Analyzing IMDB Data in Keras - Solution

In [29]:

```
# Imports
import numpy as np
import keras
from keras.datasets import imdb
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from keras.preprocessing.text import Tokenizer
import matplotlib.pyplot as plt
%matplotlib inline

np.random.seed(42)
```

1. Loading the data

This dataset comes preloaded with Keras, so one simple command will get us training and testing data. There is a parameter for how many words we want to look at. We've set it at 1000, but feel free to experiment.

In [30]:

```
# Loading the data (it's preloaded in Keras)
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=1000)

print(x_train.shape)
print(x_test.shape)
```

```
(25000,)
(25000,)
```

2. Examining the data

Notice that the data has been already pre-processed, where all the words have numbers, and the reviews come in as a vector with the words that the review contains. For example, if the word 'the' is the first one in our dictionary, and a review contains the word 'the', then there is a 1 in the corresponding vector.

The output comes as a vector of 1's and 0's, where 1 is a positive sentiment for the review, and 0 is negative.

In [31]:

```
print(x_train[0])
print(y_train[0])
```

```
[1, 14, 22, 16, 43, 530, 973, 2, 2, 65, 458, 2, 66, 2, 4, 173, 36, 256, 5, 25,
100, 43, 838, 112, 50, 670, 2, 9, 35, 480, 284, 5, 150, 4, 172, 112, 167, 2,
336, 385, 39, 4, 172, 2, 2, 17, 546, 38, 13, 447, 4, 192, 50, 16, 6, 147, 2, 19,
14, 22, 4, 2, 2, 469, 4, 22, 71, 87, 12, 16, 43, 530, 38, 76, 15, 13, 2, 4, 22,
17, 515, 17, 12, 16, 626, 18, 2, 5, 62, 386, 12, 8, 316, 8, 106, 5, 4, 2, 2, 16,
480, 66, 2, 33, 4, 130, 12, 16, 38, 619, 5, 25, 124, 51, 36, 135, 48, 25, 2, 33,
6, 22, 12, 215, 28, 77, 52, 5, 14, 407, 16, 82, 2, 8, 4, 107, 117, 2, 15, 256,
4, 2, 7, 2, 5, 723, 36, 71, 43, 530, 476, 26, 400, 317, 46, 7, 4, 2, 2, 13, 104,
88, 4, 381, 15, 297, 98, 32, 2, 56, 26, 141, 6, 194, 2, 18, 4, 226, 22, 21, 134,
476, 26, 480, 5, 144, 30, 2, 18, 51, 36, 28, 224, 92, 25, 104, 4, 226, 65, 16,
38, 2, 88, 12, 16, 283, 5, 16, 2, 113, 103, 32, 15, 16, 2, 19, 178, 32]
```

3. One-hot encoding the output

Here, we'll turn the input vectors into (0,1)-vectors. For example, if the pre-processed vector contains the number 14, then in the processed vector, the 14th entry will be 1.

In [32]:

```
# Turning the output into vector mode, each of length 1000
tokenizer = Tokenizer(num_words=1000)
x_train = tokenizer.sequences_to_matrix(x_train, mode='binary')
x_test = tokenizer.sequences_to_matrix(x_test, mode='binary')
print(x_train.shape)
print(x_test.shape)
```

```
(25000, 1000)
(25000, 1000)
```

And we'll one-hot encode the output.

In [33]:

```
# One-hot encoding the output
num_classes = 2
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
print(y_train.shape)
print(y_test.shape)
```

```
(25000, 2)
(25000, 2)
```

4. Building the model architecture

Build a model here using sequential. Feel free to experiment with different layers and sizes! Also, experiment adding dropout to reduce overfitting.

In [34]:

```
# Building the model architecture with one layer of length 100
model = Sequential()
model.add(Dense(512, activation='relu', input_dim=1000))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
model.summary()

# Compiling the model using categorical_crossentropy loss, and rmsprop optimizer.
model.compile(loss='categorical_crossentropy', optimizer='rmsprop', metrics=['accuracy'])
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_3 (Dense)              (None, 512)               512512    
_________________________________________________________________
dropout_2 (Dropout)          (None, 512)               0         
_________________________________________________________________
dense_4 (Dense)              (None, 2)                 1026      
=================================================================
Total params: 513,538.0
Trainable params: 513,538.0
Non-trainable params: 0.0
_________________________________________________________________
```

5. Training the model

Run the model here. Experiment with different batch_size, and number of epochs!

In [35]:

```
# Running and evaluating the model
hist = model.fit(x_train, y_train,
          batch_size=32,
          epochs=10,
          validation_data=(x_test, y_test),
          verbose=2)
```

```
Train on 25000 samples, validate on 25000 samples
Epoch 1/10
9s - loss: 0.3969 - acc: 0.8260 - val_loss: 0.3429 - val_acc: 0.8568
Epoch 2/10
9s - loss: 0.3339 - acc: 0.8670 - val_loss: 0.3413 - val_acc: 0.8632
Epoch 3/10
9s - loss: 0.3219 - acc: 0.8778 - val_loss: 0.3552 - val_acc: 0.8614
Epoch 4/10
9s - loss: 0.3110 - acc: 0.8853 - val_loss: 0.3718 - val_acc: 0.8602
Epoch 5/10
9s - loss: 0.3056 - acc: 0.8920 - val_loss: 0.4086 - val_acc: 0.8542
Epoch 6/10
10s - loss: 0.2951 - acc: 0.8983 - val_loss: 0.3938 - val_acc: 0.8608
Epoch 7/10
9s - loss: 0.2864 - acc: 0.9037 - val_loss: 0.4258 - val_acc: 0.8566
Epoch 8/10
9s - loss: 0.2738 - acc: 0.9100 - val_loss: 0.4733 - val_acc: 0.8509
Epoch 9/10
8s - loss: 0.2622 - acc: 0.9162 - val_loss: 0.4658 - val_acc: 0.8536
Epoch 10/10
12s - loss: 0.2520 - acc: 0.9216 - val_loss: 0.4877 - val_acc: 0.8583
```

6. Evaluating the model

This will give you the accuracy of the model. Can you get something over 85%?

In [36]:

```
score = model.evaluate(x_test, y_test, verbose=0)
print("accuracy: ", score[1])
```

```
accuracy:  0.85828
```