dog_app

January 1, 2018

1 Artificial Intelligence Nanodegree

1.1 Convolutional Neural Networks

1.2 Project: Write an Algorithm for a Dog Identification App

In this notebook, some template code has already been provided for you, and you will need to implement additional functionality to successfully complete this project. You will not need to modify the included code beyond what is requested. Sections that begin with **'(IMPLEMENTATION)'** in the header indicate that the following block of code will require additional functionality which you must provide. Instructions will be provided for each section, and the specifics of the implementation are marked in the code block with a 'TODO' statement. Please be sure to read the instructions carefully!

> **Note**: Once you have completed all of the code implementations, you need to finalize your work by exporting the iPython Notebook as an HTML document. Before exporting the notebook to html, all of the code cells need to have been run so that reviewers can see the final implementation and output. You can then export the notebook by using the menu above and navigating to","**File -> Download as -> HTML (.html)**. Include the finished document along with this notebook as your submission.

In addition to implementing code, there will be questions that you must answer which relate to the project and your implementation. Each section where you will answer a question is preceded by a **'Question X'** header. Carefully read each question and provide thorough answers in the following text boxes that begin with **'Answer:'**. Your project submission will be evaluated based on your answers to each of the questions and the implementation you provide.

> **Note:** Code and Markdown cells can be executed using the **Shift + Enter** keyboard shortcut. Markdown cells can be edited by double-clicking the cell to enter edit mode.

The rubric contains *optional* "Stand Out Suggestions" for enhancing the project beyond the minimum requirements. If you decide to pursue the "Stand Out Suggestions", you should include the code in this IPython notebook.

Step 0: Import Datasets

1.2.1 Import Dog Dataset

In the code cell below, we import a dataset of dog images. We populate a few variables through the use of the `load_files` function from the scikit-learn library: - `train_files`, `valid_files`, `test_files` - numpy arrays containing file paths to images - `train_targets`, `valid_targets`, `test_targets` - numpy arrays containing onehot-encoded classification labels - `dog_names` - list of string-valued dog breed names for translating labels

```
In [1]: from sklearn.datasets import load_files
        from keras.utils import np_utils
        import numpy as np
        from glob import glob

        # define function to load train, test, and validation datasets
        def load_dataset(path):
            data = load_files(path)
            dog_files = np.array(data['filenames'])
            dog_targets = np_utils.to_categorical(np.array(data['target']), 133)
            return dog_files, dog_targets

        # load train, test, and validation datasets
        train_files, train_targets = load_dataset('dogImages/train')
        valid_files, valid_targets = load_dataset('dogImages/valid')
        test_files, test_targets = load_dataset('dogImages/test')

        # load list of dog names
        dog_names = [item[20:-1] for item in sorted(glob("dogImages/train/*/"))]

        # print statistics about the dataset
        print('There are %d total dog categories.' % len(dog_names))
        print('There are %s total dog images.\n' % len(np.hstack([train_files, valid_files, tes
        print('There are %d training dog images.' % len(train_files))
        print('There are %d validation dog images.' % len(valid_files))
        print('There are %d test dog images.'% len(test_files))
```

```
Using TensorFlow backend.


There are 133 total dog categories.
There are 8351 total dog images.

There are 6680 training dog images.
There are 835 validation dog images.
There are 836 test dog images.
```

1.2.2 Import Human Dataset

In the code cell below, we import a dataset of human images, where the file paths are stored in the numpy array `human_files`.

In [2]:
```
import random
random.seed(8675309)

# load filenames in shuffled human dataset
human_files = np.array(glob("lfw/*/*"))
random.shuffle(human_files)

# print statistics about the dataset
print('There are %d total human images.' % len(human_files))
```

```
There are 13233 total human images.
```

Step 1: Detect Humans

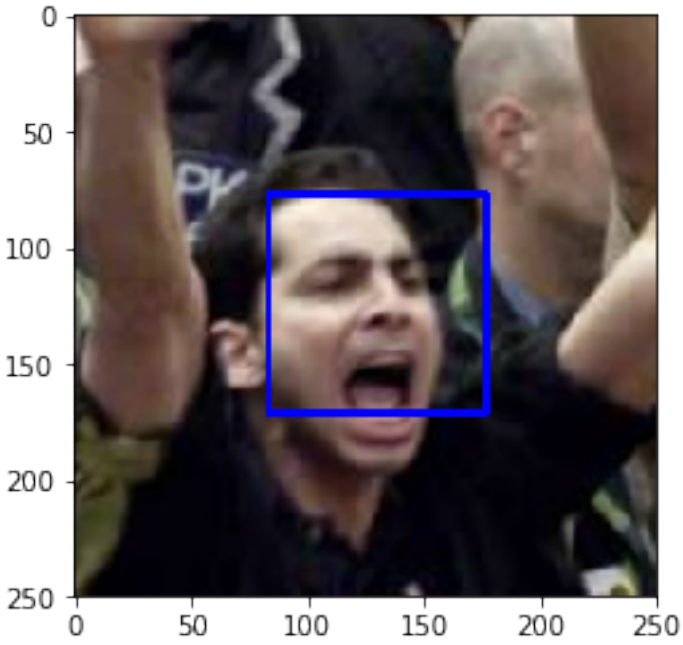
We use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images. OpenCV provides many pre-trained face detectors, stored as XML files on github. We have downloaded one of these detectors and stored it in the `haarcascades` directory.

In the next code cell, we demonstrate how to use this detector to find human faces in a sample image.

In [4]:
```
import cv2
import matplotlib.pyplot as plt
%matplotlib inline

# extract pre-trained face detector
face_cascade = cv2.CascadeClassifier('haarcascades/haarcascade_frontalface_alt.xml')

# load color (BGR) image
img = cv2.imread(human_files[4])
# convert BGR image to grayscale
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# find faces in image
faces = face_cascade.detectMultiScale(gray)

# print number of faces detected in the image
print('Number of faces detected:', len(faces))

# get bounding box for each detected face
for (x,y,w,h) in faces:
    # add bounding box to color image
    cv2.rectangle(img,(x,y),(x+w,y+h),(255,0,0),2)

# convert BGR image to RGB for plotting
cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

# display the image, along with bounding box
```

```
        plt.imshow(cv_rgb)
        plt.show()
```

```
Number of faces detected: 1
```



Before using any of the face detectors, it is standard procedure to convert the images to grayscale. The `detectMultiScale` function executes the classifier stored in `face_cascade` and takes the grayscale image as a parameter.

In the above code, `faces` is a numpy array of detected faces, where each row corresponds to a detected face. Each detected face is a 1D array with four entries that specifies the bounding box of the detected face. The first two entries in the array (extracted in the above code as `x` and `y`) specify the horizontal and vertical positions of the top left corner of the bounding box. The last two entries in the array (extracted here as `w` and `h`) specify the width and height of the box.

1.2.3 Write a Human Face Detector

We can use this procedure to write a function that returns `True` if a human face is detected in an image and `False` otherwise. This function, aptly named `face_detector`, takes a string-valued file path to an image as input and appears in the code block below.

```
In [5]: # returns "True" if face is detected in image stored at img_path
        def face_detector(img_path):
            img = cv2.imread(img_path)
            gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
            faces = face_cascade.detectMultiScale(gray)
            return len(faces) > 0
```

1.2.4 (IMPLEMENTATION) Assess the Human Face Detector

Question 1: Use the code cell below to test the performance of the `face_detector` function.
- What percentage of the first 100 images in `human_files` have a detected human face? 98% - What percentage of the first 100 images in `dog_files` have a detected human face? 11%

Ideally, we would like 100% of human images with a detected face and 0% of dog images with a detected face. You will see that our algorithm falls short of this goal, but still gives acceptable performance. We extract the file paths for the first 100 images from each of the datasets and store them in the numpy arrays `human_files_short` and `dog_files_short`.

