In [202]:
```
## Imports for the dataset and building the neural networks.
import nltk
from nltk.corpus import reuters
import keras
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import sequence
from keras.preprocessing.image import ImageDataGenerator
from keras.regularizers import l2
from keras import backend as K
from keras.models import Sequential, load_model
from keras.layers.core import Dense, Dropout, Activation
from keras.utils import np_utils
from keras.utils import to_categorical
from keras.utils import plot_model
from keras import models
from keras import optimizers
from keras import layers
from keras.datasets import reuters
from keras.models import Sequential
from keras.layers import SimpleRNN
from keras.layers import LSTM
from keras.layers import MaxPooling2D
from keras.layers import Dense
from keras.layers import Embedding
from keras.layers import Dropout
from keras.layers import Dense, Conv2D, BatchNormalization, Activat
ion
from keras.layers import AveragePooling2D, Input, Flatten
from keras.preprocessing import image
from keras.preprocessing.image import ImageDataGenerator
from keras.callbacks import ModelCheckpoint
from keras.callbacks import EarlyStopping
from keras.callbacks import ReduceLROnPlateau
from keras.callbacks import LearningRateScheduler
from keras.optimizers import Adam
from keras import regularizers
from keras.applications import VGG16
import pickle
import tensorflow as tf
from sklearn import metrics
from sklearn.model_selection import train_test_split
import numpy as np
import pandas as pd
import matplotlib
import matplotlib.pyplot as plt
import os
from urllib.request import urlretrieve
from os.path import isfile, isdir
from tqdm import tqdm
import tarfile
import json
from time import time
import chakin
```

Hyperparameters

- `max_features` and `maxlen` : Cuts off texts after this many words amongst most common words
- `max_words` : Number of words to consider as features
- `epochs` : number of iterations until the network stops learning or start overfitting
- `batch_size` : highest number that your machine has memory for. Most people set them to common sizes of memory:
- `learning_rate` : number how fast the model learns

In [203]:
```
maxlen=1000 # Cuts off texts after this many words amongst most com
mon words
max_features=1000
max_words=10000 # Number of words to consider as features
#max_features=10000
#max_words=10000
#maxlen=10000
batch_size=128
epochs = 10
learning_rate = 0.001
```

In [204]:

```
##https://towardsdatascience.com/text-classification-in-keras-part-1-a-simple-reuters-news-classifier-9558d34d01d3
# save np.load
np_load_old = np.load

# modify the default parameters of np.load
np.load = lambda *a,**k: np_load_old(*a, allow_pickle=True, **k)

#Was working: (trainingDataRaw, trainingLabelsRaw), (testingDataRaw, testingLabelsRaw) = reuters.load_data(num_words=None, test_split=0.2)
(trainingDataRaw, trainingLabelsRaw), (testingDataRaw, testingLabelsRaw) = reuters.load_data(num_words=max_features, test_split=0.2)
# restore np.load for future normal usage
np.load = np_load_old
word_index = reuters.get_word_index(path="reuters_word_index.json")

print('# of Training Samples: {}'.format(len(trainingDataRaw)))
print('# of Test Samples: {}'.format(len(testingDataRaw)))

num_classes = max(trainingLabelsRaw) + 1
print('# of Classes: {}'.format(num_classes))
# of Training Samples: 8982
# of Test Samples: 2246
# of Classes: 46
index_to_word = {}
for key, value in word_index.items():
    index_to_word[value] = key
print(' '.join([index_to_word[x] for x in trainingDataRaw[0]]))
print(trainingLabelsRaw[0])
```

```
# of Training Samples: 8982
# of Test Samples: 2246
# of Classes: 46
the of of mln loss for plc said at only ended said of could 1 traders now april 0 a after said from 1985 and from foreign 000 april 0 prices its account year a but in this mln home an states earlier and rise and revs vs 000 its 16 vs 000 a but 3 of of several and shareholders and dividend vs 000 its all 4 vs 000 1 mln agreed of april 0 are 2 states will billion total and against 000 pct dlrs
3
```

In [205]:
```
from keras.preprocessing.text import Tokenizer

tokenizer = Tokenizer(num_words=max_features)
trainingData = tokenizer.sequences_to_matrix(trainingDataRaw, mode=
'binary')
testingData = tokenizer.sequences_to_matrix(testingDataRaw, mode='b
inary')

trainingLabels = keras.utils.to_categorical(trainingLabelsRaw, num_
classes)
testingLabels = keras.utils.to_categorical(testingLabelsRaw, num_cl
asses)
print(trainingData[0])
print(len(trainingData[0]))

print(trainingLabels[0])
print(len(trainingLabels[0]))
```

```
[0. 1. 1. 0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 1. 1. 1. 0. 1. 0.
0. 1. 0.
 0. 1. 1. 0. 0. 1. 1. 0. 1. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 1. 1.
0. 0. 0.
 1. 1. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0.
0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 0. 0. 0. 0. 1. 1. 0. 0.
0. 0. 0.
 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 1. 0. 1. 0. 0. 0. 0. 0.
0. 0. 0.
 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 1. 0. 1. 0. 0. 0. 0. 0. 0.
0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0.
0. 0. 0.
 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 1. 0. 0. 0.
0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0.
0. 0. 0.
 0. 0. 0. 0. 0. 0. 1. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0.
0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0.
 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0.
 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
```

In [206]:

```
CHAKIN_INDEX = 11
NUMBER_OF_DIMENSIONS = 50
SUBFOLDER_NAME = "glove.6B"

DATA_FOLDER = "embeddings"
ZIP_FILE = os.path.join(DATA_FOLDER, "{}.zip".format(SUBFOLDER_NAM
E))
ZIP_FILE_ALT = "glove" + ZIP_FILE[5:]  # sometimes it's lowercase o
nly...
UNZIP_FOLDER = os.path.join(DATA_FOLDER, SUBFOLDER_NAME)
if SUBFOLDER_NAME[-1] == "d":
    GLOVE_FILENAME = os.path.join(
        UNZIP_FOLDER, "{}.txt".format(SUBFOLDER_NAME))
else:
    GLOVE_FILENAME = os.path.join(UNZIP_FOLDER, "{}.{}d.txt".format
(
        SUBFOLDER_NAME, NUMBER_OF_DIMENSIONS))


if not os.path.exists(ZIP_FILE) and not os.path.exists(UNZIP_FOLDE
R):
    # GloVe by Stanford is licensed Apache 2.0:
    #     https://github.com/stanfordnlp/GloVe/blob/master/LICENSE
    #     http://nlp.stanford.edu/data/glove.twitter.27B.zip
    #     Copyright 2014 The Board of Trustees of The Leland Stanfo
rd Junior University
    print("Downloading embeddings to '{}'".format(ZIP_FILE))
    chakin.download(number=CHAKIN_INDEX, save_dir='./{}'.format(DAT
A_FOLDER))
else:
    print("Embeddings already downloaded.")

if not os.path.exists(UNZIP_FOLDER):
    import zipfile
    if not os.path.exists(ZIP_FILE) and os.path.exists(ZIP_FILE_AL
T):
        ZIP_FILE = ZIP_FILE_ALT
    with zipfile.ZipFile(ZIP_FILE, "r") as zip_ref:
        print("Extracting embeddings to '{}'".format(UNZIP_FOLDER))
        zip_ref.extractall(UNZIP_FOLDER)
else:
    print("Embeddings already extracted.")
```

```
Embeddings already downloaded.
Embeddings already extracted.
```

In [207]:
```
def load_embedding_from_disks(embeddings_filename, with_indexes=Tru
e):
    """
    Read a embeddings txt file. If `with_indexes=True`,
    we return a tuple of two dictionnaries
    `(word_to_index_dict, index_to_embedding_array)`,
    otherwise we return only a direct
    `word_to_embedding_dict` dictionnary mapping
    from a string to a numpy array.
    """
    if with_indexes:
        word_to_index_dict = dict()
        index_to_embedding_array = []

    else:
        word_to_embedding_dict = dict()

    with open(embeddings_filename, 'r', encoding='utf-8') as embedd
ings_file:
        for (i, line) in enumerate(embeddings_file):

            split = line.split(' ')

            word = split[0]

            representation = split[1:]
            representation = np.array(
                [float(val) for val in representation]
            )

            if with_indexes:
                word_to_index_dict[word] = i
                index_to_embedding_array.append(representation)
            else:
                word_to_embedding_dict[word] = representation

    # Empty representation for unknown words.
    _WORD_NOT_FOUND = [0.0] * len(representation)
    if with_indexes:
        _LAST_INDEX = i + 1
        word_to_index_dict = defaultdict(
            lambda: _LAST_INDEX, word_to_index_dict)
        index_to_embedding_array = np.array(
            index_to_embedding_array + [_WORD_NOT_FOUND])
        return word_to_index_dict, index_to_embedding_array
    else:
        word_to_embedding_dict = defaultdict(lambda: _WORD_NOT_FOUN
D)
        return word_to_embedding_dict

print('\nLoading embeddings from', GLOVE_FILENAME)
word_to_index, index_to_embedding = \
    load_embedding_from_disks(GLOVE_FILENAME, with_indexes=True)
print("Embedding loaded from disks.")
```

```
Loading embeddings from embeddings/glove.6B/glove.6B.50d.txt
Embedding loaded from disks.
```

In [208]:
```
# Compare training and dev
def plot_history(history):
    accuracy = history.history['acc']
    val_accuracy = history.history['val_acc']
    loss = history.history['loss']
    val_loss = history.history['val_loss']
    epoch_number = range(1, len(accuracy) + 1)
    plt.style.use('ggplot') # Grammar of Graphics plots
    plt.figure(figsize=(20, 10))
    plt.subplot(1, 2, 1)
    plt.plot(epoch_number, accuracy, 'b', label='Training')
    plt.plot(epoch_number, val_accuracy, 'r', label='Dev')
    plt.title('Training and Dev Set Accuracy')
    plt.xlabel('Epoch Number')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.subplot(1, 2, 2)
    plt.plot(epoch_number, loss, 'b', label='Training')
    plt.plot(epoch_number, val_loss, 'r', label='Dev')
    plt.title('Training and Dev Set Loss')
    plt.xlabel('Epoch Number')
    plt.ylabel('Loss')
    plt.legend()
    plt.savefig('fig-training-process.pdf',
        papertype = 'letter', orientation ='landscape')
    plt.show()
    plt.close()

# plot confusion matrix to external file
def plot_confusion(cm_data):
    plt.figure()
    selected_cmap = sns.cubehelix_palette(light=1, as_cmap=True)
    sns_plot = sns.heatmap(cm_data, annot=True, fmt="d", \
            cmap = selected_cmap, linewidths = 0.5, cbar = False)
    sns_plot.set_yticklabels(sns_plot.get_yticklabels(), rotation =
0)
    plt.title('Confusion Matrix')
    plt.ylabel('True Digit')
    plt.xlabel('Predicted Digit')
    # plt.show() # use if plot to screen is desired
    plt.savefig('fig-confusion-matrix.pdf',
        papertype = 'letter', orientation ='landscape')
    plt.close()

def train_neural_network(session, optimizer, keep_probability, feat
ure_batch, label_batch):
    session.run(optimizer,
                feed_dict={
                    x: feature_batch,
                    y: label_batch,
                    keep_prob: keep_probability
                })

def print_stats(session, feature_batch, label_batch, cost, accurac
```

