

Artificial Intelligence Nanodegree

Convolutional Neural Networks

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 "\n", **"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!

The Road Ahead

We break the notebook into separate steps. Feel free to use the links below to navigate the notebook.

- [Step 0](#): Import Datasets
- [Step 1](#): Detect Humans
- [Step 2](#): Detect Dogs
- [Step 3](#): Create a CNN to Classify Dog Breeds (from Scratch)
- [Step 4](#): Use a CNN to Classify Dog Breeds (using Transfer Learning)
- [Step 5](#): Create a CNN to Classify Dog Breeds (using Transfer Learning)
- [Step 6](#): Write your Algorithm
- [Step 7](#): Test Your Algorithm

Step 0: Import Datasets

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, test_files])))
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.

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/**/*.jpg"))
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](http://docs.opencv.org/trunk/d7/d8b/tutorial_py_face_detection.html) (http://docs.opencv.org/trunk/d7/d8b/tutorial_py_face_detection.html) to detect human faces in images. OpenCV provides many pre-trained face detectors, stored as XML files on [github](https://github.com/opencv/opencv/tree/master/data/haarcascades) (<https://github.com/opencv/opencv/tree/master/data/haarcascades>). 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)

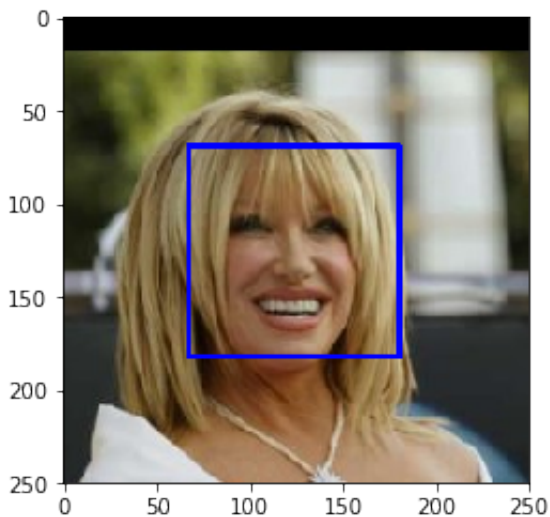
# print number of faces detected in the image
print('Number of faces detected:', len(faces))

# 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.

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
```

(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: The face detector did an excellent job of finding human faces; It found faces on 100% of the human files. On the dog files though, we expected 0 but we see 12 faces out of 100 or 12%.

In [5]:

```
%pip install progressbar
```

Defaulting to user installation because normal site-packages is not writeable

Requirement already satisfied: progressbar in /home/jupyter/.local/lib/python3.5/site-packages (2.5)

Note: you may need to restart the kernel to use updated packages.

In [6]:

```
import progressbar
```

```
human_files[0]
```

Out[6]:

```
'lfw/Javier_Vargas/Javier_Vargas_0001.jpg'
```

In [7]:

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

bar = progressbar.ProgressBar(maxval=100, widgets=[progressbar.Bar('=', '[', ']'),
    ' ', progressbar.Percentage()])
bar.start()

## Test the performance of the face_detector algorithm
human_face_Score = 0
dog_face_Score = 0

for i in range(0,100):
    bar.update(i)
    if face_detector(human_files_short[i]) > 0:
        human_face_Score += 1

    if face_detector(dog_files_short[i]) > 0:
        dog_face_Score += 1

bar.finish()

print("Out of the 100 human face files, we found ", human_face_Score, " faces.")
print("Out of the 100 dog files, we found ", dog_face_Score, " faces.")

## on the images in human_files_short and dog_files_short.
```

```
[=====
====] 100%
```

```
Out of the 100 human face files, we found 100 faces.
Out of the 100 dog files, we found 12 faces.
```


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 unnecessarily 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: It really depends on the application. For example, if the user is actually taking actions to submit an image (e.g. Application form of some sort), it would be quite reasonable to ask users to only provide images of a clear view. If the application is more automated, where we need to detect faces from pictures or videos from various daily activities, then it would not be reasonable to expect clear view of faces. To help detect images that does not necessitate an image with a clearly presented face, we can add additional training images to the set where the face is not clearly presented also.

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 [8]:

```
## (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](#) (<http://ethereon.github.io/netscope/#/gist/db945b393d40bfa26006>) 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](#) (<http://www.image-net.org/>), 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](#) (<https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a>). 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 [9]:

```
from keras.applications.resnet50 import ResNet50

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

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{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}, 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 [10]:

```
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)
```

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](https://github.com/fchollet/keras/blob/master/keras/applications/imagenet_utils.py) (https://github.com/fchollet/keras/blob/master/keras/applications/imagenet_utils.py).

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](https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a) (<https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a>).

In [11]:

```
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))
```

Write a Dog Detector

While looking at the [dictionary](https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a) (<https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a>), 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 [12]:

```
### 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))
```

(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: The dog detector performed great. Only 1 out of 100 (1%) of human faces did we erroneously detect a dog, and it detected dog faces in 100% of the dog pics.

In [13]:

```

### Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.
bar = progressbar.ProgressBar(maxval=100, widgets=[progressbar.Bar('=', '[', ']')
), ' ', progressbar.Percentage()])
bar.start()

## Test the performance of the face_detector algorithm
human_face_Score = 0
dog_face_Score = 0

for i in range(0,100):
    bar.update(i)
    if dog_detector(human_files_short[i]) > 0:
        human_face_Score += 1

    if dog_detector(dog_files_short[i]) > 0:
        dog_face_Score += 1

bar.finish()

print("Out of the 100 human face files, we found ", human_face_Score, " dogs.")
print("Out of the 100 dog files, we found ", dog_face_Score, " dogs.")

[=====
====] 100%

Out of the 100 human face files, we found  1  dogs.
Out of the 100 dog files, we found  100  dogs.

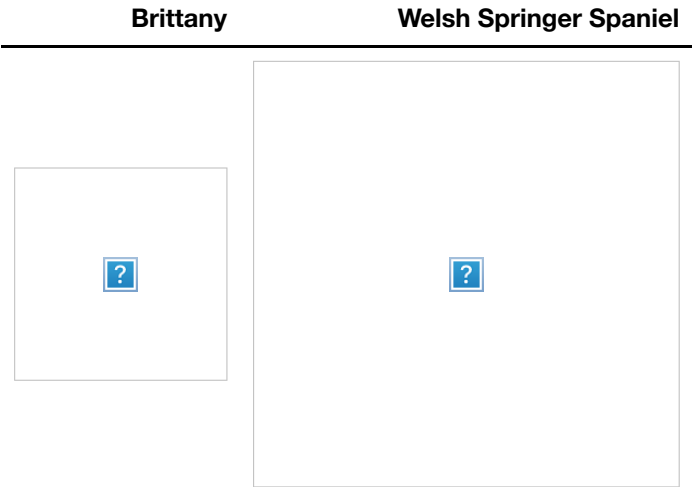
```

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

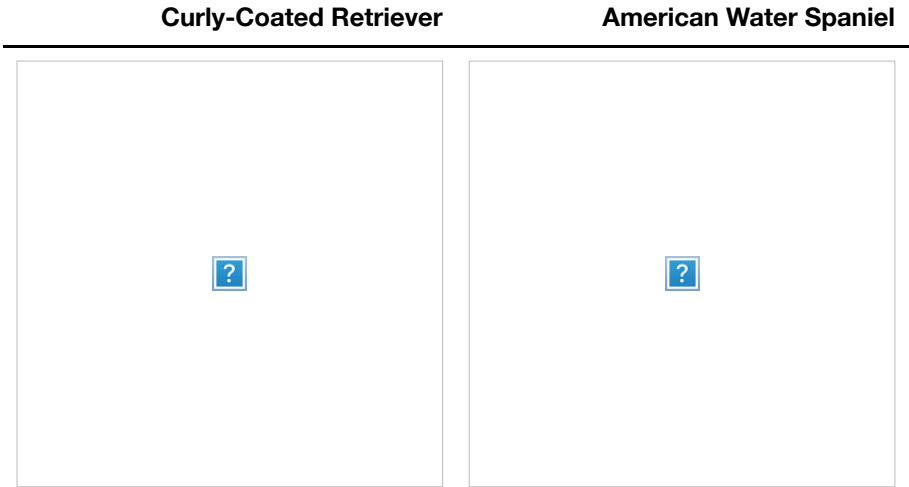
Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN that classifies dog breeds. You must create your CNN *from scratch* (so, you can't use transfer learning yet!), and you must attain a test accuracy of at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