Answer:

```
In [6]: human_files_short = human_files[:100]
        dog_files_short = train_files[:100]
        # Do NOT modify the code above this line.

        ## TODO: Test the performance of the face_detector algorithm
        ## on the images in human_files_short and dog_files_short.

        print("Perecentage of human faces detected in humans: %d%%" %
              (len([item for item in human_files_short if face_detector(item)]) / len(human_fi

        print("Perecentage of human faces detected in dogs: %d%%" %
              (len([item for item in dog_files_short if face_detector(item)]) / len(dog_files_s

Perecentage of human faces detected in humans: 98%
Perecentage of human faces detected in dogs: 11%
```

Question 2: This algorithmic choice necessitates that we communicate to the user that we accept human images only when they provide a clear view of a face (otherwise, we risk having unneccessarily frustrated users!). In your opinion, is this a reasonable expectation to pose on the user? If not, can you think of a way to detect humans in images that does not necessitate an image with a clearly presented face?

Answer: I do believe it is a reasonable expectation that the user pose. This is true because we have standard poses in our identification cards to make human recognition easier for humans and technology. However, depending on the technology, it may be a good idea to detect other human attributes (shoulders, human accessories/wearables, ect..)

We suggest the face detector from OpenCV as a potential way to detect human images in your algorithm, but you are free to explore other approaches, especially approaches that make use of deep learning :). Please use the code cell below to design and test your own face detection algorithm. If you decide to pursue this *optional* task, report performance on each of the datasets.

```
In [7]: ## (Optional) TODO: Report the performance of another
        ## face detection algorithm on the LFW dataset
        ### Feel free to use as many code cells as needed.
```

Step 2: Detect Dogs

In this section, we use a pre-trained ResNet-50 model to detect dogs in images. Our first line of code downloads the ResNet-50 model, along with weights that have been trained on ImageNet, a very large, very popular dataset used for image classification and other vision tasks. ImageNet contains over 10 million URLs, each linking to an image containing an object from one of 1000 categories. Given an image, this pre-trained ResNet-50 model returns a prediction (derived from the available categories in ImageNet) for the object that is contained in the image.

```
In [8]: from keras.applications.resnet50 import ResNet50

        # define ResNet50 model
        ResNet50_model = ResNet50(weights='imagenet')
```

1.2.5 Pre-process the Data

When using TensorFlow as backend, Keras CNNs require a 4D array (which we'll also refer to as a 4D tensor) as input, with shape

(nb_samples, rows, columns, channels),

where `nb_samples` corresponds to the total number of images (or samples), and `rows`, `columns`, and `channels` correspond to the number of rows, columns, and channels for each image, respectively.

The `path_to_tensor` function below takes a string-valued file path to a color image as input and returns a 4D tensor suitable for supplying to a Keras CNN. The function first loads the image and resizes it to a square image that is 224×224 pixels. Next, the image is converted to an array, which is then resized to a 4D tensor. In this case, since we are working with color images, each image has three channels. Likewise, since we are processing a single image (or sample), the returned tensor will always have shape

(1, 224, 224, 3).

The `paths_to_tensor` function takes a numpy array of string-valued image paths as input and returns a 4D tensor with shape

(nb_samples, 224, 224, 3).

Here, `nb_samples` is the number of samples, or number of images, in the supplied array of image paths. It is best to think of `nb_samples` as the number of 3D tensors (where each 3D tensor corresponds to a different image) in your dataset!

```
In [9]: from keras.preprocessing import image
        from tqdm import tqdm

        def path_to_tensor(img_path):
            # loads RGB image as PIL.Image.Image type
            img = image.load_img(img_path, target_size=(224, 224))
            # convert PIL.Image.Image type to 3D tensor with shape (224, 224, 3)
            x = image.img_to_array(img)
            # convert 3D tensor to 4D tensor with shape (1, 224, 224, 3) and return 4D tensor
            return np.expand_dims(x, axis=0)

        def paths_to_tensor(img_paths):
            list_of_tensors = [path_to_tensor(img_path) for img_path in tqdm(img_paths)]
            return np.vstack(list_of_tensors)
```

1.2.6 Making Predictions with ResNet-50

Getting the 4D tensor ready for ResNet-50, and for any other pre-trained model in Keras, requires some additional processing. First, the RGB image is converted to BGR by reordering the channels. All pre-trained models have the additional normalization step that the mean pixel (expressed in RGB as $[103.939, 116.779, 123.68]$ and calculated from all pixels in all images in ImageNet) must be subtracted from every pixel in each image. This is implemented in the imported function `preprocess_input`. If you're curious, you can check the code for `preprocess_input` here.

Now that we have a way to format our image for supplying to ResNet-50, we are now ready to use the model to extract the predictions. This is accomplished with the `predict` method, which returns an array whose i-th entry is the model's predicted probability that the image belongs to the i-th ImageNet category. This is implemented in the `ResNet50_predict_labels` function below.

By taking the argmax of the predicted probability vector, we obtain an integer corresponding to the model's predicted object class, which we can identify with an object category through the use of this dictionary.

```
In [10]: from keras.applications.resnet50 import preprocess_input, decode_predictions

         def ResNet50_predict_labels(img_path):
             # returns prediction vector for image located at img_path
             img = preprocess_input(path_to_tensor(img_path))
             return np.argmax(ResNet50_model.predict(img))
```

1.2.7 Write a Dog Detector

While looking at the dictionary, you will notice that the categories corresponding to dogs appear in an uninterrupted sequence and correspond to dictionary keys 151-268, inclusive, to include all categories from `'Chihuahua'` to `'Mexican hairless'`. Thus, in order to check to see if an image is predicted to contain a dog by the pre-trained ResNet-50 model, we need only check if the `ResNet50_predict_labels` function above returns a value between 151 and 268 (inclusive).

We use these ideas to complete the `dog_detector` function below, which returns `True` if a dog is detected in an image (and `False` if not).

```
In [11]: ### returns "True" if a dog is detected in the image stored at img_path
         def dog_detector(img_path):
             prediction = ResNet50_predict_labels(img_path)
             return ((prediction <= 268) & (prediction >= 151))
```

1.2.8 (IMPLEMENTATION) Assess the Dog Detector

Question 3: Use the code cell below to test the performance of your `dog_detector` function.
- What percentage of the images in `human_files_short` have a detected dog? 2% - What percentage of the images in `dog_files_short` have a detected dog? 100%

Answer:

```
In [12]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.
         print("Perecentage of dogs detected in human pictures: %d%%" %
               (len([item for item in human_files_short if dog_detector(item)]) / len(human_fi
```

```
        print("Perecentage of dogs detected in dog pictures: %d%%" %
              (len([item for item in dog_files_short if dog_detector(item)]) / len(dog_files_s
```

```
Perecentage of dogs detected in human pictures: 2%
Perecentage of dogs detected in dog pictures: 100%
```

Step 3: Create a CNN to Classify Dog Breeds (from Scratch)

Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN that classifies dog breeds. You must create your CNN *from scratch* (so, you can't use transfer learning *yet*!), and you must attain a test accuracy of at least 1%. In Step 5 of this notebook, you will have the opportunity to use transfer learning to create a CNN that attains greatly improved accuracy.

