```
model = models.Sequential()
model.add(layers.Embedding(25000, 32))
model.add(layers.LSTM(32))
model.add(layers.Dense(1, activation='sigmoid'))
```

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

```
history = model.fit(train_data,
                    train_labels,
                    epochs=6,
                    batch_size=300,
                    validation_split=0.2)
```

```
Epoch 1/6
92/92 [==============================] - 70s 759ms/step - loss: 0.6069 - accuracy: 0.6931 - val_loss: 0.5409 - val_accuracy: 0.7372
Epoch 2/6
92/92 [==============================] - 74s 811ms/step - loss: 0.4396 - accuracy: 0.8134 - val_loss: 0.4790 - val_accuracy: 0.7727
Epoch 3/6
92/92 [==============================] - 65s 703ms/step - loss: 0.3375 - accuracy: 0.8672 - val_loss: 0.5002 - val_accuracy: 0.7676
Epoch 4/6
92/92 [==============================] - 70s 764ms/step - loss: 0.2666 - accuracy: 0.8990 - val_loss: 0.5111 - val_accuracy: 0.7682
Epoch 5/6
92/92 [==============================] - 69s 755ms/step - loss: 0.2146 - accuracy: 0.9191 - val_loss: 0.5466 - val_accuracy: 0.7684
Epoch 6/6
92/92 [==============================] - 64s 699ms/step - loss: 0.1748 - accuracy: 0.9332 - val_loss: 0.8204 - val_accuracy: 0.7387
```

```
pred = model.predict(test_data)
pred = [1.0 if p>= 0.5 else 0.0 for p in pred]
```

```
print(classification_report(test_labels, pred))
```

```
271/271 [==============================] - 127s 63ms/step
              precision    recall  f1-score   support

           0       0.51      0.61      0.56      4144
           1       0.57      0.46      0.51      4504

    accuracy                           0.54      8648
   macro avg       0.54      0.54      0.53      8648
weighted avg       0.54      0.54      0.53      8648
```

Again, there is minor improvement with a different type of RNN architecture.

CNN Architecture:

```
model = models.Sequential()
model.add(layers.Embedding(25000, 128, input_length=500))
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.MaxPooling1D(5))
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.GlobalMaxPooling1D())
model.add(layers.Dense(1))
```

```
model.compile(optimizer=tf.keras.optimizers.RMSprop(learning_rate=1e-4),
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

```
history = model.fit(train_data,
                    train_labels,
                    epochs=6,
                    batch_size=300,
                    validation_split=0.2)
```

```
Epoch 1/6
92/92 [==============================] - 140s 2s/step - loss: 1.0963 - accuracy: 0.4940 - val_loss: 0.7184 - val_accuracy: 0.4263
Epoch 2/6
92/92 [==============================] - 135s 1s/step - loss: 0.6899 - accuracy: 0.5336 - val_loss: 0.6922 - val_accuracy: 0.4969
Epoch 3/6
92/92 [==============================] - 138s 1s/step - loss: 0.6844 - accuracy: 0.5624 - val_loss: 0.6753 - val_accuracy: 0.6016
Epoch 4/6
92/92 [==============================] - 137s 1s/step - loss: 0.6772 - accuracy: 0.5886 - val_loss: 0.6805 - val_accuracy: 0.5667
Epoch 5/6
92/92 [==============================] - 137s 1s/step - loss: 0.6648 - accuracy: 0.6234 - val_loss: 0.6568 - val_accuracy: 0.6152
Epoch 6/6
92/92 [==============================] - 143s 2s/step - loss: 0.6421 - accuracy: 0.6638 - val_loss: 0.6456 - val_accuracy: 0.6486
```

```
from sklearn.metrics import classification_report

pred = model.predict(test_data)
pred = [1.0 if p>= 0.5 else 0.0 for p in pred]
print(classification_report(test_labels, pred))
```

```
271/271 [==============================] - 12s 42ms/step
              precision    recall  f1-score   support

           0       0.54      0.59      0.56      4144
           1       0.59      0.54      0.56      4504

    accuracy                           0.56      8648
   macro avg       0.56      0.56      0.56      8648
weighted avg       0.57      0.56      0.56      8648
```

Here the CNN has the same results of the RNN where it performs merely a little better.

