

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 [7]:

```
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 [8]:

```
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) 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>). 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 [9]:

```
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)

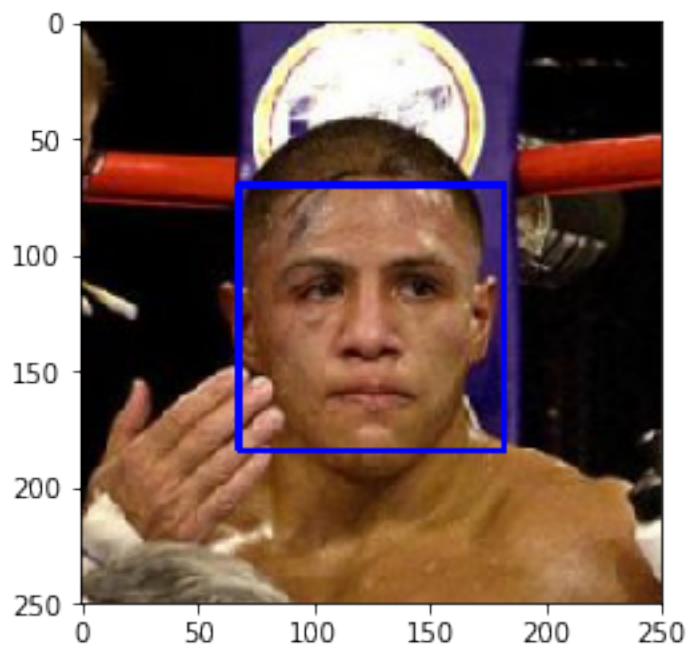
# print number of faces detected in the image
print('Number of faces detected:', len(faces))

# 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 [10]:

```
# 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:

- 98.0% of the first 100 images in `human_files` have at least one detected human face.
- 11.0% of the first 100 images in `dog_files` have at least one detected human face.

In [11]:

```
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.
def percentage_images_with_face(imageFiles):
    numberFilesContainingFace = 0
    for file in imageFiles:
        if face_detector(file):
            numberFilesContainingFace += 1
    totalNumberOfFiles = len(imageFiles)
    percentageWithFace = (numberFilesContainingFace*1.0/totalNumberOfFiles)*100
    return percentageWithFace

## Percentage of the first 100 images in human_files having at least one detected human face:
print("{}% of the first 100 images in human_files have at least one detected human face.".format((percentage_images_with_face(human_files_short))))

## Percentage of the first 100 images in dog_files having at least one detected human face:
print("{}% of the first 100 images in dog_files have at least one detected human face.".format((percentage_images_with_face(dog_files_short))))
```

98.0% of the first 100 images in human_files have at least one detected human face.

11.0% of the first 100 images in dog_files have at least one detected human face.

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: For its first version, it's okay if our feature requires that users are careful to provide only images of humans that contain clear view of a face. Generally speaking, when matching an individual with an animal they resemble, people tend to compare the person's face with that of the animal. Thanks to this, it shouldn't be terribly surprising to our users that we require a clear view of a human face.

However, in the long term, requiring the user to ensure that all human images provide clear view of a face puts an extra burden on the user's concentration and attention. This extra friction may make our feature's experience less enjoyable to some users, and may result in users being less likely to return and use this feature a second and third time. If it's too much of a hassle to do their own quality-control of images containing faces, the user might decide that our app is not worth their time.

To improve upon this deficiency, we could build a convolutional neural network and train it to detect humans in images that lack a clearly presented face. Unlike Haar cascades, which are only able to search for limited types of patterns (edge features, line features, and four-rectangle features), a CNN could on its own discover patterns that are unique to photos containing a human face, irrespective of whether that face is clearly visible or partially obscured. After its training is finished, this CNN can then use these new patterns that it discovered to locate faces in photos it hasn't seen before. The CNN will know how to find a human face, regardless of whether it is presented clearly inside the image or not.

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 [12]:

```
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

$$(nb_samples, rows, columns, channels),$$

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

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

$$(1, 224, 224, 3).$$

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

$$(nb_samples, 224, 224, 3).$$

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

In [13]:

```
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 [14]:

```
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 [15]:

```
### 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:

- 2.0% of the images in human_files_short have a detected dog.
- 100.0% of the images in dog_files_short have a detected dog.

In [16]:

```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.
def percentage_images_with_dog(imageFiles):
    numberFilesContainingDog = 0
    for file in imageFiles:
        if dog_detector(file):
            numberFilesContainingDog += 1
    totalNumberOfFiles = len(imageFiles)
    percentageWithDog = (numberFilesContainingDog*1.0/totalNumberOfFiles)*100
    return percentageWithDog

## Percentage of images in human_files_short having a detected dog:
print("{}% of the images in human_files_short have a detected dog.".format((percentage_images_with_dog(human_files_short))))

## Percentage of images in dog_files_short having a detected dog:
print("{}% of the images in dog_files_short have a detected dog.".format((percentage_images_with_dog(dog_files_short))))
```

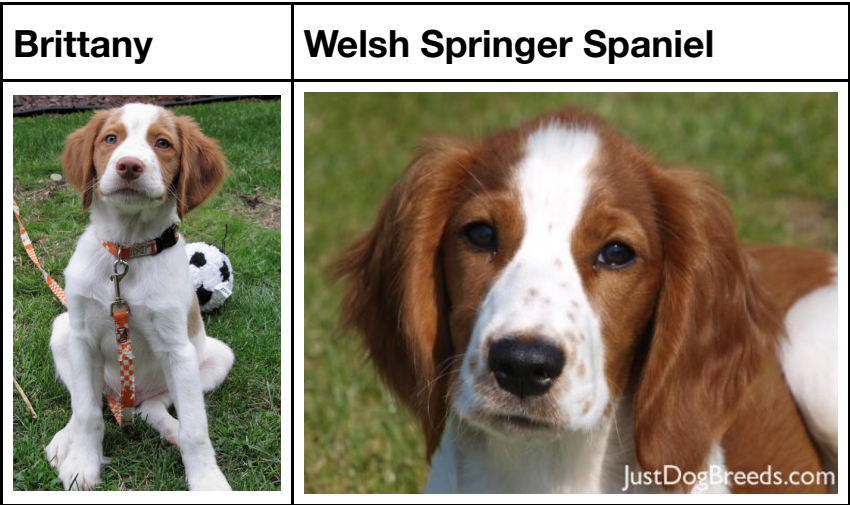
2.0% of the images in human_files_short have a detected dog.
100.0% of the images in dog_files_short have a detected dog.

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

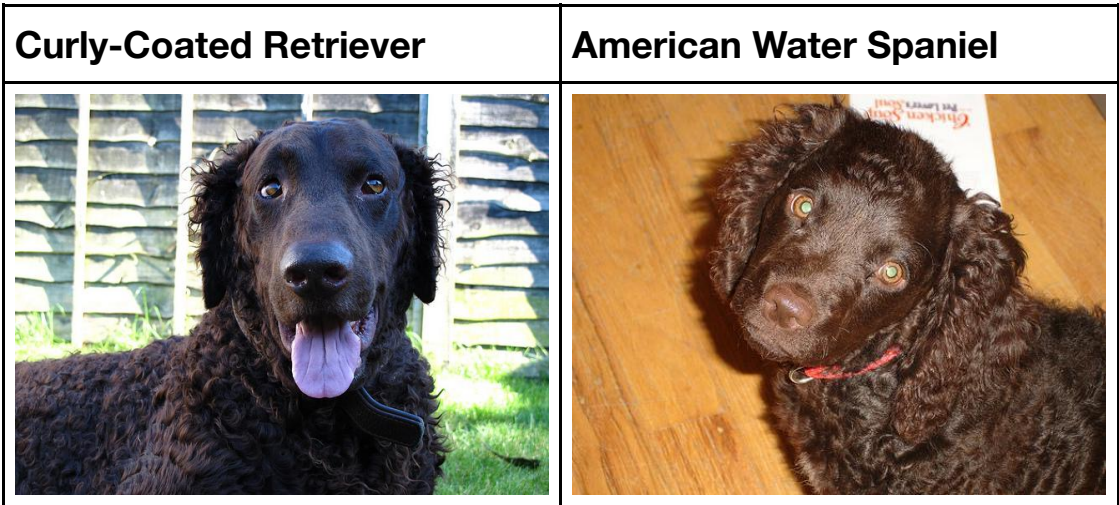
Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN that classifies dog breeds. You must create your CNN *from scratch* (so, you can't use transfer learning yet!), and you must attain a test accuracy of at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

Yellow Labrador	Chocolate Labrador	Black Labrador



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 [17]:

```
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 [00:55<00:00, 120.60it/s]
100%|██████████| 835/835 [00:06<00:00, 133.47it/s]
100%|██████████| 836/836 [00:06<00:00, 134.57it/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:

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 (GlobalAveragePooling2D)	(None, 64)	0
dense_1 (Dense)	(None, 133)	8645
Total params: 19,189.0		
Trainable params: 19,189.0		
Non-trainable params: 0.0		

INPUT

CONV

POOL

CONV

POOL

CONV

POOL

GAP

DENSE

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:

- The overall goal for my CNN architecture was to be able to gradually convert an array of image information from being fairly shallow in depth but relatively tall and wide, to being much shorter and narrower, but much deeper in depth. The idea here is that during training, my convolutional layers will be able to learn various "patterns" that are specific to images of different dog breeds. The output of my network's final convolutional layer, is basically a very long, deep array that contains all of these "patterns."
- I then flattened this third and final convolutional layer to a vector, and added a fully connected layer that contains 1,000 nodes. This fully connected layer helps my model to be able to predict a dog's breed based on whether or not an image contains the "patterns" that were learned inside the network's convolutional layers.
- My final layer contains 133 nodes to match the 133 different dog breeds into which we are categorizing dogs. I used a softmax activation function here to ensure that when making predictions my model would return a list of probabilities representing the likelihood that the dog in a certain image belonged to a particular breed.
- Notice also that after each convolutional layer I included a max pooling layer. Also notice that after both my final max pooling layer, as well as after my first fully connected layer, I included dropout layers. Inserting maxpool layers, each with a kernel size of 2 and stride of 2, decreased each preceding layer's spatial dimensions by half. Doing this helped increase the depth of subsequent layers while preventing my model from overfitting to the training images. If I had increased depth without reducing dimensionality, I would have had even more parameters to train, and having too many parameters is what often leads to overfitting.

- Adding the two dropout layers also helped to minimize overfitting. In my first dropout layer, there is a 20% chance that any node that had been in the preceding layer will be removed from the network. For my second layer, there is a 40% chance that any node at that layer will be removed during training. Together, these two dropout layers also help to reduce the number of parameters that my model will have to train.
- Finally, in regard to number of epochs, I initially ran my model for 20 epochs. However, I found that after the third epoch, I saw no further improvement (decrease) in the value of the validation data's loss function. I thus decided that 5 epochs would be good enough.

In [19]:

```
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=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=64, kernel_size=2, padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Dropout(0.2))
model.add(Flatten())
model.add(Dense(1000, activation='relu'))
model.add(Dropout(0.4))
model.add(Dense(133, activation='softmax'))

model.summary()
```

Layer (type)	Output Shape	Param #
conv2d_4 (Conv2D)	(None, 224, 224, 16)	208
max_pooling2d_5 (MaxPooling2D)	(None, 112, 112, 16)	0
conv2d_5 (Conv2D)	(None, 112, 112, 32)	2080
max_pooling2d_6 (MaxPooling2D)	(None, 56, 56, 32)	0
conv2d_6 (Conv2D)	(None, 56, 56, 64)	8256
max_pooling2d_7 (MaxPooling2D)	(None, 28, 28, 64)	0
dropout_3 (Dropout)	(None, 28, 28, 64)	0
flatten_3 (Flatten)	(None, 50176)	0
dense_3 (Dense)	(None, 1000)	50177000
dropout_4 (Dropout)	(None, 1000)	0
dense_4 (Dense)	(None, 133)	133133
Total params: 50,320,677.0		
Trainable params: 50,320,677.0		
Non-trainable params: 0.0		

Compile the Model

In [20]:

```
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>), but this is not a requirement.

In [21]:

```
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=1)
```

Train on 6680 samples, validate on 835 samples

Epoch 1/5

```
6660/6680 [=====>.] - ETA: 0s - loss: 4.9231
- acc: 0.0147Epoch 00000: val_loss improved from inf to 4.66921, saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [=====] - 46s - loss: 4.9222 - acc: 0.0148 - val_loss: 4.6692 - val_acc: 0.0347
```

Epoch 2/5

```
6660/6680 [=====>.] - ETA: 0s - loss: 4.4782
- acc: 0.0518Epoch 00001: val_loss improved from 4.66921 to 4.51312, saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [=====] - 45s - loss: 4.4769 - acc: 0.0518 - val_loss: 4.5131 - val_acc: 0.0635
```

Epoch 3/5

```
6660/6680 [=====>.] - ETA: 0s - loss: 3.7457
- acc: 0.1646Epoch 00002: val_loss improved from 4.51312 to 4.26702, saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [=====] - 45s - loss: 3.7438 - acc: 0.1648 - val_loss: 4.2670 - val_acc: 0.0814
```

Epoch 4/5

```
6660/6680 [=====>.] - ETA: 0s - loss: 2.3226
- acc: 0.4417Epoch 00003: val_loss did not improve
6680/6680 [=====] - 41s - loss: 2.3219 - acc: 0.4419 - val_loss: 4.8855 - val_acc: 0.0862
```

Epoch 5/5

```
6660/6680 [=====>.] - ETA: 0s - loss: 0.9303
- acc: 0.7685Epoch 00004: val_loss did not improve
6680/6680 [=====] - 41s - loss: 0.9303 - acc: 0.7680 - val_loss: 5.8394 - val_acc: 0.0814
```

Out[21]:

<keras.callbacks.History at 0x7f7c002cfef0>

Load the Model with the Best Validation Loss

In [22]:

```
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 [23]:

```
# 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: 8.3732%

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 [24]:

```
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 [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_5 (Dense)	(None, 133)	68229

=====
Total params: 68,229.0
Trainable params: 68,229.0
Non-trainable params: 0.0
=====

Compile the Model

In [26]:

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

Train the Model

In [27]:

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

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

Train on 6680 samples, validate on 835 samples

Epoch 1/20

```
6420/6680 [=====>..] - ETA: 0s - loss: 11.7470
- acc: 0.1366Epoch 00000: val_loss improved from inf to 10.10362, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 11.6771 - acc: 0.1409 - val_loss: 10.1036 - val_acc: 0.2263
```

Epoch 2/20

```
6500/6680 [=====>.] - ETA: 0s - loss: 9.4524
- acc: 0.3114Epoch 00001: val_loss improved from 10.10362 to 9.56507, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 9.4395 - acc
```

```
: 0.3117 - val_loss: 9.5651 - val_acc: 0.3090
Epoch 3/20
6520/6680 [=====>.] - ETA: 0s - loss: 9.0450
- acc: 0.3741Epoch 00002: val_loss improved from 9.56507 to 9.27790,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 9.0528 - acc
: 0.3743 - val_loss: 9.2779 - val_acc: 0.3365
Epoch 4/20
6460/6680 [=====>.] - ETA: 0s - loss: 8.8015
- acc: 0.4077Epoch 00003: val_loss improved from 9.27790 to 9.13364,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 8.8015 - acc
: 0.4078 - val_loss: 9.1336 - val_acc: 0.3545
Epoch 5/20
6520/6680 [=====>.] - ETA: 0s - loss: 8.6342
- acc: 0.4247Epoch 00004: val_loss improved from 9.13364 to 8.84853,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 8.6088 - acc
: 0.4263 - val_loss: 8.8485 - val_acc: 0.3737
Epoch 6/20
6520/6680 [=====>.] - ETA: 0s - loss: 8.4418
- acc: 0.4463Epoch 00005: val_loss did not improve
6680/6680 [=====] - 1s - loss: 8.4643 - acc
: 0.4452 - val_loss: 8.9055 - val_acc: 0.3772
Epoch 7/20
6500/6680 [=====>.] - ETA: 0s - loss: 8.3669
- acc: 0.4578Epoch 00006: val_loss improved from 8.84853 to 8.78878,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 8.3853 - acc
: 0.4564 - val_loss: 8.7888 - val_acc: 0.3832
Epoch 8/20
6620/6680 [=====>.] - ETA: 0s - loss: 8.1517
- acc: 0.4654Epoch 00007: val_loss improved from 8.78878 to 8.52441,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 8.1439 - acc
: 0.4662 - val_loss: 8.5244 - val_acc: 0.3916
Epoch 9/20
6500/6680 [=====>.] - ETA: 0s - loss: 7.8730
- acc: 0.4842Epoch 00008: val_loss improved from 8.52441 to 8.29596,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 7.8835 - acc
: 0.4831 - val_loss: 8.2960 - val_acc: 0.4108
Epoch 10/20
6600/6680 [=====>.] - ETA: 0s - loss: 7.6487
- acc: 0.5009Epoch 00009: val_loss improved from 8.29596 to 8.22170,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 7.6504 - acc
: 0.5006 - val_loss: 8.2217 - val_acc: 0.4168
Epoch 11/20
6400/6680 [=====>..] - ETA: 0s - loss: 7.5997
- acc: 0.5131Epoch 00010: val_loss improved from 8.22170 to 8.20368,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 7.5957 - acc
```

```
: 0.5130 - val_loss: 8.2037 - val_acc: 0.4192
Epoch 12/20
6640/6680 [=====>.] - ETA: 0s - loss: 7.5705
- acc: 0.5187Epoch 00011: val_loss improved from 8.20368 to 8.13931,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 7.5617 - acc
: 0.5192 - val_loss: 8.1393 - val_acc: 0.4299
Epoch 13/20
6420/6680 [=====>..] - ETA: 0s - loss: 7.5355
- acc: 0.5248Epoch 00012: val_loss did not improve
6680/6680 [=====] - 1s - loss: 7.5498 - acc
: 0.5237 - val_loss: 8.1560 - val_acc: 0.4240
Epoch 14/20
6580/6680 [=====>.] - ETA: 0s - loss: 7.5413
- acc: 0.5252Epoch 00013: val_loss improved from 8.13931 to 8.10905,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 7.5324 - acc
: 0.5257 - val_loss: 8.1091 - val_acc: 0.4347
Epoch 15/20
6480/6680 [=====>.] - ETA: 0s - loss: 7.5231
- acc: 0.5276Epoch 00014: val_loss improved from 8.10905 to 8.04296,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 7.5177 - acc
: 0.5278 - val_loss: 8.0430 - val_acc: 0.4431
Epoch 16/20
6580/6680 [=====>.] - ETA: 0s - loss: 7.4010
- acc: 0.5289Epoch 00015: val_loss improved from 8.04296 to 7.98424,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 7.4007 - acc
: 0.5290 - val_loss: 7.9842 - val_acc: 0.4479
Epoch 17/20
6440/6680 [=====>..] - ETA: 0s - loss: 7.3050
- acc: 0.5387Epoch 00016: val_loss improved from 7.98424 to 7.93783,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 7.3057 - acc
: 0.5383 - val_loss: 7.9378 - val_acc: 0.4335
Epoch 18/20
6500/6680 [=====>.] - ETA: 0s - loss: 7.2301
- acc: 0.5398Epoch 00017: val_loss improved from 7.93783 to 7.80622,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 7.2313 - acc
: 0.5398 - val_loss: 7.8062 - val_acc: 0.4563
Epoch 19/20
6540/6680 [=====>.] - ETA: 0s - loss: 7.1705
- acc: 0.5443Epoch 00018: val_loss improved from 7.80622 to 7.70750,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 7.1554 - acc
: 0.5455 - val_loss: 7.7075 - val_acc: 0.4479
Epoch 20/20
6440/6680 [=====>..] - ETA: 0s - loss: 7.0339
- acc: 0.5528Epoch 00019: val_loss did not improve
6680/6680 [=====] - 1s - loss: 7.0578 - acc
: 0.5515 - val_loss: 7.7520 - val_acc: 0.4539
```

Out[27]:

```
<keras.callbacks.History at 0x7f7c001615c0>
```

Load the Model with the Best Validation Loss

In [28]:

```
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 [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))) 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: 43.5407%

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)]
```

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>) bottleneck features
- ResNet-50 (<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>) bottleneck features
- Xception (<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 [31]:

```
### TODO: Obtain bottleneck features from another pre-trained CNN.
bottleneck_features = np.load('bottleneck_features/DogInceptionV3Data.npz')
train_InceptionV3 = bottleneck_features['train']
valid_InceptionV3 = bottleneck_features['valid']
test_InceptionV3 = bottleneck_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:

- I chose to build my CNN on top of the InceptionV3 bottleneck features. I believe that the lower level patterns of the images in my dataset (images of dogs) will have much in common with the lower level patterns of the dataset (imagenet) on which the InceptionV3 features were trained. Due to this, as well as due to the fact that my dataset is relatively smaller in size (6,000+ training images), I felt that the best approach was to freeze the bottleneck features of the pre-trained InceptionV3 model and make them the input layer of my CNN.
- All in all, I got my highest results for accuracy (high 70s or low 80s percentage range) when I added only one final layer to my CNN, which was a fully trainable, fully-connected layer whose shape corresponds to the list of all the dog breed categories (133). Using a softmax activation function enables my model to return a list of the probabilities that a dog appearing in a particular image matches the various dog breeds.
- I had experimented with adding two or three more dense, fully trainable layers with somewhere between 100 and 1,000 nodes each, with dropout layers following each of these fully connected layers. However, doing this caused my CNN's accuracy to dip to the low 70s percentage range. Adding these extra fully connected layers obviously caused my model to overfit too much to the training images.
- Indeed, I found that I got my best accuracy when I followed the exact transfer learning CNN structure that is recommended for the case when one is applying a pre-trained model to a small dataset with different data.
- Finally, I kept my number of epochs to 5. I found that when training over more than 5 epochs, I rarely saw an improvement in validation loss after the first 2-4 epochs.

In [32]:

```
### TODO: Define your architecture.
InceptionV3_model = Sequential()
InceptionV3_model.add(GlobalAveragePooling2D(input_shape=train_InceptionV3.shape
[1:]))
InceptionV3_model.add(Dense(133, activation='softmax'))

InceptionV3_model.summary()
```

Layer (type)	Output Shape	Param #
global_average_pooling2d_2 (None, 2048)		0
dense_6 (Dense)	(None, 133)	272517
Total params: 272,517.0		
Trainable params: 272,517.0		
Non-trainable params: 0.0		

(IMPLEMENTATION) Compile the Model

In [33]:

```
### TODO: Compile the model.
InceptionV3_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 [34]:

```
### TODO: Train the model.
checkpointer = ModelCheckpoint(filepath='saved_models/weights.best.InceptionV3.hdf5',
                               verbose=1, save_best_only=True)

InceptionV3_model.fit(train_InceptionV3, train_targets,
                      validation_data=(valid_InceptionV3, valid_targets),
                      epochs=5, batch_size=20, callbacks=[checkpointer], verbose=1)
```

Train on 6680 samples, validate on 835 samples

Epoch 1/5

```
6640/6680 [=====>.] - ETA: 0s - loss: 1.1538
- acc: 0.7101Epoch 00000: val_loss improved from inf to 0.58852, saving model to saved_models/weights.best.InceptionV3.hdf5
6680/6680 [=====] - 3s - loss: 1.1495 - acc: 0.7106 - val_loss: 0.5885 - val_acc: 0.8311
```

Epoch 2/5

```
6660/6680 [=====>.] - ETA: 0s - loss: 0.4687
- acc: 0.8587Epoch 00001: val_loss did not improve
6680/6680 [=====] - 2s - loss: 0.4697 - acc: 0.8582 - val_loss: 0.6388 - val_acc: 0.8359
```

Epoch 3/5

```
6620/6680 [=====>.] - ETA: 0s - loss: 0.3664
- acc: 0.8863Epoch 00002: val_loss did not improve
6680/6680 [=====] - 2s - loss: 0.3643 - acc: 0.8867 - val_loss: 0.6618 - val_acc: 0.8311
```

Epoch 4/5

```
6660/6680 [=====>.] - ETA: 0s - loss: 0.2928
- acc: 0.9098Epoch 00003: val_loss did not improve
6680/6680 [=====] - 2s - loss: 0.2923 - acc: 0.9100 - val_loss: 0.6324 - val_acc: 0.8419
```

Epoch 5/5

```
6640/6680 [=====>.] - ETA: 0s - loss: 0.2452
- acc: 0.9250Epoch 00004: val_loss did not improve
6680/6680 [=====] - 2s - loss: 0.2468 - acc: 0.9246 - val_loss: 0.7267 - val_acc: 0.8311
```

Out[34]:

<keras.callbacks.History at 0x7f7ac0498630>

(IMPLEMENTATION) Load the Model with the Best Validation Loss

In [35]:

```
### TODO: Load the model weights with the best validation loss.
InceptionV3_model.load_weights('saved_models/weights.best.InceptionV3.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 [36]:

```
### TODO: Calculate classification accuracy on the test dataset.
# get index of predicted dog breed for each image in test set
InceptionV3_predictions = [np.argmax(InceptionV3_model.predict(np.expand_dims(feature, axis=0))) for feature in test_InceptionV3]

# report test accuracy
test_accuracy = 100*np.sum(np.array(InceptionV3_predictions)==np.argmax(test_targets, axis=1))/len(InceptionV3_predictions)
print('Test accuracy: %.4f%%' % test_accuracy)
```

Test accuracy: 78.9474%

(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 [37]:

```
### TODO: Write a function that takes a path to an image as input
### and returns the dog breed that is predicted by the model.
def InceptionV3_predict_breed(img_path):
    # 1. extract bottleneck features
    bottleneck_feature = extract_InceptionV3(path_to_tensor(img_path))
    # 2. obtain predicted vector
    predicted_vector = InceptionV3_model.predict(bottleneck_feature)
    # 3. return dog breed that is predicted by the model
    return dog_names[np.argmax(predicted_vector)]
```

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!

```
hello, human!
```



```
You look like a ...
Chinese_shar-pei
```

(IMPLEMENTATION) Write your Algorithm

In [40]:

```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.
def dog_resemblance(imagePath):
    ## Predict the breed of the dog (or the dog breed most resembling a person's
    appearance)
    predictedBreed = InceptionV3_predict_breed(imagePath)

    if face_detector(imagePath):
        print("This person looks most similar to a {} breed of dog.".format(pred
ictedBreed))
    elif dog_detector(imagePath):
        print("This dog looks like a {} breed.".format(predictedBreed))
    else:
        print("For best results, try this feature out with an image of a person
or a dog. It appears that this image contains neither.")
    ## Display the picture
    img = cv2.imread(imagePath)
    cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    plt.imshow(cv_rgb)
    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?

(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.

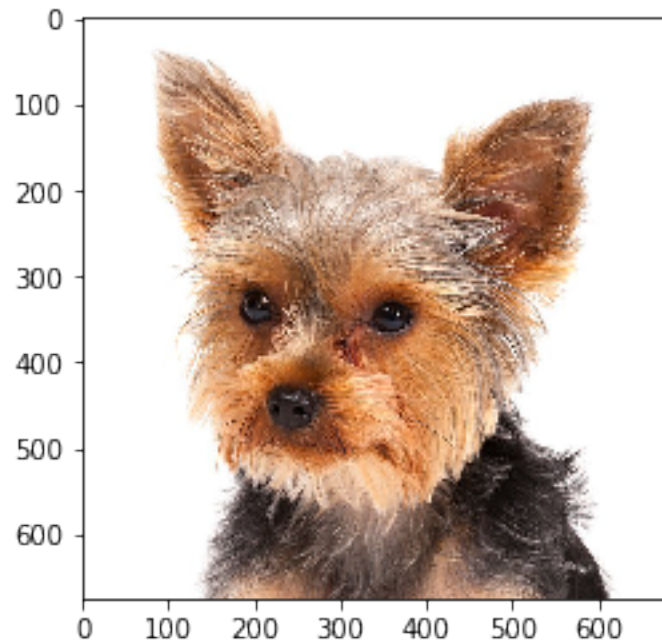
Answer: My output was slightly better than I'd expected. I was happy that my algorithm was apparently able to properly predict the correct breed of both dog images that I used to test it. If there was any one area that was a source of disappointment for me, it was that my algorithm was unable to correctly ascertain that the picture of Bill Gates was indeed that of a person's face. Thus, here are three ways I'd try and improve my algorithm:

1. Build a better face detector than that which I currently use from OpenCV. While I'm pleased with the performance of my dog detector, my face detector clearly has room for improvement.
2. If a famous person's image is given to this algorithm, it would be cool if the algorithm could identify which famous person the face in the image belongs to. For example, with the image of Barack Obama below, my algorithm would look a lot smarter if it could reply with something along the lines of "Barack Obama looks most similar to a Dachshund breed of dog," instead of merely generally stating that "this person looks most similar to a Dachshund..."
3. Finally, when given a picture that contains neither a person nor a dog, my algorithm's credibility would be enhanced if it could properly identify what the picture did actually contain. In the case of the image of the truck below, instead of saying "this is neither a dog nor a person," it would be more interesting if my algorithm could say something like "Hey this looks like a truck! Try using a picture of a dog or person instead."

In [41]:

```
## 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.  
dog_resemblance('testImages/dog1.jpg')
```

This dog looks like a Yorkshire_terrier breed.



In [42]:

```
dog_resemblance('testImages/dog2.jpg')
```

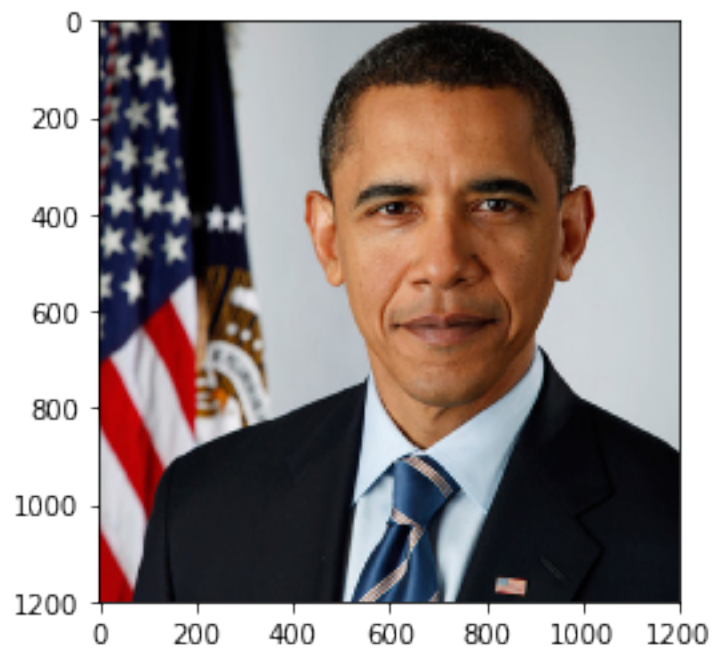
This dog looks like a Entlebucher_mountain_dog breed.



In [43]:

```
dog_resemblance('testImages/person1.jpg')
```

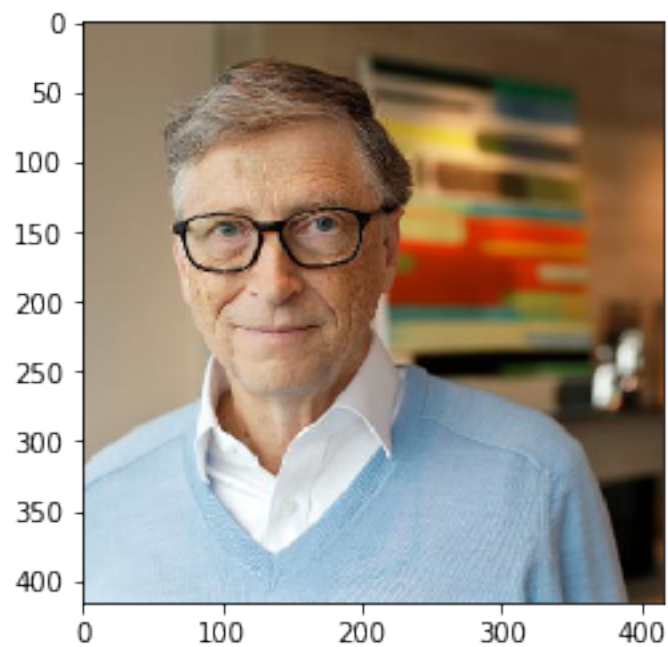
This person looks most similar to a Dachshund breed of dog.



In [44]:

```
dog_resemblance('testImages/person2.jpg')
```

For best results, try this feature out with an image of a person or a dog. It appears that this image contains neither.



In [45]:

```
dog_resemblance('testImages/nonPersonNonDog1.jpg')
```

For best results, try this feature out with an image of a person or a dog. It appears that this image contains neither.



In [46]:

```
dog_resemblance('testImages/nonPersonNonDog2.jpg')
```

For best results, try this feature out with an image of a person or a dog. It appears that this image contains neither.

