Classifying movie reviews: a binary classification example

Design a neural network to perform two-class classification or *binary classification* , of reviews form IMDB movie reviews dataset, to determine wether the reviews are positive or negative. We will use the Python library Keras to perform the classification

The IMDB Dataset

The IMDB dataset is a set of 50,000 highly polarized reviews from the Internet Movie Database. They are split into 25000 reviews each for training and testing. Each set contains equal number (50%) of positive and negative reviews.

The IMDB dataset comes packaged with Keras. It consists of reviews and their corresponding labels (0 for *negative* and 1 for *positive* review). The reviews are a sequence of words. They come preprocessed as sequence of integers, where each integer stands for a specific word in the dictionary.

The IMDB datset can be loaded directly from Keras and will usually download about 80 MB on your machine.

Import Packages

In [1]:
```
import numpy as np
from keras.datasets import imdb
from keras import models
from keras import layers
from keras import optimizers
from keras import losses
from keras import metrics


import matplotlib.pyplot as plt
%matplotlib inline
```

Loading the Data

In [2]:
```
# Load the data, keeping only 10,000 of the most frequently occuring words
(train_data, train_labels), (test_data, test_labels) = imdb.load_data(num_words = 10000)
```

In [3]:
```
# Check the first label
train_labels[0]
```

Out[3]:
```
1
```

In [4]:
```
# Since we restricted ourselves to the top 10000 frequent words, no word index should exceed
# we'll verify this below

# Here is a list of maximum indexes in every review --- we search the maximum index in this
print(type([max(sequence) for sequence in train_data]))

# Find the maximum of all max indexes
max([max(sequence) for sequence in train_data])
```

```
<class 'list'>
```

Out[4]:
```
9999
```

In [5]:
```
# Let's quickly decode a review

# step 1: load the dictionary mappings from word to integer index
word_index = imdb.get_word_index()

# step 2: reverse word index to map integer indexes to their respective words
reverse_word_index = dict([(value, key) for (key, value) in word_index.items()])

# Step 3: decode the review, mapping integer indices to words
#
# indices are off by 3 because 0, 1, and 2 are reserverd indices for "padding", "Start of se
decoded_review = ' '.join([reverse_word_index.get(i-3, '?') for i in train_data[0]])

decoded_review
```

Out[5]:
```
"? this film was just brilliant casting location scenery story direction everyone's really
suited the part they played and you could just imagine being there robert ? is an amazing a
ctor and now the same being director ? father came from the same scottish island as myself
so i loved the fact there was a real connection with this film the witty remarks throughout
the film were great it was just brilliant so much that i bought the film as soon as it was
released for ? and would recommend it to everyone to watch and the fly fishing was amazing
really cried at the end it was so sad and you know what they say if you cry at a film it mu
st have been good and this definitely was also ? to the two little boy's that played the ?
of norman and paul they were just brilliant children are often left out of the ? list i thi
nk because the stars that play them all grown up are such a big profile for the whole film
but these children are amazing and should be praised for what they have done don't you thin
k the whole story was so lovely because it was true and was someone's life after all that w
as shared with us all"
```

In [6]:
```
len(reverse_word_index)
```

Out[6]:
```
88584
```

Preparing the data

Vectorize input data

We cannot feed list of integers into our deep neural network. We will need to convert them into tensors.

To prepare our data we will One-hot Encode our lists and turn them into vectors of 0's and 1's. This would blow up all of our sequences into 10,000 dimensional vectors containing 1 at all indices corresponding to integers present in that sequence. This vector will have the element 0 at all indices which are not present in integer sequence.

Simply put, the 10,000 dimensional vector corresponding to each review, will have

- Every index corresponding to a word
- Every index vith value 1, is a word which is present in the review and is denoted by its integer counterpart
- Every index containing 0, is a word not present in the review

We will vectorize our data manually for maximum clarity. This will result in a tensors of shape (25000, 10000).

In [7]:
```
def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))    # Creates an all zero matrix of shape
    for i,sequence in enumerate(sequences):
        results[i,sequence] = 1                        # Sets specific indices of results[i]
    return results

# Vectorize training Data
X_train = vectorize_sequences(train_data)

# Vectorize testing Data
X_test = vectorize_sequences(test_data)
```

In [8]:
```
X_train[0]
```

Out[8]:
```
array([0., 1., 1., ..., 0., 0., 0.])
```

In [9]:
```
X_train.shape
```

Out[9]:
```
(25000, 10000)
```

Vectorize labels

In [10]:
```
y_train = np.asarray(train_labels).astype('float32')
y_test  = np.asarray(test_labels).astype('float32')
```

Building the network

Our input data is vectors which needs to be mapped to scaler labels (0s and 1s). This is one of the easiest setups and a simple stack of *fully-connected*, *Dense* layers with *relu* activation perform quite well.

Hidden layers

In this network we will leverage *hidden layers*. we will define our layers as such.

```
Dense(16, activation='relu')
```

The argument being passed to each `Dense` layer, `(16)` is the number of *hidden units* of a layer.

The output from a *Dense* layer with *relu* activation is genrated after a chain of *tensor* operations. This chain of operations is implemented as

```
output = relu(dot(W, input) + b)
```

Where, `W` is the *Weight matrix* and `b` is the bias (tensor).

Having 16 hidden units means that the matrix W will be of the shape (*input_Dimension* , *16*). In this case where the dimension of input vector is 10,000; the shape of Weight matrix will be (10000, 16). If you were to represent this network as graph you would see 16 neurons in this hidden layer.

To put in in laymans terms, there will be 16 balls in this layer.

Each of these balls, or *hidden units* is a dimension in the representation space of the layer. Representaion space is the set of all viable representaions for the data. Every *hidden layer* composed of its *hidden units* aims to learns one specific transformation of the data, or one feature/pattern from the data.

Hidden layers, simply put, are layers of mathematical functions each designed to produce an output specific to an intended result. Hidden layers allow for the function of a neural network to be broken down into specific transformations of the data. Each hidden layer function is specialized to produce a defined output.For example, a hidden layer functions that are used to identify human eyes and ears may be used in conjunction by subsequent layers to identify faces in images. While the functions to identify eyes alone are not enough to independently recognize objects, they can function jointly within a neural network.

Model Architecture

1. For our model we will use
    - two intermediate layers with 16 hidden units each
    - Third layer that will output the scalar sentiment prediction
2. Intermediate layers will use *relu* activation function. *relu* or Rectified linear unit function will zero out the negative values.
3. Sigmoid activation for the final layer or *output layer*. A sigmoid function "*squashes*" arbitary values into the [0,1] range.

There are formal principles that guide our approach in selecting the architectural attributes of a model. These are not

Model defination

In [11]:
```
model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

Compiling the model

In this step we will choose an *optimizer*, a *loss function*, and metrics to observe. We will go forward with

- *binary_crossentropy* loss function, commonlu used for Binary Classification
- *rmsprop* optimizer and
- *accuracy* as a measure of performance

We can pass our choices for optimizer, loss function and metrics as *strings* to the `compile` function because `rmsprop`, `binary_crossentropy` and `accuracy` come packaged with Keras.

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

One could use a customized loss function or ortimizer by passing the custom *class instance* as argument to the `loss`, `optimizer` or `mertics` fields.

In this example, we will implement our default choices, but, we will do so by passing class instances. This is exactly how we would do it, if we had customized parameters.

In [12]:
```
model.compile(
    optimizer=optimizers.RMSprop(learning_rate=0.001),
    loss = losses.binary_crossentropy,
    metrics = [metrics.binary_accuracy]
)
```

Setting up Validation

We will set aside a part of our training data for *validation* of the accuracy of the model as it trains. A *validation set* enables us to monitor the progress of our model on previously unseen data as it goes throug epochs during training.

Validation steps help us fine tune the training parameters of the `model.fit` function so as to avoid overfitting and under fitting of data.

In [13]:
```
# Input for Validation
X_val = X_train[:10000]
partial_X_train = X_train[10000:]

# Labels for validation
y_val = y_train[:10000]
partial_y_train = y_train[10000:]
```

Training our model

Initially, we will train our models for 20 epochs in mini-batches of 512 samples. We will aslo pass our *validation set* to the `fit` method.

Calling the `fit` method returns a `History` object. This object contains a member `history` which stores all data

In [14]:
```
history = model.fit(
    partial_X_train,
    partial_y_train,
    epochs=20,
    batch_size=512,
    validation_data=(X_val, y_val)
)
```

