Stoneburner, Kurt

- ## DSC 650 - Assignment 10

Links to Deep Learning Sample Code: Word Embedding Examples:

- https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/first_edition/6.1-using-word-embeddings.ipynb (https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/first_edition/6.1-using-word-embeddings.ipynb)

RNN and LSTM Examples

- https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/first_edition/6.2-understanding-recurrent-neural-networks.ipynb (https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/first_edition/6.2-understanding-recurrent-neural-networks.ipynb)

ngram reference:

- https://www.analyticsvidhya.com/blog/2021/09/what-are-n-grams-and-how-to-implement-them-in-python/ (https://www.analyticsvidhya.com/blog/2021/09/what-are-n-grams-and-how-to-implement-them-in-python/)

pad_sequence: Used to transform lists within lists (2D array) to have a uniform inner dimension. Essentially, padding smaller arrays to the size of the largest, or trimming all arrays to max len

This function transforms a list (of length num_samples) of sequences (lists of integers) into a 2D Numpy array of shape (num_samples, num_timesteps). num_timesteps is either the maxlen argument if provided, or the length of the longest sequence in the list.

- https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/sequence/pad_sequences (https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/sequence/pad_sequences)

Working with tensors was a lost cause for this assignment. Keeping these for reference Convert Numpy Array to Tensor:

- https://www.projectpro.io/recipes/convert-numpy-array-tensor (https://www.projectpro.io/recipes/convert-numpy-array-tensor)

Convert Tensor to Numpy Array:

- https://www.delftstack.com/howto/numpy/python-convert-tensor-to-numpy-array/ (https://www.delftstack.com/howto/numpy/python-convert-tensor-to-numpy-array/)

Basic Text Preparation for modeling: Tokenize, ngram, Convert Text to numeric categorical (integer) value. Use Keras pad_sequences to convert integer lists to uniform lists. Feed into model.

In [1]:

```python
import os
from pathlib import Path
import sys
# //*** Imports and Load Data
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import time

from tensorflow import keras
import tensorflow as tf
import datetime
from tensorflow.keras.optimizers import RMSprop
from tensorflow.keras import preprocessing
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense
from tensorflow.keras.layers import Embedding


#//*** Reusing Code from assignment 04
from chardet.universaldetector import UniversalDetector
from bs4 import BeautifulSoup


import re

#//*** Use the whole window in the IPYNB editor
from IPython.core.display import display, HTML
display(HTML("<style>.container { width:100% !important; }</style>"))

#//*** Maximize columns and rows displayed by pandas
pd.set_option('display.max_rows', 100)
pd.set_option('display.max_columns', None)
```

In [2]:

```python
#//*****************************************************
#//*** Plot a Fitted Models History of Loss and Accuracy
#//*****************************************************
def plot_model_history(input_history):
    loss_key, acc_key = list(input_history.history.keys())[:2]
    val_loss_key, val_acc_key = list(input_history.history.keys())[2:

    acc = input_history.history[acc_key]
    loss = input_history.history[loss_key]

    val_loss = input_history.history[val_loss_key]
    val_acc = input_history.history[val_acc_key]

    epochs = range(1, len(loss) + 1)
    plt.plot(epochs, acc, "b", label="Training Accuracy")
    plt.title("Training Accuracy\nAccuracy should go up")
    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    plt.legend()
    plt.show()

    plt.plot(epochs, loss, "bo", label="Training Loss")

    plt.title("Training Loss \nLoss should go down")
    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    plt.legend()
    plt.show()


    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 Validation Set Accuracy
    plt.clf()

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

```

In [3]:

```
#//*** Get Working Directory
current_dir = Path(os.getcwd()).absolute()

#//*** Go up Two folders
project_dir = current_dir.parents[2]

#//*** IMDB Data Path
imdb_path = project_dir.joinpath("dsc650/data/external/imdb/aclImdb")

file_path = imdb_path.joinpath("train/pos")

#//*** Grab the first positive review text for testing
file_path = file_path.joinpath(os.listdir(file_path)[0])

with open(file_path,'r') as f:
    sample_text = f.read()

print(sample_text)
```

```
Bromwell High is a cartoon comedy. It ran at the same time as some oth
er programs about school life, such as "Teachers". My 35 years in the
teaching profession lead me to believe that Bromwell High's satire is
much closer to reality than is "Teachers". The scramble to survive fin
```

In [4]:

```python
#//*** Randomly assign 20% of the training Data and copy to a validat
import os, pathlib, shutil, random

val_dir = imdb_path.joinpath("val")
train_dir = imdb_path.joinpath("train")
test_dir = imdb_path.joinpath("test")

for category in ("neg", "pos"):
    #//*** Skip if val folder exists (Delete Folder to resample)
    if os.path.exists(val_dir.joinpath(category)):
        break

    os.makedirs(val_dir.joinpath(category))
    files = os.listdir(train_dir.joinpath(category))
    random.Random(1337).shuffle(files)
    num_val_samples = int(0.2 * len(files))
    val_files = files[-num_val_samples:]
    for fname in val_files:
        shutil.move(train_dir / category / fname,
                    val_dir / category / fname)

```

Load IMDB Dataset

In [5]:

```python
#//*** Use Universal Detector to determine file encoding.
#//*** Borrowed from Assignment04
def read_file_with_encoding(filepath):

    detector = UniversalDetector()

    try:
        with open(filepath) as f:
            return f.read()
    except UnicodeDecodeError:
        detector.reset()
        with open(filepath, 'rb') as f:
            for line in f.readlines():
                detector.feed(line)
                if detector.done:
                    break
        detector.close()
        encoding = detector.result['encoding']
        with open(filepath, encoding=encoding) as f:
            return f.read()

#//*** Borrowed from Assignment04
def parse_html_payload(payload):
    """
    This function uses Beautiful Soup to read HTML data
    and return the text.  If the payload is plain text, then
    Beautiful Soup will return the original content
    """
    soup = BeautifulSoup(payload, 'html.parser')
    return str(soup.get_text()).encode('utf-8').decode('utf-8')

def load_dataset(dir_path):

    text = []
    targets = []

    #//*** Crawl the neg and pos folders
    for category in ("neg", "pos"):
        files = os.listdir(dir_path.joinpath(category))

        #//*** Loop through each file in the folder
        for file in files:
            try:
                #//*** Add processed file to text
                text.append(
                    #//*** Strip HTML Tags
                    parse_html_payload(
                        #//*** Read File from disk. Function uses Uni
                        read_file_with_encoding(
                            dir_path.joinpath(category).joinpath(file

                #//*** Append Target Value
                if category == 'neg':
                    targets.append(0)
                else:
                    targets.append(1)
            except:
                print(f"Dropping File: {file} due to decoding issues"

    #//*** Targets returned as Numpy float32 array
    return text, np.asarray( targets).astype('float32')
```