In [209]:
```
def normalize(x):
    """
        argument
            - x: input image data in numpy array [32, 32, 3]
        return
            - normalized x
    """
    min_val = np.min(x)
    max_val = np.max(x)
    x = (x-min_val) / (max_val-min_val)
    return x

def one_hot_encode(x):
    """
        argument
            - x: a list of labels
        return
            - one hot encoding matrix (number of labels, number of
class)
    """
    encoded = np.zeros((len(x), 47))

    for idx, val in enumerate(x):
        encoded[idx][val] = 1

    return encoded

def evaluate_model(model, train_features, train_labels, test_featur
es, test_labels):
    # evaluate fitted model on the full training set
    train_loss, train_acc = model.evaluate(train_features,train_lab
els,verbose = 3)
    print('\nFull training set accuracy:', \
        '{:6.4f}'.format(np.round(train_acc, decimals = 4)))
    # evaluate the fitted model on the hold-out test set
    test_loss, test_acc = model.evaluate(test_features, test_label
s, verbose = 3)
    print('Hold-out test set accuracy:', \
        '{:6.4f}'.format(np.round(test_acc, decimals = 4)))

def load_label_names():
    return ['cocoa','grain','veg-oil','earn','acq','wheat','copper
','housing','money-supply',
              'coffee','sugar','trade','reserves','ship','cotton','
carcass','crude','nat-gas',
              'cpi','money-fx','interest','gnp','meal-feed','alum
','oilseed','gold','tin',
              'strategic-metal','livestock','retail','ipi','iron-st
eel','rubber','heat','jobs',
              'lei','bop','zinc','orange','pet-chem','dlr','gas','s
ilver','wpi','hog','lead']

def classification_report(model, test_features, test_labels):
    # examine the predicted values within a precision/recall framew
```

```
Targets:
```

Out[209]:

```
['cocoa',
 'grain',
 'veg-oil',
 'earn',
 'acq',
 'wheat',
 'copper',
 'housing',
 'money-supply',
 'coffee',
 'sugar',
 'trade',
 'reserves',
 'ship',
 'cotton',
 'carcass',
 'crude',
 'nat-gas',
 'cpi',
 'money-fx',
 'interest',
 'gnp',
 'meal-feed',
 'alum',
 'oilseed',
 'gold',
 'tin',
 'strategic-metal',
 'livestock',
 'retail',
 'ipi',
 'iron-steel',
 'rubber',
 'heat',
 'jobs',
 'lei',
 'bop',
 'zinc',
 'orange',
 'pet-chem',
 'dlr',
 'gas',
 'silver',
 'wpi',
 'hog',
 'lead']
```

Model with no embedding space

Word vectors obtained from one-hot encoding.

Using the in-built keras functions that strip special characters and take into account the

N most important words. These are high-dimensional and sparse.

Quite high accuracy since this is a word association problem and the semantic relationships aren't as important.

The resulting space has no structure.

In [210]:
```
noEmbeddingModel = Sequential()
noEmbeddingModel.add(Dense(512, input_shape=(max_features,)))
noEmbeddingModel.add(Activation('relu'))
noEmbeddingModel.add(Dropout(0.3))
noEmbeddingModel.add(Dense(num_classes))
noEmbeddingModel.add(Activation('softmax')) #Softmax is used for mu
lti-class classification
noEmbeddingModel.compile(loss='categorical_crossentropy', optimizer
='adam', metrics=['accuracy'])
noEmbeddingModel.summary()
```

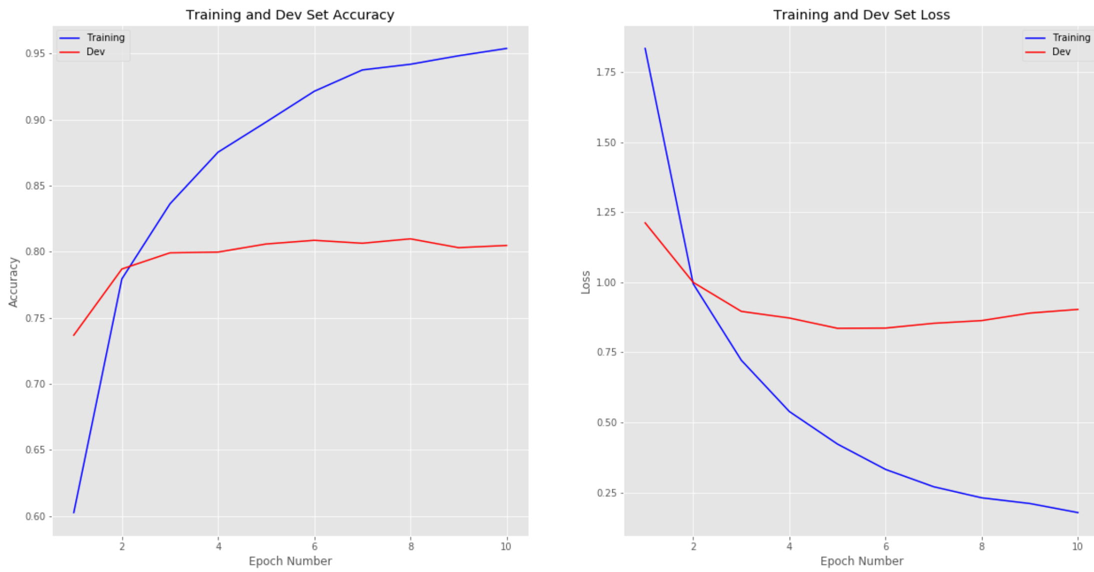
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_60 (Dense)             (None, 512)               512512
_________________________________________________________________
activation_31 (Activation)   (None, 512)               0
_________________________________________________________________
dropout_25 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_61 (Dense)             (None, 46)                23598
_________________________________________________________________
activation_32 (Activation)   (None, 46)                0
=================================================================
Total params: 536,110
Trainable params: 536,110
Non-trainable params: 0
_________________________________________________________________
```

In [211]:
```
testingData.shape
```

Out[211]:
```
(2246, 1000)
```

In [212]:
```
model_name = "NoEmbeddingModel.h5"
train_model(noEmbeddingModel, model_name, trainingData, trainingLab
els, testingData, testingLabels)
```

```
Training...
Epoch 00010: early stopping
Saved best trained model at NoEmbeddingModel.h5

Time of execution for training (seconds):     15.222

Full training set accuracy: 0.9334
Hold-out test set accuracy: 0.7890
```



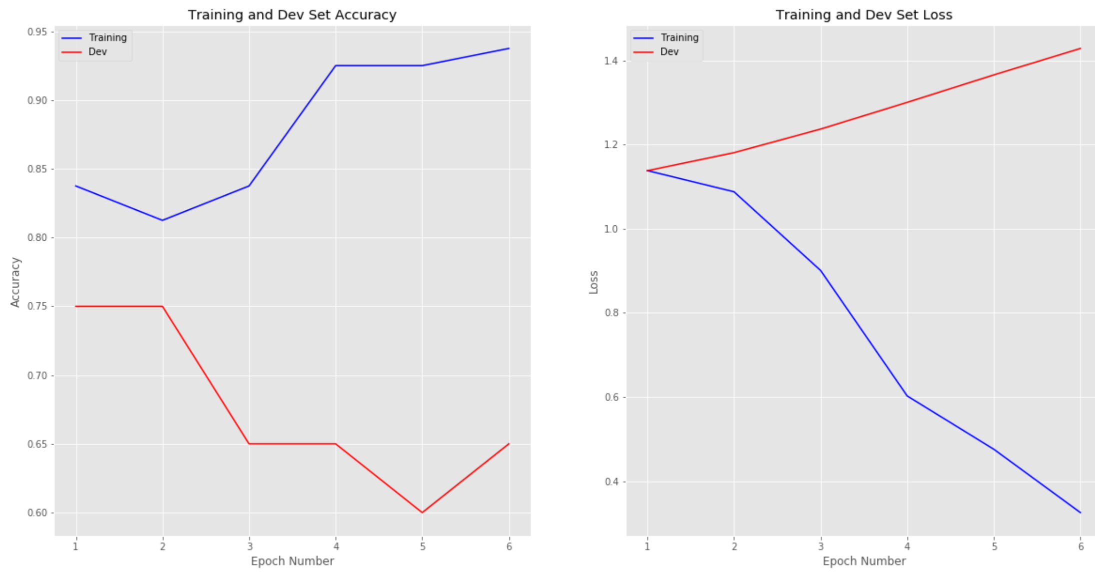
Out[212]:
```
<keras.callbacks.History at 0x7f2e62a8b8d0>
```

Model with no embedding space: Working with a small number of cases

Surprisingling high accuracy still. High training set accuracy due to overfitting.

In [213]:
```
model_name="NoEmbeddingSmall.h5"
train_model(noEmbeddingModel, model_name, trainingData[-100:], trainingLabels[-100:], testingData, testingLabels)
```

```
Training...
Epoch 00006: early stopping
Saved best trained model at NoEmbeddingSmall.h5

Time of execution for training (seconds):      0.069

Full training set accuracy: 0.9100
Hold-out test set accuracy: 0.7752
```



Out[213]: <keras.callbacks.History at 0x7f2dc881bbe0>

Model with Learned Embedding Space

Learned embeddings that are low-dimensional floating-point vectors that are learned from these data.

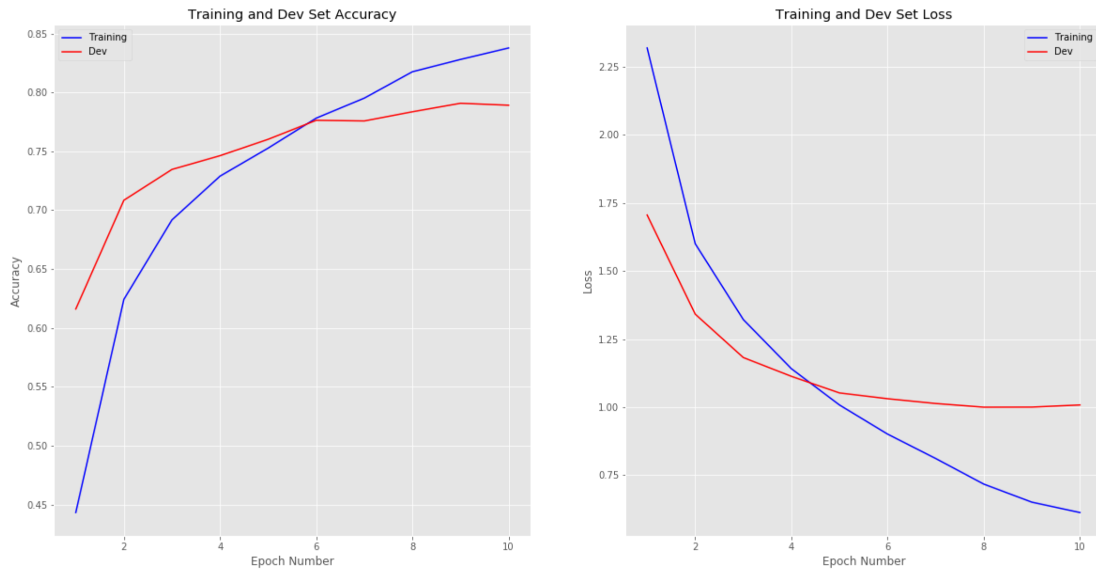
Some semantic relationships between the words are coded as vectors.

In [214]:
```
learnedEmbeddingModel = Sequential()

#learnedEmbeddingModel.add(Embedding(max_features, 8, input_length=
maxlen))
learnedEmbeddingModel.add(Embedding(max_features, embedding_dim, in
put_length=maxlen))
learnedEmbeddingModel.add(Flatten())
learnedEmbeddingModel.add(Dense(num_classes))
learnedEmbeddingModel.add(Activation('relu'))
learnedEmbeddingModel.add(Dropout(0.3))
learnedEmbeddingModel.add(Dense(num_classes, activation='softmax'))
learnedEmbeddingModel.compile(optimizer='adam', loss='categorical_c
rossentropy', metrics=['acc'])
learnedEmbeddingModel.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_25 (Embedding)     (None, 1000, 50)          50000
_________________________________________________________________
flatten_14 (Flatten)         (None, 50000)             0
_________________________________________________________________
dense_62 (Dense)             (None, 46)                2300046
_________________________________________________________________
activation_33 (Activation)   (None, 46)                0
_________________________________________________________________
dropout_26 (Dropout)         (None, 46)                0
_________________________________________________________________
dense_63 (Dense)             (None, 46)                2162
=================================================================
Total params: 2,352,208
Trainable params: 2,352,208
Non-trainable params: 0
_________________________________________________________________
```

In [215]:
```
model_name = "LearnedEmbedding.h5"
train_model(learnedEmbeddingModel, model_name, trainingData, traini
ngLabels, testingData, testingLabels)
```

```
Training...
Saved best trained model at LearnedEmbedding.h5

Time of execution for training (seconds):     22.840

Full training set accuracy: 0.8898
Hold-out test set accuracy: 0.7796
```



Out[215]: <keras.callbacks.History at 0x7f2e180af160>

Model with Pre-Trained Embedding Layer

The pretrained embedding layer captures the generic aspects of the language

In [216]:
```
#https://medium.com/@sabber/classifying-yelp-review-comments-using-
cnn-lstm-and-pre-trained-glove-word-embeddings-part-3-53fcea9a17fa
embeddings_index = dict()
f = open(GLOVE_FILENAME)
for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
f.close()
```

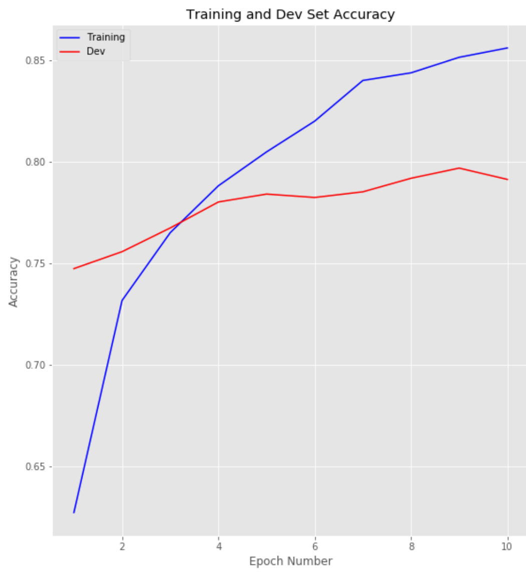
In [217]:
```
embedding_dim = 50

embedding_matrix = np.zeros((max_words, embedding_dim))
for word, i in word_index.items():
    embedding_vector = embeddings_index.get(word)
    if i < max_words:
        if embedding_vector is not None:
            # Words not found in embedding index will be all-zeros.
            embedding_matrix[i] = embedding_vector
```

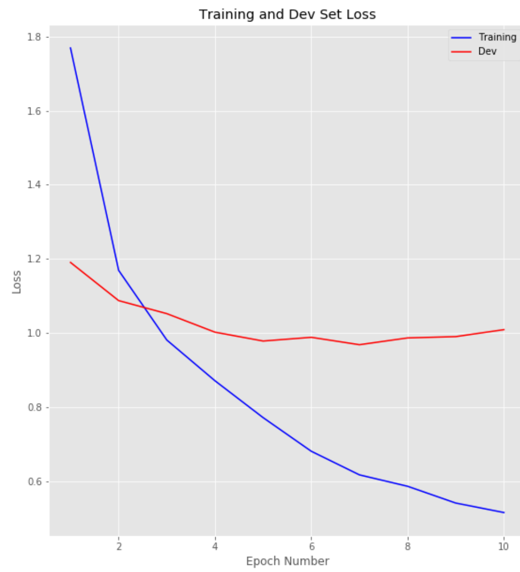
In [218]:
```
pretrainedModel = Sequential()
pretrainedModel.add(Embedding(max_words, embedding_dim, input_length=maxlen, weights=[embedding_matrix],trainable=False))
pretrainedModel.add(Flatten())
pretrainedModel.add(Dense(32, activation='relu'))
pretrainedModel.add(Dropout(0.3))
pretrainedModel.add(Dense(46, activation='softmax'))
pretrainedModel.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['acc'])
pretrainedModel.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
embedding_26 (Embedding)     (None, 1000, 50)          500000    
_________________________________________________________________
flatten_15 (Flatten)         (None, 50000)             0         
_________________________________________________________________
dense_64 (Dense)             (None, 32)                1600032   
_________________________________________________________________
dropout_27 (Dropout)         (None, 32)                0         
_________________________________________________________________
dense_65 (Dense)             (None, 46)                1518      
=================================================================
Total params: 2,101,550
Trainable params: 1,601,550
Non-trainable params: 500,000
_________________________________________________________________
```

In [219]:
```
model_name="pretrained.h5"
train_model(pretrainedModel, model_name, trainingData, trainingLabe
ls, testingData, testingLabels)
```