```
Epoch 1/20
30/30 [==============================] - 2s 37ms/step - loss: 0.5156 - binary_accuracy: 0.7
949 - val_loss: 0.3994 - val_binary_accuracy: 0.8627
Epoch 2/20
30/30 [==============================] - 0s 16ms/step - loss: 0.3162 - binary_accuracy: 0.8
993 - val_loss: 0.3088 - val_binary_accuracy: 0.8886
Epoch 3/20
30/30 [==============================] - 0s 16ms/step - loss: 0.2303 - binary_accuracy: 0.9
267 - val_loss: 0.2881 - val_binary_accuracy: 0.8869
Epoch 4/20
30/30 [==============================] - 0s 15ms/step - loss: 0.1831 - binary_accuracy: 0.9
411 - val_loss: 0.2792 - val_binary_accuracy: 0.8897
Epoch 5/20
30/30 [==============================] - 0s 15ms/step - loss: 0.1514 - binary_accuracy: 0.9
501 - val_loss: 0.2767 - val_binary_accuracy: 0.8884
Epoch 6/20
30/30 [==============================] - 0s 15ms/step - loss: 0.1214 - binary_accuracy: 0.9
642 - val_loss: 0.3112 - val_binary_accuracy: 0.8781
Epoch 7/20
30/30 [==============================] - 0s 15ms/step - loss: 0.1041 - binary_accuracy: 0.9
694 - val_loss: 0.3267 - val_binary_accuracy: 0.8804
Epoch 8/20
30/30 [==============================] - 0s 14ms/step - loss: 0.0830 - binary_accuracy: 0.9
779 - val_loss: 0.3235 - val_binary_accuracy: 0.8807
Epoch 9/20
30/30 [==============================] - 0s 14ms/step - loss: 0.0707 - binary_accuracy: 0.9
798 - val_loss: 0.3542 - val_binary_accuracy: 0.8775
Epoch 10/20
30/30 [==============================] - 0s 14ms/step - loss: 0.0553 - binary_accuracy: 0.9
858 - val_loss: 0.3724 - val_binary_accuracy: 0.8774
Epoch 11/20
30/30 [==============================] - 0s 14ms/step - loss: 0.0457 - binary_accuracy: 0.9
889 - val_loss: 0.4186 - val_binary_accuracy: 0.8712
Epoch 12/20
30/30 [==============================] - 0s 15ms/step - loss: 0.0381 - binary_accuracy: 0.9
915 - val_loss: 0.4310 - val_binary_accuracy: 0.8773
Epoch 13/20
30/30 [==============================] - 0s 15ms/step - loss: 0.0305 - binary_accuracy: 0.9
941 - val_loss: 0.4663 - val_binary_accuracy: 0.8751
Epoch 14/20
30/30 [==============================] - 0s 14ms/step - loss: 0.0242 - binary_accuracy: 0.9
953 - val_loss: 0.5045 - val_binary_accuracy: 0.8726
Epoch 15/20
30/30 [==============================] - 0s 15ms/step - loss: 0.0192 - binary_accuracy: 0.9
966 - val_loss: 0.5289 - val_binary_accuracy: 0.8713
Epoch 16/20
30/30 [==============================] - 0s 14ms/step - loss: 0.0162 - binary_accuracy: 0.9
971 - val_loss: 0.5600 - val_binary_accuracy: 0.8704
Epoch 17/20
30/30 [==============================] - 0s 14ms/step - loss: 0.0111 - binary_accuracy: 0.9
987 - val_loss: 0.6111 - val_binary_accuracy: 0.8679
Epoch 18/20
30/30 [==============================] - 0s 14ms/step - loss: 0.0082 - binary_accuracy: 0.9
995 - val_loss: 0.6720 - val_binary_accuracy: 0.8633
Epoch 19/20
30/30 [==============================] - 0s 14ms/step - loss: 0.0106 - binary_accuracy: 0.9
978 - val_loss: 0.6709 - val_binary_accuracy: 0.8653
Epoch 20/20
30/30 [==============================] - 0s 14ms/step - loss: 0.0039 - binary_accuracy: 0.9
999 - val_loss: 0.6942 - val_binary_accuracy: 0.8653
```

At the end of training we have attained a training accuracy of 99.85% and validation accuracy of 86.57%

Now that we have trained our network, we will observe its performance metrics stored in the `History` object.

Calling the `fit` method returns a `History` object. This object has an sttribute `history` which is a dictionary containing four enteries: one per monitored metric.

In [15]:
```
history_dict = history.history
history_dict.keys()
```

Out[15]:
```
dict_keys(['loss', 'binary_accuracy', 'val_loss', 'val_binary_accuracy'])
```

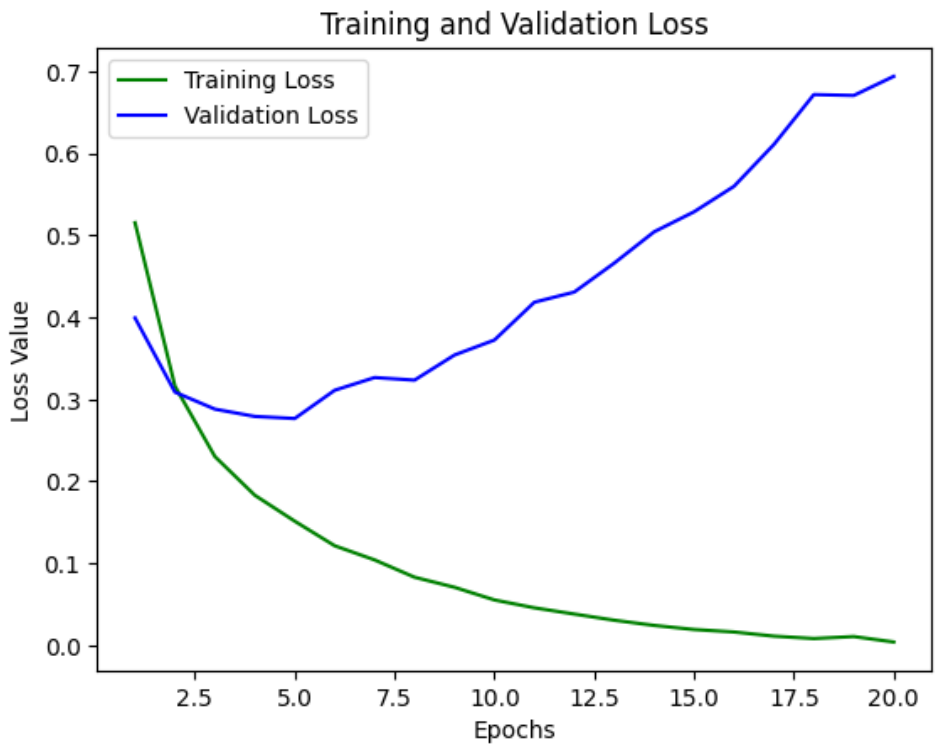
`history_dict` contains values of

- Training loss
- Trainining Accuracy
- Validation Loss
- Validation Accuracy

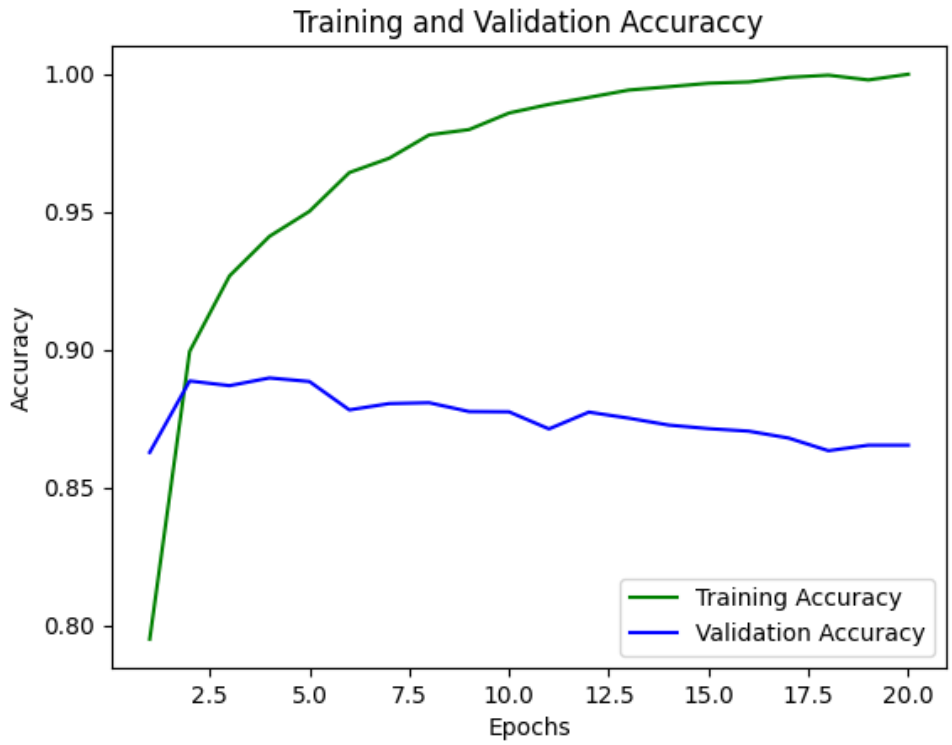
at the end of each epoch.

Let's use Matplotlib to plot Training and validation losses and Traing and Validation Accuracy side by side.

In [16]:
```
# Plotting losses
loss_values = history_dict['loss']
val_loss_values = history_dict['val_loss']

epochs = range(1, len(loss_values) + 1)

plt.plot(epochs, loss_values, 'g', label="Training Loss")
plt.plot(epochs, val_loss_values, 'b', label="Validation Loss")

plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss Value')
plt.legend()

plt.show()
```



In [17]:
```
# Training and Validation Accuracy

acc_values = history_dict['binary_accuracy']
val_acc_values = history_dict['val_binary_accuracy']

epochs = range(1, len(loss_values) + 1)

plt.plot(epochs, acc_values, 'g', label="Training Accuracy")
plt.plot(epochs, val_acc_values, 'b', label="Validation Accuracy")

plt.title('Training and Validation Accuraccy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.show()
```



We observe that *minimum validation loss* and *maximum validation Accuracy* is achived at around 3-5 epochs. After that we observe 2 trends:

- increase in validation loss and decrese in training loss
- decrease in validation accuracy and increase in training accuracy

This implies that the model is getting better at classifying the sentiment of the training data, but making consistently worse predictions when it encounters new, previously unseed data. This is the hallmark of *Overfitting*. After the 5th epoch the model begins to fit too closely to the trainning data.

To address overfitting, we will reduce the number of epochs to somewhere between 3 and 5. These results may vary depending on your machine and due to the very nature of the random assignment of weights that may vary from model to mode.

In our case we will stop training after 3 epochs.

Retraining our model

In [18]:
```
model.fit(
    partial_X_train,
    partial_y_train,
    epochs=3,
    batch_size=512,
    validation_data=(X_val, y_val)
)
```

```
Epoch 1/3
30/30 [==============================] - 2s 50ms/step - loss: 0.0053 - binary_accuracy: 0.9
996 - val_loss: 0.7299 - val_binary_accuracy: 0.8648
Epoch 2/3
30/30 [==============================] - 0s 15ms/step - loss: 0.0040 - binary_accuracy: 0.9
993 - val_loss: 0.7694 - val_binary_accuracy: 0.8659
Epoch 3/3
30/30 [==============================] - 0s 15ms/step - loss: 0.0019 - binary_accuracy: 0.9
999 - val_loss: 0.8005 - val_binary_accuracy: 0.8638
```

Out[18]:
```
<keras.callbacks.History at 0x7f1e00ba11e0>
```

In the end we achive a *training accuracy* of 99% and a *validation accuray* of 86%

Model Evaluation

In [19]:
```
# Making Predictions for testing data
np.set_printoptions(suppress=True)
result = model.predict(X_test)
```

```
782/782 [==============================] - 1s 1ms/step
```

In [20]:
```
result
```

Out[20]:
```
array([[0.00426335],
       [0.9999999 ],
       [0.99748594],
       ...,
       [0.000199  ],
       [0.02019035],
       [0.5236855 ]], dtype=float32)
```

In [21]:
```
y_pred = np.zeros(len(result))
for i, score in enumerate(result):
    y_pred[i] = np.round(score)
```

In [22]:
```
mae = metrics.mean_absolute_error(y_pred, y_test)
mae
```

Out[22]:
```
<tf.Tensor: shape=(), dtype=float32, numpy=0.15012>
```