Be careful with adding too many trainable layers! More parameters means longer training, which means you are more likely to need a GPU to accelerate the training process. Thankfully, Keras provides a handy estimate of the time that each epoch is likely to take; you can extrapolate this estimate to figure out how long it will take for your algorithm to train.

We mention that the task of assigning breed to dogs from images is considered exceptionally challenging. To see why, consider that *even a human* would have great difficulty in distinguishing between a Brittany and a Welsh Springer Spaniel.

Brittany	Welsh Springer Spaniel

It is not difficult to find other dog breed pairs with minimal inter-class variation (for instance, Curly-Coated Retrievers and American Water Spaniels).

Curly-Coated Retriever	American Water Spaniel

Likewise, recall that labradors come in yellow, chocolate, and black. Your vision-based algorithm will have to conquer this high intra-class variation to determine how to classify all of these different shades as the same breed.

Yellow Labrador	Chocolate Labrador

We also mention that random chance presents an exceptionally low bar: setting aside the fact that the classes are slightly imabalanced, a random guess will provide a correct answer roughly 1 in 133 times, which corresponds to an accuracy of less than 1%.

Remember that the practice is far ahead of the theory in deep learning. Experiment with many different architectures, and trust your intuition. And, of course, have fun!

1.2.9 Pre-process the Data

We rescale the images by dividing every pixel in every image by 255.

```
In [13]: from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True

         # pre-process the data for Keras
         train_tensors = paths_to_tensor(train_files).astype('float32')/255
         valid_tensors = paths_to_tensor(valid_files).astype('float32')/255
         test_tensors = paths_to_tensor(test_files).astype('float32')/255
```

```
100%|| 6680/6680 [00:53<00:00, 124.14it/s]
100%|| 835/835 [00:06<00:00, 139.16it/s]
100%|| 836/836 [00:05<00:00, 139.87it/s]
```

1.2.10 (IMPLEMENTATION) Model Architecture

Create a CNN to classify dog breed. At the end of your code cell block, summarize the layers of your model by executing the line:

```
model.summary()
```

We have imported some Python modules to get you started, but feel free to import as many modules as you need. If you end up getting stuck, here's a hint that specifies a model that trains relatively fast on CPU and attains >1% test accuracy in 5 epochs:

Question 4: Outline the steps you took to get to your final CNN architecture and your reasoning at each step. If you chose to use the hinted architecture above, describe why you think that CNN architecture should work well for the image classification task.

Answer: I initially began using the example example CNN architecture. It had fewer parameters than my architecture and trained relatively fast. However, the example architecture did not produce accuracy > 1%. Using larger filters, kernel_size, and strides dramatically increased accuracy. The relu activation function was used per suggestion of Udacity AIND instructors. After reviewing the keras documentation and examples, I chose to use a densely connected neural network followed by a relu activation function which increased the model's accuracy to 10%.

```
In [18]: from keras.layers import Conv2D, MaxPooling2D, GlobalAveragePooling2D
         from keras.layers import Dropout, Flatten, Dense, Activation
         from keras.models import Sequential

         model = Sequential()

         model.add(Conv2D(filters=32,kernel_size=3,strides=3,activation='relu', padding='same'
                          input_shape=(224, 224, 3)))
         model.add(MaxPooling2D(pool_size=(2,2)))
         model.add(Conv2D(filters=32,kernel_size=3,strides=3,activation='relu'))
         model.add(MaxPooling2D(pool_size=(2,2)))
         model.add(Dropout(0.5))
```

```
        model.add(Flatten())
        model.add(Dense(133))
        model.add(Activation('relu'))
        model.add(Dropout(0.2))
        model.add(Dense(133))
        model.add(Activation('softmax'))

        model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_9 (Conv2D)            (None, 75, 75, 32)        896
_________________________________________________________________
max_pooling2d_10 (MaxPooling (None, 37, 37, 32)        0
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 12, 12, 32)        9248
_________________________________________________________________
max_pooling2d_11 (MaxPooling (None, 6, 6, 32)          0
_________________________________________________________________
dropout_9 (Dropout)          (None, 6, 6, 32)          0
_________________________________________________________________
flatten_6 (Flatten)          (None, 1152)              0
_________________________________________________________________
dense_9 (Dense)              (None, 133)               153349
_________________________________________________________________
activation_58 (Activation)   (None, 133)               0
_________________________________________________________________
dropout_10 (Dropout)         (None, 133)               0
_________________________________________________________________
dense_10 (Dense)             (None, 133)               17822
_________________________________________________________________
activation_59 (Activation)   (None, 133)               0
=================================================================
Total params: 181,315.0
Trainable params: 181,315.0
Non-trainable params: 0.0
_________________________________________________________________
```

1.2.11 Compile the Model

```
In [19]: model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy
```

1.2.12 (IMPLEMENTATION) Train the Model

Train your model in the code cell below. Use model checkpointing to save the model that attains the best validation loss.

