Project 3 Hot Dog Or Not Hot Dog

November 27, 2017

0.0.1 By Dev Ojha, Brandon Hsieh, Eva Su

In [3]:
```
import keras
import numpy as np
import os
import matplotlib.pyplot as plt
from scipy import misc
from keras.models import Model, Sequential
from keras.layers import Dense, Activation, Flatten, Dropout
import pickle
%matplotlib inline
```

In [4]:
```
test_img_fnames = os.listdir('data/data/test')
train_img_fnames = os.listdir('data/data/train')
percentage_data_size = 5 / 100
print(len(test_img_fnames), len(train_img_fnames))
```

```
2500 22500
```

1 Load in the data

First we need to write a function to load into memory each of the images, and then resize them to (224, 224, 3). scipy.misc.imread and scipy.misc.imresize will be useful for this. You should also rescale the images so that they are on a scale of 0 to 1, meaning each pixel value should be between 0 and 1. This can be achieved by dividing by an appropriate constant.

In [5]:
```
def load_img(filename):
    img = misc.imread(filename)
    img = misc.imresize(img, (224,224))
    return img / 255
```

In [4]:
```
test_imgs = [load_img(os.path.join('data/test', fname)) for fname in test_img_fnames[:in
test_imgs = np.stack(test_imgs)
```

In [7]:
```
train_imgs = [load_img(os.path.join('data/data/train', fname)) for fname in train_img_fn
train_imgs = np.stack(train_imgs)
```

In [8]:
```
with open('data/train_labels.pkl', 'rb') as f:
    train_labels = pickle.load(f)
```

2 Load Pretrained VGG network

Now in order to generate the CNN featurizations of our images we need to load a pretrained network. Note that running this network will take a long time on CPU so you can feel free to skip this section and just load the featurizations I provided in the next section

```
In [ ]: VGG16 = keras.applications.vgg16.VGG16(input_shape=(224, 224, 3), weights='imagenet', in
```

```
In [ ]: VGG16.summary()
```

```
In [ ]: featurized_training_data = VGG16.predict(train_imgs, verbose=1)
```

```
In [ ]: featurized_test_data = VGG16.predict(test_imgs, verbose=1)
```

```
In [ ]: # Save featurizations, I changed filenames so as not to overwrite the original
        import pickle
        with open('featurized_train_imgs_gen.pkl', 'wb') as f:
            pickle.dump(featurized_training_data, f)
        with open('featurized_test_imgs_gen.pkl', 'wb') as f:
            pickle.dump(featurized_test_data, f)
```

3 Load featurizations

```
In [11]: import pickle
         with open('featurized_train_imgs.pkl', 'rb') as f:
             featurized_training_data = pickle.load(f)
         with open('featurized_test_imgs.pkl', 'rb') as f:
             featurized_test_data = pickle.load(f)
```

4 Create you network

Now we need to create a network to take in the featurizations and output a label of dog or not dog. To do this you should use Keras' Sequential model. We will need to flatten our (7,7,512) feature input into a vector (HINT: lookup flatten in keras documentation) and then add a Dense layer with some number of neurons (play around with the number of neurons to improve your performance). Then finally we need a Dense layer with 1 neuron and a sigmoid activation to represent our label output. You might want to use more or less model.add calls than have been provided

```
In [12]: model = Sequential()
         model.add(keras.layers.Flatten(input_shape=(7,7,512)))
         model.add(keras.layers.Dense(60, activation="tanh"))
         model.add(keras.layers.Dense(10, activation="relu"))
         model.add(keras.layers.Dense(1, activation="sigmoid"))
```

```
In [14]: model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_1 (Flatten)          (None, 25088)             0
_________________________________________________________________
dense_1 (Dense)              (None, 60)                1505340
_________________________________________________________________
dense_2 (Dense)              (None, 10)                610
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 11
=================================================================
Total params: 1,505,961
Trainable params: 1,505,961
Non-trainable params: 0
_________________________________________________________________
```

5 Now we need to train the network

You need to compile the model first, and then use the fit function. You should use binary crossentropy as your loss.

```
In [15]: model.compile(optimizer="sgd", loss="binary_crossentropy")
```

```
In [16]: model.fit(featurized_training_data, train_labels, epochs=15)
```

```
Epoch 1/15
10000/10000 [==============================] - 10s - loss: 0.5887
Epoch 2/15
10000/10000 [==============================] - 9s - loss: 0.3964
Epoch 3/15
10000/10000 [==============================] - 9s - loss: 0.3149
Epoch 4/15
10000/10000 [==============================] - 8s - loss: 0.2818
Epoch 5/15
10000/10000 [==============================] - 8s - loss: 0.2707
Epoch 6/15
10000/10000 [==============================] - 8s - loss: 0.2409
Epoch 7/15
10000/10000 [==============================] - 8s - loss: 0.2199
Epoch 8/15
10000/10000 [==============================] - 8s - loss: 0.2114
Epoch 9/15
10000/10000 [==============================] - 8s - loss: 0.2017
Epoch 10/15
10000/10000 [==============================] - 8s - loss: 0.1980
Epoch 11/15
10000/10000 [==============================] - 8s - loss: 0.1802
```

```
Epoch 12/15
10000/10000 [==============================] - 8s - loss: 0.1770
Epoch 13/15
10000/10000 [==============================] - 10s - loss: 0.1631
Epoch 14/15
10000/10000 [==============================] - 8s - loss: 0.1716
Epoch 15/15
10000/10000 [==============================] - 8s - loss: 0.1490
```

```
Out[16]: <keras.callbacks.History at 0x1ca80110780>
```

Now we need to predict labels for the test set and output it to file. Use keras predict for this. Remember that the predictions are real values between 0 and 1 and you should be outputting just 0 or 1, not a value between.

```
In [85]: predictions = model.predict(featurized_test_data)
```

```
In [86]: i = 'Y'
         with open('test_labels_%s.txt' % i, 'w') as f:
             f.write("Id,Label\n")
             for i, val in enumerate(predictions):
                 prediction = 0 if val < .5 else 1
                 f.write(str(i))
                 f.write(',')
                 f.write(str(prediction))
                 f.write('\n')
```

```
In [2]: ## This is a method we made to compare our generated files, since we have a limited numb
        ## We look at how much generated files disagree, and compare those disagreeements with s
        # Add disagreeing images manually here. cats are 1, dogs are 0
        known_dic = {0:0, 16:1, 23:1, 59:1, 79:1, 85:0, 89:1, 93:0, 107:1, 116:1, 132:1, 154:0,
                     279:1, 271:1, 215:0, 202:1, 195:0, 174:0, 154:0, 335:1, 366:1, 350:1, 351:1,
                     913:0, 923:1,2313:1, 2319:0, 2321:1, 2496: 0, 2461:0}
        def compare_files(f1, f2):
            f1 = open(f1,'r')
            f2 = open(f2,'r')
            f1.readline()
            f2.readline()
            count, netdif, unknowns, wrong, both_wrong = 0, 0, 0, 0, 0
            for line_num in range(2500):
                mod_1 = f1.readline()
                pred_1 = int(mod_1[mod_1.index(",")+1:][:-1])
                mod_2 = f2.readline()
                pred_2 = int(mod_2[mod_2.index(",")+1:][:-1])
                if pred_2 != pred_1:
                    netdif += 1
                    if line_num in known_dic:
                        if known_dic[line_num] == pred_1:
```

```
                        count += 1
                    else:
                        count -= 1
                        # print(line_num)
                        wrong += 1
                else:
                    #print("Disagreement on %s , with unknown result" % line_num)
                    unknowns += 1
            else:
                if line_num in known_dic and known_dic[line_num] != mod_2: both_wrong += 1;

        print("model 1 scored " + str(count) + " more than model 2")
        print("Net number of different lines is " + str(netdif))
        print("model 1 had %s wrong lines" % str(wrong))
        print("on top of that, both models had %s wrong" % both_wrong)
        print("Number of unknown lines is " + str(unknowns))

    compare_files('test_labels_2.txt','test_labels_1.txt')
```

```
model 1 scored 9 more than model 2
Net number of different lines is 151
model 1 had 12 wrong lines
on top of that, both models had 1 wrong
Number of unknown lines is 118
```

6 Part 2

```
In [17]: from sklearn.decomposition import PCA
```

Now we want to run PCA on our images and our featurizations to see the difference. To save time we will just run it on the first 500 images. Take the first 500 images and 500 featurizations and reshape them into shape (500, x).

```
In [31]: original_imgs = np.reshape(train_imgs[:500], (500, 224*224*3))
         featurized = np.reshape(featurized_training_data[:500], (500, 7*7*512))
```

Run PCA with 2 components on the original images

```
In [34]: pca_orig = PCA(n_components=2)
         pca_orig.fit(original_imgs)
```

```
Out[34]: PCA(copy=True, iterated_power='auto', n_components=2, random_state=None,
           svd_solver='auto', tol=0.0, whiten=False)
```

Run PCA with 2 components on the featurizations

```
In [35]: pca_featurized = PCA(n_components=2)
         pca_featurized = pca_featurized.fit(featurized)
```

Project the original images and featurizations onto the 2 principal components. (HINT: look at the PCA.transform function)

```
In [36]: projected_orig_imgs = pca_orig.transform(original_imgs)
         projected_featurized = pca_featurized.transform(featurized)
```

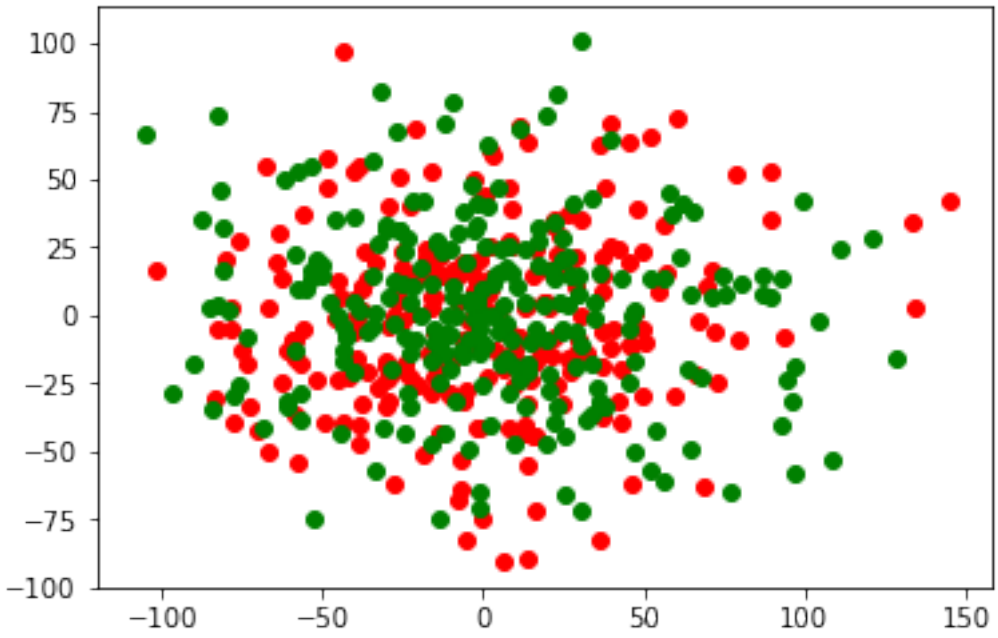
Find the indices of the labels that are cats and the indices that are dogs. np.where will make this very easy

```
In [45]: labels = train_labels[:500]
         cat_inds = np.where(labels == 1)
         dog_inds = np.where(labels == 0)
```

Plot the PCA of the original images and the PCA of the featurization

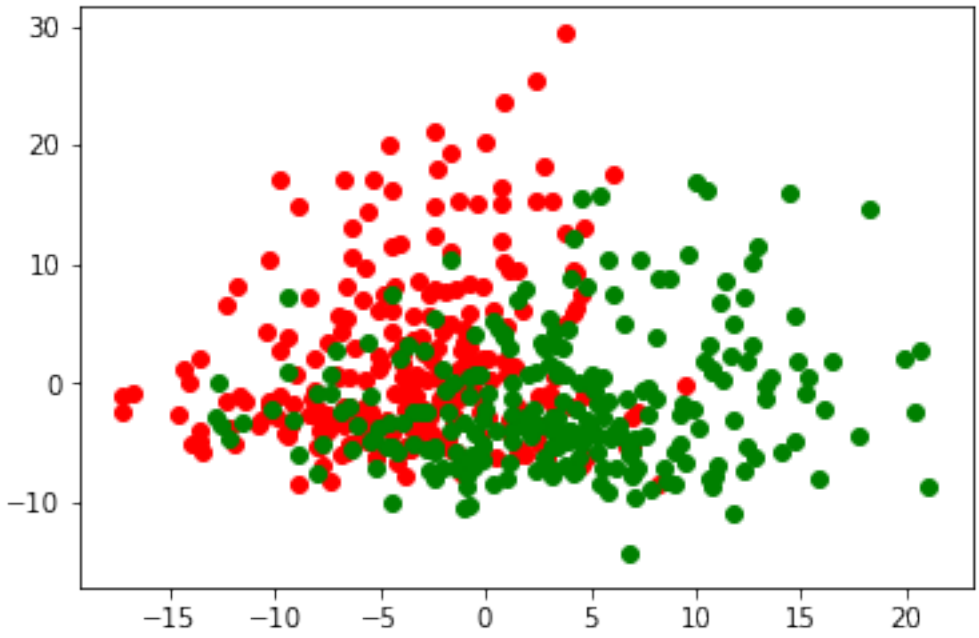
```
In [46]: plt.scatter(projected_orig_imgs[cat_inds ,0], projected_orig_imgs[cat_inds, 1], c='red'
         plt.scatter(projected_orig_imgs[dog_inds, 0], projected_orig_imgs[dog_inds, 1], c='gree
```

```
Out[46]: <matplotlib.collections.PathCollection at 0x1ca814e9d68>
```




```
In [47]: plt.scatter(projected_featurized[cat_inds ,0], projected_featurized[cat_inds, 1], c='re
         plt.scatter(projected_featurized[dog_inds, 0], projected_featurized[dog_inds, 1], c='gr
```

```
Out[47]: <matplotlib.collections.PathCollection at 0x1ca83b2aa58>
```



Question 1: Describe the differences you see between the two plots in terms of the clustering of the classes?

The partition between the two clusters is more defined in the PCA projection of the featurized data. In the PCA projection of the featurized data, the green dots are concentrated to the bottom right, and the red dots are concentrated to the top left. The overlap between the data clusters is smaller than in the PCA projection of the original images. There is no clear clustering in the PCA projection of he original images.

Question 2: Based on your answer to Question 1, describe why it is useful to featurize with a CNN before training a fully connnected model to classify the images.

Its useful to featurize the data with a CNN before training a fully connected model because it is easier to identify clusters in this data (as evident by the PCA projections), and therefore the fully connected model will be able to classify the data better.

An intuitive reason, aside from PCA for why this works is that essentially there exists data in the featurized version of he data that better indicates whether an image is a dog or a cat than just the values of each of the pixels in the image. We would expect this to be true, but this can also be seen from the plots. If we knew the location of a point on each plot, we would have a better idea if it was a dog or cat using the featurized data's PCA plot than if we used the original image's PCA plot. A similar logic holds for the neural network, as the PCA plots are showing us that the features do identify the data better than the original image pixel values since the clusters are more identifiable.