Assignment 10.1

In [27]:

```
#//*** Vectorize a corpus
class Vectorizer:
    def __init__(self,**kwargs):
        self.corpus_tokens = []
        self.corpus_ngrams = []

        self.max_tokens = None
        self.ngram_size = 1
        self.tidyup = True

        self.max_element_count = -1

        for key,value in kwargs.items():
            if key =="max_tokens":
                self.max_tokens = value

            if key == "ngrams":
                self.ngram_size = value

            if key == "tidyup":
                self.tidyup = value


        #//*** One Hot Encoding Dictionaries
        #//*** Key = Token Index, Value = Word
        self.ngram_index = {}

        #//*** Key = Word, Value = Token Index
        self.vocabulary_index = {}

    def tokenize(self,raw_text):
        #//*** Initialize Output Tokens
        tokens = []

        #//*** Split Text into words
        for x in re.split("\s",raw_text):

            #//*** Findall Non text characters in each word
            non_text = re.findall("\W",x)

            #//*** Remove non_text Characters
            for i in non_text:
                x = x.replace(i,"")

            #//*** If X has length, append out
            if len(x) > 0:
                tokens.append(x.lower())
        return tokens

    def build_ngrams(self):
        if self.ngram_size <= 0:
            print("Ngram size must be an integer > 0")
            print("Quitting!")
            return None

        #//*** Using unigrams, use tokens
        if self.ngram_size == 1:
            self.corpus_ngrams = self.corpus_tokens
            return

        self.corpus_ngrams = []

        #//*** Get each token group from corpus_tokens
        for token in self.corpus_tokens:

            loop_ngram = []

            #//*** Use an index based range to loop through tokens
            for x in range(0,len(token) ):

                #//*** Check if index + ngram_size exceeds the lengt
                if x+self.ngram_size <= len(token):

                    result = ""

                    #//*** Build the ngram
                    for y in range(self.ngram_size):
                        #print(self.tokens[x+y])
                        result += token[x+y] + " "

                    loop_ngram.append(result[:-1])

                else:
                    break

            #//*** Grab Token Element Count, Keep the greatest count
            if len(loop_ngram) > self.max_element_count:
                self.max_element_count = len(loop_ngram)


            #//*** Token group ngram is built. Add loop_ngram to corp
            self.corpus_ngrams.append(loop_ngram)



    def build_vocabulary(self,corpus):
        if not isinstance(corpus,list) :
            print("Vectorizer Requires a corpus (list of text):")
            return None

        self.tokens = []

        print("Tokenizing...")
        #//*** Tokenize each text entry in the corpus
        for raw_text in corpus:
            self.corpus_tokens.append(self.tokenize(raw_text))

        print("Building ngrams...")
        #//*** Build ngrams (Defaults to unigrams)
        self.build_ngrams()

        word_freq = {}

        print("Building Vocabulary...")
        #//*** Build dictionary of unique words
        #//*** Loop through each element of the corpus
        for element in self.corpus_ngrams:


            #//*** Grab Token Element Count, Keep the greatest count
            if len(element) > self.max_element_count:
                self.max_element_count = len(element)

            #//*** Process each individual ngram
            for ngram in element:



                #//*** Add unique words to dictionaries
                if ngram not in self.vocabulary_index.keys():
                    index = len(self.ngram_index.values())
                    self.ngram_index[ index ] = ngram
                    self.vocabulary_index [ ngram ] = index

                    #//*** Initialize Word Frequency
                    word_freq[ ngram ] = 1
                else:
                    #//*** Increment Word Frequency
                    word_freq[ ngram ] += 1

        #//*** END for element in self.corpus_ngrams:
        if self.max_tokens != None:

            #//*** Check if token count exceeds max tokens
            if self.max_tokens < len(self.ngram_index.items()):

                print("Sorting Word Frequency...")
                #//*** Sort the Word Frequency Dictionary. Keep the
                word_freq = dict(sorted(word_freq.items(), key=lambda

                print("Building Token Dictionary")
                #//*** Get list of keys that are lowest frequency
                for key in list(word_freq.keys())[self.max_tokens:]:
                    #//*** Delete Low Frequency ngrams
                    del word_freq[ key ]

                self.ngram_index = {}
                self.vocabulary_index = {}

                print("Rebuilding Vocabulary")
                #//*** Rebuild ngram_index & vocabulary_index
                for ngram in word_freq.keys():
                    index = len(self.ngram_index.values())
                    self.ngram_index[ index ] = ngram
                    self.vocabulary_index [ ngram ] = index

            #//*** END Trim Low Frequency ngrams
        self.word_freq = word_freq

    #//*** One Hot encode the corpus.
    #//*** Handling the corpus as a whole increases processing speed
    #//*** Hot encode to a sparse tensor to for increased encoding sp
    def one_hot_encode(self,corpus):

        #//*** Encoded Results
        results = []

        #//*** Set the Max array size to the total number of items i
        array_size = len(self.ngram_index.keys())


        start_time = datetime.datetime.now()
        count = 0



        for element in corpus:
            #//*** hot encode each ngram
            result = []
            for ngram in element:

                #//*** Skip words not in self.vocabulary_index
                #//*** These are skipped due to max_tokens limitatior
                if ngram not in self.vocabulary_index.keys():
                    continue

                sparse_tensor = tf.SparseTensor(indices=[[0,self.voca
                #index = self.vocabulary_index[ngram]

                #base_array = np.zeros(array_size, dtype=int)

                #base_array [index] = 1


                #//*** Add the one-hot-encoded word to encoded text
                result.append(sparse_tensor)


            #//*** END for ngram in tokens:

            result = tf.sparse.concat(axis=1, sp_inputs=result)
            #//*** concat Sparse Matrix
            results.append( result )

            count += 1



            #//*** Print a status update every 1000 items
            if count % 100 == 0:
                print(f"{count} / {len(corpus)} Encoded: {datetime.da

        #//*** Concat List of Sparse Matrixes into a sparse matrix
        #results =  tf.sparse.concat(axis=1, sp_inputs=results)

        print(f"Encoding Complete: {datetime.datetime.now() - start_t

        return results

    #//***
    def integer_encode(self,corpus):
        #//*** Encoded Results
        results = []

        #//*** Set the Max array size to the total number of items in
        array_size = len(self.ngram_index.keys())


        start_time = datetime.datetime.now()
        count = 0



        for element in corpus:
            #//*** hot encode each ngram
            result = []
            for ngram in element:

                #//*** Skip words not in self.vocabulary_index
                #//*** These are skipped due to max_tokens limitatio
                if ngram not in self.vocabulary_index.keys():
                    continue

                #//*** Get integer value of ngram from dictionary.
                #//*** Add to result
                result.append(self.vocabulary_index[ngram])


            #//*** END for ngram in tokens:

            #//*** result is a complete encoded element
            results.append( np.array(result).astype(np.float32) )

            count += 1



            #//*** Print a status update every 1000 items
            if count % 5000 == 0:
                print(f"{count} / {len(corpus)} Encoded: {datetime.da

        print(f"Encoding Complete: {datetime.datetime.now() - start_t

        #//*** results is a collection of encoded elements
        return np.array(results,dtype=object)


    def encode(self,corpus,encoding='int'):

        if not isinstance(corpus,list) :
            print("Vectorizer Requires a corpus (list of text):")
            return None

        self.corpus_tokens = []
        self.corpus_ngrams = []
        print("Tokenizing...")
        #//*** Tokenize each text entry in the corpus
        for raw_text in corpus:
            self.corpus_tokens.append(self.tokenize(raw_text))

        print("Building ngrams...")
        #//*** Build ngrams (Defaults to unigrams)
        self.build_ngrams()

        if encoding == 'onehot':
            print("One Hot Coding....")

            #//*** One Hot Encode Values. These are actually sparse t
            encoded = self.one_hot_encode(self.corpus_ngrams)

        if encoding == 'int':
            print("Interger encoding....")

            #//*** Convert ngrams to integers. These are actually spa
            encoded = self.integer_encode(self.corpus_ngrams)

            #//*** Convert lists to Numpy array of float 32 type. Thi
            #encoded = np.asarray(encoded).astype('float32')

        #//*** TidyUp (Delete) ngrams and Tokens
        if self.tidyup:
            self.corpus_tokens = []
            self.corpus_ngrams = []


        return encoded

    #//*** Convert One-Hot-Encoding to text
    def decode(self,elements):

        results = []

        #//*** For Each element in Corpus

        decoded = ""

        #//*** For Each ngram (word(s)) in Elements
        for ngram in elements:

            #//*** Grab Index of 1 from sparse tensor
            index = ngram.indices[0].numpy()[1]

            #ngram = list(ngram.numpy())

            decoded += self.ngram_index[ index ] + " "

        #//*** END for ngram in elements:
        results.append( decoded[:-1])

        #//*** END for elements in corpus:
        return results

print("Loading Raw Validation Set")
val_x_train, val_y_train = load_dataset(val_dir)

#//*** Test the Vectorizer with some sample data
vectorizer = Vectorizer(max_tokens=100,ngrams=2, tidyup=False)
vectorizer.build_vocabulary(val_x_train[:5])
start_time = datetime.datetime.now()

temp_vals = vectorizer.encode(val_x_train[:5],encoding='onehot')
```

```

print(f"Run Time: {datetime.datetime.now() - start_time}")


int_vals = vectorizer.encode(val_x_train[:5],encoding='int')
print("Integer Encoding:")
print(int_vals)
print()
print()

print("Sample Text: (First 500 Chars)")
for element in val_x_train[:5]:
    print(element[:500])
    print("====")
print()
print()

print("Tokens: (First 100 tokens)")
for token in vectorizer.corpus_tokens:
    print(token[:100])
    print("====")
print()
print()

print("ngrams: (First 50 tokens)")
for token in vectorizer.corpus_ngrams:
    print(token[:100])
    print("====")
print()
print()
print("Small one hot encoded Sample:")
print(temp_vals)
print()
print()
print("Encoded Vocabulary")
print(vectorizer.vocabulary_index)
print()
print()
print("Decoded Text from vocabulary (limited by max tokens)")
for result in vectorizer.decode(temp_vals):
    print(result)
    print()

del temp_vals
del int_vals
del vectorizer