```
Training...
Saved best trained model at pretrained.h5

Time of execution for training (seconds):     20.107

Full training set accuracy: 0.9050
Hold-out test set accuracy: 0.7738
```





Out[219]: <keras.callbacks.History at 0x7f2e6ce770f0>

Pre-Trained Models: Working with a small number of cases

Pre-Trained embeddings are useful when there aren't enough data to learn the features

This shows similar accuracy as above which makes sense since the embeddings help with the learning.

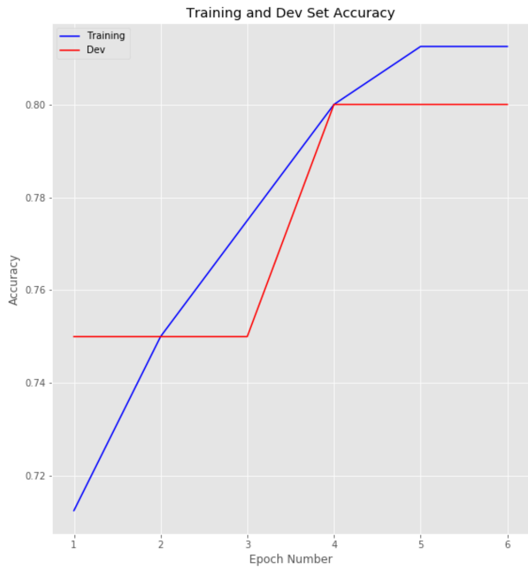
The embeddings are computed using word-occurence statistics (what words co-occur in sentences or documents)

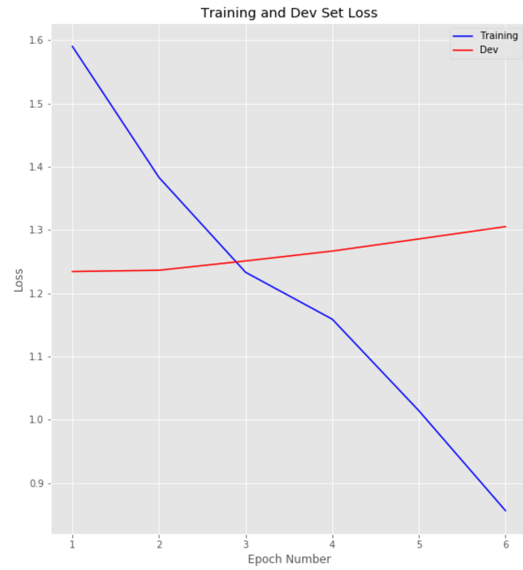
[https://nlp.stanford.edu/projects/glove (https://nlp.stanford.edu/projects/glove)](https://nlp.stanford.edu/projects/glove). Developed by Stanford researchers in 2014 based on factorizing a matrix of

word co-occurences statistics.

In [220]:
```
model_name="pretrainedSmall.h5"
train_model(pretrainedModel, model_name, trainingData[-100:], trainingLabels[-100:], testingData, testingLabels)
```

```
Training...
Epoch 00006: early stopping
Saved best trained model at pretrainedSmall.h5

Time of execution for training (seconds):      0.152

Full training set accuracy: 0.9000
Hold-out test set accuracy: 0.7663
```





Out[220]: <keras.callbacks.History at 0x7f2e185fbcc0>

SimpleRNNs

Processes sequences by iterating through the sequence elements and maintaining a state containing

information relative to what it has seen so far.

Takes inputs of (batch_size, timesteps, input_features)

SimpleRNNs aren't good at processing longs sequences, such as text. Due to the vanishing gradient problem.

Refer to Hochreiter and Schmidhuber (1997).

SimpleRNNs aren't very good at processing long sequences, like text. Other types of recurrent layers perform much better. Let's take a look at some more advanced layers.

In [221]:
```
embedding_matrix = np.zeros((max_words, embedding_dim))
for word, i in word_index.items():
    embedding_vector = embeddings_index.get(word)
    if i < max_words:
        if embedding_vector is not None:
            # Words not found in embedding index will be all-zeros.
            embedding_matrix[i] = embedding_vector

trainingDataSeq = sequence.pad_sequences(trainingDataRaw, maxlen=ma
xlen)
testingDataSeq = sequence.pad_sequences(testingDataRaw, maxlen=maxl
en)
print('input_train shape:', trainingDataSeq.shape)
print('input_test shape:', testingDataSeq.shape)
```

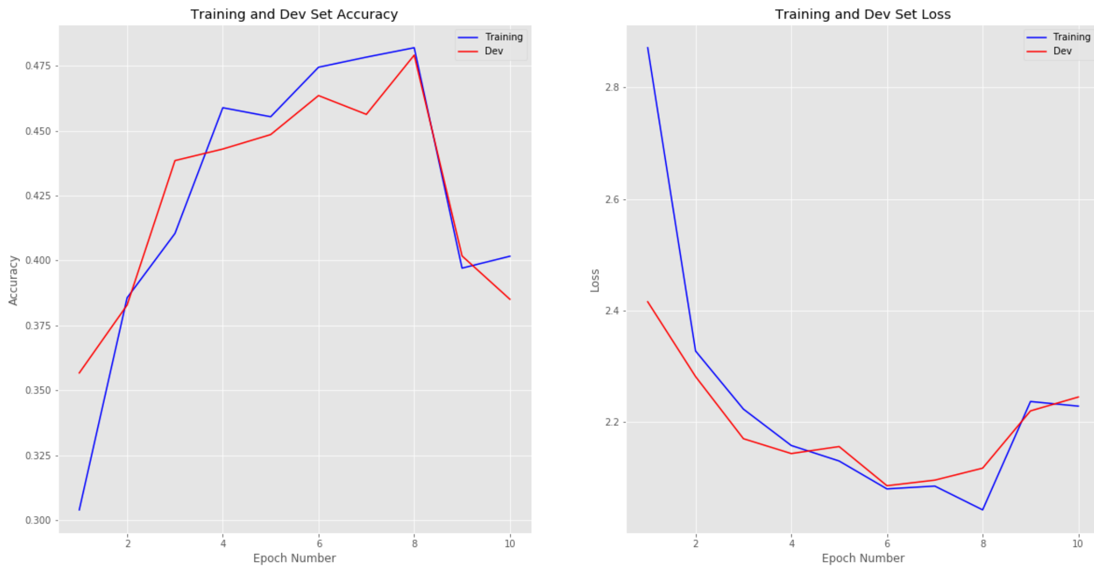
```
input_train shape: (8982, 1000)
input_test shape: (2246, 1000)
```

In [222]:
```
simpleRNNModel = Sequential()
simpleRNNModel.add(Embedding(max_words, embedding_dim, input_length
=maxlen, weights=[embedding_matrix],trainable=False))
simpleRNNModel.add(SimpleRNN(32, return_sequences=True))
simpleRNNModel.add(SimpleRNN(32, return_sequences=True))
simpleRNNModel.add(SimpleRNN(32, return_sequences=True))
simpleRNNModel.add(SimpleRNN(32))
simpleRNNModel.add(Dense(num_classes, activation='softmax'))
simpleRNNModel.compile(optimizer='adam', loss='categorical_crossent
ropy', metrics=['acc'])

simpleRNNModel.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
embedding_27 (Embedding)     (None, 1000, 50)          500000    
_________________________________________________________________
simple_rnn_33 (SimpleRNN)    (None, 1000, 32)          2656      
_________________________________________________________________
simple_rnn_34 (SimpleRNN)    (None, 1000, 32)          2080      
_________________________________________________________________
simple_rnn_35 (SimpleRNN)    (None, 1000, 32)          2080      
_________________________________________________________________
simple_rnn_36 (SimpleRNN)    (None, 32)                2080      
_________________________________________________________________
dense_66 (Dense)             (None, 46)                1518      
=================================================================
Total params: 510,414
Trainable params: 10,414
Non-trainable params: 500,000
_________________________________________________________________
```

In [223]:
```
model_name="simpleRNN.h5"
train_model(simpleRNNModel, model_name, trainingDataSeq, trainingLa
bels, testingDataSeq, testingLabels)
```