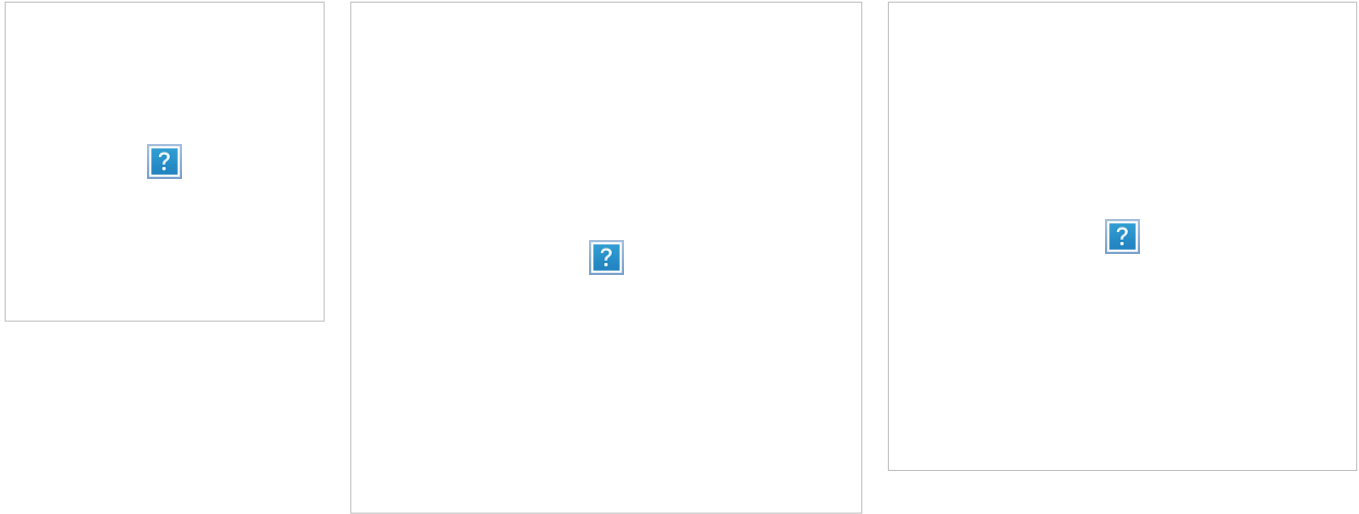


It is not difficult to find other dog breed pairs with minimal inter-class variation (for instance, Curly-Coated Retrievers and American Water Spaniels).



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.





We also mention that random chance presents an exceptionally low bar: setting aside the fact that the classes are slightly imbalanced, 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!

Pre-process the Data

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

In [14]:

```
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 [00:31<00:00, 213.64it/s]
100%|██████████| 835/835 [00:03<00:00, 241.58it/s]
100%|██████████| 836/836 [00:03<00:00, 248.63it/s]
```

(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: As we've learned in the class lectures, fully connected Multi layer perceptron neural network works for simple image classification, but due to the high number of parameters it needs to train, it has the drawback of resource intensive. For detecting the dog breed images which are more sophisticated and has tons of different features in different areas, we will start with a convolution layer. And also as seen in the examples in the lectures, we will need at least a few layers with filters for simpler lines or shapes, to eventually layers with filters for more sophisticated facial features.

After each convolution layer, we add pooling layers to reduce over-fitting.

In [15]:

```
from keras.layers import Conv2D, MaxPooling2D, GlobalAveragePooling2D
from keras.layers import Dropout, Flatten, Dense
from keras.models import Sequential

model = Sequential()

### Define the architecture based on the sample
model.add(Conv2D(filters=16, kernel_size=2, padding='same', activation='relu', input_shape=(224,224,3)))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=32, kernel_size=2, padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=16, kernel_size=2, padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(GlobalAveragePooling2D())
model.add(Dense(133, activation='softmax'))

model.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
=====		
conv2d_1 (Conv2D)	(None, 224, 224, 16)	208
max_pooling2d_2 (MaxPooling2D)	(None, 112, 112, 16)	0
conv2d_2 (Conv2D)	(None, 112, 112, 32)	2080
max_pooling2d_3 (MaxPooling2D)	(None, 56, 56, 32)	0
conv2d_3 (Conv2D)	(None, 56, 56, 16)	2064
max_pooling2d_4 (MaxPooling2D)	(None, 28, 28, 16)	0
global_average_pooling2d_1 (GlobalAveragePooling2D)	(None, 16)	0
dense_1 (Dense)	(None, 133)	2261
=====		
Total params: 6,613		
Trainable params: 6,613		
Non-trainable params: 0		

Compile the Model

In [16]:

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

(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 \(https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html\)](https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html), but this is not a requirement.

In [17]:

```
from keras.callbacks import ModelCheckpoint

### 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

6680/6680 [=====] - 12s 2ms/step - loss: 4.8872 - accuracy: 0.0103 - val_loss: 4.8757 - val_accuracy: 0.0096

Epoch 00001: val_loss improved from inf to 4.87571, saving model to saved_models/weights.best.from_scratch.hdf5

Epoch 2/20

6680/6680 [=====] - 12s 2ms/step - loss: 4.8724 - accuracy: 0.0115 - val_loss: 4.8654 - val_accuracy: 0.0108

Epoch 00002: val_loss improved from 4.87571 to 4.86536, saving model to saved_models/weights.best.from_scratch.hdf5

Epoch 3/20

6680/6680 [=====] - 12s 2ms/step - loss: 4.8604 - accuracy: 0.0139 - val_loss: 4.8576 - val_accuracy: 0.0132

Epoch 00003: val_loss improved from 4.86536 to 4.85761, saving model to saved_models/weights.best.from_scratch.hdf5
Epoch 4/20
6680/6680 [=====] - 12s 2ms/step - loss: 4.8381 - accuracy: 0.0169 - val_loss: 4.8343 - val_accuracy: 0.0180

Epoch 00004: val_loss improved from 4.85761 to 4.83426, saving model to saved_models/weights.best.from_scratch.hdf5
Epoch 5/20
6680/6680 [=====] - 12s 2ms/step - loss: 4.8099 - accuracy: 0.0160 - val_loss: 4.8194 - val_accuracy: 0.0240

Epoch 00005: val_loss improved from 4.83426 to 4.81944, saving model to saved_models/weights.best.from_scratch.hdf5
Epoch 6/20
6680/6680 [=====] - 12s 2ms/step - loss: 4.7870 - accuracy: 0.0178 - val_loss: 4.8063 - val_accuracy: 0.0204

Epoch 00006: val_loss improved from 4.81944 to 4.80635, saving model to saved_models/weights.best.from_scratch.hdf5
Epoch 7/20
6680/6680 [=====] - 12s 2ms/step - loss: 4.7657 - accuracy: 0.0199 - val_loss: 4.7876 - val_accuracy: 0.0192

Epoch 00007: val_loss improved from 4.80635 to 4.78760, saving model to saved_models/weights.best.from_scratch.hdf5
Epoch 8/20
6680/6680 [=====] - 12s 2ms/step - loss: 4.7474 - accuracy: 0.0210 - val_loss: 4.7748 - val_accuracy: 0.0240

Epoch 00008: val_loss improved from 4.78760 to 4.77479, saving model to saved_models/weights.best.from_scratch.hdf5
Epoch 9/20
6680/6680 [=====] - 12s 2ms/step - loss: 4.7293 - accuracy: 0.0219 - val_loss: 4.7659 - val_accuracy: 0.0180

Epoch 00009: val_loss improved from 4.77479 to 4.76591, saving model to saved_models/weights.best.from_scratch.hdf5
Epoch 10/20
6680/6680 [=====] - 12s 2ms/step - loss: 4.7112 - accuracy: 0.0217 - val_loss: 4.7500 - val_accuracy: 0.0168