You are welcome to augment the training data, but this is not a requirement.

```
In [20]: from keras.callbacks import ModelCheckpoint

         ### TODO: specify the number of epochs that you would like to use to train the model.

         epochs = 20

         ### Do NOT modify the code below this line.

         checkpointer = ModelCheckpoint(filepath='saved_models/weights.best.from_scratch.hdf5'
                                        verbose=1, save_best_only=True)

         model.fit(train_tensors, train_targets,
                   validation_data=(valid_tensors, valid_targets),
                   epochs=epochs, batch_size=20, callbacks=[checkpointer], verbose=1)

Train on 6680 samples, validate on 835 samples
Epoch 1/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.8825 - acc: 0.0120Epoch 00000:
6680/6680 [==============================] - 31s - loss: 4.8823 - acc: 0.0120 - val_loss: 4.845
Epoch 2/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.7383 - acc: 0.0261Epoch 00001:
6680/6680 [==============================] - 30s - loss: 4.7381 - acc: 0.0260 - val_loss: 4.562
Epoch 3/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.4712 - acc: 0.0443Epoch 00002:
6680/6680 [==============================] - 30s - loss: 4.4706 - acc: 0.0443 - val_loss: 4.328
Epoch 4/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.2883 - acc: 0.0565Epoch 00003:
6680/6680 [==============================] - 30s - loss: 4.2880 - acc: 0.0564 - val_loss: 4.207
Epoch 5/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.1498 - acc: 0.0728Epoch 00004:
6680/6680 [==============================] - 30s - loss: 4.1502 - acc: 0.0728 - val_loss: 4.168
Epoch 6/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.0339 - acc: 0.0850Epoch 00005:
6680/6680 [==============================] - 30s - loss: 4.0355 - acc: 0.0850 - val_loss: 4.057
Epoch 7/20
6660/6680 [============================>.] - ETA: 0s - loss: 3.9507 - acc: 0.0949Epoch 00006:
6680/6680 [==============================] - 31s - loss: 3.9507 - acc: 0.0949 - val_loss: 4.011
Epoch 8/20
6660/6680 [============================>.] - ETA: 0s - loss: 3.8700 - acc: 0.1135Epoch 00007:
6680/6680 [==============================] - 30s - loss: 3.8703 - acc: 0.1135 - val_loss: 4.045
Epoch 9/20
6660/6680 [============================>.] - ETA: 0s - loss: 3.7987 - acc: 0.1234Epoch 00008:
6680/6680 [==============================] - 30s - loss: 3.8007 - acc: 0.1232 - val_loss: 3.979
Epoch 10/20
6660/6680 [============================>.] - ETA: 0s - loss: 3.7084 - acc: 0.1402Epoch 00009:
6680/6680 [==============================] - 31s - loss: 3.7089 - acc: 0.1401 - val_loss: 3.924
```

```
Epoch 11/20
6660/6680 [============================>.] - ETA: 0s - loss: 3.6592 - acc: 0.1449Epoch 00010:
6680/6680 [==============================] - 30s - loss: 3.6613 - acc: 0.1448 - val_loss: 3.909
Epoch 12/20
6660/6680 [============================>.] - ETA: 0s - loss: 3.5930 - acc: 0.1560Epoch 00011:
6680/6680 [==============================] - 30s - loss: 3.5925 - acc: 0.1564 - val_loss: 3.894
Epoch 13/20
6660/6680 [============================>.] - ETA: 0s - loss: 3.5402 - acc: 0.1745Epoch 00012:
6680/6680 [==============================] - 30s - loss: 3.5396 - acc: 0.1746 - val_loss: 3.904
Epoch 14/20
6660/6680 [============================>.] - ETA: 0s - loss: 3.4776 - acc: 0.1749Epoch 00013:
6680/6680 [==============================] - 30s - loss: 3.4781 - acc: 0.1751 - val_loss: 3.900
Epoch 15/20
6660/6680 [============================>.] - ETA: 0s - loss: 3.4386 - acc: 0.1880Epoch 00014:
6680/6680 [==============================] - 30s - loss: 3.4389 - acc: 0.1880 - val_loss: 3.932
Epoch 16/20
6660/6680 [============================>.] - ETA: 0s - loss: 3.3965 - acc: 0.1878Epoch 00015:
6680/6680 [==============================] - 30s - loss: 3.3943 - acc: 0.1882 - val_loss: 3.924
Epoch 17/20
6660/6680 [============================>.] - ETA: 0s - loss: 3.3575 - acc: 0.2041Epoch 00016:
6680/6680 [==============================] - 30s - loss: 3.3561 - acc: 0.2042 - val_loss: 3.923
Epoch 18/20
6660/6680 [============================>.] - ETA: 0s - loss: 3.2833 - acc: 0.2075Epoch 00017:
6680/6680 [==============================] - 30s - loss: 3.2860 - acc: 0.2072 - val_loss: 3.874
Epoch 19/20
6660/6680 [============================>.] - ETA: 0s - loss: 3.2483 - acc: 0.2144Epoch 00018:
6680/6680 [==============================] - 30s - loss: 3.2485 - acc: 0.2142 - val_loss: 3.950
Epoch 20/20
6660/6680 [============================>.] - ETA: 0s - loss: 3.1841 - acc: 0.2335Epoch 00019:
6680/6680 [==============================] - 30s - loss: 3.1850 - acc: 0.2334 - val_loss: 3.914
```

```
Out[20]: <keras.callbacks.History at 0x7fa28d808f98>
```

1.2.13 Load the Model with the Best Validation Loss

```
In [21]: model.load_weights('saved_models/weights.best.from_scratch.hdf5')
```

1.2.14 Test the Model

Try out your model on the test dataset of dog images. Ensure that your test accuracy is greater than 1%.

```
In [22]: # get index of predicted dog breed for each image in test set
         dog_breed_predictions = [np.argmax(model.predict(np.expand_dims(tensor, axis=0))) for

         # report test accuracy
         test_accuracy = 100*np.sum(np.array(dog_breed_predictions)==np.argmax(test_targets, a
         print('Test accuracy: %.4f%%' % test_accuracy)
```

```
Test accuracy: 10.6459%
```

Step 4: Use a CNN to Classify Dog Breeds

To reduce training time without sacrificing accuracy, we show you how to train a CNN using transfer learning. In the following step, you will get a chance to use transfer learning to train your own CNN.

1.2.15 Obtain Bottleneck Features

```
In [23]: bottleneck_features = np.load('bottleneck_features/DogVGG16Data.npz')
         train_VGG16 = bottleneck_features['train']
         valid_VGG16 = bottleneck_features['valid']
         test_VGG16 = bottleneck_features['test']
```

1.2.16 Model Architecture

The model uses the the pre-trained VGG-16 model as a fixed feature extractor, where the last convolutional output of VGG-16 is fed as input to our model. We only add a global average pooling layer and a fully connected layer, where the latter contains one node for each dog category and is equipped with a softmax.

```
In [24]: VGG16_model = Sequential()
         VGG16_model.add(GlobalAveragePooling2D(input_shape=train_VGG16.shape[1:]))
         VGG16_model.add(Dense(133, activation='softmax'))

         VGG16_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
global_average_pooling2d_1 ( (None, 512)               0
_________________________________________________________________
dense_11 (Dense)             (None, 133)               68229
=================================================================
Total params: 68,229.0
Trainable params: 68,229.0
Non-trainable params: 0.0
_________________________________________________________________
```

1.2.17 Compile the Model

```
In [25]: VGG16_model.compile(loss='categorical_crossentropy', optimizer='rmsprop', metrics=['a
```