```
Training...
Saved best trained model at simpleRNN.h5

Time of execution for training (seconds):   1402.606

Full training set accuracy: 0.4006
Hold-out test set accuracy: 0.3931
```



Out[223]: <keras.callbacks.History at 0x7f2e67a4f128>

Long Short-Term Memory RNNs

LSTMs save information from the sequence to be ported later at a different point. So, it prevents older signals from vanaishing.

Accuracy is much better than the SimpleRNN. I only trained using 3 epochs since it took so long to train.

Not performing better because the long-term structure of the text isn't as important with this problem because we're more concerned with word counts and mapping to the categories in this problem. The fully-connected approach does this fine.

The strength of LSTMs are more apparent with other problems like question-answering and machine translation.

In [224]:
```
LSTMModel = Sequential()
LSTMModel.add(Embedding(max_words, embedding_dim, input_length=maxlen, weights=[embedding_matrix],trainable=False))
LSTMModel.add(LSTM(32))
LSTMModel.add(Dense(num_classes, activation='softmax'))
LSTMModel.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['acc'])
LSTMModel.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
embedding_28 (Embedding)     (None, 1000, 50)          500000    
_________________________________________________________________
lstm_4 (LSTM)                (None, 32)                10624     
_________________________________________________________________
dense_67 (Dense)             (None, 46)                1518      
=================================================================
Total params: 512,142
Trainable params: 12,142
Non-trainable params: 500,000
_________________________________________________________________
```

In [225]:
```
model_name="LSTM.h5"
#train_model(LSTMModel, model_name, trainingDataSeq, trainingLabels, testingDataSeq, testingLabels)
```

Much better accuracy since LSTM suffers less from the vanishing-gradient problem. Slightly better than the full-connected approach.

Not performing better as hyperparameters not tuned and I didn't include regularization.

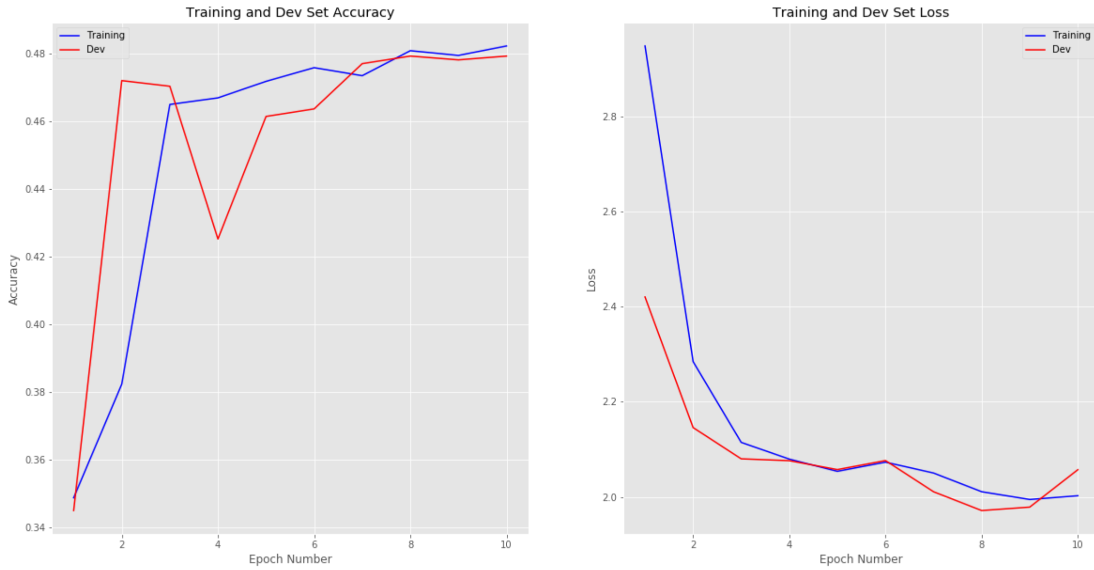
Problem is well-solved looking at the frequency of words in the text when matching to the categories.

... this is what the fully-connected solution does.

RNNs and, in particular, LSTMs are better at machine translation and question-answering.

In [226]:
```
begin_time = time()

## Compile and run the dense model
checkpointer = ModelCheckpoint(filepath=model_name,
                               monitor = 'val_acc',
                               verbose=2,
                               save_best_only=True)

early_stopping = EarlyStopping(monitor='val_loss',
                          min_delta=0,
                          patience=2,
                          verbose=2, mode='auto')

print('Training...')
## Training model
history = LSTMModel.fit(trainingDataSeq, trainingLabels,
                    batch_size=batch_size, epochs=epochs,
                    callbacks=[checkpointer, early_stopping],
                    verbose=2,
                    validation_split=0.2)

print('Saved best trained model at %s ' % model_name)

execution_time = time() - begin_time
print('\nTime of execution for training (seconds):', \
    '{:10.3f}'.format(np.round(execution_time, decimals = 3)))
evaluate_model(LSTMModel, trainingDataSeq, trainingLabels, testingD
ataSeq, testingLabels)
plot_history(history)
```

```
Training...
Train on 7185 samples, validate on 1797 samples
Epoch 1/10
 - 100s - loss: 2.9474 - acc: 0.3488 - val_loss: 2.4200 - val_ac
c: 0.3450

Epoch 00001: val_acc improved from -inf to 0.34502, saving model
to LSTM.h5
Epoch 2/10
 - 91s - loss: 2.2842 - acc: 0.3823 - val_loss: 2.1456 - val_acc:
0.4719

Epoch 00002: val_acc improved from 0.34502 to 0.47190, saving mod
el to LSTM.h5
Epoch 3/10
 - 91s - loss: 2.1144 - acc: 0.4649 - val_loss: 2.0799 - val_acc:
0.4702

Epoch 00003: val_acc did not improve from 0.47190
Epoch 4/10
 - 91s - loss: 2.0793 - acc: 0.4668 - val_loss: 2.0760 - val_acc:
0.4252

Epoch 00004: val_acc did not improve from 0.47190
Epoch 5/10
 - 91s - loss: 2.0536 - acc: 0.4717 - val_loss: 2.0572 - val_acc:
0.4613

Epoch 00005: val_acc did not improve from 0.47190
Epoch 6/10
 - 91s - loss: 2.0729 - acc: 0.4757 - val_loss: 2.0762 - val_acc:
0.4636

Epoch 00006: val_acc did not improve from 0.47190
Epoch 7/10
 - 91s - loss: 2.0500 - acc: 0.4733 - val_loss: 2.0107 - val_acc:
0.4769

Epoch 00007: val_acc improved from 0.47190 to 0.47691, saving mod
el to LSTM.h5
Epoch 8/10
 - 91s - loss: 2.0108 - acc: 0.4807 - val_loss: 1.9712 - val_acc:
0.4791

Epoch 00008: val_acc improved from 0.47691 to 0.47913, saving mod
el to LSTM.h5
Epoch 9/10
 - 90s - loss: 1.9944 - acc: 0.4793 - val_loss: 1.9784 - val_acc:
0.4780

Epoch 00009: val_acc did not improve from 0.47913
Epoch 10/10
 - 91s - loss: 2.0023 - acc: 0.4821 - val_loss: 2.0570 - val_acc:
0.4791
```


Training and Dev Set Accuracy
Training
Dev
Accuracy
Epoch Number
Training and Dev Set Loss
Training
Dev
Loss
Epoch Number