Epoch 00010: val_loss improved from 4.76591 to 4.75003, saving model to saved_models/weights.best.from_scratch.hdf5
Epoch 11/20
6680/6680 [=====] - 12s 2ms/step - loss: 4.7004 - accuracy: 0.0257 - val_loss: 4.7508 - val_accuracy: 0.0204

Epoch 00011: val_loss did not improve from 4.75003
Epoch 12/20

6680/6680 [=====] - 12s 2ms/step - loss: 4.6855 - accuracy: 0.0269 - val_loss: 4.7612 - val_accuracy: 0.0228

Epoch 00012: val_loss did not improve from 4.75003
Epoch 13/20
6680/6680 [=====] - 12s 2ms/step - loss: 4.6750 - accuracy: 0.0295 - val_loss: 4.7268 - val_accuracy: 0.0204

Epoch 00013: val_loss improved from 4.75003 to 4.72677, saving model to saved_models/weights.best.from_scratch.hdf5
Epoch 14/20
6680/6680 [=====] - 12s 2ms/step - loss: 4.6630 - accuracy: 0.0301 - val_loss: 4.7236 - val_accuracy: 0.0251

Epoch 00014: val_loss improved from 4.72677 to 4.72357, saving model to saved_models/weights.best.from_scratch.hdf5
Epoch 15/20
6680/6680 [=====] - 12s 2ms/step - loss: 4.6570 - accuracy: 0.0307 - val_loss: 4.7526 - val_accuracy: 0.0263

Epoch 00015: val_loss did not improve from 4.72357
Epoch 16/20
6680/6680 [=====] - 12s 2ms/step - loss: 4.6416 - accuracy: 0.0326 - val_loss: 4.7186 - val_accuracy: 0.0192

Epoch 00016: val_loss improved from 4.72357 to 4.71861, saving model to saved_models/weights.best.from_scratch.hdf5
Epoch 17/20
6680/6680 [=====] - 12s 2ms/step - loss: 4.6324 - accuracy: 0.0352 - val_loss: 4.7352 - val_accuracy: 0.0287

Epoch 00017: val_loss did not improve from 4.71861
Epoch 18/20
6680/6680 [=====] - 12s 2ms/step - loss: 4.6230 - accuracy: 0.0335 - val_loss: 4.7144 - val_accuracy: 0.0275

Epoch 00018: val_loss improved from 4.71861 to 4.71442, saving model to saved_models/weights.best.from_scratch.hdf5
Epoch 19/20
6680/6680 [=====] - 12s 2ms/step - loss: 4.6118 - accuracy: 0.0401 - val_loss: 4.7117 - val_accuracy: 0.0240

Epoch 00019: val_loss improved from 4.71442 to 4.71165, saving model to saved_models/weights.best.from_scratch.hdf5
Epoch 20/20
6680/6680 [=====] - 12s 2ms/step - loss: 4.6037 - accuracy: 0.0392 - val_loss: 4.6822 - val_accuracy: 0.0263

Epoch 00020: val_loss improved from 4.71165 to 4.68221, saving model to saved_models/weights.best.from_scratch.hdf5

Out[17]:

```
<keras.callbacks.callbacks.History at 0x7efe3041ec18>
```

Load the Model with the Best Validation Loss

In [18]:

```
model.load_weights('saved_models/weights.best.from_scratch.hdf5')
```

Test the Model

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

In [19]:

```
# 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 tensor in test_tensors]

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

Test accuracy: 4.1866%

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.

Obtain Bottleneck Features

In [20]:

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

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 [21]:

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

VGG16_model.summary()
```

Model: "sequential_2"

Layer (type)	Output Shape	Param #
global_average_pooling2d_2 ((None, 512)		0
dense_2 (Dense)	(None, 133)	68229
Total params: 68,229		
Trainable params: 68,229		
Non-trainable params: 0		

Compile the Model

In [22]:

```
VGG16_model.compile(loss='categorical_crossentropy', optimizer='rmsprop', metrics=['accuracy'])
```

Train the Model

In [23]:

```
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 830 samples

Epoch 1/20

6680/6680 [=====] - 2s 226us/step - loss: 7.6849 - accuracy: 0.2331 - val_loss: 3.3543 - val_accuracy: 0.4419

Epoch 00001: val_loss improved from inf to 3.35429, saving model to saved_models/weights.best.VGG16.hdf5

Epoch 2/20

6680/6680 [=====] - 1s 213us/step - loss: 2.1757 - accuracy: 0.6000 - val_loss: 2.4155 - val_accuracy: 0.5928

Epoch 00002: val_loss improved from 3.35429 to 2.41548, saving model to saved_models/weights.best.VGG16.hdf5

Epoch 3/20

6680/6680 [=====] - 1s 209us/step - loss: 1.2362 - accuracy: 0.7415 - val_loss: 2.0150 - val_accuracy: 0.6467

Epoch 00003: val_loss improved from 2.41548 to 2.01503, saving model to saved_models/weights.best.VGG16.hdf5

Epoch 4/20

6680/6680 [=====] - 1s 214us/step - loss: 0.7838 - accuracy: 0.8211 - val_loss: 2.0559 - val_accuracy: 0.6671

Epoch 00004: val_loss did not improve from 2.01503

Epoch 5/20

6680/6680 [=====] - 1s 211us/step - loss: 0.5708 - accuracy: 0.8615 - val_loss: 1.7644 - val_accuracy: 0.6970

Epoch 00005: val_loss improved from 2.01503 to 1.76438, saving model to saved_models/weights.best.VGG16.hdf5

Epoch 6/20

6680/6680 [=====] - 1s 207us/step - loss: 0.4004 - accuracy: 0.8916 - val_loss: 1.7614 - val_accuracy: 0.7126

Epoch 00006: val_loss improved from 1.76438 to 1.76138, saving model to saved_models/weights.best.VGG16.hdf5

Epoch 7/20

6680/6680 [=====] - 1s 213us/step - loss: 0.3026 - accuracy: 0.9186 - val_loss: 1.7606 - val_accuracy: 0.7317

Epoch 00007: val_loss improved from 1.76138 to 1.76056, saving model to saved_models/weights.best.VGG16.hdf5

Epoch 8/20

6680/6680 [=====] - 1s 213us/step - loss: 0.2293 - accuracy: 0.9344 - val_loss: 1.8998 - val_accuracy: 0.6994

Epoch 00008: val_loss did not improve from 1.76056

Epoch 9/20

6680/6680 [=====] - 1s 210us/step - loss: 0.1680 - accuracy: 0.9484 - val_loss: 1.8850 - val_accuracy: 0.7317

Epoch 00009: val_loss did not improve from 1.76056
Epoch 10/20
6680/6680 [=====] - 1s 213us/step - loss: 0.1361 - accuracy: 0.9608 - val_loss: 1.9635 - val_accuracy: 0.7305

Epoch 00010: val_loss did not improve from 1.76056
Epoch 11/20
6680/6680 [=====] - 1s 211us/step - loss: 0.1029 - accuracy: 0.9699 - val_loss: 1.9441 - val_accuracy: 0.7353

Epoch 00011: val_loss did not improve from 1.76056
Epoch 12/20
6680/6680 [=====] - 1s 212us/step - loss: 0.0787 - accuracy: 0.9743 - val_loss: 1.9695 - val_accuracy: 0.7210

Epoch 00012: val_loss did not improve from 1.76056
Epoch 13/20
6680/6680 [=====] - 1s 212us/step - loss: 0.0671 - accuracy: 0.9783 - val_loss: 1.7971 - val_accuracy: 0.7497

Epoch 00013: val_loss did not improve from 1.76056
Epoch 14/20
6680/6680 [=====] - 1s 211us/step - loss: 0.0566 - accuracy: 0.9832 - val_loss: 1.8758 - val_accuracy: 0.7425

Epoch 00014: val_loss did not improve from 1.76056
Epoch 15/20
6680/6680 [=====] - 1s 209us/step - loss: 0.0445 - accuracy: 0.9873 - val_loss: 1.8968 - val_accuracy: 0.7485

Epoch 00015: val_loss did not improve from 1.76056
Epoch 16/20
6680/6680 [=====] - 1s 211us/step - loss: 0.0338 - accuracy: 0.9900 - val_loss: 1.9435 - val_accuracy: 0.7569

Epoch 00016: val_loss did not improve from 1.76056
Epoch 17/20
6680/6680 [=====] - 1s 215us/step - loss: 0.0304 - accuracy: 0.9916 - val_loss: 2.1784 - val_accuracy: 0.7281

Epoch 00017: val_loss did not improve from 1.76056
Epoch 18/20
6680/6680 [=====] - 1s 209us/step - loss: 0.0250 - accuracy: 0.9921 - val_loss: 2.0573 - val_accuracy: 0.7485

Epoch 00018: val_loss did not improve from 1.76056
Epoch 19/20
6680/6680 [=====] - 1s 214us/step - loss: 0.0280 - accuracy: 0.9918 - val_loss: 2.0959 - val_accuracy: 0.7473