1.2.18 Train the Model

```
In [27]: checkpointer = ModelCheckpoint(filepath='saved_models/weights.best.VGG16.hdf5',
                                        verbose=1, save_best_only=True)

         VGG16_model.fit(train_VGG16, train_targets,
                   validation_data=(valid_VGG16, valid_targets),
                   epochs=20, batch_size=20, callbacks=[checkpointer], verbose=1)
```

```
Train on 6680 samples, validate on 835 samples
Epoch 1/20
6440/6680 [===========================>..] - ETA: 0s - loss: 8.0786 - acc: 0.4943Epoch 00000: v
6680/6680 [==============================] - 1s - loss: 8.1009 - acc: 0.4927 - val_loss: 8.9733
Epoch 2/20
6620/6680 [===========================>.] - ETA: 0s - loss: 8.0690 - acc: 0.4962Epoch 00001: v
6680/6680 [==============================] - 1s - loss: 8.0689 - acc: 0.4963 - val_loss: 8.9761
Epoch 3/20
6480/6680 [============================>.] - ETA: 0s - loss: 8.0322 - acc: 0.4972Epoch 00002: v
6680/6680 [==============================] - 1s - loss: 8.0209 - acc: 0.4981 - val_loss: 8.9712
Epoch 4/20
6600/6680 [============================>.] - ETA: 0s - loss: 7.9944 - acc: 0.5023Epoch 00003: v
6680/6680 [==============================] - 1s - loss: 8.0025 - acc: 0.5018 - val_loss: 8.9626
Epoch 5/20
6620/6680 [============================>.] - ETA: 0s - loss: 7.9809 - acc: 0.5024Epoch 00004: v
6680/6680 [==============================] - 1s - loss: 7.9840 - acc: 0.5022 - val_loss: 8.9041
Epoch 6/20
6460/6680 [============================>.] - ETA: 0s - loss: 7.8789 - acc: 0.5025Epoch 00005: v
6680/6680 [==============================] - 1s - loss: 7.8732 - acc: 0.5028 - val_loss: 8.8384
Epoch 7/20
6440/6680 [===========================>..] - ETA: 0s - loss: 7.8011 - acc: 0.5110Epoch 00006: v
6680/6680 [==============================] - 1s - loss: 7.8059 - acc: 0.5108 - val_loss: 8.7524
Epoch 8/20
6440/6680 [===========================>..] - ETA: 0s - loss: 7.7902 - acc: 0.5130Epoch 00007: v
6680/6680 [==============================] - 1s - loss: 7.7834 - acc: 0.5135 - val_loss: 8.7781
Epoch 9/20
6660/6680 [============================>.] - ETA: 0s - loss: 7.7738 - acc: 0.5156Epoch 00008: v
6680/6680 [==============================] - 1s - loss: 7.7773 - acc: 0.5153 - val_loss: 8.7595
Epoch 10/20
6540/6680 [============================>.] - ETA: 0s - loss: 7.7917 - acc: 0.5147Epoch 00009: v
6680/6680 [==============================] - 1s - loss: 7.7684 - acc: 0.5162 - val_loss: 8.7597
Epoch 11/20
6460/6680 [============================>.] - ETA: 0s - loss: 7.7816 - acc: 0.5139Epoch 00010: v
6680/6680 [==============================] - 1s - loss: 7.7569 - acc: 0.5156 - val_loss: 8.7455
Epoch 12/20
6600/6680 [============================>.] - ETA: 0s - loss: 7.6322 - acc: 0.5209Epoch 00011: v
6680/6680 [==============================] - 1s - loss: 7.6495 - acc: 0.5199 - val_loss: 8.6440
Epoch 13/20
6440/6680 [===========================>..] - ETA: 0s - loss: 7.5842 - acc: 0.5231Epoch 00012: v
6680/6680 [==============================] - 1s - loss: 7.5674 - acc: 0.5241 - val_loss: 8.6027
```

```
Epoch 14/20
6520/6680 [============================>.] - ETA: 0s - loss: 7.5241 - acc: 0.5282Epoch 00013: v
6680/6680 [==============================] - 1s - loss: 7.5282 - acc: 0.5278 - val_loss: 8.5876
Epoch 15/20
6620/6680 [============================>.] - ETA: 0s - loss: 7.5070 - acc: 0.5311Epoch 00014: v
6680/6680 [==============================] - 1s - loss: 7.5052 - acc: 0.5311 - val_loss: 8.5671
Epoch 16/20
6440/6680 [===========================>..] - ETA: 0s - loss: 7.4578 - acc: 0.5315Epoch 00015: v
6680/6680 [==============================] - 1s - loss: 7.4421 - acc: 0.5323 - val_loss: 8.5021
Epoch 17/20
6640/6680 [============================>.] - ETA: 0s - loss: 7.3918 - acc: 0.5384Epoch 00016: v
6680/6680 [==============================] - 1s - loss: 7.3861 - acc: 0.5388 - val_loss: 8.4230
Epoch 18/20
6640/6680 [============================>.] - ETA: 0s - loss: 7.3718 - acc: 0.5407Epoch 00017: v
6680/6680 [==============================] - 1s - loss: 7.3759 - acc: 0.5404 - val_loss: 8.4197
Epoch 19/20
6460/6680 [============================>.] - ETA: 0s - loss: 7.3611 - acc: 0.5395Epoch 00018: v
6680/6680 [==============================] - 1s - loss: 7.3533 - acc: 0.5398 - val_loss: 8.3203
Epoch 20/20
6620/6680 [============================>.] - ETA: 0s - loss: 7.2884 - acc: 0.5429Epoch 00019: v
6680/6680 [==============================] - 1s - loss: 7.2833 - acc: 0.5433 - val_loss: 8.4272
```

```
Out[27]: <keras.callbacks.History at 0x7fa2d14e20b8>
```

1.2.19 Load the Model with the Best Validation Loss

```
In [28]: VGG16_model.load_weights('saved_models/weights.best.VGG16.hdf5')
```

1.2.20 Test the Model

Now, we can use the CNN to test how well it identifies breed within our test dataset of dog images.
We print the test accuracy below.

```
In [29]: # get index of predicted dog breed for each image in test set
         VGG16_predictions = [np.argmax(VGG16_model.predict(np.expand_dims(feature, axis=0))) i

         # report test accuracy
         test_accuracy = 100*np.sum(np.array(VGG16_predictions)==np.argmax(test_targets, axis=1
         print('Test accuracy: %.4f%%' % test_accuracy)
```

```
Test accuracy: 43.7799%
```

1.2.21 Predict Dog Breed with the Model

```
In [30]: from extract_bottleneck_features import *

         def VGG16_predict_breed(img_path):
```

```
    # extract bottleneck features
    bottleneck_feature = extract_VGG16(path_to_tensor(img_path))
    # obtain predicted vector
    predicted_vector = VGG16_model.predict(bottleneck_feature)
    # return dog breed that is predicted by the model
    return dog_names[np.argmax(predicted_vector)]
```

```
In [49]: # Predict first 100 images of training set
         correct_predictions = [train_files[ix] for ix in range(len(train_files[:100])) if VGG

         print("Percentage of dog images correctly predicted (First 100 Training Set): %d%%" %
              len(correct_predictions))
```

```
Percentage of dog images correctly predicted (First 100 Training Set): 70%
```

Step 5: Create a CNN to Classify Dog Breeds (using Transfer Learning)

You will now use transfer learning to create a CNN that can identify dog breed from images. Your CNN must attain at least 60% accuracy on the test set.

In Step 4, we used transfer learning to create a CNN using VGG-16 bottleneck features. In this section, you must use the bottleneck features from a different pre-trained model. To make things easier for you, we have pre-computed the features for all of the networks that are currently available in Keras: - VGG-19 bottleneck features - ResNet-50 bottleneck features - Inception bottleneck features - Xception bottleneck features

The files are encoded as such:

```
Dog{network}Data.npz
```

where `{network}`, in the above filename, can be one of `VGG19`, `Resnet50`, `InceptionV3`, or `Xception`. Pick one of the above architectures, download the corresponding bottleneck features, and store the downloaded file in the `bottleneck_features/` folder in the repository.

1.2.22 (IMPLEMENTATION) Obtain Bottleneck Features

In the code block below, extract the bottleneck features corresponding to the train, test, and validation sets by running the following:

```
bottleneck_features = np.load('bottleneck_features/Dog{network}Data.npz')
train_{network} = bottleneck_features['train']
valid_{network} = bottleneck_features['valid']
test_{network} = bottleneck_features['test']
```

```
In [51]: ### TODO: Obtain bottleneck features from another pre-trained CNN.

         bottleneck_features = np.load('bottleneck_features/DogResnet50Data.npz')
         train_ResNet50 = bottleneck_features['train']
         valid_ResNet50 = bottleneck_features['valid']
         test_ResNet50 = bottleneck_features['test']
```

1.2.23 (IMPLEMENTATION) Model Architecture

Create a CNN to classify dog breed. At the end of your code cell block, summarize the layers of your model by executing the line:

```
<your model's name>.summary()
```

Question 5: Outline the steps you took to get to your final CNN architecture and your reasoning at each step. Describe why you think the architecture is suitable for the current problem.

