TextClassification2

April 22, 2023

Justin Hardy | JEH180008 | Dr. Mazidi | CS 4395.001

The purpose of this assignment is to explore deep learning, and implement them in problems related to Natural Language Processing.

1 Imports

[1]:
```
import pandas
import numpy
import math
import sys
import os
import re as regex
import tensorflow as tf
from keras import layers, models, regularizers, initializers
from keras.preprocessing.text import Tokenizer
from keras.layers.preprocessing.text_vectorization import TextVectorization
from sklearn.preprocessing import LabelEncoder
```

2 About the Data Set

The data set I'll be using for this assignment is a data sed I'd found on Kaggle, which is a data set that contains 50 thousand amazon reviews and their corresponding sentiments (negative/positive).

You can click this link to view the data set: IMDB Reviews Data Set

3 Reading in the Data Set

We'll start by reading in the data and putting it into a keras dataset object. Then split the data into train, test, and validate at an 70/20/10 split.

[2]:
```
# Read in the data set
df = pandas.read_csv('data/IMDB Dataset.csv', header=0, encoding='utf-8',␣
 ↪keep_default_na=False)

# Split into train, test, validate
numpy.random.seed(1234) # seed for reproducibility
```

```
i = numpy.random.rand(len(df)) < 0.7
train = df[i]
test_val = df[~i]
i = numpy.random.rand(len(test_val)) < 0.67
test = test_val[i]
val = test_val[~i]

# Print train/test/val shapes
print(train.shape)
print(test.shape)
print(val.shape)
print()
print('train preview:\n' + str(train.head()))
```

```
(34947, 2)
(10016, 2)
(5037, 2)

train preview:
                                              review sentiment
0  One of the other reviewers has mentioned that …  positive
1  A wonderful little production. <br /><br />The…  positive
2  I thought this was a wonderful way to spend ti…  positive
5  Probably my all-time favorite movie, a story o…  positive
6  I sure would like to see a resurrection of a u…  positive
```

4 Text Preprocessing

To process the text, we'll need to tokenize & fit the train data's feature column, as well as encode the train data's label column.

[3]:
```
# Specify model settings
num_labels = 2
vocab_size = 30000
batch_size = 100

# Fit the tokenizer to the training data
tokenizer = Tokenizer(num_words=vocab_size)
tokenizer.fit_on_texts(train.review)

# Convert review texts to matrices, which will be our features
x_train = tokenizer.texts_to_matrix(train.review, mode='tfidf')
x_test = tokenizer.texts_to_matrix(test.review, mode='tfidf')
x_val = tokenizer.texts_to_matrix(val.review, mode='tfidf')

# Fit the encoder to the training data
encoder = LabelEncoder()
```

```
encoder.fit(train.sentiment)
y_train = encoder.transform(train.sentiment)
y_test = encoder.transform(test.sentiment)
y_val = encoder.transform(val.sentiment)

# Print shapes
print('train shapes:', x_train.shape, y_train.shape)
print('test shapes:', x_test.shape, y_test.shape)
print('val shapes:', x_val.shape, y_val.shape)
```

```
train shapes: (34947, 30000) (34947,)
test shapes: (10016, 30000) (10016,)
val shapes: (5037, 30000) (5037,)
```

5 Training The Models

For the Machine Learning models, we'll create a basic sequential model and an RNN model, making two attempts at each. The first attempt will be a simple version of the model, while the second attempt will be my attempt at an improved version of the simple model. Any things I tried that didn't make it into the final version of the second attempt will be noted in my explanation of the model.

5.1 Basic Sequential Model (first attempt)

In this attempt, I'll create a basic sequential model using Keras.

5.1.1 Training

[4]:
```
# Specificy model settings
epochs = 30

# Create the sequential model & fit
sm1 = models.Sequential()
sm1.add(layers.Dense(32, input_dim=vocab_size, kernel_initializer='normal',␣
 ↪activation='relu'))
sm1.add(layers.Dense(1, kernel_initializer='normal', activation='sigmoid'))
sm1.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
sm1_history = sm1.fit(x_train, y_train, batch_size=batch_size, epochs=epochs,␣
 ↪verbose=1, validation_data=(x_val, y_val))
```

```
Epoch 1/30
350/350 [==============================] - 14s 37ms/step - loss: 0.3361 -
accuracy: 0.8584 - val_loss: 0.2672 - val_accuracy: 0.8976
Epoch 2/30
350/350 [==============================] - 5s 14ms/step - loss: 0.1099 -
accuracy: 0.9623 - val_loss: 0.3131 - val_accuracy: 0.8972
Epoch 3/30
350/350 [==============================] - 5s 14ms/step - loss: 0.0405 -
```

```
accuracy: 0.9893 - val_loss: 0.4039 - val_accuracy: 0.8914
Epoch 4/30
350/350 [==============================] - 5s 13ms/step - loss: 0.0145 -
accuracy: 0.9971 - val_loss: 0.4784 - val_accuracy: 0.8918
Epoch 5/30
350/350 [==============================] - 5s 13ms/step - loss: 0.0055 -
accuracy: 0.9995 - val_loss: 0.5477 - val_accuracy: 0.8924
Epoch 6/30
350/350 [==============================] - 5s 13ms/step - loss: 0.0026 -
accuracy: 0.9998 - val_loss: 0.5950 - val_accuracy: 0.8910
Epoch 7/30
350/350 [==============================] - 5s 13ms/step - loss: 0.0015 -
accuracy: 1.0000 - val_loss: 0.6346 - val_accuracy: 0.8900
Epoch 8/30
350/350 [==============================] - 5s 13ms/step - loss: 9.5743e-04 -
accuracy: 1.0000 - val_loss: 0.6666 - val_accuracy: 0.8898
Epoch 9/30
350/350 [==============================] - 5s 14ms/step - loss: 6.6766e-04 -
accuracy: 1.0000 - val_loss: 0.6965 - val_accuracy: 0.8892
Epoch 10/30
350/350 [==============================] - 5s 13ms/step - loss: 4.8596e-04 -
accuracy: 1.0000 - val_loss: 0.7230 - val_accuracy: 0.8900
Epoch 11/30
350/350 [==============================] - 5s 13ms/step - loss: 3.6203e-04 -
accuracy: 1.0000 - val_loss: 0.7487 - val_accuracy: 0.8896
Epoch 12/30
350/350 [==============================] - 5s 13ms/step - loss: 2.7589e-04 -
accuracy: 1.0000 - val_loss: 0.7724 - val_accuracy: 0.8894
Epoch 13/30
350/350 [==============================] - 5s 13ms/step - loss: 2.1346e-04 -
accuracy: 1.0000 - val_loss: 0.7949 - val_accuracy: 0.8894
Epoch 14/30
350/350 [==============================] - 5s 13ms/step - loss: 1.6714e-04 -
accuracy: 1.0000 - val_loss: 0.8168 - val_accuracy: 0.8884
Epoch 15/30
350/350 [==============================] - 5s 14ms/step - loss: 1.3208e-04 -
accuracy: 1.0000 - val_loss: 0.8386 - val_accuracy: 0.8886
Epoch 16/30
350/350 [==============================] - 5s 14ms/step - loss: 1.0514e-04 -
accuracy: 1.0000 - val_loss: 0.8589 - val_accuracy: 0.8884
Epoch 17/30
350/350 [==============================] - 5s 14ms/step - loss: 8.4208e-05 -
accuracy: 1.0000 - val_loss: 0.8795 - val_accuracy: 0.8878
Epoch 18/30
350/350 [==============================] - 5s 15ms/step - loss: 6.7712e-05 -
accuracy: 1.0000 - val_loss: 0.8999 - val_accuracy: 0.8874
Epoch 19/30
350/350 [==============================] - 5s 14ms/step - loss: 5.4651e-05 -
```

```
accuracy: 1.0000 - val_loss: 0.9199 - val_accuracy: 0.8870
Epoch 20/30
350/350 [==============================] - 5s 14ms/step - loss: 4.4246e-05 -
accuracy: 1.0000 - val_loss: 0.9393 - val_accuracy: 0.8870
Epoch 21/30
350/350 [==============================] - 5s 14ms/step - loss: 3.5883e-05 -
accuracy: 1.0000 - val_loss: 0.9589 - val_accuracy: 0.8870
Epoch 22/30
350/350 [==============================] - 5s 14ms/step - loss: 2.9215e-05 -
accuracy: 1.0000 - val_loss: 0.9786 - val_accuracy: 0.8860
Epoch 23/30
350/350 [==============================] - 5s 14ms/step - loss: 2.3834e-05 -
accuracy: 1.0000 - val_loss: 0.9977 - val_accuracy: 0.8860
Epoch 24/30
350/350 [==============================] - 5s 14ms/step - loss: 1.9439e-05 -
accuracy: 1.0000 - val_loss: 1.0168 - val_accuracy: 0.8860
Epoch 25/30
350/350 [==============================] - 5s 14ms/step - loss: 1.5916e-05 -
accuracy: 1.0000 - val_loss: 1.0364 - val_accuracy: 0.8852
Epoch 26/30
350/350 [==============================] - 5s 14ms/step - loss: 1.3033e-05 -
accuracy: 1.0000 - val_loss: 1.0549 - val_accuracy: 0.8852
Epoch 27/30
350/350 [==============================] - 5s 14ms/step - loss: 1.0682e-05 -
accuracy: 1.0000 - val_loss: 1.0740 - val_accuracy: 0.8852
Epoch 28/30
350/350 [==============================] - 5s 14ms/step - loss: 8.7572e-06 -
accuracy: 1.0000 - val_loss: 1.0929 - val_accuracy: 0.8856
Epoch 29/30
350/350 [==============================] - 5s 13ms/step - loss: 7.2006e-06 -
accuracy: 1.0000 - val_loss: 1.1115 - val_accuracy: 0.8854
Epoch 30/30
350/350 [==============================] - 5s 13ms/step - loss: 5.9149e-06 -
accuracy: 1.0000 - val_loss: 1.1303 - val_accuracy: 0.8856
```

5.1.2 Evaluation

[5]:
```
# Evaluate Accuracy
sm1_score = sm1.evaluate(x_test, y_test, batch_size=batch_size, verbose=1)
print('Accuracy:', sm1_score[1])
```

```
101/101 [==============================] - 1s 5ms/step - loss: 1.1396 -
accuracy: 0.8819
Accuracy: 0.8818889856338501
```

As we can see, the algorithm achieved an 88% accuracy. From the verbose output of the model training, we can observe that the model without a doubt overfitted the train data, resulting in perfect prediction accuracy on the train data, at the cost of prediction accuracy on data outside of the train data (ie test & validate). In other words, as the model continued to train, it became less

and less effective at generalizing the data. We'll look to remedy that in our next attempt.

5.2 Basic Sequential Model (Second Attempt)

With this attempt, I wanted to do my best to prevent the model from overfitting, while improving its accuracy.

5.2.1 Training

[6]:
```
# Specificy model settings
epochs = 30

# Create the sequential model & fit
sm2 = models.Sequential()
sm2.add(layers.Dense(64, input_dim=vocab_size, kernel_initializer='normal',␣
 ↪activation='relu', kernel_regularizer=regularizers.l2(l2=0.001)))
sm2.add(layers.Dropout(0.75))
sm2.add(layers.Dense(32, input_dim=vocab_size, kernel_initializer='normal',␣
 ↪activation='relu', kernel_regularizer=regularizers.l2(l2=0.001)))
sm2.add(layers.Dropout(0.5))
sm2.add(layers.Dense(1, kernel_initializer='normal', activation='sigmoid'))
sm2.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
sm2_history = sm2.fit(x_train, y_train, batch_size=batch_size, epochs=epochs,␣
 ↪verbose=1, validation_data=(x_val, y_val))
```

```
Epoch 1/30
350/350 [==============================] - 16s 42ms/step - loss: 1.2152 -
accuracy: 0.7633 - val_loss: 0.5606 - val_accuracy: 0.8972
Epoch 2/30
350/350 [==============================] - 9s 27ms/step - loss: 0.5880 -
accuracy: 0.8948 - val_loss: 0.5755 - val_accuracy: 0.8986
Epoch 3/30
350/350 [==============================] - 9s 26ms/step - loss: 0.5733 -
accuracy: 0.9097 - val_loss: 0.6002 - val_accuracy: 0.8999
Epoch 4/30
350/350 [==============================] - 9s 26ms/step - loss: 0.5741 -
accuracy: 0.9134 - val_loss: 0.6125 - val_accuracy: 0.9003
Epoch 5/30
350/350 [==============================] - 9s 26ms/step - loss: 0.5760 -
accuracy: 0.9182 - val_loss: 0.6255 - val_accuracy: 0.8984
Epoch 6/30
350/350 [==============================] - 9s 27ms/step - loss: 0.5929 -
accuracy: 0.9166 - val_loss: 0.6407 - val_accuracy: 0.8999
Epoch 7/30
350/350 [==============================] - 9s 26ms/step - loss: 0.5850 -
accuracy: 0.9203 - val_loss: 0.6406 - val_accuracy: 0.9001
Epoch 8/30
350/350 [==============================] - 9s 26ms/step - loss: 0.5911 -
```

```
accuracy: 0.9200 - val_loss: 0.6433 - val_accuracy: 0.9009
Epoch 9/30
350/350 [==============================] - 9s 26ms/step - loss: 0.5901 -
accuracy: 0.9205 - val_loss: 0.6493 - val_accuracy: 0.8987
Epoch 10/30
350/350 [==============================] - 9s 26ms/step - loss: 0.5962 -
accuracy: 0.9206 - val_loss: 0.6643 - val_accuracy: 0.8982
Epoch 11/30
350/350 [==============================] - 9s 26ms/step - loss: 0.6016 -
accuracy: 0.9209 - val_loss: 0.6569 - val_accuracy: 0.8968
Epoch 12/30
350/350 [==============================] - 9s 27ms/step - loss: 0.5999 -
accuracy: 0.9207 - val_loss: 0.6666 - val_accuracy: 0.8968
Epoch 13/30
350/350 [==============================] - 10s 28ms/step - loss: 0.6051 -
accuracy: 0.9206 - val_loss: 0.6539 - val_accuracy: 0.9001
Epoch 14/30
350/350 [==============================] - 10s 28ms/step - loss: 0.5949 -
accuracy: 0.9203 - val_loss: 0.6493 - val_accuracy: 0.8993
Epoch 15/30
350/350 [==============================] - 10s 27ms/step - loss: 0.5920 -
accuracy: 0.9229 - val_loss: 0.6577 - val_accuracy: 0.9019
Epoch 16/30
350/350 [==============================] - 10s 27ms/step - loss: 0.5946 -
accuracy: 0.9248 - val_loss: 0.6616 - val_accuracy: 0.8970
Epoch 17/30
350/350 [==============================] - 10s 28ms/step - loss: 0.5985 -
accuracy: 0.9227 - val_loss: 0.6554 - val_accuracy: 0.8995
Epoch 18/30
350/350 [==============================] - 10s 29ms/step - loss: 0.5916 -
accuracy: 0.9231 - val_loss: 0.6541 - val_accuracy: 0.8993
Epoch 19/30
350/350 [==============================] - 10s 28ms/step - loss: 0.5852 -
accuracy: 0.9241 - val_loss: 0.6532 - val_accuracy: 0.9037
Epoch 20/30
350/350 [==============================] - 10s 27ms/step - loss: 0.6010 -
accuracy: 0.9231 - val_loss: 0.6655 - val_accuracy: 0.8964
Epoch 21/30
350/350 [==============================] - 10s 28ms/step - loss: 0.5958 -
accuracy: 0.9227 - val_loss: 0.6519 - val_accuracy: 0.8991
Epoch 22/30
350/350 [==============================] - 10s 28ms/step - loss: 0.5917 -
accuracy: 0.9226 - val_loss: 0.6467 - val_accuracy: 0.8970
Epoch 23/30
350/350 [==============================] - 10s 27ms/step - loss: 0.5905 -
accuracy: 0.9237 - val_loss: 0.6500 - val_accuracy: 0.8991
Epoch 24/30
350/350 [==============================] - 10s 28ms/step - loss: 0.5942 -
```

```
accuracy: 0.9233 - val_loss: 0.6550 - val_accuracy: 0.8982
Epoch 25/30
350/350 [==============================] - 9s 27ms/step - loss: 0.5967 -
accuracy: 0.9210 - val_loss: 0.6639 - val_accuracy: 0.8924
Epoch 26/30
350/350 [==============================] - 9s 27ms/step - loss: 0.5943 -
accuracy: 0.9249 - val_loss: 0.6557 - val_accuracy: 0.8962
Epoch 27/30
350/350 [==============================] - 9s 27ms/step - loss: 0.5963 -
accuracy: 0.9216 - val_loss: 0.6531 - val_accuracy: 0.8972
Epoch 28/30
350/350 [==============================] - 9s 27ms/step - loss: 0.5959 -
accuracy: 0.9241 - val_loss: 0.6532 - val_accuracy: 0.8997
Epoch 29/30
350/350 [==============================] - 9s 27ms/step - loss: 0.5982 -
accuracy: 0.9226 - val_loss: 0.6597 - val_accuracy: 0.9001
Epoch 30/30
350/350 [==============================] - 9s 27ms/step - loss: 0.5912 -
accuracy: 0.9253 - val_loss: 0.6539 - val_accuracy: 0.9005
```

5.2.2 Evaluation

[7]:
```
# Evaluate Accuracy
sm2_score = sm2.evaluate(x_test, y_test, batch_size=batch_size, verbose=1)
print('Accuracy:', sm2_score[1])
```

```
101/101 [==============================] - 1s 7ms/step - loss: 0.6663 -
accuracy: 0.8961
Accuracy: 0.8960663080215454
```

I approached this attempt by applying various overfit-reduction techniques to my model. Essentially, I'd split the model between two dense layers of various sizes, and applied an L2 Regularizer to it. L2 Regularization (AKA Ridge Regularization) helps to prevent overfitting by forcing the function to reduce weight magnitude. Additionally, I'd added a Dropout layer between the two dense layers to randomly drop a percentage of the neurons in the layer during the epoch. With this, I'd observed that lower values don't affect the overfitting model as much.

Overall, I believe this attempt better generalizes the data than the first attempt, especially as it achieve approximately 90% accuracy on the test data.

5.3 CNN + Pretrained Embeddings

In this attempt, I'll create another sequential model using an CNN layer, and experiment with some embeddings.

5.3.1 GloVe Embedding

[8]:
```
# Get glove path
path_to_glove_file = os.path.join("data", "glove.6B.100d.txt")

# Load embedding word vectors
embeddings_index = {}
with open(path_to_glove_file, encoding='utf8') as f:
    for line in f:
        word, coefs = line.split(maxsplit=1)
        coefs = numpy.fromstring(coefs, "f", sep=" ")
        embeddings_index[word] = coefs

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

```
Found 400000 word vectors.
```

[9]:
```
# Set up the vectorizer
vectorizer = TextVectorization(max_tokens=vocab_size,␣
 ↪output_sequence_length=200)
text_ds = tf.data.Dataset.from_tensor_slices(train.review).batch(100)
vectorizer.adapt(text_ds)
voc = vectorizer.get_vocabulary()
word_index = dict(zip(voc, range(len(voc))))

# Create embedding matrix
num_tokens = len(voc) + 2
embedding_dim = 100
hits = misses = 0

embedding_matrix = numpy.zeros((num_tokens, embedding_dim))
for word, i in word_index.items():
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        embedding_matrix[i] = embedding_vector
        hits += 1
    else:
        misses += 1
print("Converted %d words (%d misses)" % (hits, misses))
```

```
Converted 27131 words (2869 misses)
```

5.3.2 Training

[10]:
```
#  Specificy model settings
epochs = 10
batch_size = 100
```

```
# Create the sequential model
cnn1 = models.Sequential()
cnn1.add(layers.Embedding(num_tokens, embedding_dim,␣
 ↪embeddings_initializer=initializers.Constant(embedding_matrix),␣
 ↪trainable=False))
cnn1.add(layers.Conv1D(128, 5, activation='relu'))
cnn1.add(layers.MaxPooling1D(5))
cnn1.add(layers.Conv1D(128, 5, activation='relu'))
cnn1.add(layers.GlobalMaxPooling1D())
cnn1.add(layers.Dense(128, activation='relu'))
cnn1.add(layers.Dropout(0.5))
cnn1.add(layers.Dense(2, activation='softmax'))
cnn1.summary()

# Update train, test, validate to use new vectorizer
x_train = vectorizer(numpy.array([[s] for s in train.review])).numpy()
x_val = vectorizer(numpy.array([[s] for s in val.review])).numpy()
x_test = vectorizer(numpy.array([[s] for s in test.review])).numpy()

train.sentiment = train.sentiment.astype('category').cat.codes
test.sentiment = test.sentiment.astype('category').cat.codes
val.sentiment = val.sentiment.astype('category').cat.codes
y_train = numpy.array(train.sentiment)
y_val = numpy.array(val.sentiment)
y_test = numpy.array(test.sentiment)

# Fit the model
cnn1.compile(loss='sparse_categorical_crossentropy', optimizer='rmsprop',␣
 ↪metrics=['accuracy'])
cnn1_history = cnn1.fit(x_train, y_train, batch_size=batch_size, epochs=epochs,␣
 ↪verbose=1, validation_data=(x_val, y_val))
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, None, 100)         3000200

 conv1d (Conv1D)             (None, None, 128)         64128

 max_pooling1d (MaxPooling1D  (None, None, 128)        0
 )

 conv1d_1 (Conv1D)           (None, None, 128)         82048

 global_max_pooling1d (Globa  (None, 128)              0
 lMaxPooling1D)
```

```
 dense_5 (Dense)             (None, 128)               16512

 dropout_2 (Dropout)         (None, 128)               0

 dense_6 (Dense)             (None, 2)                 258

=================================================================
Total params: 3,163,146
Trainable params: 162,946
Non-trainable params: 3,000,200
_________________________________________________________________
Epoch 1/10

C:\Users\Justi\AppData\Local\Temp\ipykernel_47420\3171375614.py:22:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  train.sentiment = train.sentiment.astype('category').cat.codes
C:\Users\Justi\AppData\Local\Temp\ipykernel_47420\3171375614.py:23:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  test.sentiment = test.sentiment.astype('category').cat.codes
C:\Users\Justi\AppData\Local\Temp\ipykernel_47420\3171375614.py:24:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  val.sentiment = val.sentiment.astype('category').cat.codes

350/350 [==============================] - 19s 53ms/step - loss: 0.5959 -
accuracy: 0.6758 - val_loss: 0.4562 - val_accuracy: 0.7812
Epoch 2/10
350/350 [==============================] - 18s 51ms/step - loss: 0.4366 -
accuracy: 0.7983 - val_loss: 0.4420 - val_accuracy: 0.7852
Epoch 3/10
350/350 [==============================] - 18s 51ms/step - loss: 0.3808 -
accuracy: 0.8282 - val_loss: 0.3648 - val_accuracy: 0.8358
Epoch 4/10
350/350 [==============================] - 18s 51ms/step - loss: 0.3373 -
```

```
accuracy: 0.8524 - val_loss: 0.3995 - val_accuracy: 0.8227
Epoch 5/10
350/350 [==============================] - 18s 51ms/step - loss: 0.2957 -
accuracy: 0.8720 - val_loss: 0.3558 - val_accuracy: 0.8406
Epoch 6/10
350/350 [==============================] - 18s 51ms/step - loss: 0.2533 -
accuracy: 0.8946 - val_loss: 0.3825 - val_accuracy: 0.8338
Epoch 7/10
350/350 [==============================] - 18s 52ms/step - loss: 0.2147 -
accuracy: 0.9108 - val_loss: 0.6107 - val_accuracy: 0.7741
Epoch 8/10
350/350 [==============================] - 18s 51ms/step - loss: 0.1816 -
accuracy: 0.9274 - val_loss: 0.4914 - val_accuracy: 0.8376
Epoch 9/10
350/350 [==============================] - 18s 51ms/step - loss: 0.1531 -
accuracy: 0.9429 - val_loss: 0.6339 - val_accuracy: 0.8114
Epoch 10/10
350/350 [==============================] - 18s 51ms/step - loss: 0.1314 -
accuracy: 0.9507 - val_loss: 0.5528 - val_accuracy: 0.8436
```

5.3.3 Evaluation

[11]:
```
# Predict off of the test data
cnn1_score = cnn1.evaluate(x_test, y_test, batch_size=batch_size, verbose=1)
print('Accuracy:', cnn1_score[1])
```

```
101/101 [==============================] - 2s 18ms/step - loss: 0.5608 -
accuracy: 0.8314
Accuracy: 0.8313698172569275
```

As you can see, the CNN model performed worse overall than my previous attempts - scoring an 83% accuracy - but it is likely a sign that the choice in layers and their values needs to be altered to get better accuracy. It took me a while to figure out how to load up the pretrained embedding, but it was interesting to be able to try it out at least once here. I recall for Machine Learning, CNN performed really well on image classification problems, and I can anticipate RNN performing better for text classification problems.

6 Final Remarks

It's pretty clear that Deep Learning does extremely well in text classification problems as a whole. In comparison to my previous text classification assignment - which used a similar review-based sentiment analysis data set - My first two sequential models performed better than most of the models I'd created in that previous assignment, albeit the sizes of both data sets are not equal. Additionally, I believe that if I spent more time with the CNN model I'd created, that I would be able to get it to perform better than its preceding sequential models.