```

```
Loading Raw Validation Set
Tokenizing...
Building ngrams...
Building Vocabulary...
Sorting Word Frequency...
Building Token Dictionary
Rebuilding Vocabulary
Tokenizing...
Building ngrams...
One Hot Coding....
Encoding Complete: 0:00:00.046011
Run Time: 0:00:00.051012
Tokenizing...
```

```
Building ngrams...
Interger encoding....
Encoding Complete: 0:00:00
Integer Encoding:
[array([ 2., 76., 77.,  5., 78., 79., 80., 81., 82., 21., 83., 84., 8
5.,
```

In [7]:

```python
print("Loading Raw Validation Set")
raw_val_x_train, val_y_train = load_dataset(val_dir)

print("Loading Raw Train Data")
raw_x_train, y_train = load_dataset(train_dir)

print("Loading Raw Test Data")
raw_x_test, y_test = load_dataset(test_dir)
print("Done")

val_y_train = np.array(val_y_train,dtype=object)
y_train = np.array(y_train,dtype=object)
y_test = np.array(y_test,dtype=object)

y_train = np.asarray(y_train).astype(np.int)
```

```
Loading Raw Validation Set
Loading Raw Train Data
Dropping File: 7714_1.txt due to decoding issues
Dropping File: 11351_9.txt due to decoding issues
Dropping File: 8263_9.txt due to decoding issues
Loading Raw Test Data
Dropping File: 4414_1.txt due to decoding issues
Dropping File: 6973_1.txt due to decoding issues
Dropping File: 2464_10.txt due to decoding issues
Dropping File: 5281_10.txt due to decoding issues
Done
```

In [8]:

```python



#//*** Test the Vectorizer with some sample data
max_tokens = 20000
#maxlen = 1000 #//*** Limit reviews to this length, Leave blank to us

ngrams = 1

#//*** Initialize vectorizer
vectorizer = Vectorizer(max_tokens=max_tokens,ngrams=ngrams)

#//*** Build Vocabulary based on the training text
vectorizer.build_vocabulary(raw_x_train)

#//*** maxlen required: This is maximum number of tokens/ngrams to us
#//*** pads all articles to the same word count. This required to hav

maxlen = vectorizer.max_element_count #//*** Sets maxlen to the large


```

```
#//*** Encode Validation, training and test data

print("Encoding Validation Data...")
val_x_train = vectorizer.encode(raw_val_x_train)
print(vectorizer.max_element_count)
print("Encoding Training Data...")
x_train = vectorizer.encode(raw_x_train)

print("Encoding Test Data...")
x_test = vectorizer.encode(raw_x_test)


print("Padding Validation...")
val_x_train = preprocessing.sequence.pad_sequences(val_x_train, maxle

print("Padding Training Data...")
x_train = preprocessing.sequence.pad_sequences(x_train, maxlen=maxlen

print("Padding Test Data...")
x_test = preprocessing.sequence.pad_sequences(x_test, maxlen=maxlen)

val_y_train = np.asarray(val_y_train).astype(np.int)
x_test = np.asarray(x_test).astype(np.int)
y_test = np.asarray(y_test).astype(np.int)

print("Done")

print(type(val_x_train),type(val_x_train[0]),x_train.shape)
print(type(val_y_train),type(val_y_train[0]),y_train.shape)
print(type(x_test),type(x_test[0]))
print(type(y_test),type(y_test[0]))
print(x_test.shape,y_test.shape)
```

```
Tokenizing...
Building ngrams...
Building Vocabulary...
Sorting Word Frequency...
Building Token Dictionary
Rebuilding Vocabulary
Encoding Validation Data...
Tokenizing...
Building ngrams...
Interger encoding....
5000 / 5000 Encoded: 0:00:00.496111
Encoding Complete: 0:00:00.496111
2450
Encoding Training Data...
Tokenizing...
Building ngrams...
Interger encoding....
5000 / 19997 Encoded: 0:00:00.484107
10000 / 19997 Encoded: 0:00:00.977218
15000 / 19997 Encoded: 0:00:01.466329
Encoding Complete: 0:00:01.963440
Encoding Test Data...
Tokenizing...
```

```
Building ngrams...
Interger encoding....
5000 / 24996 Encoded: 0:00:00.477108
10000 / 24996 Encoded: 0:00:00.949213
15000 / 24996 Encoded: 0:00:01.431323
20000 / 24996 Encoded: 0:00:01.913430
Encoding Complete: 0:00:02.384537
Padding Validation...
Padding Training Data...
Padding Test Data...
Done
<class 'numpy.ndarray'> <class 'numpy.ndarray'> (19997, 2450)
<class 'numpy.ndarray'> <class 'numpy.int32'> (19997,)
<class 'numpy.ndarray'> <class 'numpy.ndarray'>
<class 'numpy.ndarray'> <class 'numpy.int32'>
(24996, 2450) (24996,)
```

In [9]:

```python
print(type(val_x_train),type(val_x_train[0]),x_train.shape)
print(type(val_y_train),type(val_y_train[0]),y_train.shape)
print(type(x_test),type(x_test[0]))
print(type(y_test),type(y_test[0]))
print(x_test.shape,y_test.shape)
```

```
<class 'numpy.ndarray'> <class 'numpy.ndarray'> (19997, 2450)
<class 'numpy.ndarray'> <class 'numpy.int32'> (19997,)
<class 'numpy.ndarray'> <class 'numpy.ndarray'>
<class 'numpy.ndarray'> <class 'numpy.int32'>
(24996, 2450) (24996,)
```

https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/first_edition/6.1-using-word-embeddings.ipynb (https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/first_edition/6.1-using-word-embeddings.ipynb)

In [10]:

```
"""

from keras.datasets import imdb
from keras import preprocessing

# Number of words to consider as features
max_features = 10000
# Cut texts after this number of words
# (among top max_features most common words)
maxlen = 20

# Load the data as lists of integers.
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=max_f

# This turns our lists of integers
# into a 2D integer tensor of shape `(samples, maxlen)`
#x_train = preprocessing.sequence.pad_sequences(x_train, maxlen=maxle
#x_test = preprocessing.sequence.pad_sequences(x_test, maxlen=maxlen)
print(type(x_train),type(x_train[0]))
print(type(y_train),type(y_train[0]))
"""
```

pad_sequence: Used to transform lists within lists (2D array) to have a uniform inner dimension. Essentially, padding smaller arrays to the size of the largest, or trimming all arrays to max len

This function transforms a list (of length num_samples) of sequences (lists of integers) into a 2D Numpy array of shape (num_samples, num_timesteps). num_timesteps is either the maxlen argument if provided, or the length of the longest sequence in the list.

- https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/sequence/pad_sequences (https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/sequence/pad_sequences)

10.2

Using listings 6.16, 6.17, and 6.18 in Deep Learning with Python as a guide, train a sequential model with embeddings on the IMDB data found in data/external/imdb/. Produce the model performance metrics and training and validation accuracy curves within the Jupyter notebook.

In [11]:

```


def get_sequential_embedded_model():
    model = Sequential()
    # We specify the maximum input length to our Embedding layer
    # so we can later flatten the embedded inputs
    model.add(Embedding(max_tokens, 8, input_length=maxlen))
    # After the Embedding layer,
    # our activations have shape `(samples, maxlen, 8)`.

    # We flatten the 3D tensor of embeddings
    # into a 2D tensor of shape `(samples, maxlen * 8)`
    model.add(Flatten())

    # We add the classifier on top
    model.add(Dense(1, activation='sigmoid'))
    model.compile(optimizer='rmsprop', loss='binary_crossentropy', me

    return model

model = get_sequential_embedded_model()
model.summary()

#//*** Code to check the data type expected for each model layer
[print(i.shape, i.dtype) for i in model.inputs]
[print(o.shape, o.dtype) for o in model.outputs]
[print(l.name, l.input_shape, l.dtype) for l in model.layers]

#//*** Reference for recasting lists as an np.array of float32 type
#x = np.asarray(x).astype('float32')

```

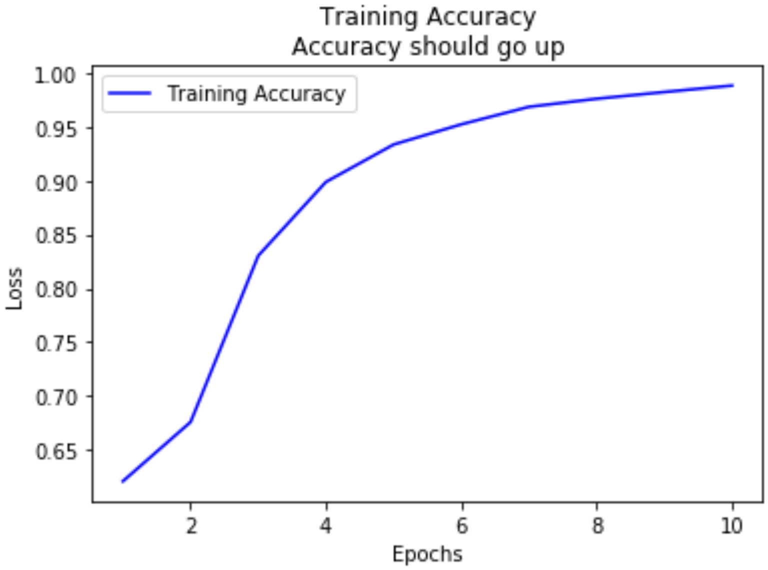
```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
embedding (Embedding)        (None, 2450, 8)           160000    
_________________________________________________________________
flatten (Flatten)            (None, 19600)             0         
_________________________________________________________________
dense (Dense)                (None, 1)                 19601     
=================================================================
Total params: 179,601
Trainable params: 179,601
Non-trainable params: 0
_________________________________________________________________
(None, 2450) <dtype: 'float32'>
(None, 1) <dtype: 'float32'>
embedding (None, 2450) float32
flatten (None, 2450, 8) float32
dense (None, 19600) float32
```

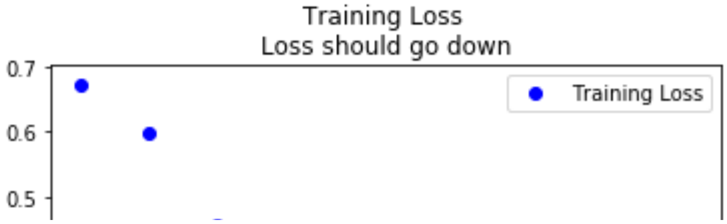
Out[11]: [None, None, None]

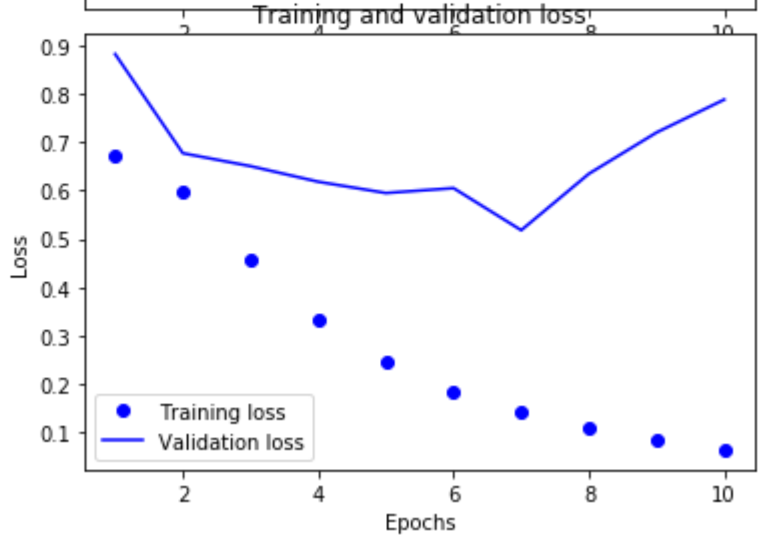
In [12]:

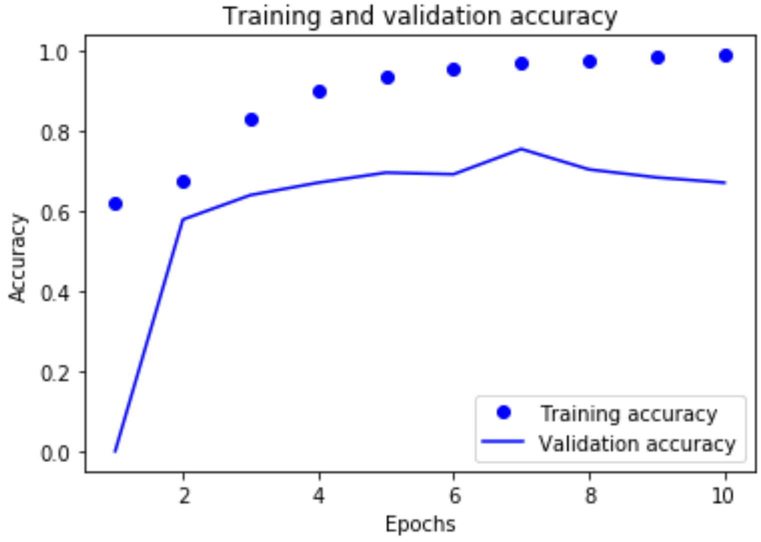
```
history = model.fit(val_x_train,val_y_train,
                    epochs=10,
                    batch_size=32,
                    validation_split=0.2
                   )

score = model.evaluate(np.asarray(x_test).astype(np.int), np.asarray(

print('Test loss:', score[0])
print('Test accuracy:', score[1])

plot_model_history(history)
```

```
Epoch 1/10
125/125 [==============================] - 2s 9ms/step - loss: 0.6708
- acc: 0.6208 - val_loss: 0.8819 - val_acc: 0.0000e+00
Epoch 2/10
125/125 [==============================] - 1s 7ms/step - loss: 0.5973
- acc: 0.6758 - val_loss: 0.6769 - val_acc: 0.5780
Epoch 3/10
125/125 [==============================] - 1s 8ms/step - loss: 0.4582
- acc: 0.8303 - val_loss: 0.6502 - val_acc: 0.6390
Epoch 4/10
125/125 [==============================] - 1s 7ms/step - loss: 0.3333
- acc: 0.8988 - val_loss: 0.6180 - val_acc: 0.6700
Epoch 5/10
125/125 [==============================] - 1s 7ms/step - loss: 0.2458
- acc: 0.9335 - val_loss: 0.5947 - val_acc: 0.6950
Epoch 6/10
125/125 [==============================] - 1s 7ms/step - loss: 0.1850
- acc: 0.9523 - val_loss: 0.6048 - val_acc: 0.6910
Epoch 7/10
125/125 [==============================] - 1s 7ms/step - loss: 0.1410
- acc: 0.9685 - val_loss: 0.5178 - val_acc: 0.7540
Epoch 8/10
125/125 [==============================] - 1s 7ms/step - loss: 0.1088
- acc: 0.9760 - val_loss: 0.6345 - val_acc: 0.7030
Epoch 9/10
125/125 [==============================] - 1s 7ms/step - loss: 0.0836
- acc: 0.9822 - val_loss: 0.7198 - val_acc: 0.6830
Epoch 10/10
125/125 [==============================] - 1s 7ms/step - loss: 0.0642
- acc: 0.9883 - val_loss: 0.7878 - val_acc: 0.6700
Test loss: 0.4089915454387665
Test accuracy: 0.8279724717140198
```









In [13]:

```
history = model.fit(x_train,y_train,
                    epochs=8,
                    batch_size=32,
                    validation_split=0.2
                   )

score = model.evaluate(np.asarray(x_test).astype(np.int), np.asarray(

print('Test loss:', score[0])
print('Test accuracy:', score[1])

```

```
Epoch 1/8
500/500 [==============================] - 4s 8ms/step - loss: 0.2878
- acc: 0.8827 - val_loss: 0.4963 - val_acc: 0.7975
Epoch 2/8
500/500 [==============================] - 4s 8ms/step - loss: 0.2160
- acc: 0.9155 - val_loss: 0.5547 - val_acc: 0.7803
Epoch 3/8
500/500 [==============================] - 4s 8ms/step - loss: 0.1730
- acc: 0.9362 - val_loss: 0.4601 - val_acc: 0.8198
Epoch 4/8
500/500 [==============================] - 4s 8ms/step - loss: 0.1423
- acc: 0.9492 - val_loss: 0.4608 - val_acc: 0.8290
Epoch 5/8
500/500 [==============================] - 4s 8ms/step - loss: 0.1186
- acc: 0.9595 - val_loss: 0.6408 - val_acc: 0.7710
Epoch 6/8
500/500 [==============================] - 4s 8ms/step - loss: 0.0997
```

In [ ]:

10.3

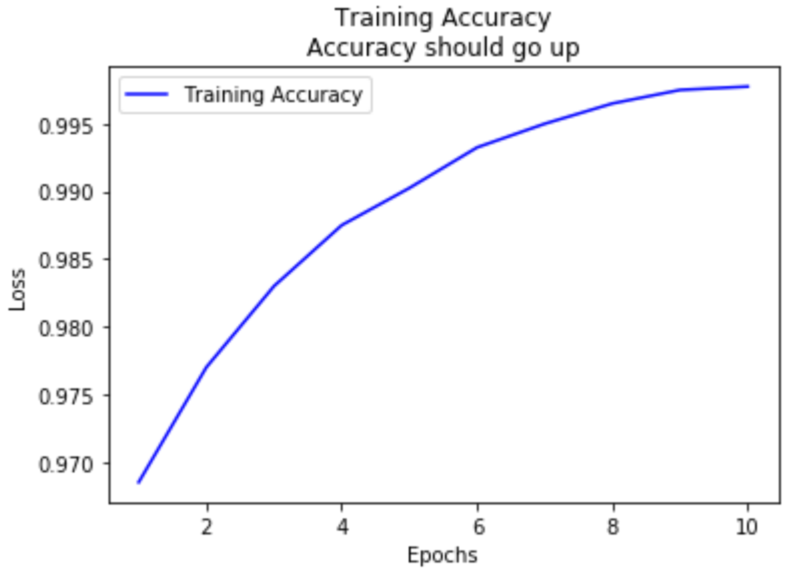
Using listing 6.27 in Deep Learning with Python as a guide, fit the same data with an LSTM layer. Produce the model performance metrics and training and validation accuracy curves within the Jupyter notebook.

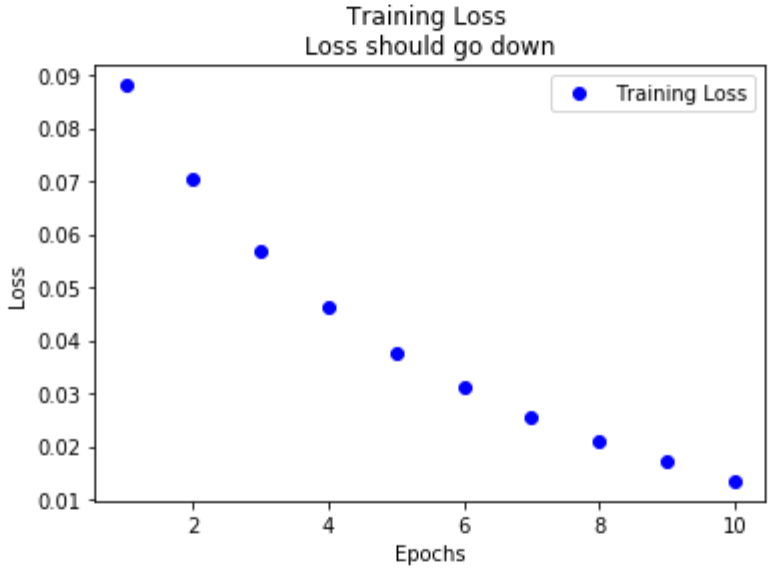
In [14]:

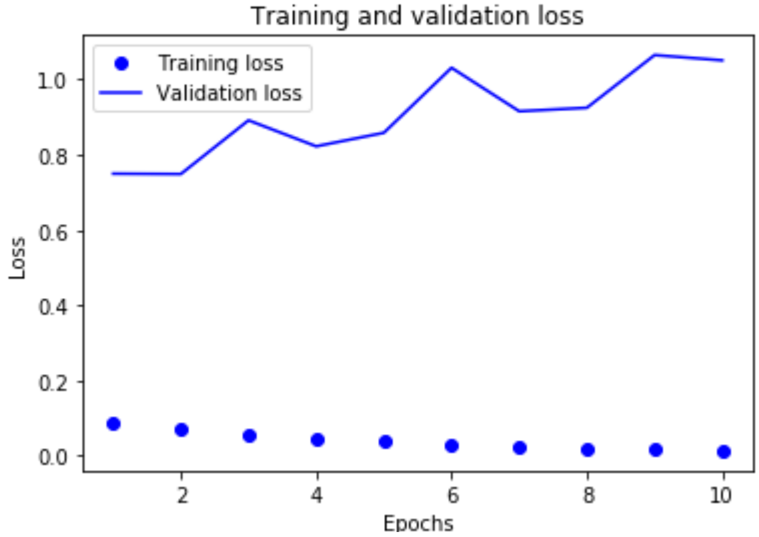
```python


from keras.models import Sequential
from keras.layers import Embedding, SimpleRNN,LSTM

def get_lstm_model():

    model = Sequential()
    model.add(Embedding(max_tokens, 32))
    model.add(LSTM(32))
    model.add(Dense(1, activation='sigmoid'))

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


history = model.fit(val_x_train, val_y_train,
                    epochs=10,
                    batch_size=128,
                    validation_split=0.2)

score = model.evaluate(np.asarray(x_test).astype(np.int), np.asarray(

print('Test loss:', score[0])
print('Test accuracy:', score[1])

plot_model_history(history)
```

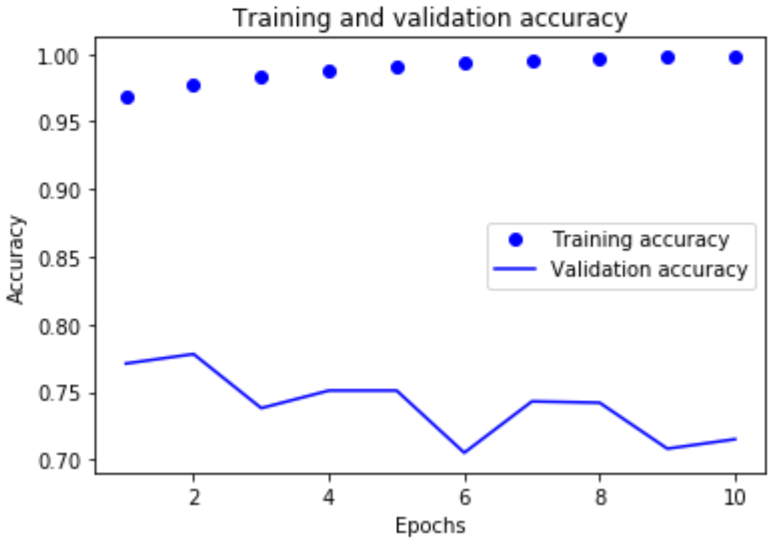
```
Epoch 1/10
32/32 [==============================] - 1s 27ms/step - loss: 0.0881 -
acc: 0.9685 - val_loss: 0.7499 - val_acc: 0.7710
Epoch 2/10
32/32 [==============================] - 1s 26ms/step - loss: 0.0703 -
acc: 0.9770 - val_loss: 0.7489 - val_acc: 0.7780
Epoch 3/10
32/32 [==============================] - 1s 26ms/step - loss: 0.0569 -
acc: 0.9830 - val_loss: 0.8917 - val_acc: 0.7380
Epoch 4/10
32/32 [==============================] - 1s 26ms/step - loss: 0.0464 -
acc: 0.9875 - val_loss: 0.8226 - val_acc: 0.7510
Epoch 5/10
32/32 [==============================] - 1s 26ms/step - loss: 0.0377 -
acc: 0.9902 - val_loss: 0.8585 - val_acc: 0.7510
Epoch 6/10
32/32 [==============================] - 1s 26ms/step - loss: 0.0313 -
acc: 0.9933 - val_loss: 1.0313 - val_acc: 0.7050
Epoch 7/10
32/32 [==============================] - 1s 27ms/step - loss: 0.0255 -
acc: 0.9950 - val_loss: 0.9157 - val_acc: 0.7430
Epoch 8/10
32/32 [==============================] - 1s 26ms/step - loss: 0.0208 -
acc: 0.9965 - val_loss: 0.9251 - val_acc: 0.7420
Epoch 9/10
32/32 [==============================] - 1s 26ms/step - loss: 0.0171 -
acc: 0.9975 - val_loss: 1.0651 - val_acc: 0.7080
```

```
Epoch 10/10
32/32 [==============================] - 1s 26ms/step - loss: 0.0135 -
acc: 0.9977 - val_loss: 1.0511 - val_acc: 0.7150
Test loss: 0.4457803964614868
Test accuracy: 0.8531364798545837
```


Training Accuracy
Accuracy should go up


Training Loss
Loss should go down


Training and validation loss



In [15]:

```
history = model.fit(x_train, y_train,
                    epochs=8,
                    batch_size=128,
                    validation_split=0.2)

score = model.evaluate(np.asarray(x_test).astype(np.int), np.asarray(

print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
Epoch 1/8
125/125 [==============================] - 3s 27ms/step - loss: 0.0648
- acc: 0.9773 - val_loss: 0.8485 - val_acc: 0.7442
Epoch 2/8
125/125 [==============================] - 3s 26ms/step - loss: 0.0523
- acc: 0.9834 - val_loss: 0.7618 - val_acc: 0.7663
Epoch 3/8
125/125 [==============================] - 3s 26ms/step - loss: 0.0432
- acc: 0.9874 - val_loss: 0.8154 - val_acc: 0.7508
Epoch 4/8
125/125 [==============================] - 3s 26ms/step - loss: 0.0359
- acc: 0.9905 - val_loss: 0.7709 - val_acc: 0.7663
Epoch 5/8
125/125 [==============================] - 3s 26ms/step - loss: 0.0297
- acc: 0.9924 - val_loss: 0.8048 - val_acc: 0.7607
Epoch 6/8
125/125 [==============================] - 3s 26ms/step - loss: 0.0247
- acc: 0.9942 - val_loss: 0.9268 - val_acc: 0.7372
Epoch 7/8
125/125 [==============================] - 3s 26ms/step - loss: 0.0203
- acc: 0.9955 - val_loss: 0.9174 - val_acc: 0.7420
Epoch 8/8
125/125 [==============================] - 3s 26ms/step - loss: 0.0167
- acc: 0.9966 - val_loss: 0.8256 - val_acc: 0.7688
Test loss: 0.46850863099098206
Test accuracy: 0.8548167943954468
```

10.4

Using listing 6.46 in Deep Learning with Python as a guide, fit the same data with a simple 1D convnet. Produce the model performance metrics and training and validation accuracy curves within the Jupyter notebook.

In [24]:

```
from keras import layers

def get_conv1d_model():
    model = Sequential()
    model.add(layers.Embedding(max_tokens, 128, input_length=maxlen))
    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=RMSprop(learning_rate=1e-4),
                  loss='binary_crossentropy',
                  metrics=['acc'])

    model.summary()
    return model


model = get_conv1d_model()
history = model.fit(val_x_train, val_y_train,
                    epochs=30,
                    batch_size=128,
                    validation_split=0.2)

score = model.evaluate(np.asarray(x_test).astype(np.int), np.asarray(

print('Test loss:', score[0])
print('Test accuracy:', score[1])

plot_model_history(history)
```

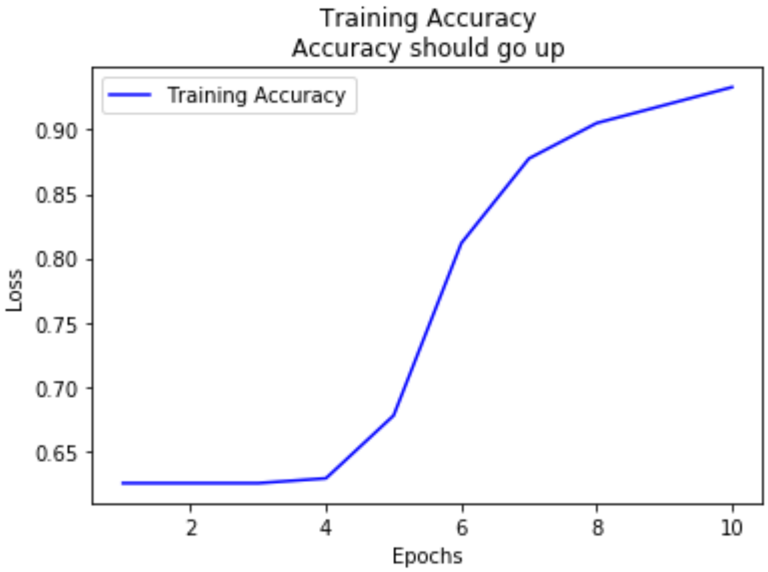
```
Model: "sequential_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
```

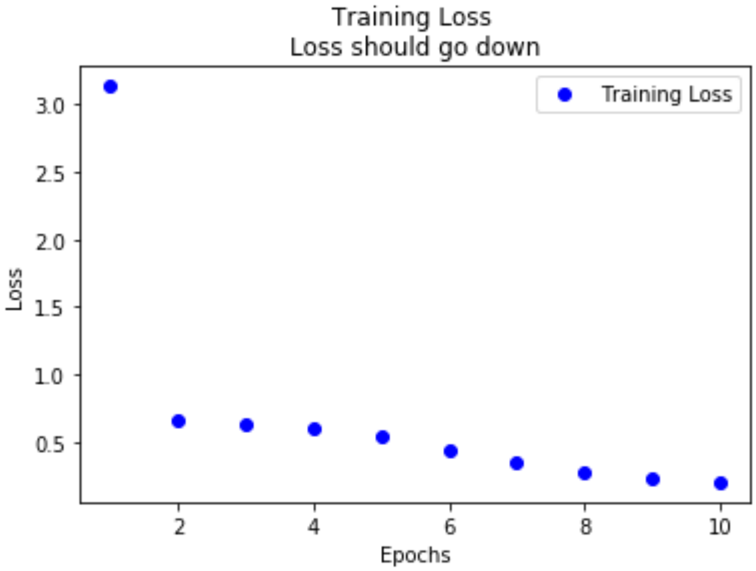
In [25]:

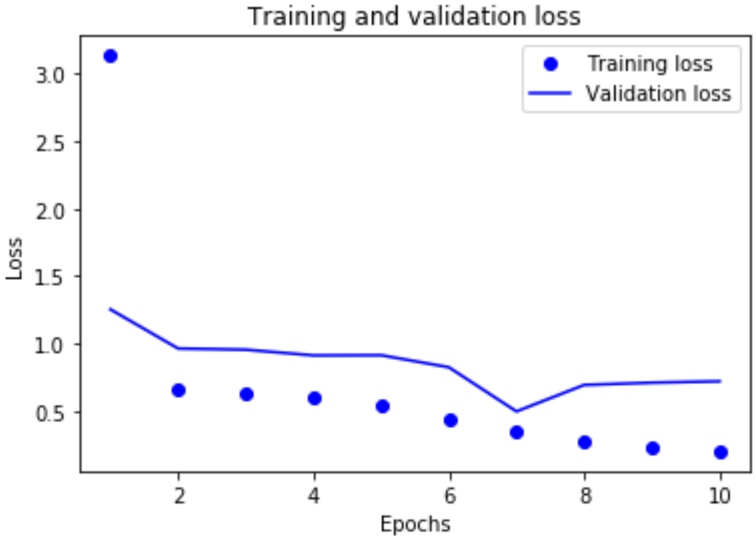
```
history = model.fit(x_train, y_train,
                    epochs=10,
                    batch_size=128,
                    validation_split=0.2)

score = model.evaluate(np.asarray(x_test).astype(np.int), np.asarray(

print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

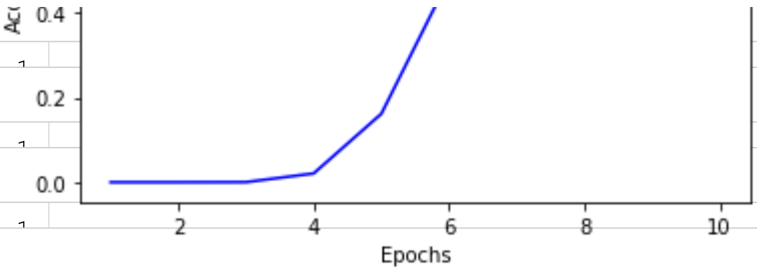
```
Epoch 1/10
125/125 [==============================] - 99s 789ms/step - loss: 3.13
18 - acc: 0.6251 - val_loss: 1.2540 - val_acc: 0.0000e+00
Epoch 2/10
125/125 [==============================] - 97s 775ms/step - loss: 0.65
54 - acc: 0.6251 - val_loss: 0.9645 - val_acc: 0.0000e+00
Epoch 3/10
125/125 [==============================] - 99s 789ms/step - loss: 0.63
06 - acc: 0.6251 - val_loss: 0.9557 - val_acc: 2.5000e-04
Epoch 4/10
125/125 [==============================] - 101s 809ms/step - loss: 0.5
968 - acc: 0.6288 - val_loss: 0.9135 - val_acc: 0.0207
Epoch 5/10
125/125 [==============================] - 98s 786ms/step - loss: 0.54
06 - acc: 0.6777 - val_loss: 0.9148 - val_acc: 0.1615
Epoch 6/10
125/125 [==============================] - 99s 794ms/step - loss: 0.44
34 - acc: 0.8117 - val_loss: 0.8259 - val_acc: 0.4803
Epoch 7/10
125/125 [==============================] - 98s 781ms/step - loss: 0.34
56 - acc: 0.8776 - val_loss: 0.4988 - val_acc: 0.7897
Epoch 8/10
125/125 [==============================] - 97s 779ms/step - loss: 0.28
25 - acc: 0.9052 - val_loss: 0.6948 - val_acc: 0.7120
Epoch 9/10
125/125 [==============================] - 97s 779ms/step - loss: 0.23
84 - acc: 0.9191 - val_loss: 0.7116 - val_acc: 0.7322
Epoch 10/10
125/125 [==============================] - 100s 798ms/step - loss: 0.2
044 - acc: 0.9331 - val_loss: 0.7217 - val_acc: 0.7487
Test loss: 0.4732949435710907
Test accuracy: 0.8379340767860413
```

```
In [26]: plot_model_history(history)
```









In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

```
# //*** CODE HERE
```