```
from tensorflow.keras.layers.experimental.preprocessing import TextVectorization
vectorizer = TextVectorization(max_tokens=25000, output_sequence_length=500)
text_ds = tf.data.Dataset.from_tensor_slices(train.Text).batch(128)
vectorizer.adapt(text_ds)
```

```
voc = vectorizer.get_vocabulary()
word_index = dict(zip(voc, range(len(voc))))
```

```
test = ["the", "cat", "sat", "on", "the", "mat"]
[word_index[w] for w in test]
```

```
[2, 2332, 2602, 15, 2, 19089]
```

```
from tensorflow.keras import layers

EMBEDDING_DIM = 128
MAX_SEQUENCE_LENGTH = 200

embedding_layer = layers.Embedding(len(word_index) + 1,
                            EMBEDDING_DIM,
                            input_length=MAX_SEQUENCE_LENGTH)
```

```
x_train = vectorizer(np.array([[s] for s in train.Text])).numpy()
y_train = np.array(train_labels)
```

CNN Architecture with different type of Embedding & More Layers

```
from tensorflow import keras
int_sequences_input = keras.Input(shape=(None,), dtype="int64")
embedded_sequences = embedding_layer(int_sequences_input)
x = layers.Conv1D(128, 5, activation="relu")(embedded_sequences)
x = layers.MaxPooling1D(5)(x)
x = layers.Conv1D(128, 5, activation="relu")(x)
x = layers.MaxPooling1D(5)(x)
x = layers.Conv1D(128, 5, activation="relu")(x)
x = layers.GlobalMaxPooling1D()(x)
x = layers.Dense(128, activation="relu")(x)
x = layers.Dropout(0.5)(x)
preds = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(int_sequences_input, preds)
```

```
model.compile(
    loss="binary_crossentropy", optimizer="rmsprop", metrics=["acc"]
)
model.fit(x_train, y_train, batch_size=128, epochs=6, validation_split=0.2)
```

```
Epoch 1/6
215/215 [==============================] - 308s 1s/step - loss: 0.5944 - acc: 0.6512 - val_loss: 0.4578 - val_acc: 0.7784
Epoch 2/6
215/215 [==============================] - 314s 1s/step - loss: 0.2893 - acc: 0.8851 - val_loss: 0.5407 - val_acc: 0.7662
Epoch 3/6
215/215 [==============================] - 300s 1s/step - loss: 0.1415 - acc: 0.9454 - val_loss: 0.8470 - val_acc: 0.7321
Epoch 4/6
215/215 [==============================] - 282s 1s/step - loss: 0.0782 - acc: 0.9687 - val_loss: 0.9535 - val_acc: 0.7486
Epoch 5/6
215/215 [==============================] - 259s 1s/step - loss: 0.0521 - acc: 0.9770 - val_loss: 1.1756 - val_acc: 0.7445
Epoch 6/6
215/215 [==============================] - 264s 1s/step - loss: 0.0418 - acc: 0.9803 - val_loss: 1.4497 - val_acc: 0.7545
<keras.callbacks.History at 0x7f095a84b940>
```

```
test = df[~i]
```

```
data_test = test.Text
```

```
test_final = vectorizer(np.array([[s] for s in data_test])).numpy()
print(test_final.shape)
```

```
(8648, 500)
```

```
pred = model.predict(test_final)
pred = [1.0 if p>= 0.5 else 0.0 for p in pred]
print(classification_report(test.Sentiment, pred))
```

```
271/271 [==============================] - 24s 87ms/step
              precision    recall  f1-score   support

           0       0.90      0.90      0.90      4144
           1       0.91      0.90      0.91      4504

    accuracy                           0.90      8648
   macro avg       0.90      0.90      0.90      8648
weighted avg       0.90      0.90      0.90      8648
```

This proves that with the right complex model and proper embedding that the results can be significantly better.

Overall, for a simple sequential model it is able to produce results that are decent. For very simple RNN and CNN models, it produces results that are inadequate but presentable. It may be that the data needs to be preprocessed differently or the model itself needs to be increased in complexity. Additionally, when using RNN and CNN these models take a significant amount of time. Finally, for the final model of this assignment, using a larger model and appropriate embedding proved to be worthwhile although the time investment was more vast than the other models we had used before it.

✓ 23s completed at 3:14 PM