```
Epoch 00019: val_loss did not improve from 1.76056
Epoch 20/20
6680/6680 [=====] - 1s 212us/step - loss: 0
.0198 - accuracy: 0.9951 - val_loss: 2.1164 - val_accuracy: 0.7533
```

```
Epoch 00020: val_loss did not improve from 1.76056
```

```
Out[23]:
```

```
<keras.callbacks.callbacks.History at 0x7efe3037fa58>
```

Load the Model with the Best Validation Loss

```
In [24]:
```

```
VGG16_model.load_weights('saved_models/weights.best.VGG16.hdf5')
```

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 [25]:
```

```
# 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))) for feature in test_VGG16]

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

```
Test accuracy: 71.8900%
```

Predict Dog Breed with the Model

In [26]:

```
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)]
```

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 \(https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/DogVGG19Data.npz\)](https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/DogVGG19Data.npz)
bottleneck features
- [ResNet-50 \(https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/DogResnet50Data.npz\)](https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/DogResnet50Data.npz)
bottleneck features
- [Inception \(https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/DogInceptionV3Data.npz\)](https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/DogInceptionV3Data.npz)
bottleneck features
- [Xception \(https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/DogXceptionData.npz\)](https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/DogXceptionData.npz)
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.

(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 [27]:

```
### Obtain bottleneck features from another pre-trained CNN.
bottleneck19_features = np.load('bottleneck_features/DogVGG19Data.npz')
train_VGG19 = bottleneck19_features['train']
valid_VGG19 = bottleneck19_features['valid']
test_VGG19 = bottleneck19_features['test']
```

(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 seeing VGG16 that has pretty good performance at above 70%, I want to see how VGG19, with 3 additional layers, would affect the performance. Similar to VGG16, we want to use transfer learning and keep most of the model and just replace the final layers.

In [28]:

```
### Define your architecture.
VGG19_model = Sequential()
VGG19_model.add(GlobalAveragePooling2D(input_shape=train_VGG19.shape[1:]))
VGG19_model.add(Dense(133, activation='softmax'))

VGG19_model.summary()
```

Model: "sequential_3"

Layer (type)	Output Shape	Param #
=====		
global_average_pooling2d_3 (None, 512)		0
=====		
dense_3 (Dense)	(None, 133)	68229
=====		
Total params: 68,229		
Trainable params: 68,229		
Non-trainable params: 0		
=====		

(IMPLEMENTATION) Compile the Model

In [29]:

```
### Compile the model.
VGG19_model.compile(loss='categorical_crossentropy', optimizer='rmsprop', metrics=['accuracy'])
```

(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 \(https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html\)](https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html), but this is not a requirement.

In [30]:

```
### Train the model.
checkpointer19 = 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=[checkpointer19], verbose=1)
```

Train on 6680 samples, validate on 835 samples

Epoch 1/20

6680/6680 [=====] - 1s 218us/step - loss: 7.2159 - accuracy: 0.2519 - val_loss: 3.5131 - val_accuracy: 0.4503

Epoch 00001: val_loss improved from inf to 3.51306, saving model to saved_models/weights.best.VGG19.hdf5

Epoch 2/20

6680/6680 [=====] - 1s 209us/step - loss: 1.9925 - accuracy: 0.6166 - val_loss: 2.4172 - val_accuracy: 0.5928

Epoch 00002: val_loss improved from 3.51306 to 2.41718, saving model to saved_models/weights.best.VGG19.hdf5

Epoch 3/20

6680/6680 [=====] - 1s 212us/step - loss: 1.1142 - accuracy: 0.7484 - val_loss: 2.1447 - val_accuracy: 0.6383

Epoch 00003: val_loss improved from 2.41718 to 2.14472, saving model to saved_models/weights.best.VGG19.hdf5

Epoch 4/20

6680/6680 [=====] - 1s 209us/step - loss: 0.7548 - accuracy: 0.8243 - val_loss: 1.9702 - val_accuracy: 0.6862

Epoch 00004: val_loss improved from 2.14472 to 1.97020, saving model to saved_models/weights.best.VGG19.hdf5

Epoch 5/20

6680/6680 [=====] - 1s 209us/step - loss: 0.5061 - accuracy: 0.8734 - val_loss: 1.8905 - val_accuracy: 0.6958

Epoch 00005: val_loss improved from 1.97020 to 1.89054, saving model to saved_models/weights.best.VGG19.hdf5

Epoch 6/20

6680/6680 [=====] - 1s 209us/step - loss: 0.3744 - accuracy: 0.8990 - val_loss: 1.7596 - val_accuracy: 0.7162

Epoch 00006: val_loss improved from 1.89054 to 1.75964, saving model to saved_models/weights.best.VGG19.hdf5

Epoch 7/20

6680/6680 [=====] - 1s 208us/step - loss: 0.2821 - accuracy: 0.9174 - val_loss: 1.8685 - val_accuracy: 0.7222

Epoch 00007: val_loss did not improve from 1.75964

Epoch 8/20

6680/6680 [=====] - 1s 210us/step - loss: 0.2148 - accuracy: 0.9401 - val_loss: 1.8563 - val_accuracy: 0.7246

Epoch 00008: val_loss did not improve from 1.75964

Epoch 9/20

6680/6680 [=====] - 1s 211us/step - loss: 0.1525 - accuracy: 0.9528 - val_loss: 1.8576 - val_accuracy: 0.7329

Epoch 00009: val_loss did not improve from 1.75964

Epoch 10/20

6680/6680 [=====] - 1s 211us/step - loss: 0.1157 - accuracy: 0.9645 - val_loss: 1.9129 - val_accuracy: 0.7365

Epoch 00010: val_loss did not improve from 1.75964

Epoch 11/20

6680/6680 [=====] - 1s 214us/step - loss: 0.0913 - accuracy: 0.9723 - val_loss: 1.8924 - val_accuracy: 0.7341

Epoch 00011: val_loss did not improve from 1.75964

Epoch 12/20

6680/6680 [=====] - 1s 212us/step - loss: 0.0684 - accuracy: 0.9781 - val_loss: 1.8888 - val_accuracy: 0.7377

Epoch 00012: val_loss did not improve from 1.75964

Epoch 13/20

6680/6680 [=====] - 2s 230us/step - loss: 0

.0599 - accuracy: 0.9820 - val_loss: 1.9350 - val_accuracy: 0.7329

Epoch 00013: val_loss did not improve from 1.75964

Epoch 14/20

6680/6680 [=====] - 2s 249us/step - loss: 0

.0515 - accuracy: 0.9844 - val_loss: 2.0033 - val_accuracy: 0.7509

Epoch 00014: val_loss did not improve from 1.75964

Epoch 15/20

6680/6680 [=====] - 2s 242us/step - loss: 0

.0386 - accuracy: 0.9876 - val_loss: 2.0630 - val_accuracy: 0.7317

Epoch 00015: val_loss did not improve from 1.75964

Epoch 16/20

6680/6680 [=====] - 2s 244us/step - loss: 0

.0374 - accuracy: 0.9882 - val_loss: 2.0308 - val_accuracy: 0.7437

Epoch 00016: val_loss did not improve from 1.75964

Epoch 17/20

6680/6680 [=====] - 2s 248us/step - loss: 0

.0295 - accuracy: 0.9915 - val_loss: 2.0871 - val_accuracy: 0.7257

Epoch 00017: val_loss did not improve from 1.75964

Epoch 18/20

6680/6680 [=====] - 2s 242us/step - loss: 0

.0251 - accuracy: 0.9919 - val_loss: 2.0683 - val_accuracy: 0.7557

Epoch 00018: val_loss did not improve from 1.75964

Epoch 19/20

6680/6680 [=====] - 2s 231us/step - loss: 0

.0226 - accuracy: 0.9933 - val_loss: 2.0495 - val_accuracy: 0.7497

Epoch 00019: val_loss did not improve from 1.75964

Epoch 20/20

6680/6680 [=====] - 1s 210us/step - loss: 0

.0191 - accuracy: 0.9942 - val_loss: 2.0873 - val_accuracy: 0.7497

Epoch 00020: val_loss did not improve from 1.75964

Out[30]:

<keras.callbacks.callbacks.History at 0x7efe302d6e10>

(IMPLEMENTATION) Load the Model with the Best Validation Loss

In [31]:

```
### Load the model weights with the best validation loss.
VGG19_model.load_weights('saved_models/weights.best.VGG19.hdf5')
```

(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 [32]:

```
### 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))) for feature in test_VGG19]

# report test accuracy
test_accuracy = 100*np.sum(np.array(VGG19_predictions)==np.argmax(test_targets,
axis=1))/len(VGG19_predictions)
print('Test accuracy: %.4f%%' % test_accuracy)
```

Test accuracy: 73.3254%

(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 [33]:

```
### Write a function that takes a path to an image as input
### and returns the dog breed that is predicted by the model.

def VGG19_predict_dogbreed(imgPath):
    bottleneck_feature = extract_VGG19(path_to_tensor(imgPath))
    predicted_vector = VGG19_model.predict(bottleneck_feature)
    return dog_names[np.argmax(predicted_vector)]
```

In [34]:

```
from IPython.display import Image
```

In [35]:

```
testImage = "dogImages/test/123.Pomeranian/Pomeranian_07861.jpg"
Image(filename=testImage)
```

Out[35]:



In [36]:

```
breed = VGG19_predict_dogbreed(testImage)
print("This pic's breed is: ", breed)
```

This pic's breed is: Pomeranian

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!



(IMPLEMENTATION) Write your Algorithm

In [37]:

```
### Method to predict dog breed, or what a human resembles

from IPython.display import Image

def which_dog_are_you(imgPath):
    #Detect dog if we can find one:
    print("Testing file name: ", imgPath)

    if dog_detector(imgPath):
        #Show the dog picture and return breed name
        img = cv2.imread(imgPath)
        cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        plt.imshow(cv_rgb)
        plt.show()

        breed = VGG19_predict_dogbreed(imgPath)
        print("Found a dog. The predicted breed is: ", breed)
        return breed

    else:
        #Detect and print the face if we found one:
        img = cv2.imread(imgPath)
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        faces = face_cascade.detectMultiScale(gray)
        # 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()

        if len(faces) > 0:
            breed = VGG19_predict_dogbreed(imgPath)
            print("Found a human face. It most resembles: ", breed)
            return breed
        else:
            print("Sorry, I didn't find any human or dog faces.")
```

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?

(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.

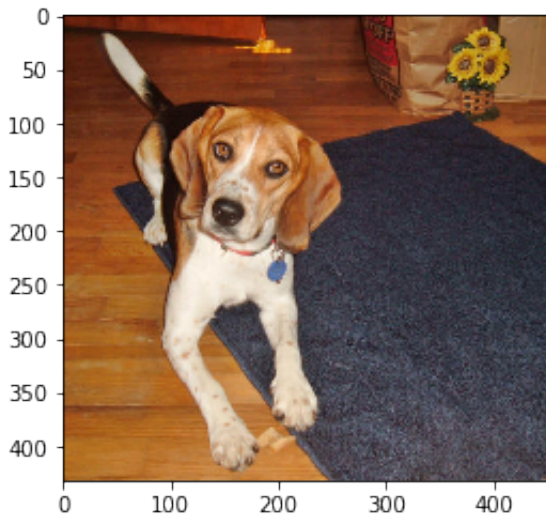
Answer: The output is similar to what I expected. First of all, the human vs dog face detection works quite well, with real human and real dog pictures. The dog breed itself, as expected is a hard problem as many dogs have similar features, and if the pictures are small, it's very difficult to tell what breed it is. Overallly it did come up with relatively similar breeds though. Some possible ways to improve algorithm:

1. It may help with the prediction if we keep the pictures at a larger size than 224x224
2. It will require expensive GPU times, but it would be better if we can actually include models from multiple algorithms or optimizers.
3. Some images actually have multiple dogs / humans. The algorithm didn't quite define how to handle these. The most sophisticate way may be to identify both and pass the sub image to the breed classification algorithms

In [38]:

```
## Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.
## Quick unit tests
which_dog_are_you("dogImages/test/016.Beagle/Beagle_01130.jpg")
# which_dog_are_you("lfw/Britney_Spears/Britney_Spears_0001.jpg")
```

Testing file name: dogImages/test/016.Beagle/Beagle_01130.jpg



Found a dog. The predicted breed is: Beagle

Out[38]:

'Beagle'

In [39]:

```
#Test all the files in test_images
import os

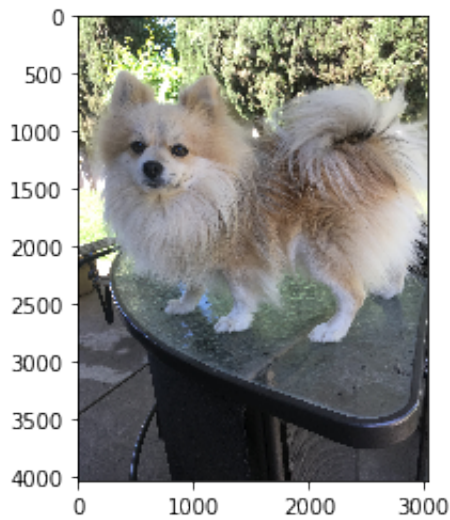
testFolder = "test_images"
listOfFiles = os.listdir(testFolder)
for entry in listOfFiles:
    if entry.endswith(".jpg"):
        print (which_dog_are_you(testFolder + "/" + entry))
        print ("\n\n")
```

Testing file name: test_images/human_cat_taylor_swift.jpg



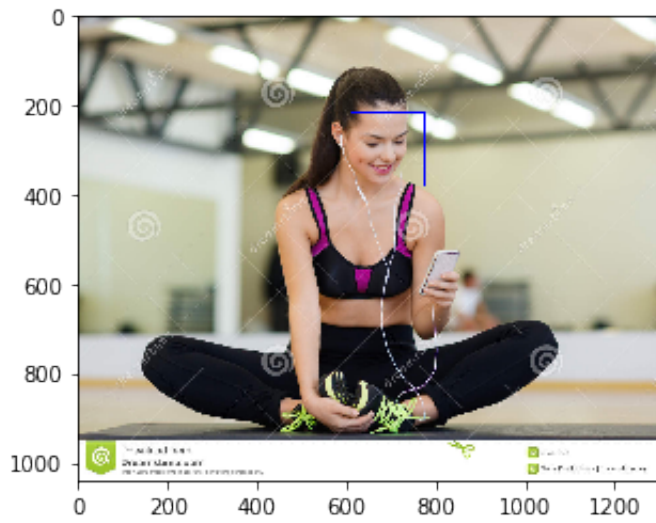
Found a dog. The predicted breed is: Italian_greyhound
Italian_greyhound

Testing file name: test_images/dog_Pomeranian_Snowball_parents2.jpg



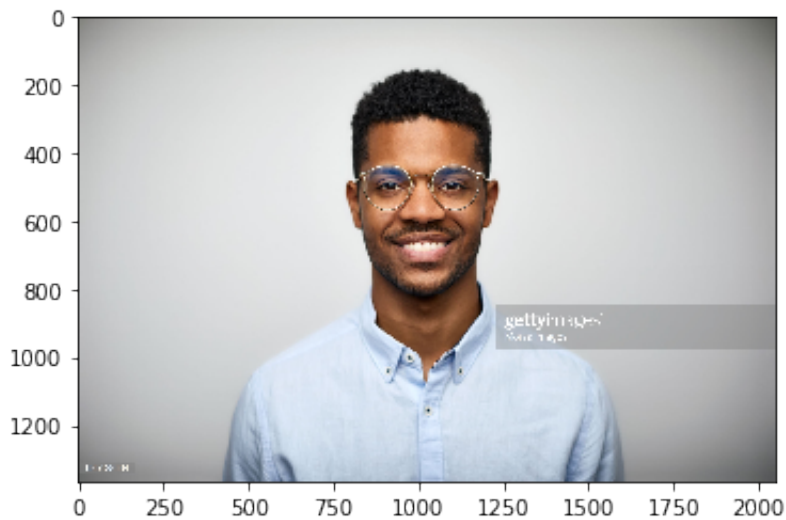
Found a dog. The predicted breed is: Pomeranian
Pomeranian

Testing file name: test_images/human_gym_woman.jpg



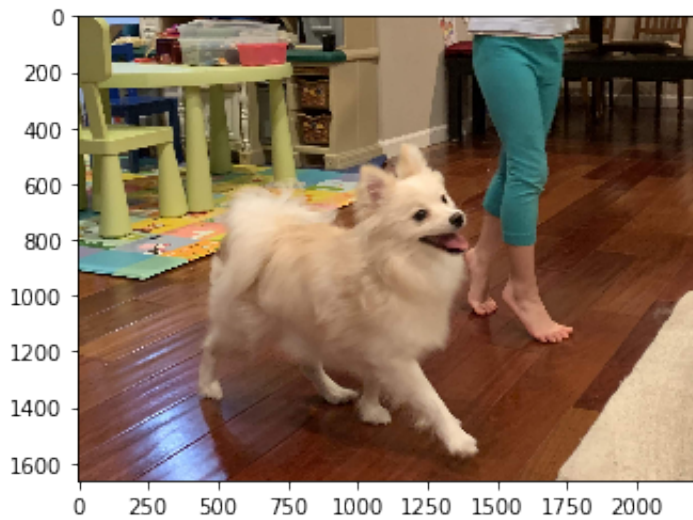
Found a human face. It most resembles: Xoloitzcuintli
Xoloitzcuintli

Testing file name: test_images/human_man-gettyimages-985138634.jpg



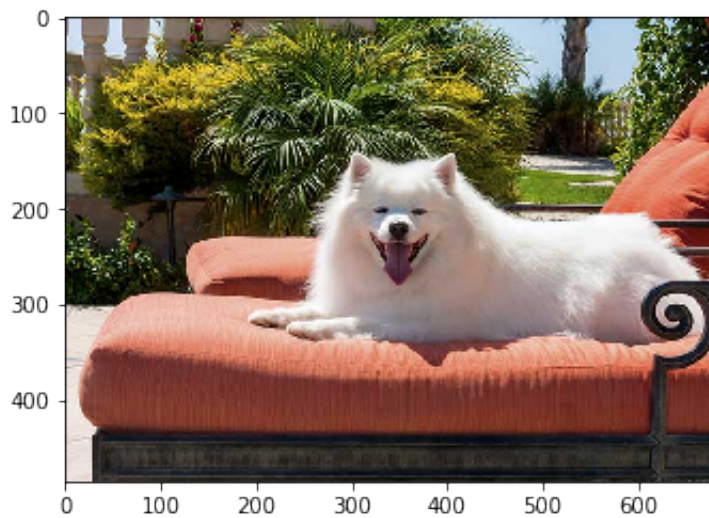
Found a human face. It most resembles: Norwegian_lundehund
Norwegian_lundehund

Testing file name: test_images/dog_Pomeranian_Snowball_2.jpg



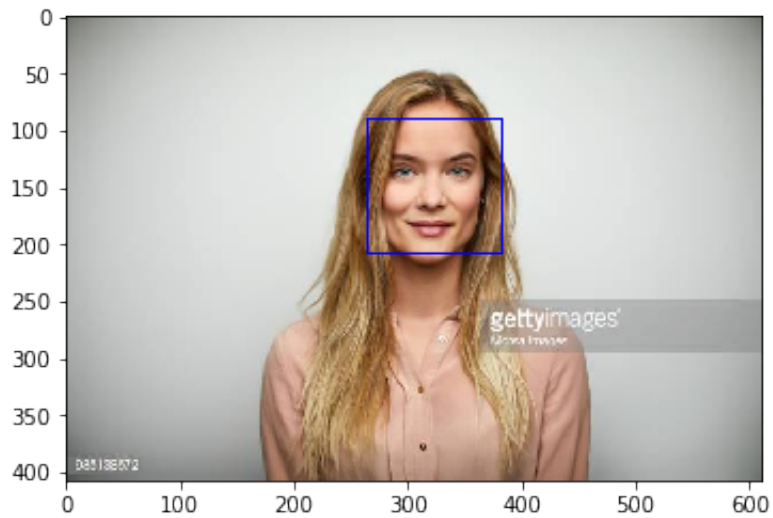
Found a dog. The predicted breed is: Norwegian_buhund
Norwegian_buhund

Testing file name: test_images/dog_American_Eskimo.jpg



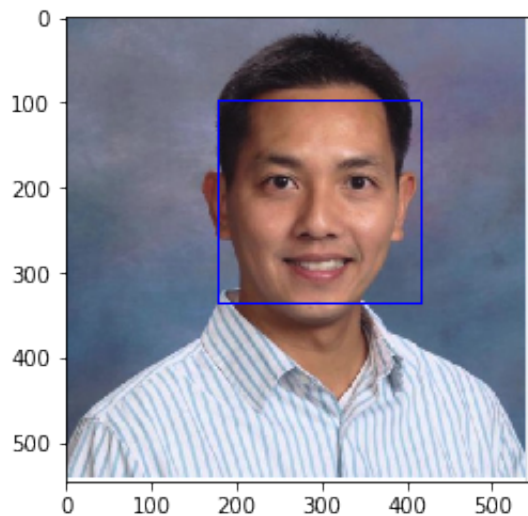
Found a dog. The predicted breed is: American_eskimo_dog
American_eskimo_dog

Testing file name: test_images/human_woman-gettyimages-985138672.jpg
g



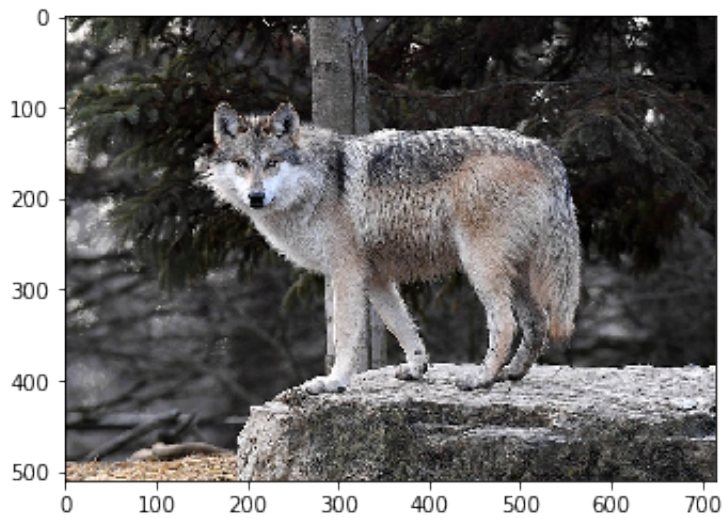
Found a human face. It most resembles: Yorkshire_terrier
Yorkshire_terrier

Testing file name: test_images/human_Ronald_Leung.jpg



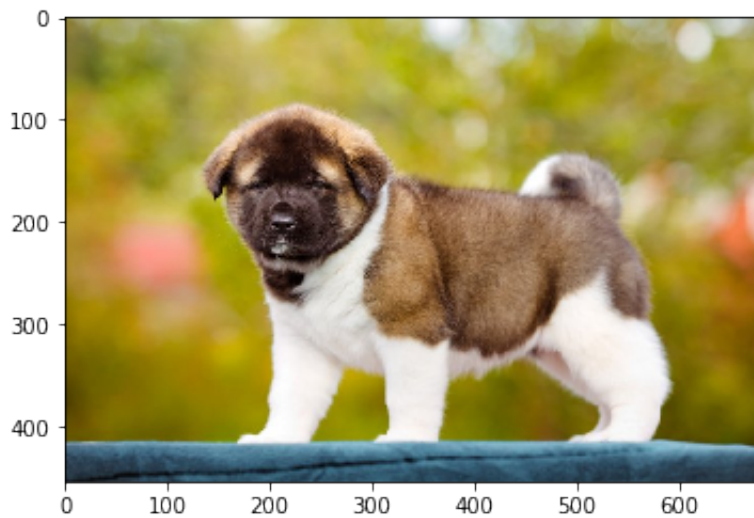
Found a human face. It most resembles: Japanese_chin
Japanese_chin

Testing file name: test_images/wolf_mexican_grey_wolf.jpg



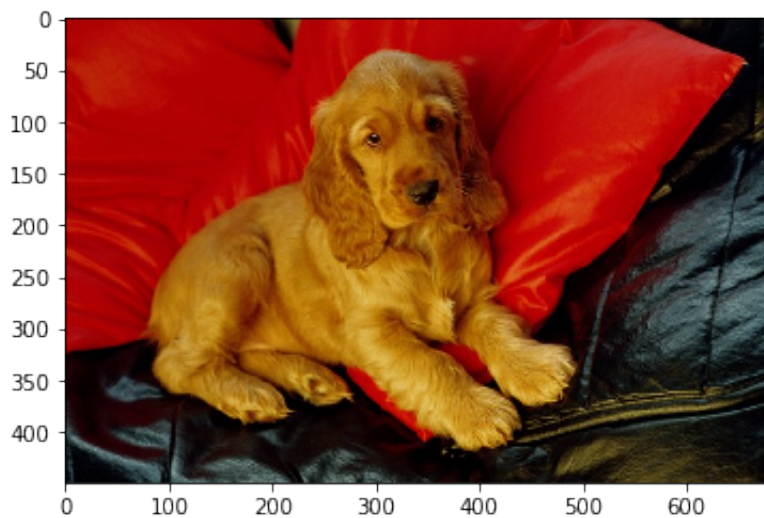
Sorry, I didn't find any human or dog faces.
None

Testing file name: test_images/dog_akita.jpg



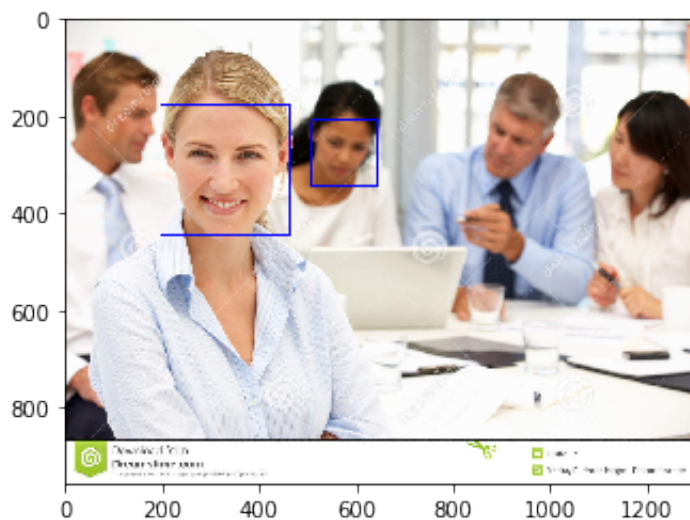
Found a dog. The predicted breed is: Akita
Akita

Testing file name: test_images/dog_Cocker_Spaniel.jpg



Found a dog. The predicted breed is: Chesapeake_bay_retriever
Chesapeake_bay_retriever

Testing file name: test_images/human_office.jpg



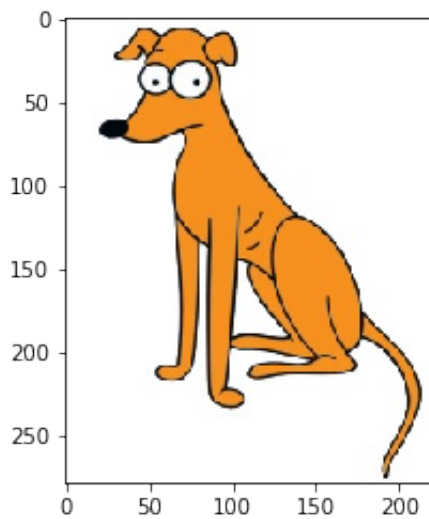
Found a human face. It most resembles: Cardigan_welsh_corgi
Cardigan_welsh_corgi

Testing file name: test_images/human_santa.jpg



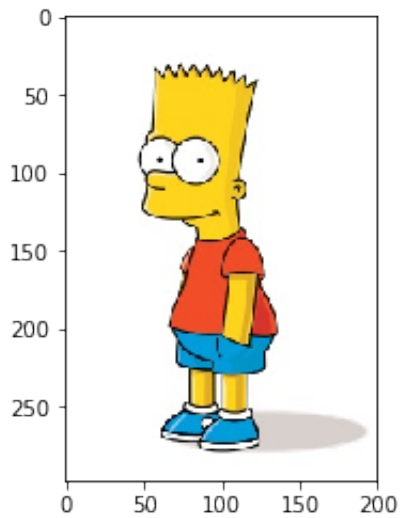
Found a human face. It most resembles: Great_dane
Great_dane

Testing file name: test_images/anime_Simpsons_Santa_Little_Helper.j
pg



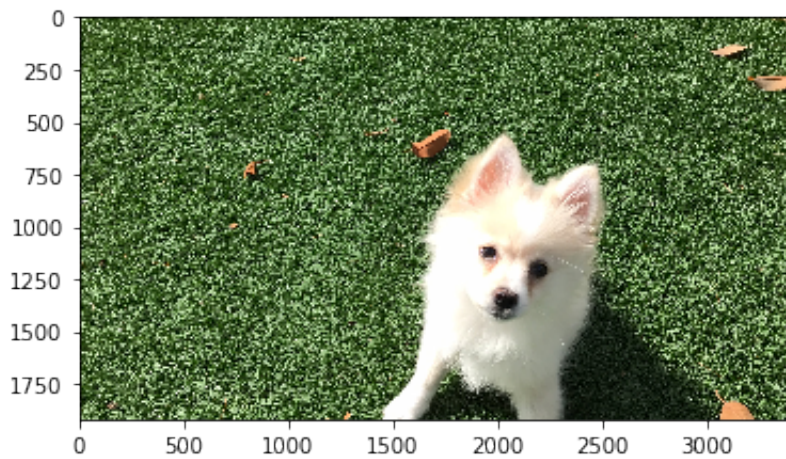
Sorry, I didn't find any human or dog faces.
None

Testing file name: test_images/anime_Bart_Simpson_200px.jpg



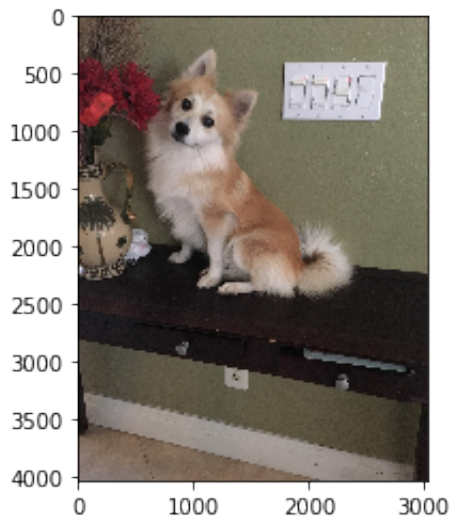
Sorry, I didn't find any human or dog faces.
None

Testing file name: test_images/dog_Pomeranian_Snowball_1.jpg



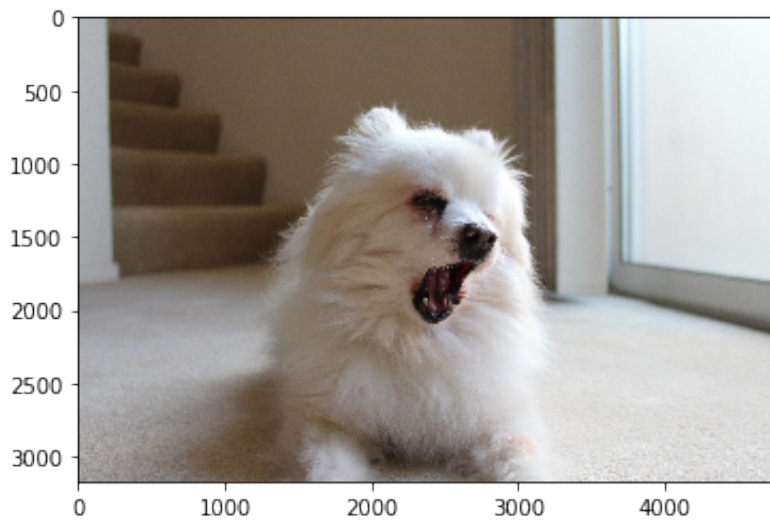
Found a dog. The predicted breed is: American_eskimo_dog
American_eskimo_dog

Testing file name: test_images/dog_Pomeranian_Snowball_parents1.jpg



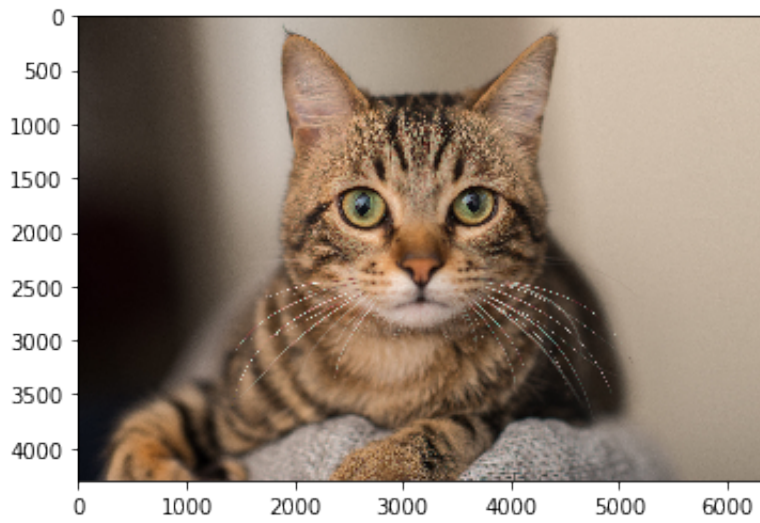
Found a dog. The predicted breed is: Norwegian_buhund
Norwegian_buhund

Testing file name: test_images/dog_Pomeranian_fluffy2.jpg



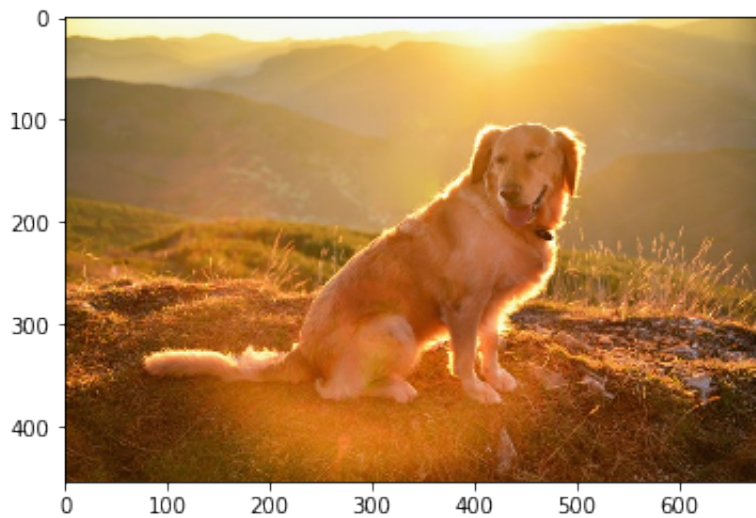
Found a dog. The predicted breed is: Bichon_frise
Bichon_frise

Testing file name: test_images/cat.jpg



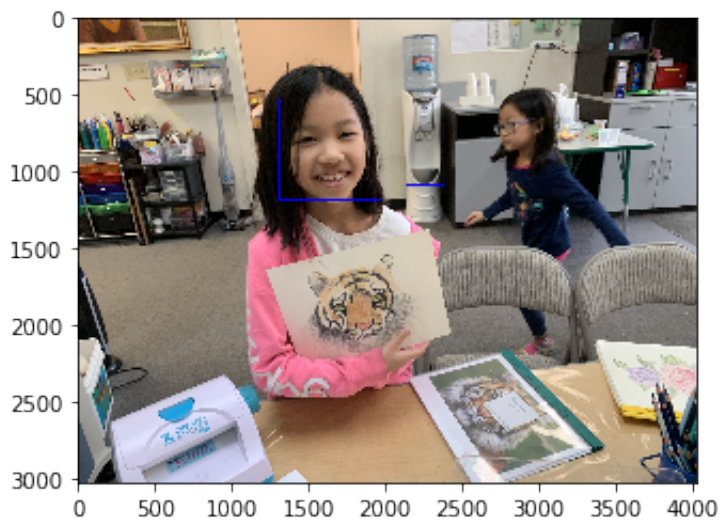
Sorry, I didn't find any human or dog faces.
None

Testing file name: test_images/dog_Golden_Retriever.jpg



Found a dog. The predicted breed is: Chesapeake_bay_retriever
Chesapeake_bay_retriever

Testing file name: test_images/human_Alyssa.jpg



Found a human face. It most resembles: Chihuahua
Chihuahua

Testing file name: test_images/young-guy-sitting-panel-hugging-retriever-dog-isolated-white-background-172415241.jpg



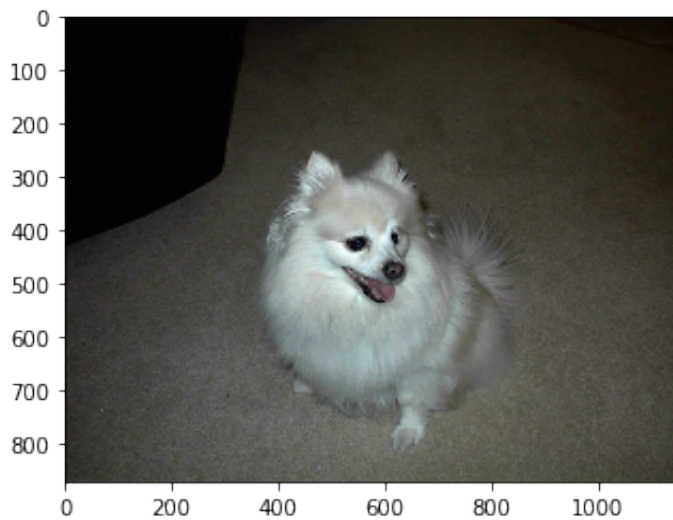
Found a dog. The predicted breed is: Anatolian_shepherd_dog
Anatolian_shepherd_dog

Testing file name: test_images/dog_Beagle.jpg



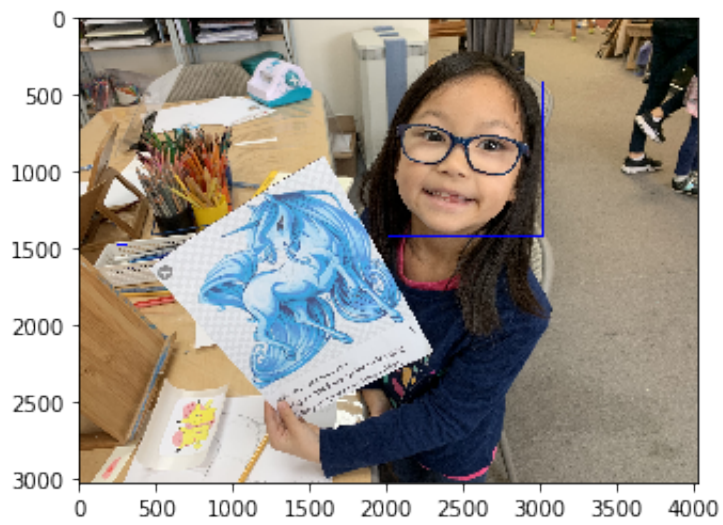
Found a dog. The predicted breed is: German_pinscher
German_pinscher

Testing file name: test_images/dog_Pomeranian_fluffy.jpg



Found a dog. The predicted breed is: American_eskimo_dog
American_eskimo_dog

Testing file name: test_images/human_Annabelle.jpg



Found a human face. It most resembles: English_toy_spaniel
English_toy_spaniel

In []: