Data Scientist Nanodegree

May 9, 2019

1 Project: Write an Algorithm for a Dog Identification App

This notebook walks you through one of the most popular Udacity projects across machine learning and artificial intellegence nanodegree programs. The goal is to classify images of dogs according to their breed.

If you are looking for a more guided capstone project related to deep learning and convolutional neural networks, this might be just it. Notice that even if you follow the notebook to creating your classifier, you must still create a blog post or deploy an application to fulfill the requirements of the capstone project.

Also notice, you may be able to use only parts of this notebook (for example certain coding portions or the data) without completing all parts and still meet all requirements of the capstone project.

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.

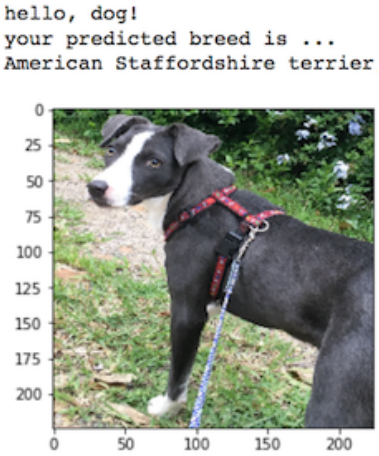
In addition to implementing code, there will be questions that you must answer which relate to the project and your implementation. Each section where you will answer a question is preceded by a **'Question X'** header. Carefully read each question and provide thorough answers in the following text boxes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

Why We're Here

In this notebook, you will make the first steps towards developing an algorithm that could be used as part of a mobile or web app. At the end of this project, your code will accept any user-supplied image as input. If a dog is detected in the image, it will provide an estimate of the dog's breed. If a human is detected, it will provide an estimate of the dog breed that is most resembling. The image below displays potential sample output of your finished project (... but we expect that each student's algorithm will behave differently!).



In this real-world setting, you will need to piece together a series of models to perform different tasks; for instance, the algorithm that detects humans in an image will be different from the CNN that infers dog breed. There are many points of possible failure, and no perfect algorithm exists. Your imperfect solution will nonetheless create a fun user experience!

Step 0: Import Datasets

1.0.1 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.
```

1.0.2 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.
```

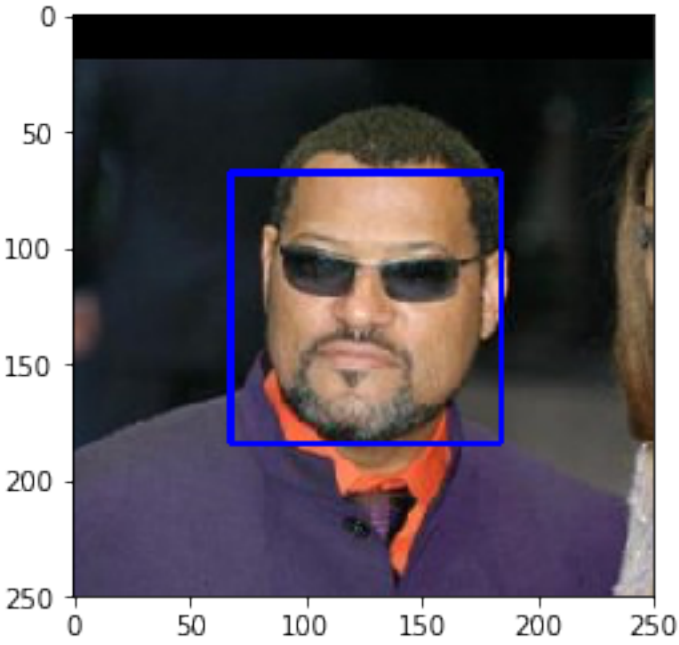
Step 1: Detect Humans

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 [3]:
```
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[3])
# 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.

1.0.3 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 [4]: # 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
```

1.0.4 (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?
- What percentage of the first 100 images in `dog_files` have a detected human face?

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 [5]: 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.
        humans=0
        dogs=0
        for n in range(100):
            if face_detector(human_files[n])==True:
                humans+=1
            if face_detector(dog_files_short[n])==True:
                dogs+=1

        humansperc=(humans*100)/100
        dogsperc=(dogs*100)/100

        print("The percentage of detected human faces, using the first 100 images, is:", str(hu
        print("The percentage of detected dog faces, using the first 100 images, is:", str(dogs
```

```
The percentage of detected human faces, using the first 100 images, is: 99.0 %
The percentage of detected dog faces, using the first 100 images, is: 12.0 %
```

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:

No, it is not a reasonable expectation. It would be good to impose the use of an algorithm in order to obtain its best performance, however, this is not the case for most scenarios. The easiest way to increase the robustness of the algorithm is by adjusting the parameters of the Haar Cascades face detection algorithm, especially the scaleFactor and the minNeighbors parameters. This procedure is presented next:

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.*

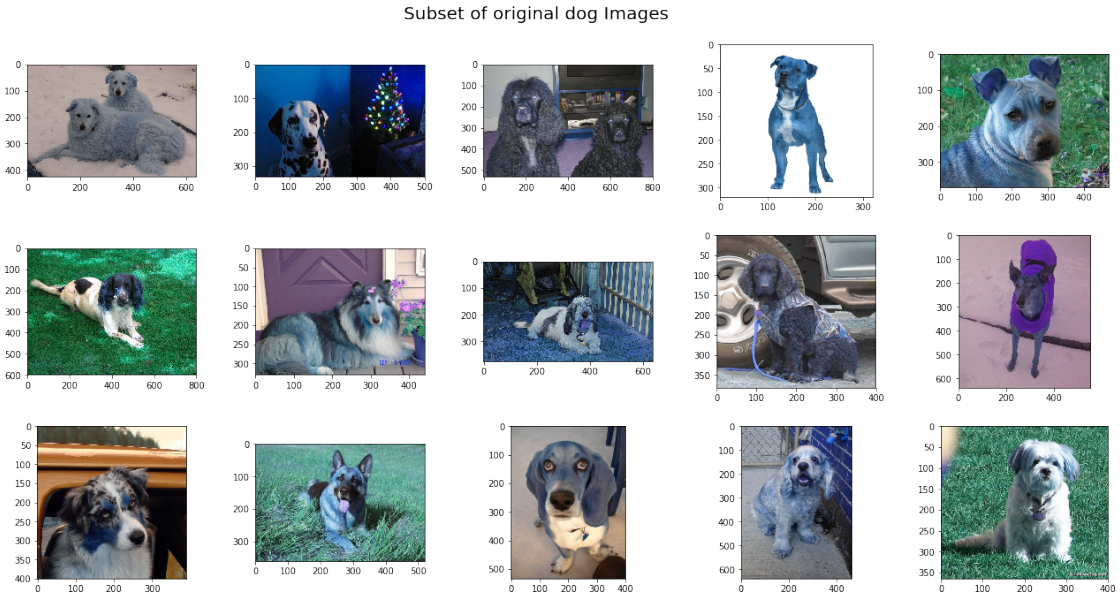
```
In [6]: #*****************************************Preparing the data**************************
        #Importing libraries
        import keras
```

```
from keras.preprocessing.image import ImageDataGenerator

#Creating an array to store images found in the human_files
dataH=np.ndarray(shape=(len(human_files),250,250,3), dtype='uint8')

#Storing images
for n in range(len(human_files)):
    img=cv2.imread(human_files[n])
    dataH[n,:,:,:] = img

#Creating an ImageDataGenerator, to create images with faces that are sligtly more dif
#the originals human images
datagen_reg = ImageDataGenerator(
    width_shift_range=0.35,  # randomly shift images horizontally (35% of total width)
    height_shift_range=0.35,  # randomly shift images vertically (35% of total height)
    rotation_range=20,    #Range to rotate the images
    horizontal_flip=True) # randomly flip images horizontally

#Creating an ImageDataGenerator, to create images with faces that are more difficult t
#the oroginals human images
datagen_bad = ImageDataGenerator(
    width_shift_range=0.4,  # randomly shift images horizontally (40% of total width)
    height_shift_range=0.4,  # randomly shift images vertically (40% of total height)
    rotation_range=25,    #Range to rotate the images
    horizontal_flip=True) # randomly flip images horizontally

#fit augmented images generators with the data
datagen_reg.fit(dataH)
datagen_bad.fit(dataH)

#Creating arrays to store the original and augmented images.
NumberOfImages = 100
x_train_norH = np.ndarray(shape=(NumberOfImages,250,250,3), dtype='uint8')  #Original
x_train_regH = np.ndarray(shape=(NumberOfImages,250,250,3), dtype='uint8')  #Moderate
x_train_badH = np.ndarray(shape=(NumberOfImages,250,250,3), dtype='uint8')  #Highly au

#Creating hard to detect faces images
#Only human images were augmented, since dogs images already have bad performances whe
for n in range(NumberOfImages):
        x_train_norH[n,:,:,:]=dataH[n,:,:,:]

for x_batch in datagen_reg.flow(dataH[NumberOfImages:NumberOfImages*2,:,:,:], batch_siz
    for n in range(NumberOfImages):
        x_train_regH[n,:,:,:]=x_batch[n]
    break;

for x_batch in datagen_bad.flow(dataH[NumberOfImages*2:,:,:,:], batch_size=NumberOfImag
    for n in range(NumberOfImages):
```

```
            x_train_badH[n,:,:,:]=x_batch[n]
        break;

    #Removing unnecessary data and printing an end indicator
    del dataH
    print("Done")
```

```
Done
```

In [7]:
```
# Visualizing a subset of original dog images
fig = plt.figure(figsize=(20,10))
for i in range(15):
    ax = fig.add_subplot(3, 5, i+1)
    img = cv2.imread(train_files[i])/255
    ax.imshow(img)
plt.subplots_adjust(top=0.92, bottom=0.08, left=0.10, right=0.95, hspace=0.25,
                    wspace=0.35)

fig.suptitle('Subset of original dog Images', fontsize=20)
plt.show()
```



In [8]:
```
#*********************************Plotting images used to test performances**********
#Importing libraries
import matplotlib.pyplot as plt

#Getting subsets of the originals and augmented images
x_train_subset_norH = x_train_norH[:15,:,:,:]/255
```

```
x_train_subset_regH = x_train_regH[:15,:,:,:]/255
x_train_subset_badH = x_train_badH[:15,:,:,:]/255

# visualize subset of original dog images
fig = plt.figure(figsize=(20,2))
for i in range(15):
    ax = fig.add_subplot(1, 15, i+1)
    img = cv2.imread(train_files[i])/255
    ax.imshow(img)
fig.suptitle('Subset of original dog Images', fontsize=20)
plt.show()

# visualize subset of original human images
fig = plt.figure(figsize=(20,2))
for i in range(0, len(x_train_subset_norH)):
    ax = fig.add_subplot(1, 15, i+1)
    ax.imshow(x_train_subset_norH[i])
fig.suptitle('Subset of original human images', fontsize=20)
plt.show()

# visualize subset of moderate augmented human images
fig = plt.figure(figsize=(20,2))
for x_batch in datagen_reg.flow(x_train_subset_regH, batch_size=15):
    for i in range(0, 15):
        ax = fig.add_subplot(1, 15, i+1)
        ax.imshow(x_batch[i])
    fig.suptitle('Subset of augmented human images, with a regular face detection quali
    plt.show()
    break;

# visualize subset of moderate highly augmented human images
fig = plt.figure(figsize=(20,2))
for x_batch in datagen_bad.flow(x_train_subset_badH, batch_size=15):
    for i in range(0, 15):
        ax = fig.add_subplot(1, 15, i+1)
        ax.imshow(x_batch[i])
    fig.suptitle('Subset of augmented human images, with a bad face detection quality'
    plt.show()
    break;

#Removing unnecessary data
del x_train_subset_norH, x_train_subset_regH, x_train_subset_badH, datagen_reg, datagen
```

Subset of original human images

Subset of augmented human images, with a regular face detection quality

Subset of augmented human images, with a bad face detection quality

In [9]:
```
#**************************Function to test the Haars face detection algorithm perfor

def testingperformance(nordata, regdata, baddata, dogdata, scaleFactor, minNeighbors):

    CountsD=np.zeros(shape=(2,1))
    CountsH=np.zeros(shape=(2,3))

    for n in range(100):
        img_nor = nordata[n,:,:,:]
        img_reg = regdata[n,:,:,:]
        img_bad = baddata[n,:,:,:]
        img_norD=cv2.imread(dogdata[n])
        gray_nor = cv2.cvtColor(img_nor, cv2.COLOR_BGR2GRAY)
        gray_reg = cv2.cvtColor(img_reg, cv2.COLOR_BGR2GRAY)
        gray_bad = cv2.cvtColor(img_bad, cv2.COLOR_BGR2GRAY)
        gray_norD = cv2.cvtColor(img_norD, cv2.COLOR_BGR2GRAY)

        #With dog not modified images
        if (len(face_cascade.detectMultiScale(gray_norD))>0)==True:
            CountsD[0,0]=CountsD[0,0]+1
        if (len(face_cascade.detectMultiScale(gray_norD,scaleFactor,minNeighbors))>0)==

            CountsD[1,0]=CountsD[1,0]+1

        #With human not modified images
```

```
            if (len(face_cascade.detectMultiScale(gray_nor))>0)==True:
                CountsH[0,0]=CountsH[0,0]+1
            if (len(face_cascade.detectMultiScale(gray_nor,scaleFactor,minNeighbors))>0)==
                CountsH[1,0]=CountsH[1,0]+1

            #With human regular modified images
            if (len(face_cascade.detectMultiScale(gray_reg))>0)==True:
                CountsH[0,1]=CountsH[0,1]+1
            if (len(face_cascade.detectMultiScale(gray_reg,scaleFactor,minNeighbors))>0)==
                CountsH[1,1]=CountsH[1,1]+1

            #With human bad modified images
            if (len(face_cascade.detectMultiScale(gray_bad))>0)==True:
                CountsH[0,2]=CountsH[0,2]+1
            if (len(face_cascade.detectMultiScale(gray_bad,scaleFactor,minNeighbors))>0)==
                CountsH[1,2]=CountsH[1,2]+1

        return CountsD, CountsH
```

In [10]:
```
#**********************Printing results, evaluating the performance of the Haars face

#Obtaining results
scaleFactor = 1.01
minNeighbors = 1
[CountsD,CountsH] = testingperformance(x_train_norH, x_train_regH, x_train_badH, trai

#Printing results
ObjectsList = ["Dogs", "Humans"]
for k in range(2):
    Temp='       '
    print("**********************************",ObjectsList[k],' images face detectio
          "**********************************",'\n')
    print(end="                                                  "),
    if k==0:
        Counter=CountsD
        Labels = ["Original Images"]
    else:
        Counter=CountsH
        Labels = ["Original Images" , "Moderate augmented images", "Highly augmented
    for j in range(0,Counter.shape[1]):
        print(Labels[j],  end="   "),
    print()
    for j in range(0,Counter.shape[0]):
        if j==0:
            print('Default Haar detector performance(%): ', end="      "),
        else:
            print('Modified Haar detector performance(%):', end="      "),
        for i in range(0,Counter.shape[1]):
```

```
                print("%4.f" %Counter[j,i],  end="                    "),
            print()
        print()

    #Removing unnecessary data
    del x_train_norH, x_train_regH, x_train_badH
```

```
********************************** Dogs  images face detection performances ****************

                                        Original Images
Default Haar detector performance(%):         12
Modified Haar detector performance(%):        91

********************************** Humans  images face detection performances **************

                                        Original Images   Moderate augmented images   Highly a
Default Haar detector performance(%):         99                      82                     5
Modified Haar detector performance(%):       100                      98                     7
```

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 [11]: from keras.applications.resnet50 import ResNet50

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

1.0.5 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

$$(\text{nb}_samples\text{nb}_samples\text{nb}_samples_{\text{nb}_samples,\text{rows},\text{columns},\text{channels}),}$$

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

$$(\text{nb}_samples\text{nb}_samples\text{nb}_samples_{\text{nb}_samples, 224, 224, 3)}.$$

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 [12]: 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)
```

1.0.6 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 [13]: 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))
```

1.0.7 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 [14]: ### 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))
```

1.0.8 (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?
- What percentage of the images in `dog_files_short` have a detected dog?

Answer:

```
In [15]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.
         Dog_Detector_Human_Counter = 0
         Dog_Detector_Dog_Counter = 0

         for n in range(len(human_files_short)):
             if dog_detector(human_files_short[n])==True:
                 Dog_Detector_Human_Counter+=1
             if dog_detector(dog_files_short[n])==True:
                 Dog_Detector_Dog_Counter+=1

         print("The performance of dog_detector with human images is:", str(Dog_Detector_Human_
         print("The performance of dog_detector with dog images is:", str(Dog_Detector_Dog_Cou

The performance of dog_detector with human images is: 1 %
The performance of dog_detector with dog images is: 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.

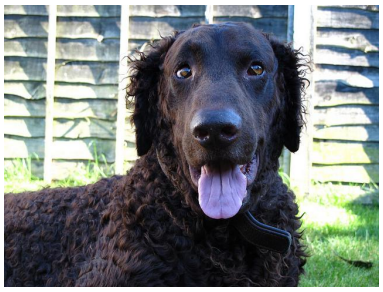
We mention that the task of assigning breed to dogs from images is considered exceptionally challenging. To see why, consider that *even a human* would have great difficulty in distinguishing between a Brittany and a Welsh Springer Spaniel.

Brittany	Welsh Springer Spaniel

.5  .5 

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

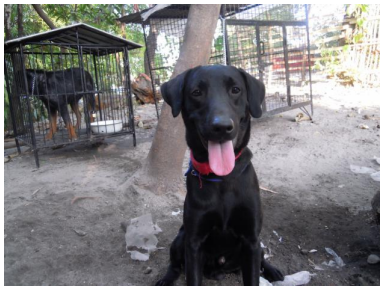
Curly-Coated Retriever	American Water Spaniel

.5  .5 

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!

1.0.9 Pre-process the Data

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

```
In [16]: 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 [01:09<00:00, 96.57it/s]
100%|| 835/835 [00:08<00:00, 103.42it/s]
100%|| 836/836 [00:07<00:00, 106.29it/s]
```

1.0.10 (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 CNN architecture was chosen, followed by a fully connected layer to classify the dog breeds. This architecture was chosen in order to use a CNN to extract the spatial features from the images, while the fully connected layer was used to relate those features with specific dog breeds. Thus, three similar but different architectures were tested, in order to determine the best

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 223, 223, 16)	208
max_pooling2d_1 (MaxPooling2	(None, 111, 111, 16)	0
conv2d_2 (Conv2D)	(None, 110, 110, 32)	2080
max_pooling2d_2 (MaxPooling2	(None, 55, 55, 32)	0
conv2d_3 (Conv2D)	(None, 54, 54, 64)	8256
max_pooling2d_3 (MaxPooling2	(None, 27, 27, 64)	0
global_average_pooling2d_1 (	(None, 64)	0
dense_1 (Dense)	(None, 133)	8645

Total params: 19,189.0
Trainable params: 19,189.0
Non-trainable params: 0.0

Sample CNN

parameters for the mentioned architecture, such as the number of filters and nodes on each layer. The results can be seen in the *test model* subsection.

As mentioned before, three different architectures were tested. The hinted architecture was used as a base with a fully connected layer as a complement. The differences between the mentioned architectures are the number of filters and nodes used in the fully connected layer. The selected architecture was the one that gave better accuracy during its evaluation.

Selected network architecture

```
In [17]: from keras.layers import Conv2D, MaxPooling2D, GlobalAveragePooling2D
         from keras.layers import Dropout, Flatten, Dense
         from keras.models import Sequential

         model = Sequential()

         ### TODO: Define your architecture.
         model.add(Conv2D(filters=32, kernel_size=2, padding='same', activation='relu',
                                  input_shape=(224, 224,3)))
         model.add(MaxPooling2D(pool_size=2))
         model.add(Conv2D(filters=64, kernel_size=2, padding='same', activation='relu'))
         model.add(MaxPooling2D(pool_size=2))
         model.add(Conv2D(filters=128, kernel_size=2, padding='same', activation='relu'))
         model.add(Dropout(0.3))
         model.add(Flatten())
         model.add(Dense(200, activation='relu'))
         model.add(Dropout(0.4))
         model.add(Dense(133, activation='softmax'))

         model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
```

```
conv2d_1 (Conv2D)            (None, 224, 224, 32)      416
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 112, 112, 32)      0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 112, 112, 64)      8256
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 56, 56, 64)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 56, 56, 128)       32896
_________________________________________________________________
dropout_1 (Dropout)          (None, 56, 56, 128)       0
_________________________________________________________________
flatten_1 (Flatten)          (None, 401408)            0
_________________________________________________________________
dense_1 (Dense)              (None, 200)               80281800
_________________________________________________________________
dropout_2 (Dropout)          (None, 200)               0
_________________________________________________________________
dense_2 (Dense)              (None, 133)               26733
=================================================================
Total params: 80,350,101
Trainable params: 80,350,101
Non-trainable params: 0
_________________________________________________________________
```

1.0.11 Compile the Model

```
In [18]: model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy
```

1.0.12 (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 [19]: from keras.callbacks import ModelCheckpoint

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

         epochs = 5

         ### 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=2)
```

```
Train on 6680 samples, validate on 835 samples
Epoch 1/5
 - 49s - loss: 8.3561 - acc: 0.0096 - val_loss: 4.8305 - val_acc: 0.0287

Epoch 00001: val_loss improved from inf to 4.83053, saving model to saved_models/weights.best.
Epoch 2/5
 - 46s - loss: 4.7188 - acc: 0.0289 - val_loss: 4.5108 - val_acc: 0.0479

Epoch 00002: val_loss improved from 4.83053 to 4.51084, saving model to saved_models/weights.be
Epoch 3/5
 - 46s - loss: 4.2255 - acc: 0.0837 - val_loss: 4.3997 - val_acc: 0.0563

Epoch 00003: val_loss improved from 4.51084 to 4.39969, saving model to saved_models/weights.be
Epoch 4/5
 - 48s - loss: 3.2263 - acc: 0.2490 - val_loss: 4.7002 - val_acc: 0.0743

Epoch 00004: val_loss did not improve from 4.39969
Epoch 5/5
 - 48s - loss: 1.9450 - acc: 0.5219 - val_loss: 5.2365 - val_acc: 0.0695

Epoch 00005: val_loss did not improve from 4.39969
```

```
Out[19]: <keras.callbacks.History at 0x21fd95785f8>
```

1.0.13 Load the Model with the Best Validation Loss

```
In [20]: model.load_weights('saved_models/weights.best.from_scratch.hdf5')
```

1.0.14 Test the Model

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

```
In [21]: #_________________________________________________________________
         #Layer (type)                 Output Shape              Param #
         #=================================================================
         #conv2d_40 (Conv2D)           (None, 224, 224, 16)      208
         #_________________________________________________________________
         #max_pooling2d_35 (MaxPooling (None, 112, 112, 16)      0
         #_________________________________________________________________
         #conv2d_41 (Conv2D)           (None, 112, 112, 32)      2080
         #_________________________________________________________________
         #max_pooling2d_36 (MaxPooling (None, 56, 56, 32)        0
         #_________________________________________________________________
         #conv2d_42 (Conv2D)           (None, 56, 56, 64)        8256
```

```
#_________________________________________________________________
#global_average_pooling2d_4 ( (None, 64)                0
#_________________________________________________________________
#dense_18 (Dense)            (None, 133)               8645
#=================================================================
#Total params: 19,189
#Trainable params: 19,189
#Non-trainable params: 0

# 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: 5.6220%
```

In [22]:
```
#_________________________________________________________________
#Layer (type)                 Output Shape              Param #
#=================================================================
#conv2d_37 (Conv2D)           (None, 224, 224, 16)      208
#_________________________________________________________________
#max_pooling2d_33 (MaxPooling (None, 112, 112, 16)      0
#_________________________________________________________________
#conv2d_38 (Conv2D)           (None, 112, 112, 32)      2080
#_________________________________________________________________
#max_pooling2d_34 (MaxPooling (None, 56, 56, 32)        0
#_________________________________________________________________
#conv2d_39 (Conv2D)           (None, 56, 56, 64)        8256
#_________________________________________________________________
#dropout_14 (Dropout)         (None, 56, 56, 64)        0
#_________________________________________________________________
#flatten_9 (Flatten)          (None, 200704)            0
#_________________________________________________________________
#dense_16 (Dense)             (None, 200)               40141000
#_________________________________________________________________
#dropout_15 (Dropout)         (None, 200)               0
#_________________________________________________________________
#dense_17 (Dense)             (None, 133)               26733
#=================================================================
#Total params: 40,178,277
#Trainable params: 40,178,277
#Non-trainable params: 0
#_________________________________________________________________

# 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, ax
        print('Test accuracy: %.4f%%' % test_accuracy)
```

```
Test accuracy: 5.6220%
```

```
In [23]: #_________________________________________________________________
         #Layer (type)                 Output Shape              Param #
         #=================================================================
         #conv2d_1 (Conv2D)            (None, 224, 224, 32)      416
         #_________________________________________________________________
         #max_pooling2d_2 (MaxPooling2 (None, 112, 112, 32)      0
         #_________________________________________________________________
         #conv2d_2 (Conv2D)            (None, 112, 112, 64)      8256
         #_________________________________________________________________
         #max_pooling2d_3 (MaxPooling2 (None, 56, 56, 64)        0
         #_________________________________________________________________
         #conv2d_3 (Conv2D)            (None, 56, 56, 128)       32896
         #_________________________________________________________________
         #dropout_1 (Dropout)          (None, 56, 56, 128)       0
         #_________________________________________________________________
         #flatten_2 (Flatten)          (None, 401408)            0
         #_________________________________________________________________
         #dense_1 (Dense)              (None, 200)               80281800
         #_________________________________________________________________
         #dropout_2 (Dropout)          (None, 200)               0
         #_________________________________________________________________
         #dense_2 (Dense)              (None, 133)               26733
         #=================================================================
         #Total params: 80,350,101
         #Trainable params: 80,350,101
         #Non-trainable params: 0
         #_________________________________________________________________

         # 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, ax
         print('Test accuracy: %.4f%%' % test_accuracy)
```

```
Test accuracy: 5.6220%
```

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.

1.0.15 Obtain Bottleneck Features

```
In [24]: bottleneck_features = np.load('bottleneck_features/DogVGG16Data.npz')
         train_VGG16 = bottleneck_features['train']
         valid_VGG16 = bottleneck_features['valid']
         test_VGG16 = bottleneck_features['test']
```

1.0.16 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 [25]: 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_3 (Dense)              (None, 133)               68229
=================================================================
Total params: 68,229
Trainable params: 68,229
Non-trainable params: 0
_________________________________________________________________
```

1.0.17 Compile the Model

```
In [26]: VGG16_model.compile(loss='categorical_crossentropy', optimizer='rmsprop', metrics=['a
```

1.0.18 Train the Model

```
In [27]: from keras.callbacks import ModelCheckpoint

         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=2)

Train on 6680 samples, validate on 835 samples
Epoch 1/20
 - 7s - loss: 12.8884 - acc: 0.1036 - val_loss: 11.6352 - val_acc: 0.1653

Epoch 00001: val_loss improved from inf to 11.63525, saving model to saved_models/weights.best
Epoch 2/20
 - 2s - loss: 11.0729 - acc: 0.2263 - val_loss: 10.8531 - val_acc: 0.2287

Epoch 00002: val_loss improved from 11.63525 to 10.85306, saving model to saved_models/weights
Epoch 3/20
 - 2s - loss: 10.4925 - acc: 0.2880 - val_loss: 10.5936 - val_acc: 0.2587

Epoch 00003: val_loss improved from 10.85306 to 10.59361, saving model to saved_models/weights
Epoch 4/20
 - 2s - loss: 10.2818 - acc: 0.3204 - val_loss: 10.5015 - val_acc: 0.2766

Epoch 00004: val_loss improved from 10.59361 to 10.50152, saving model to saved_models/weights
Epoch 5/20
 - 2s - loss: 9.9821 - acc: 0.3365 - val_loss: 10.0789 - val_acc: 0.2958

Epoch 00005: val_loss improved from 10.50152 to 10.07885, saving model to saved_models/weights
Epoch 6/20
 - 2s - loss: 9.5278 - acc: 0.3662 - val_loss: 9.8827 - val_acc: 0.3114

Epoch 00006: val_loss improved from 10.07885 to 9.88269, saving model to saved_models/weights.h
Epoch 7/20
 - 2s - loss: 9.2873 - acc: 0.3904 - val_loss: 9.8402 - val_acc: 0.3018

Epoch 00007: val_loss improved from 9.88269 to 9.84022, saving model to saved_models/weights.be
Epoch 8/20
 - 2s - loss: 9.0482 - acc: 0.4028 - val_loss: 9.5252 - val_acc: 0.3198

Epoch 00008: val_loss improved from 9.84022 to 9.52516, saving model to saved_models/weights.be
Epoch 9/20
 - 2s - loss: 8.6521 - acc: 0.4229 - val_loss: 9.1951 - val_acc: 0.3329

Epoch 00009: val_loss improved from 9.52516 to 9.19511, saving model to saved_models/weights.be
Epoch 10/20
 - 2s - loss: 8.3162 - acc: 0.4494 - val_loss: 8.9837 - val_acc: 0.3533

Epoch 00010: val_loss improved from 9.19511 to 8.98366, saving model to saved_models/weights.be
Epoch 11/20
 - 3s - loss: 8.1702 - acc: 0.4714 - val_loss: 8.8714 - val_acc: 0.3629
```

```
Epoch 00011: val_loss improved from 8.98366 to 8.87138, saving model to saved_models/weights.be
Epoch 12/20
 - 3s - loss: 8.0543 - acc: 0.4814 - val_loss: 8.7788 - val_acc: 0.3808

Epoch 00012: val_loss improved from 8.87138 to 8.77882, saving model to saved_models/weights.be
Epoch 13/20
 - 2s - loss: 7.9379 - acc: 0.4880 - val_loss: 8.5725 - val_acc: 0.3952

Epoch 00013: val_loss improved from 8.77882 to 8.57253, saving model to saved_models/weights.be
Epoch 14/20
 - 2s - loss: 7.7803 - acc: 0.5003 - val_loss: 8.4388 - val_acc: 0.4000

Epoch 00014: val_loss improved from 8.57253 to 8.43881, saving model to saved_models/weights.be
Epoch 15/20
 - 2s - loss: 7.5521 - acc: 0.5108 - val_loss: 8.2090 - val_acc: 0.4084

Epoch 00015: val_loss improved from 8.43881 to 8.20896, saving model to saved_models/weights.be
Epoch 16/20
 - 2s - loss: 7.4003 - acc: 0.5257 - val_loss: 8.1670 - val_acc: 0.4132

Epoch 00016: val_loss improved from 8.20896 to 8.16698, saving model to saved_models/weights.be
Epoch 17/20
 - 2s - loss: 7.3009 - acc: 0.5298 - val_loss: 7.9010 - val_acc: 0.4263

Epoch 00017: val_loss improved from 8.16698 to 7.90097, saving model to saved_models/weights.be
Epoch 18/20
 - 2s - loss: 7.0004 - acc: 0.5496 - val_loss: 7.7796 - val_acc: 0.4275

Epoch 00018: val_loss improved from 7.90097 to 7.77965, saving model to saved_models/weights.be
Epoch 19/20
 - 2s - loss: 6.7618 - acc: 0.5591 - val_loss: 7.6269 - val_acc: 0.4467

Epoch 00019: val_loss improved from 7.77965 to 7.62693, saving model to saved_models/weights.be
Epoch 20/20
 - 2s - loss: 6.5645 - acc: 0.5725 - val_loss: 7.4153 - val_acc: 0.4551

Epoch 00020: val_loss improved from 7.62693 to 7.41533, saving model to saved_models/weights.be


Out[27]: <keras.callbacks.History at 0x222145c0860>
```

1.0.19 Load the Model with the Best Validation Loss

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

1.0.20 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))) :

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

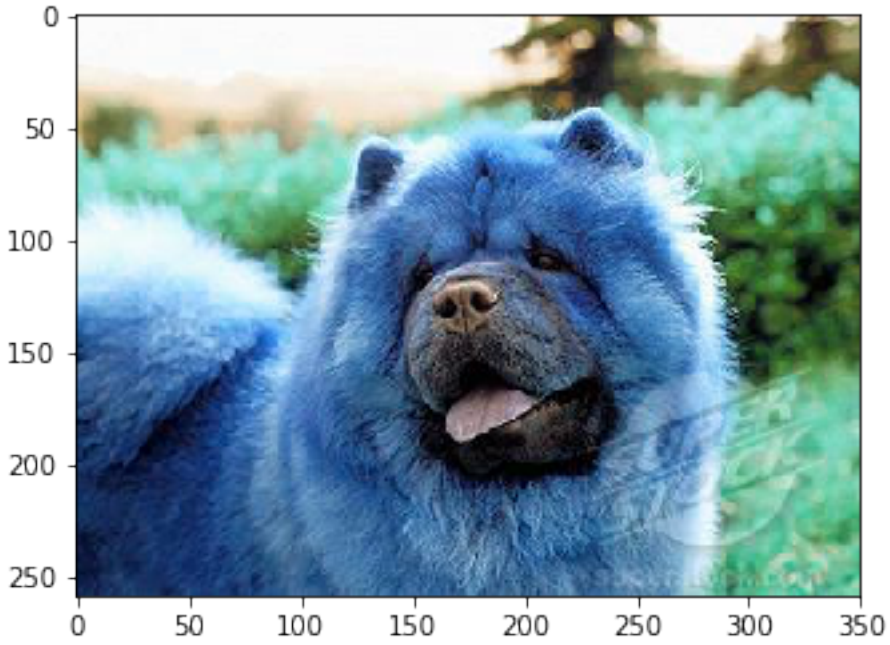
```
Test accuracy: 44.9761%
```

1.0.21 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 [31]: print("The dog in the fallowing image is a: ",VGG16_predict_breed(test_files[9]))
         img = cv2.imread(test_files[9])
         plt.imshow(img)
         plt.show()
```

```
The dog in the fallowing image is a:  Akita
```



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.

1.0.22 (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 [32]: ### TODO: Obtain bottleneck features from another pre-trained CNN.
         bottleneck_features = np.load('bottleneck_features/DogVGG19Data.npz')
         train_VGG19 = bottleneck_features['train']
         valid_VGG19 = bottleneck_features['valid']
         test_VGG19 = bottleneck_features['test']
```

1.0.23 (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:

I used a *VGG19* network because I wanted to have a metric to compare the performance of the selected network, and the best metric I available was the *VGG16* network used in the previous sections. As a result, the *VGG19* network was tested in the same conditions as was implemented the *VGG16*, getting an accuracy around *60%*. Finally, fully connected layers were added in

ascendent order, following the procedure implemented to answer the 4th question. Additionally, different amount of nodes were used in the implemented extra layers. Nonetheless, two is the maximum number of fully connected layers used, since it was enough to get the desired performance, as shown next:

In [33]:
```
### TODO: Define your architecture.
VGG19_model = Sequential()
VGG19_model.add(GlobalAveragePooling2D(input_shape=train_VGG19.shape[1:]))
VGG19_model.add(Dense(300, activation='relu'))
VGG19_model.add(Dropout(0.4))
VGG19_model.add(Dense(133, activation='softmax'))

VGG19_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
global_average_pooling2d_2 ( (None, 512)               0
_________________________________________________________________
dense_4 (Dense)              (None, 300)               153900
_________________________________________________________________
dropout_3 (Dropout)          (None, 300)               0
_________________________________________________________________
dense_5 (Dense)              (None, 133)               40033
=================================================================
Total params: 193,933
Trainable params: 193,933
Non-trainable params: 0
_________________________________________________________________
```

1.0.24 (IMPLEMENTATION) Compile the Model

In [34]:
```
### TODO: Compile the model.
VGG19_model.compile(loss='categorical_crossentropy', optimizer='rmsprop', metrics=['a
```

1.0.25 (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.

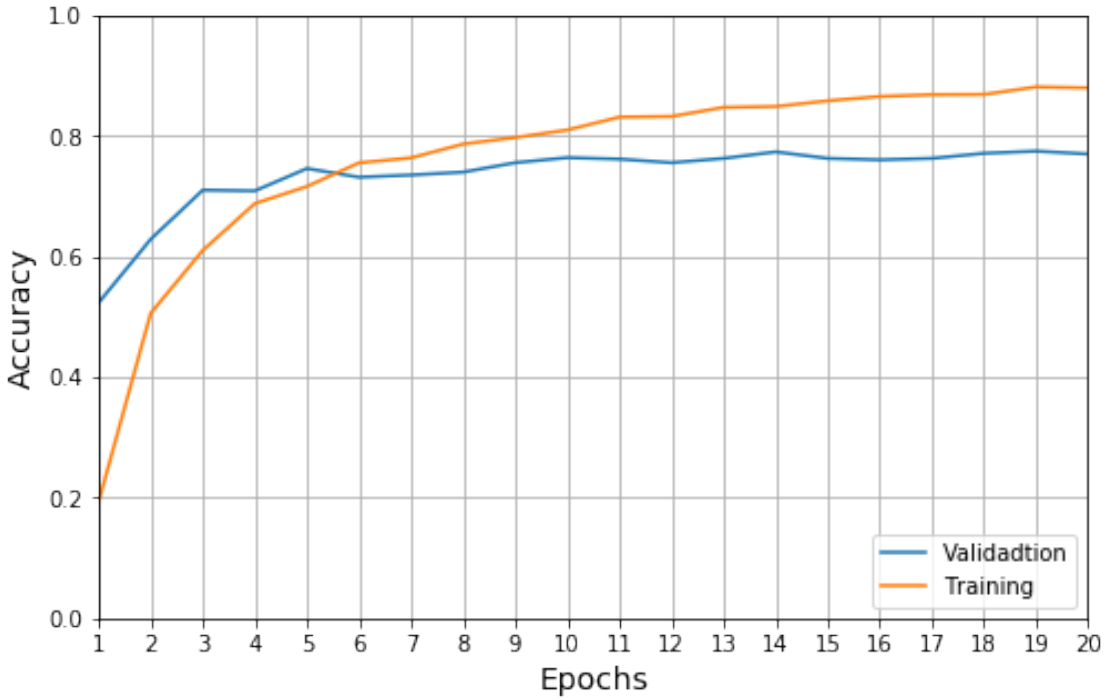
You are welcome to augment the training data, but this is not a requirement.

In [35]:
```
### TODO: Train the model.
checkpointer = ModelCheckpoint(filepath='saved_models/weights.best.VGG19.hdf5',
                               verbose=1, save_best_only=True)

VGG19_model.fit(train_VGG19, train_targets,
          validation_data=(valid_VGG19, valid_targets),
          epochs=20, batch_size=20, callbacks=[checkpointer], verbose=2)
```

```
Train on 6680 samples, validate on 835 samples
Epoch 1/20
 - 4s - loss: 5.2184 - acc: 0.1928 - val_loss: 1.8403 - val_acc: 0.5234

Epoch 00001: val_loss improved from inf to 1.84031, saving model to saved_models/weights.best.
Epoch 2/20
 - 2s - loss: 1.9938 - acc: 0.5060 - val_loss: 1.2946 - val_acc: 0.6287

Epoch 00002: val_loss improved from 1.84031 to 1.29460, saving model to saved_models/weights.be
Epoch 3/20
 - 2s - loss: 1.4973 - acc: 0.6102 - val_loss: 1.0196 - val_acc: 0.7102

Epoch 00003: val_loss improved from 1.29460 to 1.01957, saving model to saved_models/weights.be
Epoch 4/20
 - 2s - loss: 1.2153 - acc: 0.6880 - val_loss: 1.0321 - val_acc: 0.7090

Epoch 00004: val_loss did not improve from 1.01957
Epoch 5/20
 - 3s - loss: 1.0825 - acc: 0.7163 - val_loss: 0.9431 - val_acc: 0.7461

Epoch 00005: val_loss improved from 1.01957 to 0.94309, saving model to saved_models/weights.be
Epoch 6/20
 - 3s - loss: 0.9510 - acc: 0.7555 - val_loss: 1.0437 - val_acc: 0.7317

Epoch 00006: val_loss did not improve from 0.94309
Epoch 7/20
 - 2s - loss: 0.9069 - acc: 0.7638 - val_loss: 1.0186 - val_acc: 0.7353

Epoch 00007: val_loss did not improve from 0.94309
Epoch 8/20
 - 2s - loss: 0.7940 - acc: 0.7867 - val_loss: 1.2505 - val_acc: 0.7401

Epoch 00008: val_loss did not improve from 0.94309
Epoch 9/20
 - 2s - loss: 0.7973 - acc: 0.7975 - val_loss: 1.1187 - val_acc: 0.7557

Epoch 00009: val_loss did not improve from 0.94309
Epoch 10/20
 - 2s - loss: 0.7315 - acc: 0.8100 - val_loss: 1.0903 - val_acc: 0.7641

Epoch 00010: val_loss did not improve from 0.94309
Epoch 11/20
 - 2s - loss: 0.6779 - acc: 0.8314 - val_loss: 1.1885 - val_acc: 0.7617

Epoch 00011: val_loss did not improve from 0.94309
Epoch 12/20
 - 2s - loss: 0.6911 - acc: 0.8325 - val_loss: 1.2992 - val_acc: 0.7557
```

```
Epoch 00012: val_loss did not improve from 0.94309
Epoch 13/20
 - 2s - loss: 0.6279 - acc: 0.8475 - val_loss: 1.1896 - val_acc: 0.7629

Epoch 00013: val_loss did not improve from 0.94309
Epoch 14/20
 - 2s - loss: 0.6149 - acc: 0.8490 - val_loss: 1.2339 - val_acc: 0.7737

Epoch 00014: val_loss did not improve from 0.94309
Epoch 15/20
 - 2s - loss: 0.5842 - acc: 0.8584 - val_loss: 1.3508 - val_acc: 0.7629

Epoch 00015: val_loss did not improve from 0.94309
Epoch 16/20
 - 2s - loss: 0.5841 - acc: 0.8656 - val_loss: 1.3394 - val_acc: 0.7605

Epoch 00016: val_loss did not improve from 0.94309
Epoch 17/20
 - 2s - loss: 0.5512 - acc: 0.8686 - val_loss: 1.3624 - val_acc: 0.7629

Epoch 00017: val_loss did not improve from 0.94309
Epoch 18/20
 - 2s - loss: 0.5683 - acc: 0.8690 - val_loss: 1.3127 - val_acc: 0.7713

Epoch 00018: val_loss did not improve from 0.94309
Epoch 19/20
 - 2s - loss: 0.5236 - acc: 0.8814 - val_loss: 1.4130 - val_acc: 0.7749

Epoch 00019: val_loss did not improve from 0.94309
Epoch 20/20
 - 2s - loss: 0.5542 - acc: 0.8799 - val_loss: 1.2984 - val_acc: 0.7701

Epoch 00020: val_loss did not improve from 0.94309


Out[35]: <keras.callbacks.History at 0x22002172be0>

In [43]: #Generating vectors containing the training and validation accuracy
         ValAccu = [0.5234, 0.6287, 0.7102, 0.7090, 0.7461, 0.7317, 0.7353, 0.7401, 0.7557, 0.7
                    0.7737, 0.7629, 0.7605, 0.7629, 0.7713, 0.7749, 0.7701]
         TrainAccu = [0.1928, 0.5060, 0.6102, 0.6880, 0.7163, 0.7555, 0.7638, 0.7867, 0.7975, 0
                      0.8490, 0.8584, 0.8656, 0.8686, 0.8690, 0.8814, 0.8799]

         #Plotting the training and validation accuracies
         plt.figure(figsize=(8,5))
         plt.plot(range(1,21), ValAccu)
         plt.plot(range(1,21), TrainAccu, )
         plt.xticks(range(1,22, 1))
```

```
plt.xlim([1,20])
plt.ylim([0,1])
plt.grid(True)

plt.legend(('Validadtion', 'Training'),
           loc='lower right')
plt.xlabel('Epochs', fontsize=14)
plt.ylabel('Accuracy', fontsize=14);
```



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

In [37]:
```
### TODO: Load the model weights with the best validation loss.
VGG19_model.load_weights('saved_models/weights.best.VGG19.hdf5')
```

1.0.27 (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 [38]:
```
### TODO: Calculate classification accuracy on the test dataset.
# get index of predicted dog breed for each image in test set
VGG19_predictions = [np.argmax(VGG19_model.predict(np.expand_dims(feature, axis=0))) 

# report test accuracy
test_accuracy = 100*np.sum(np.array(VGG19_predictions)==np.argmax(test_targets, axis=
print('Test accuracy: %.4f%%' % test_accuracy)
```

```
Test accuracy: 75.2392%
```

1.0.28 (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

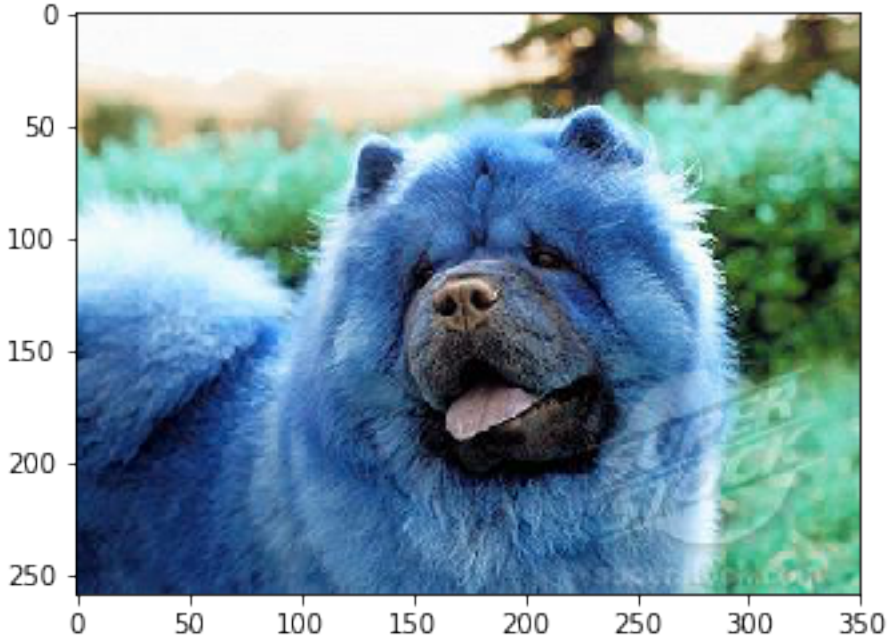
```
extract_{network}
```

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

In [39]:
```
### TODO: Write a function that takes a path to an image as input
### and returns the dog breed that is predicted by the model.
from extract_bottleneck_features import *

def VGG19_predict_breed(img_path):

    """
    Function that takes a path to an image as input and returns the dog breed that is

    Inputs:
        img_path: Image file path.

    Returns:
        dog_names: dog breed that is predicted by the model.
    """

    # extract bottleneck features
    bottleneck_feature = extract_VGG19(path_to_tensor(img_path))
    # obtain predicted vector
    predicted_vector = VGG19_model.predict(bottleneck_feature)
    # return dog breed that is predicted by the model
    return dog_names[np.argmax(predicted_vector)]
```

In [40]:
```
#Testing the previous function
print("The dog in the following image is a: ",VGG19_predict_breed(test_files[9]))
img = cv2.imread(test_files[9])
plt.imshow(img)
plt.show()
```

```
The dog in the following image is a:  Chow_chow
```



Step 6: Write your Algorithm

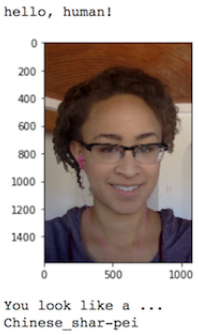
Write an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then, - if a **dog** is detected in the image, return the predicted breed. - if a **human** is detected in the image, return the resembling dog breed. - if **neither** is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

1.0.29 (IMPLEMENTATION) Write your Algorithm

```
In [41]: ### TODO: Write your algorithm.
         ### Feel free to use as many code cells as needed.
         import sys

         def HumanDogOrNeither(SelImageName,ScaleFactor=1.05,minNeighbors=3):

             """
             Function to predict dog breeds from dogs and humans images.
```



Sample Human Output

```
    Inputs:
        SelImageName: Image file path.
        ScaleFactor: Parameter specifying how much the image size is reduced at each
        minNeighbors: Parameter specifying how many neighbors each candidate rectangl

    Returns:
        If a dog of a human image is used as input, the function return the more simi
    """
    #Pre-processing the selected image:
    SelImage = cv2.imread(SelImageName)
    grayImage = cv2.cvtColor(SelImage, cv2.COLOR_BGR2GRAY)

    #Identifying if there is a dog or a human in the selected image:
    def checkingImage(SelImageName,ScaleFactor,minNeighbors):
        if dog_detector(SelImageName)==True:
            print('The breed of the dog in the following image is: ',end=''),
        else:
            print(len(face_cascade.detectMultiScale(grayImage,ScaleFactor,minNeighbors
            if (len(face_cascade.detectMultiScale(grayImage,ScaleFactor,minNeighbors)
                print('The person in the following image resembles a: ')
            else:
                print('There are not humans or dogs in the following image',end=''),
                fig = plt.figure(figsize=(8,8))
                plt.imshow(SelImage)
                plt.show()
                raise ValueError(" ")

    try:
        checkingImage(SelImageName,ScaleFactor,minNeighbors)
    except :
        return

    #Predicting dog breeds using the CNN trained in question 5.
    Dog_breed_key=VGG19_predict_breed(SelImageName)
```

```
#Printing results:
print(Dog_breed_key)

#Plotting selected imaged, next to the dog_breed that resembles the human or dog
fig = plt.figure(figsize=(8,8))
plt.imshow(SelImage)
plt.show()
```

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?

1.0.30 (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:

Results obtained are satisfactory, as expected after getting an accuracy for the proposed network of 72%. However, the algorithm can improve considering the following points:

1.-A bigger dataset can be used to train the network, with this the chosen network could relate more features to the breeds.

2.-More epochs to train the network. Using more epochs would probably allow the network to learn better how to recognize dog breeds in an image.

3.-Probably by making the architecture of the network bigger and more complex, but that would require more resources such as RAM memory and GPU processing.

In [42]:
```
## 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.
import os

#Name for the images to test
TestImagesFiles={1:"Chamo",2:"Jr",3:"Oso",4:"Lobo",5:"Chama",6:"Mini",7:"Monkey",8:"Sa
                 10:"Robert", 11:"Quetzalcoatl",12:"Rei",13:"Finito",14:"Miyamoto",15:'

#Testing the dog breed classification algorithm
for n in TestImagesFiles.keys():
    TelImageName = (os.getcwd()+"\MyImages\\"+TestImagesFiles[n]+".jpg")
    HumanDogOrNeither(TelImageName)
```

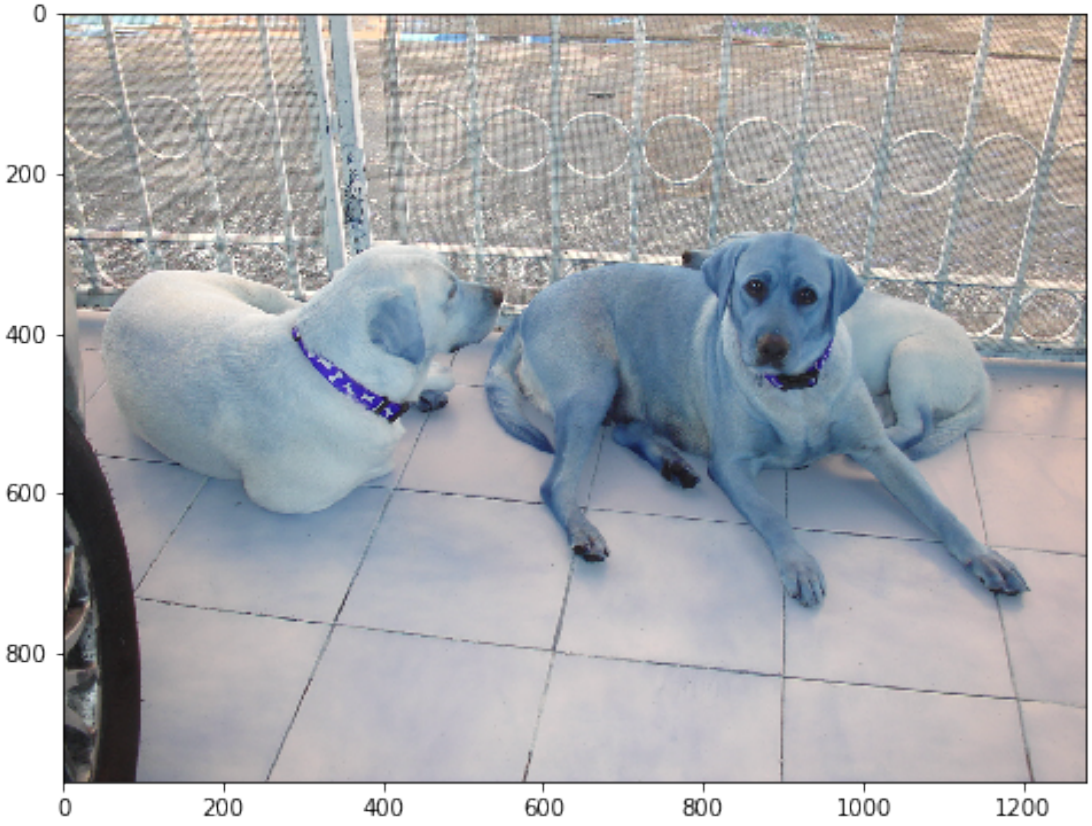
```
The breed of the dog in the following image is: Dalmatian
```

```
The breed of the dog in the following image is: Labrador_retriever
```

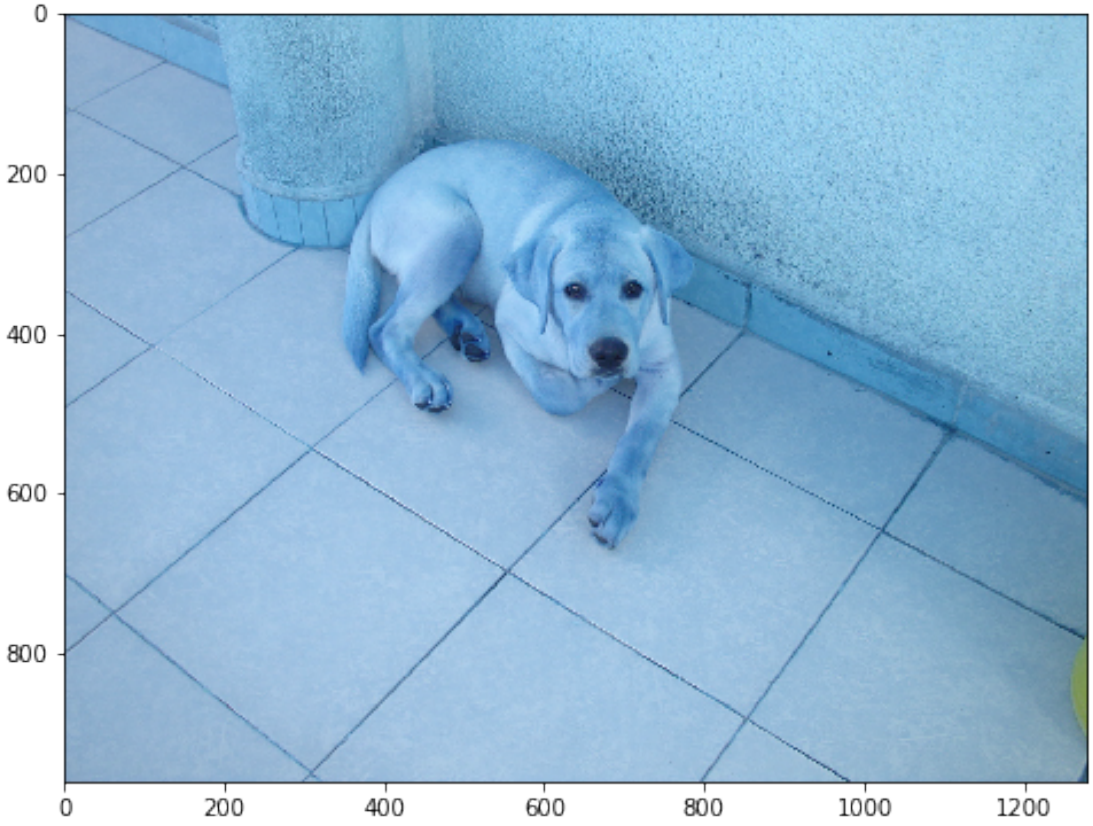
```
The breed of the dog in the following image is: Chow_chow
```

```
The breed of the dog in the following image is: German_shepherd_dog
```

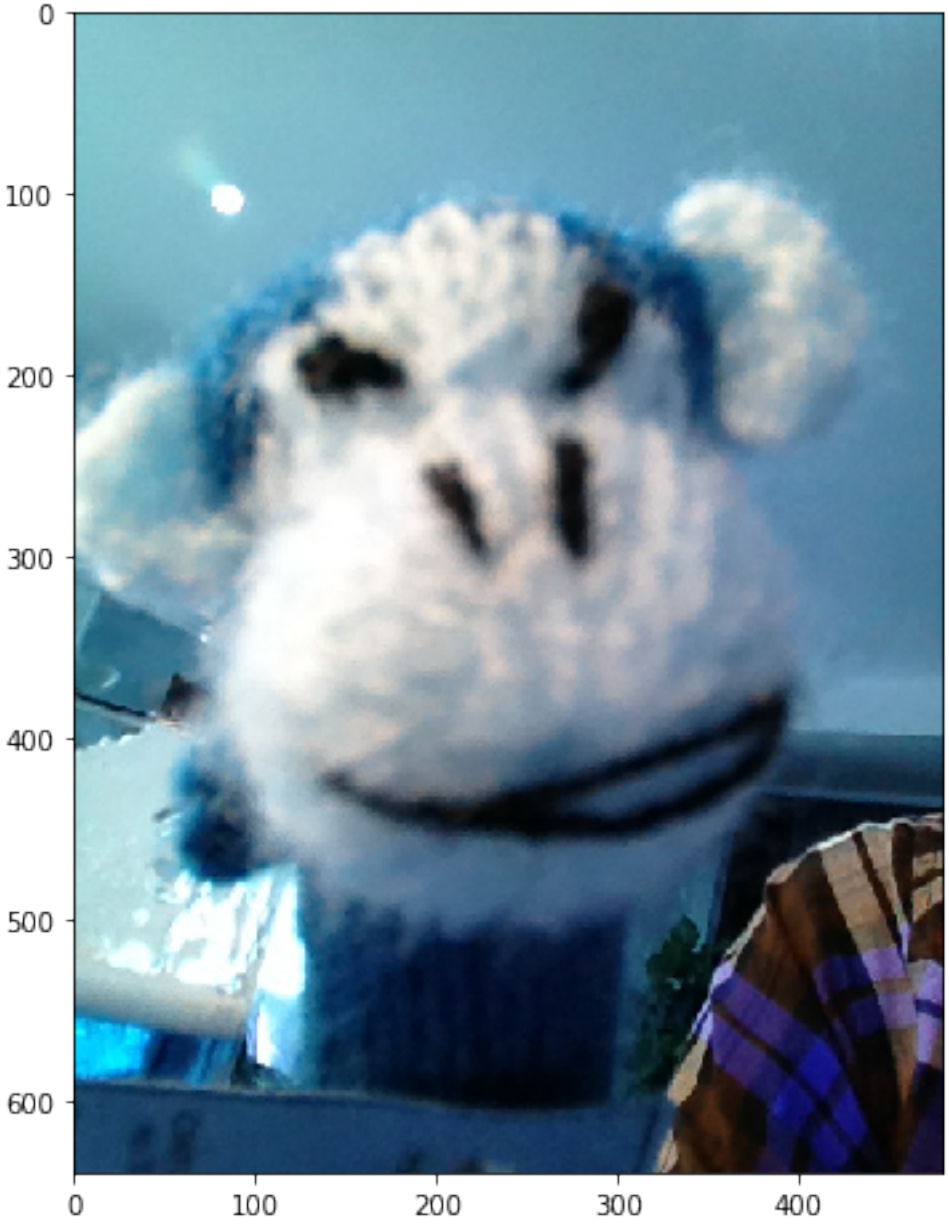
```
The breed of the dog in the following image is: Beagle
```



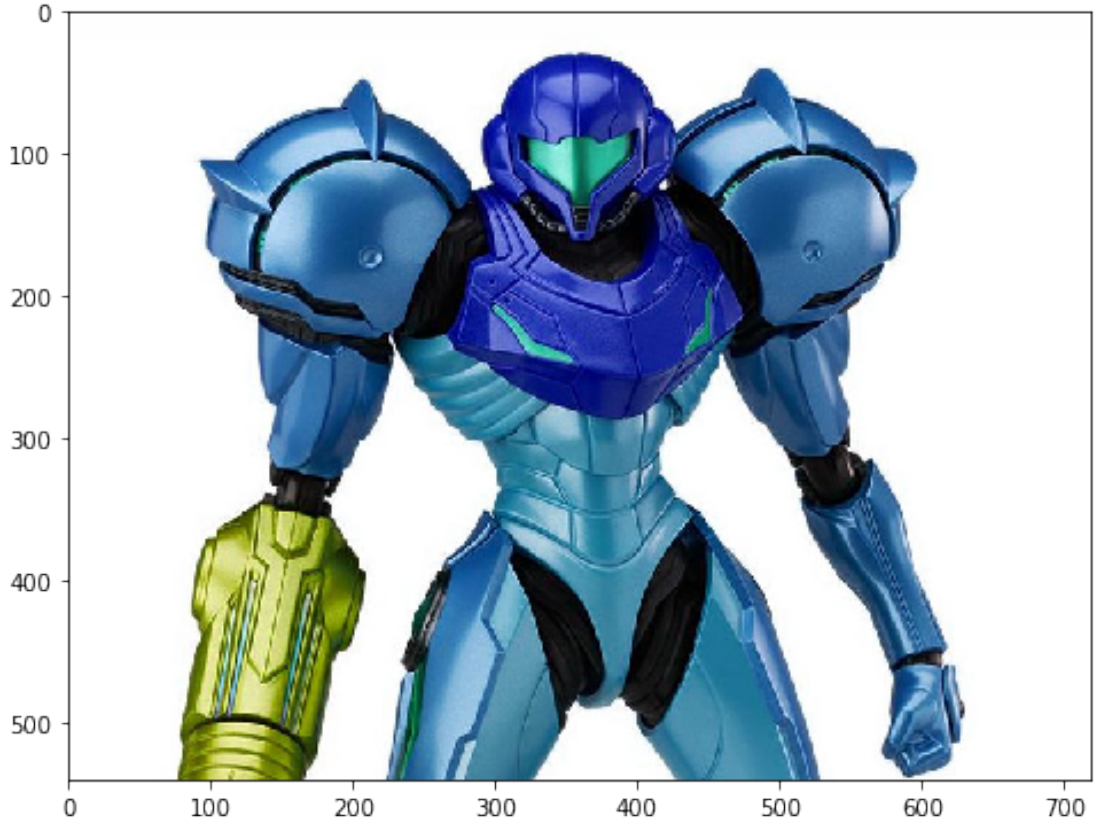
```
The breed of the dog in the following image is: Labrador_retriever
```



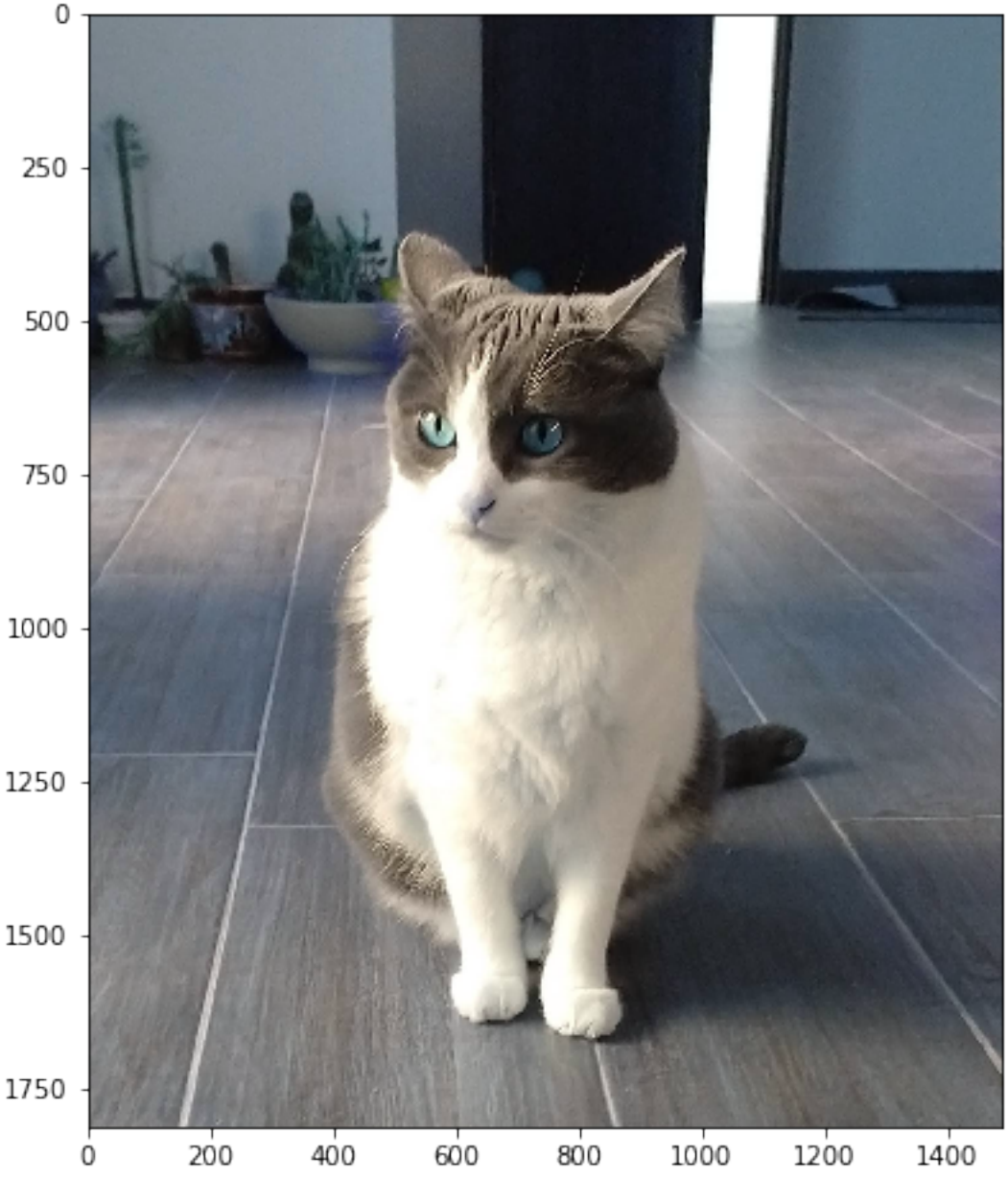
```
There are not humans or dogs in the following image
```



There are not humans or dogs in the following image

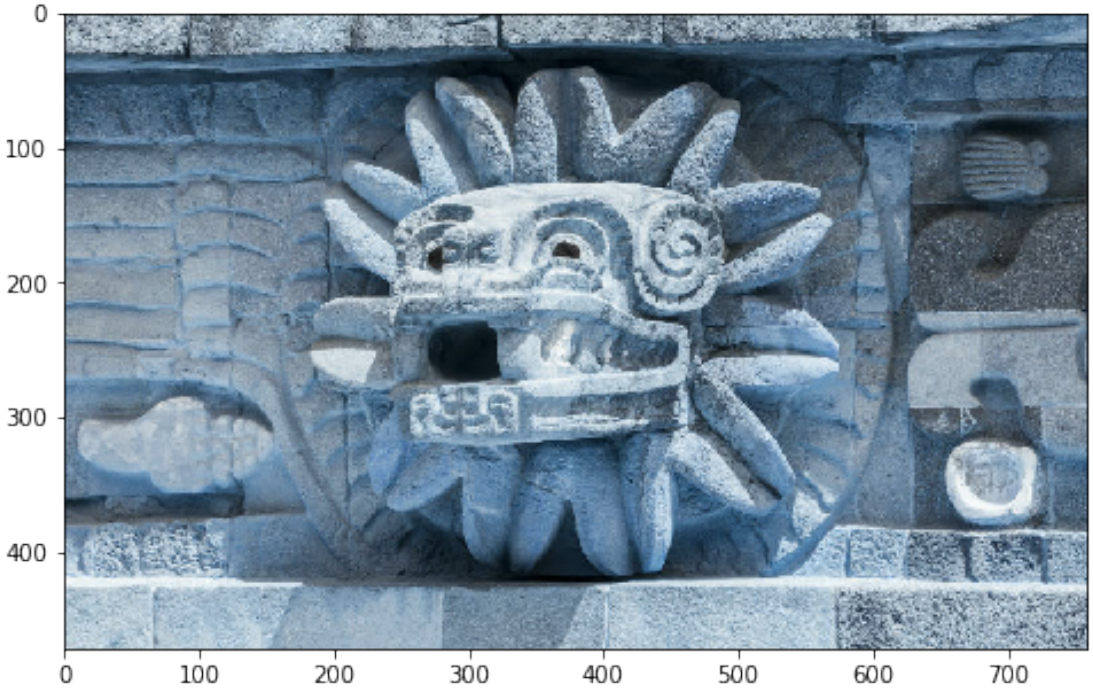


There are not humans or dogs in the following image

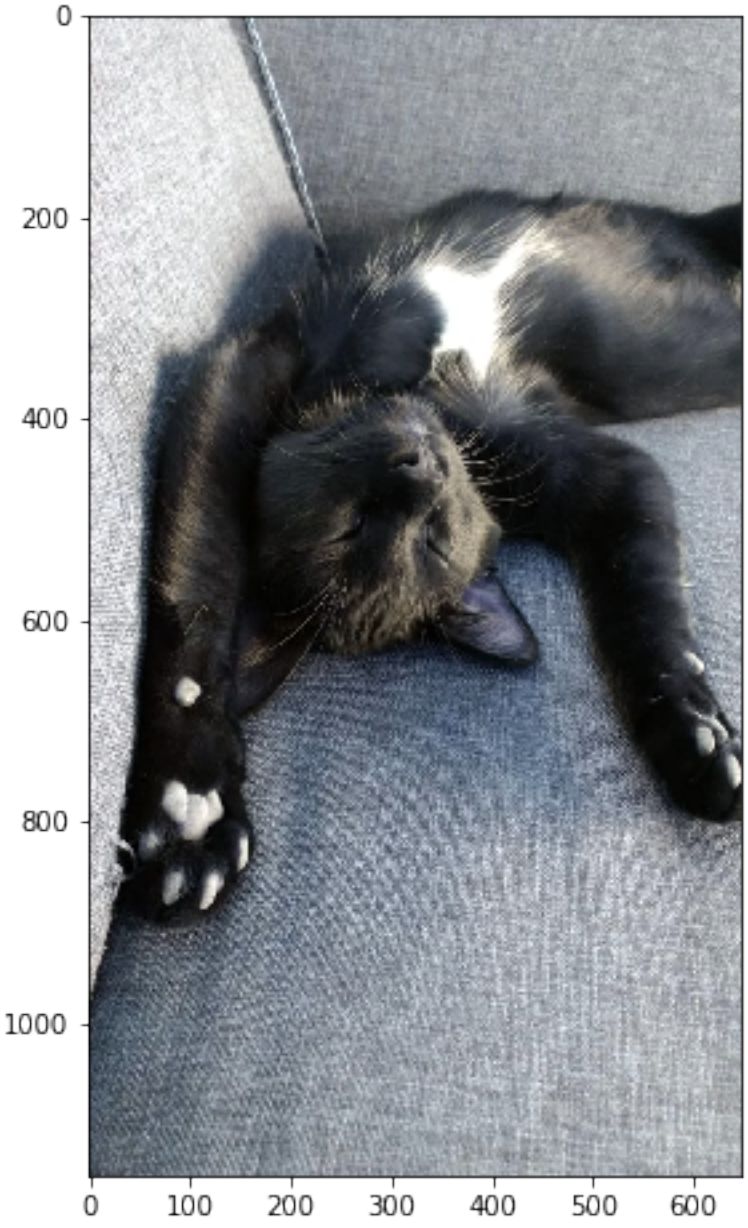


```
The person in the following image resembles a:
Irish_water_spaniel
```

There are not humans or dogs in the following image



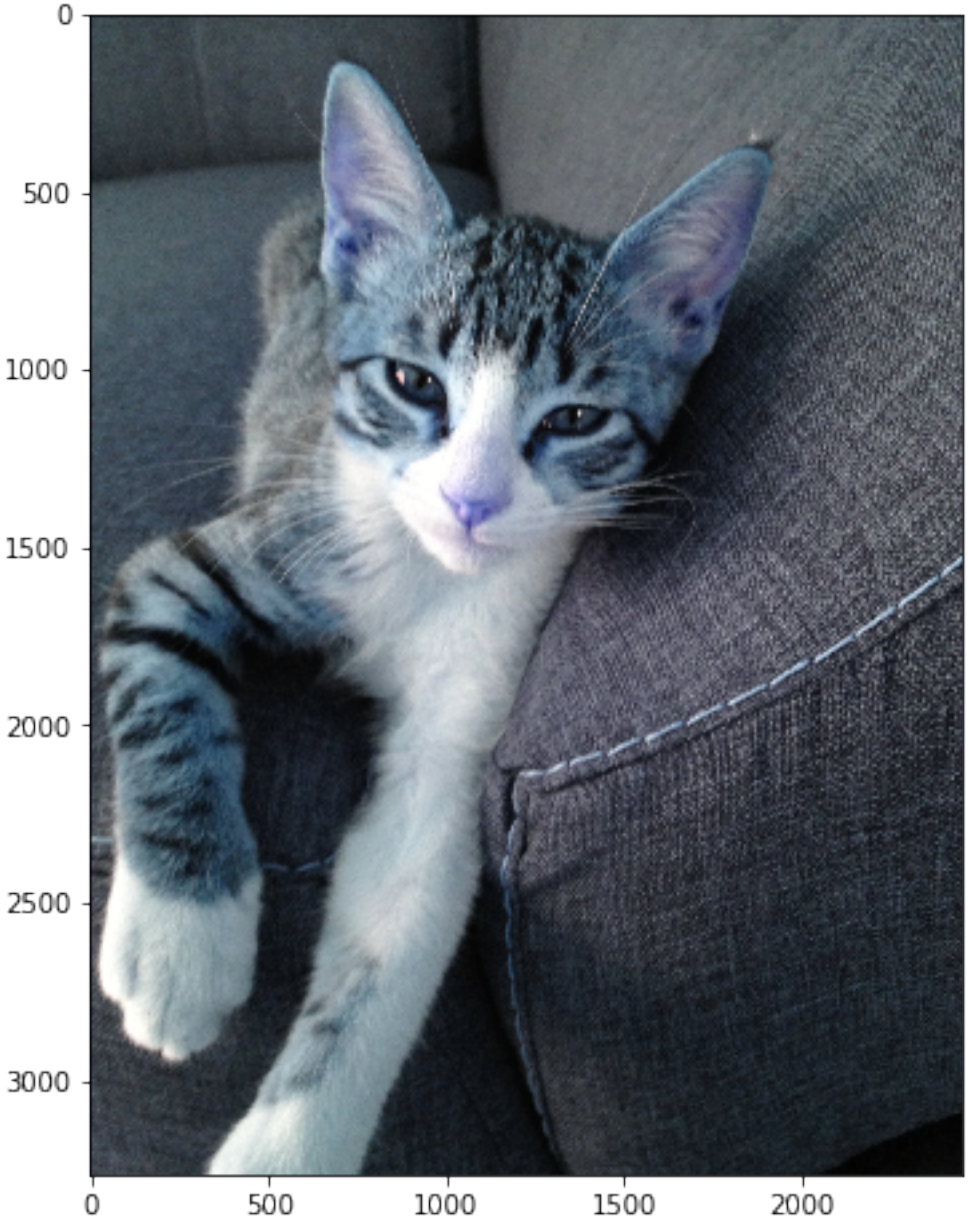
There are not humans or dogs in the following image



```
The person in the following image resembles a:
Pharaoh_hound
```

```
The person in the following image resembles a:
Chihuahua
```

There are not humans or dogs in the following image



```
The person in the following image resembles a:
Pharaoh_hound
```