Answer: After observing the impresive transfer learning results in the last example, I decided to base my architecture off of it. After comparing the four bottlenecks I chose to adopt ResNet50 due to it's speed and accuracy (https://www.pyimagesearch.com/2017/03/20/imagenet-vggnet-resnet-inception-xception-keras/). After applying a global average pooling layer and a dense layer the accuracy jumped to 84%. After experimenting with a few activation functions softmax provided the greatest accuracy

In [52]:
```
### TODO: Define your architecture.

        ResNet50_model = Sequential()
        ResNet50_model.add(GlobalAveragePooling2D(input_shape=train_ResNet50.shape[1:]))
        ResNet50_model.add(Dense(133, activation='softmax'))

        ResNet50_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
global_average_pooling2d_2 ( (None, 2048)              0
_________________________________________________________________
dense_12 (Dense)             (None, 133)               272517
=================================================================
Total params: 272,517.0
Trainable params: 272,517.0
Non-trainable params: 0.0
_________________________________________________________________
```

1.2.24 (IMPLEMENTATION) Compile the Model

In [53]:
```
### TODO: Compile the model.

        ResNet50_model.compile(loss='categorical_crossentropy', optimizer='rmsprop', metrics=
```

1.2.25 (IMPLEMENTATION) Train the Model

Train your model in the code cell below. Use model checkpointing to save the model that attains the best validation loss.

You are welcome to augment the training data, but this is not a requirement.

In [54]:
```
### TODO: Train the model.

checkpointer = ModelCheckpoint(filepath='saved_models/weights.best.ResNet50.hdf5',
                               verbose=1, save_best_only=True)

ResNet50_model.fit(train_ResNet50, train_targets,
          validation_data=(valid_ResNet50, valid_targets),
          epochs=20, batch_size=20, callbacks=[checkpointer], verbose=1)
```

```
Train on 6680 samples, validate on 835 samples
Epoch 1/20
6640/6680 [============================>.] - ETA: 0s - loss: 1.6164 - acc: 0.6023Epoch 00000:
6680/6680 [==============================] - 3s - loss: 1.6129 - acc: 0.6027 - val_loss: 0.8338
Epoch 2/20
6560/6680 [============================>.] - ETA: 0s - loss: 0.4347 - acc: 0.8675Epoch 00001:
6680/6680 [==============================] - 1s - loss: 0.4353 - acc: 0.8669 - val_loss: 0.6734
Epoch 3/20
6600/6680 [============================>.] - ETA: 0s - loss: 0.2643 - acc: 0.9152Epoch 00002:
6680/6680 [==============================] - 1s - loss: 0.2638 - acc: 0.9154 - val_loss: 0.6800
Epoch 4/20
6540/6680 [============================>.] - ETA: 0s - loss: 0.1705 - acc: 0.9456Epoch 00003:
6680/6680 [==============================] - 1s - loss: 0.1689 - acc: 0.9463 - val_loss: 0.6974
Epoch 5/20
6540/6680 [============================>.] - ETA: 0s - loss: 0.1229 - acc: 0.9641Epoch 00004:
6680/6680 [==============================] - 1s - loss: 0.1254 - acc: 0.9632 - val_loss: 0.7066
Epoch 6/20
6460/6680 [============================>.] - ETA: 0s - loss: 0.0872 - acc: 0.9724Epoch 00005:
6680/6680 [==============================] - 1s - loss: 0.0874 - acc: 0.9726 - val_loss: 0.7358
Epoch 7/20
6560/6680 [============================>.] - ETA: 0s - loss: 0.0611 - acc: 0.9809Epoch 00006:
6680/6680 [==============================] - 1s - loss: 0.0617 - acc: 0.9807 - val_loss: 0.6926
Epoch 8/20
6660/6680 [============================>.] - ETA: 0s - loss: 0.0473 - acc: 0.9853Epoch 00007:
6680/6680 [==============================] - 1s - loss: 0.0472 - acc: 0.9853 - val_loss: 0.7102
Epoch 9/20
6520/6680 [============================>.] - ETA: 0s - loss: 0.0336 - acc: 0.9914Epoch 00008:
6680/6680 [==============================] - 1s - loss: 0.0331 - acc: 0.9916 - val_loss: 0.7158
Epoch 10/20
6480/6680 [============================>.] - ETA: 0s - loss: 0.0252 - acc: 0.9934Epoch 00009:
6680/6680 [==============================] - 1s - loss: 0.0254 - acc: 0.9933 - val_loss: 0.7109
Epoch 11/20
6600/6680 [============================>.] - ETA: 0s - loss: 0.0201 - acc: 0.9947Epoch 00010:
6680/6680 [==============================] - 1s - loss: 0.0200 - acc: 0.9948 - val_loss: 0.7741
Epoch 12/20
6480/6680 [============================>.] - ETA: 0s - loss: 0.0148 - acc: 0.9965Epoch 00011:
6680/6680 [==============================] - 1s - loss: 0.0161 - acc: 0.9961 - val_loss: 0.8243
Epoch 13/20
6600/6680 [============================>.] - ETA: 0s - loss: 0.0137 - acc: 0.9959Epoch 00012:
```

```
6680/6680 [==============================] - 1s - loss: 0.0136 - acc: 0.9960 - val_loss: 0.8448
Epoch 14/20
6660/6680 [==============================>.] - ETA: 0s - loss: 0.0108 - acc: 0.9973Epoch 00013: v
6680/6680 [==============================] - 1s - loss: 0.0108 - acc: 0.9973 - val_loss: 0.8606
Epoch 15/20
6540/6680 [==============================>.] - ETA: 0s - loss: 0.0099 - acc: 0.9977Epoch 00014: v
6680/6680 [==============================] - 1s - loss: 0.0098 - acc: 0.9978 - val_loss: 0.8278
Epoch 16/20
6560/6680 [==============================>.] - ETA: 0s - loss: 0.0082 - acc: 0.9983Epoch 00015: v
6680/6680 [==============================] - 1s - loss: 0.0092 - acc: 0.9982 - val_loss: 0.8564
Epoch 17/20
6480/6680 [==============================>.] - ETA: 0s - loss: 0.0073 - acc: 0.9980Epoch 00016: v
6680/6680 [==============================] - 1s - loss: 0.0072 - acc: 0.9981 - val_loss: 0.8879
Epoch 18/20
6520/6680 [==============================>.] - ETA: 0s - loss: 0.0061 - acc: 0.9980Epoch 00017: v
6680/6680 [==============================] - 1s - loss: 0.0062 - acc: 0.9979 - val_loss: 0.9006
Epoch 19/20
6540/6680 [==============================>.] - ETA: 0s - loss: 0.0057 - acc: 0.9977Epoch 00018: v
6680/6680 [==============================] - 1s - loss: 0.0056 - acc: 0.9978 - val_loss: 0.8836
Epoch 20/20
6660/6680 [==============================>.] - ETA: 0s - loss: 0.0062 - acc: 0.9982Epoch 00019: v
6680/6680 [==============================] - 1s - loss: 0.0062 - acc: 0.9982 - val_loss: 0.9359
```

```
Out[54]: <keras.callbacks.History at 0x7f9f7d4e0c18>
```

1.2.26 (IMPLEMENTATION) Load the Model with the Best Validation Loss

```
In [55]: ### TODO: Load the model weights with the best validation loss.

         ResNet50_model.load_weights('saved_models/weights.best.ResNet50.hdf5')
```

1.2.27 (IMPLEMENTATION) Test the Model

Try out your model on the test dataset of dog images. Ensure that your test accuracy is greater than 60%.

```
In [56]: ### TODO: Calculate classification accuracy on the test dataset.

         # get index of predicted dog breed for each image in test set
         ResNet50_predictions = [np.argmax(ResNet50_model.predict(np.expand_dims(feature, axis=

         # report test accuracy
         test_accuracy = 100*np.sum(np.array(ResNet50_predictions)==np.argmax(test_targets, axi
         print('Test accuracy: %.4f%%' % test_accuracy)
```

```
Test accuracy: 80.2632%
```

1.2.28 (IMPLEMENTATION) Predict Dog Breed with the Model

Write a function that takes an image path as input and returns the dog breed (`Affenpinscher`, `Afghan_hound`, etc) that is predicted by your model.

Similar to the analogous function in Step 5, your function should have three steps: 1. Extract the bottleneck features corresponding to the chosen CNN model. 2. Supply the bottleneck features as input to the model to return the predicted vector. Note that the argmax of this prediction vector gives the index of the predicted dog breed. 3. Use the `dog_names` array defined in Step 0 of this notebook to return the corresponding breed.

The functions to extract the bottleneck features can be found in `extract_bottleneck_features.py`, and they have been imported in an earlier code cell. To obtain the bottleneck features corresponding to your chosen CNN architecture, you need to use the function

`extract_{network}`

where `{network}`, in the above filename, should be one of `VGG19`, `Resnet50`, `InceptionV3`, or `Xception`.

In [59]:
```
### TODO: Write a function that takes a path to an image as input
### and returns the dog breed that is predicted by the model.

def Resnet50_predict_breed(img_path):
    # extract bottleneck features
    bottleneck_feature = extract_Resnet50(path_to_tensor(img_path))
    # obtain predicted vector
    predicted_vector = ResNet50_model.predict(bottleneck_feature)
    # return dog breed that is predicted by the model
    return dog_names[np.argmax(predicted_vector)]
```

In [122]:
```
# Predict first 10 images of training set
correct_predictions = [train_files[ix] for ix in range(len(train_files[:10])) if Resn

print("Percentage of dog images correctly predicted (First 10 Training Set): %d0%%" %
     len(correct_predictions))
```

```
Percentage of dog images correctly predicted (First 10 Training Set): 80%
```

Step 6: Write your Algorithm

Write an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then, - if a **dog** is detected in the image, return the predicted breed. - if a **human** is detected in the image, return the resembling dog breed. - if **neither** is detected in the image, provide output that indicates an error.

You are welcome to write your own functions for detecting humans and dogs in images, but feel free to use the `face_detector` and `dog_detector` functions developed above. You are **required** to use your CNN from Step 5 to predict dog breed.

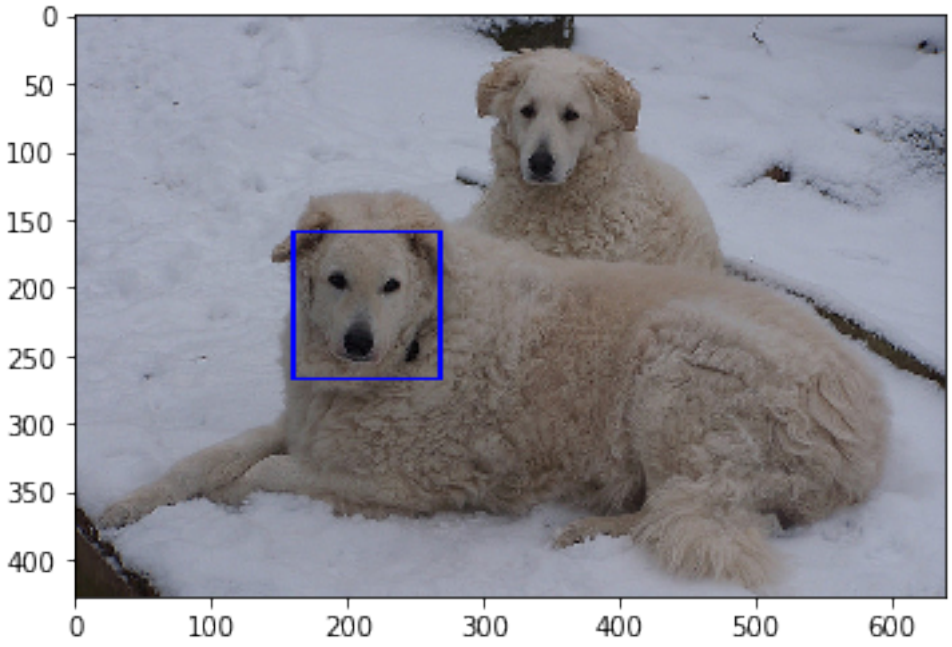
Some sample output for our algorithm is provided below, but feel free to design your own user experience!

1.2.29 (IMPLEMENTATION) Write your Algorithm

In [110]:
```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.
def custom_detection(img_path):
    # extract pre-trained face detector
    face_cascade = cv2.CascadeClassifier('haarcascades/haarcascade_frontalface_alt.xm

    # load color (BGR) image
    img = cv2.imread(img_path)
    # convert BGR image to grayscale
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # find faces in image
    faces = face_cascade.detectMultiScale(gray)

    # print number of faces detected in the image
    face_present = face_detector(img_path)
    try:
        dog_breed = Resnet50_predict_breed(img_path)
    except:
        dog_breed = None

    print('Face Present:', face_present)

    # get bounding box for each detected face
    for (x,y,w,h) in faces:
        # add bounding box to color image
        cv2.rectangle(img,(x,y),(x+w,y+h),(255,0,0),2)

    # convert BGR image to RGB for plotting
    cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

    # display the image, along with bounding box
    plt.imshow(cv_rgb)
    plt.show()

    # Print dog breed name
    if dog_breed:
        print("Dog breed doppleganger is: ", dog_breed)
    else:
        print("Unfortunately, this image did not have a dogbreed doppleganger")
```

In [69]:
```
custom_detection(train_files[0])
```

```
Face Present: True
```




```
Dog breed doppleganger is:  Kuvasz
```

Step 7: Test Your Algorithm

In this section, you will take your new algorithm for a spin! What kind of dog does the algorithm think that **you** look like? If you have a dog, does it predict your dog's breed accurately? If you have a cat, does it mistakenly think that your cat is a dog?

1.2.30 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

Test your algorithm at least six images on your computer. Feel free to use any images you like. Use at least two human and two dog images.

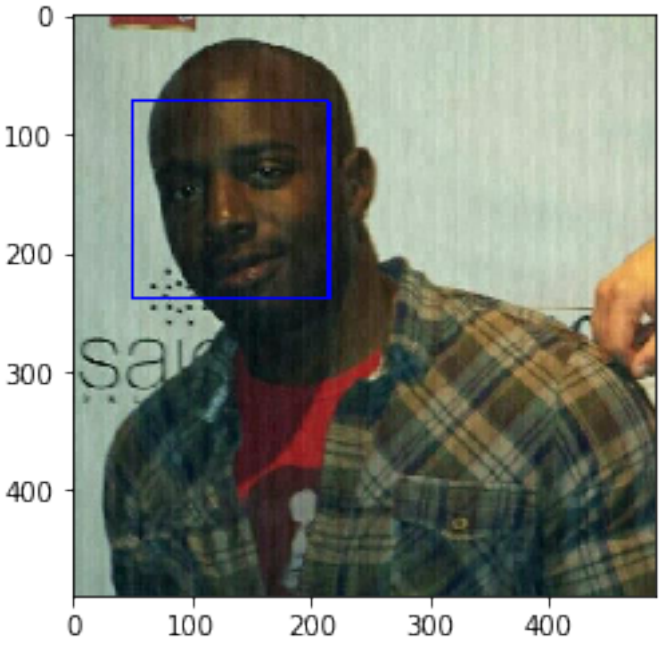
Question 6: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

Answer:

```
In [ ]: ## TODO: Execute your algorithm from Step 6 on
        ## at least 6 images on your computer.
        ## Feel free to use as many code cells as needed.

In [108]: # Example 1 -- Me :-D
          import urllib
```

```
        urllib.request.urlretrieve("https://pbs.twimg.com/profile_images/608416574417039362/1
                                   filename="ronaldeddings.jpg")
        custom_detection("ronaldeddings.jpg")
```

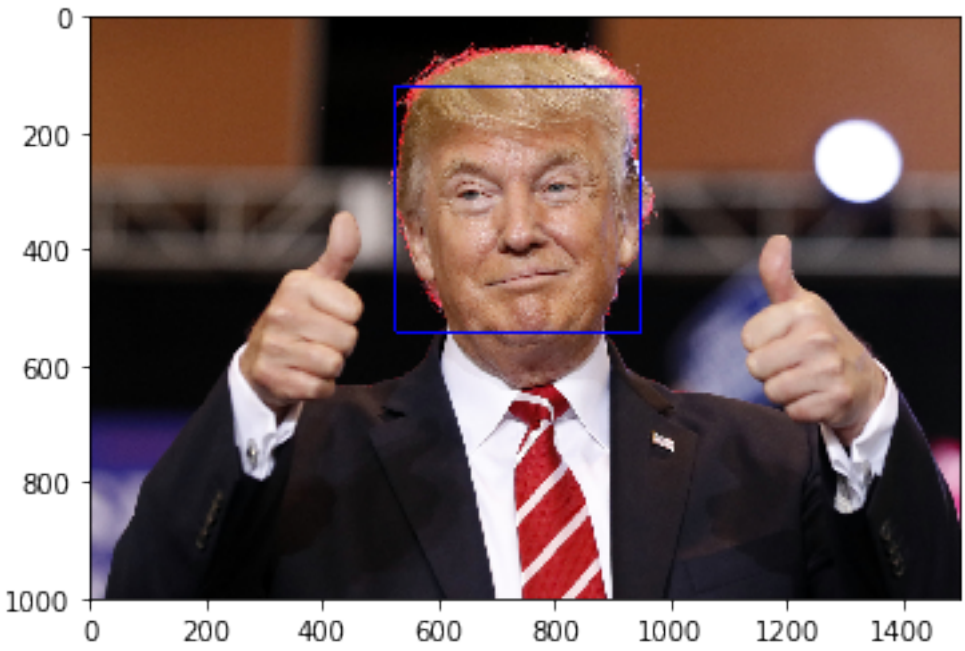
```
Face Present: True
```




```
Dog breed doppleganger is:  Dogue_de_bordeaux
```

```
In [109]: # Example 2 -- President Donald Trump

        urllib.request.urlretrieve("http://d279m997dpfwgl.cloudfront.net/wp/2017/08/trumpthu
                                   filename="donaldtrump.jpg")
        custom_detection("donaldtrump.jpg")
```

```
Face Present: True
```




```
Dog breed doppleganger is:  Boston_terrier
```

```
In [112]: # Example 3 -- Grumpy cat

          urllib.request.urlretrieve("https://www.grumpycats.com/images/about/tardar.jpg",
                                     filename="grumpy_cat.jpg")
          custom_detection("grumpy_cat.jpg")
```

```
Face Present: False
```


```
Dog breed doppleganger is:  Japanese_chin
```

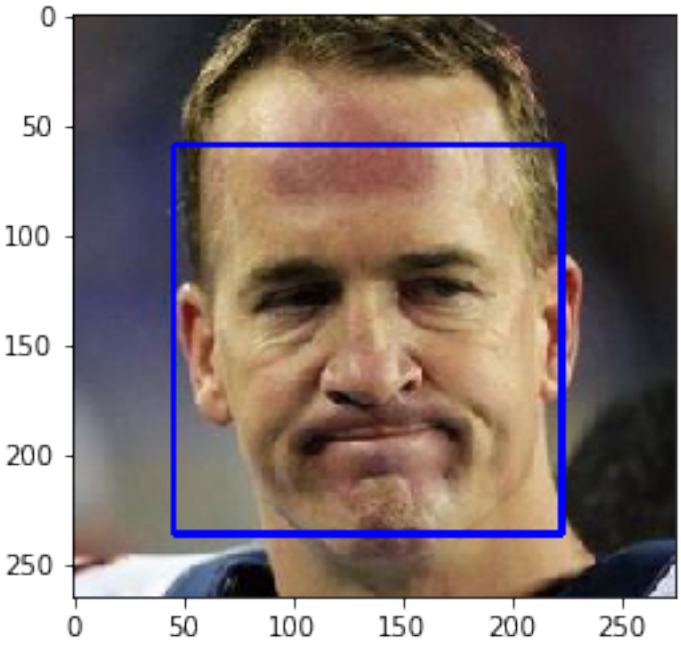
```
In [114]: # Example 4 -- Picture of a dog that looks like Payton Manning
          urllib.request.urlretrieve("https://i.pinimg.com/originals/88/c5/fa/88c5fa6f3d5375f3
                                     filename="payton_dog.jpg")
          custom_detection("payton_dog.jpg")
```

```
Face Present: False
```


```
Dog breed doppleganger is:  Dogue_de_bordeaux
```

```
In [115]: # Example 5 -- Picture of Payton Manning
          urllib.request.urlretrieve("https://pbs.twimg.com/media/B0WsDlVIEAA6x8V.jpg",
                                     filename="payton_manning.jpg")
          custom_detection("payton_manning.jpg")
```

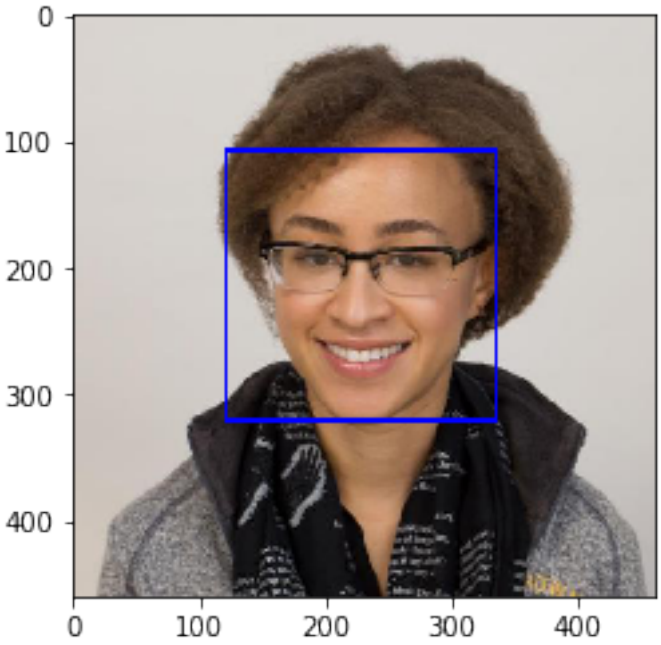
```
Face Present: True
```




```
Dog breed doppleganger is:  Boston_terrier
```

```
In [116]: # Example 5 -- Picture of Alexis Cook from Udacity -- Thanks for the lesson Alexis!
          urllib.request.urlretrieve("https://avatars3.githubusercontent.com/u/10624937?s=460&v
                                     filename="alexis_cook.jpg")
          custom_detection("alexis_cook.jpg")
```

```
Face Present: True
```




```
Dog breed doppleganger is:  Xoloitzcuintli
```

In [ ]:

In [ ]: