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.

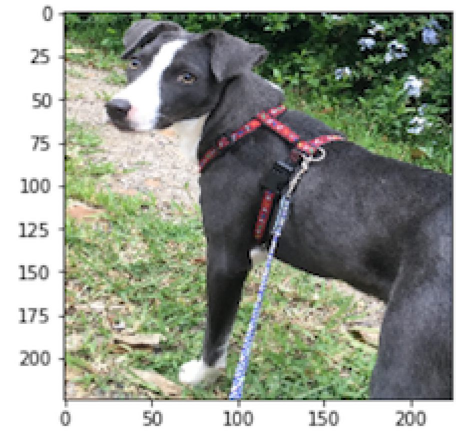
> **Note:** Code and Markdown cells can be executed using the **Shift + Enter** keyboard shortcut. Markdown cells can be edited by double-clicking the cell to enter edit mode.

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!).

```
hello, dog!
your predicted breed is ...
American Staffordshire terrier
```



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

In [5]:
```
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, v
alid_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 [6]:
```
import random
random.seed(8675309)

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

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

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

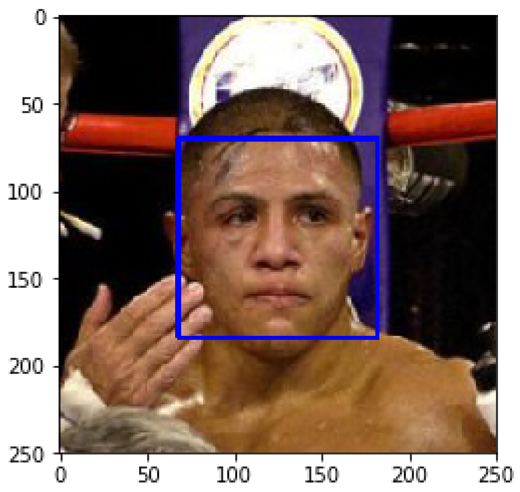
Step 1: Detect Humans

We use OpenCV's implementation of Haar feature-based cascade classifiers (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 [7]:
```
import cv2
import matplotlib.pyplot as plt
%matplotlib inline

# extract pre-trained face detector
face_cascade = cv2.CascadeClassifier('haarcascades/haarcascade_frontalfa
ce_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 [8]:
```
# 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% of the first 100 images in `human_files` have a detected human face

11% of the first 100 images in `dog_files` have a detected human face

In [9]:
```
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.
human_counter = 0
dog_counter = 0

face_detected = 0
for human in human_files_short:
    face_detected = 1 if face_detector(human) else 0
    human_counter += face_detected

print('Number of faces detected in human files:', human_counter)
print('percentage of human faces detected in dog files', human_counter/(
len(human_files_short)))

for dog in dog_files_short:
    face_detected = 1 if face_detector(dog) else 0
    dog_counter += face_detected

print('Number of faces detected in dog files:', dog_counter)
print('percentage of human faces detected in dog files', dog_counter/(le
n(dog_files_short)))
```

```
Number of faces detected in human files: 98
percentage of human faces detected in dog files 0.98
Number of faces detected in dog files: 11
percentage of human faces detected in dog files 0.11
```

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

Answer:

This is a reasonable expectation to pose on the user, though not ideal. Machine learning at current stage can do a lot more advanced face detection, including detecting faces presented from different angels, in different lighting, with various degrees of distortion etc. Google and Facebook, among others, are also able to do accurate face-recognition.

Different algorithms have been developed to detect human faces in different conditions. One of them is called face landmark estimation. It basically maps out a human face into 68 points / landmarks, the shape of the face, positions and contours of eyes, nose, mouth, etc. When the map is constructed, the image can be scaled, rotated and transformed images to obtain a roughly front and center face view.

OpenFace (https://cmusatyalab.github.io/openface/) has a very popular library and is doing a great job detecting faces and recognize public figures (such as Barack Obama and Will Farrell). I have not able to install the openface library to try it out.

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 [19]:
```
## (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 [10]:
```
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 [11]:
```
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 ret
urn 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).

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).

In [12]:
```python
from keras.applications.resnet50 import preprocess_input, decode_predict
ions

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), 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 [13]:
```python
### 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% of of the images in `human_files_short` have a detected dog

100% of the images in `dog_files_short` have a detected dog

In [14]:
```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.
human_counter = 0
dog_counter = 0

face_detected = 0
for human in human_files_short:
    face_detected = 1 if dog_detector(human) else 0
    human_counter += face_detected

print('Number of dog faces detected in human files:', human_counter)
print('percentage of dog faces detected in human files', human_counter/(
len(human_files_short)))

for dog in dog_files_short:
    face_detected = 1 if dog_detector(dog) else 0
    dog_counter += face_detected

print('Number of dog faces detected in dog files:', dog_counter)
print('percentage of dog faces detected in dog files', dog_counter/(len(
dog_files_short)))
```

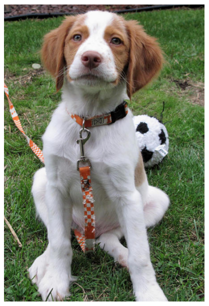
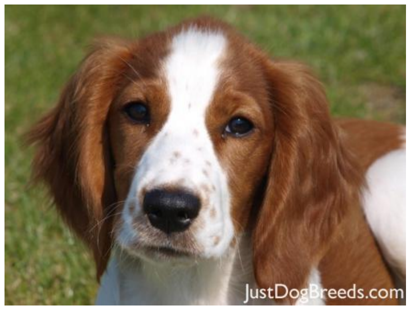
```
Number of dog faces detected in human files: 2
percentage of dog faces detected in human files 0.02
Number of dog faces detected in dog files: 100
percentage of dog faces detected in dog files 1.0
```

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

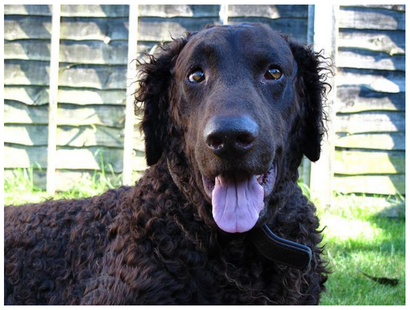
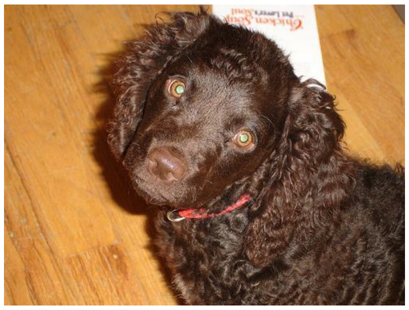
Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN that classifies dog breeds. You must create your CNN *from scratch* (so, you can't use transfer learning *yet*!), and you must attain a test accuracy of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

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

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

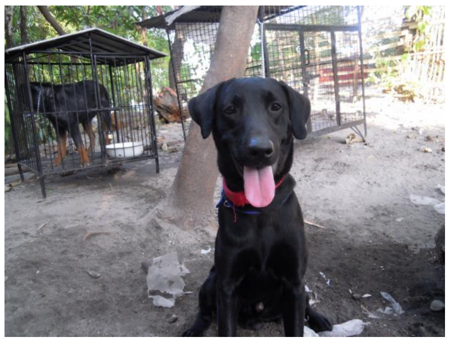
Brittany	Welsh Springer Spaniel
	

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

Curly-Coated Retriever	American Water Spaniel
	

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

Yellow Labrador	Chocolate Labrador	Black Labrador

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 imabalanced, a random guess will provide a correct answer roughly 1 in 133 times, which corresponds to an accuracy of less than 1%.

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

Pre-process the Data

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

In [15]:
```python
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

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

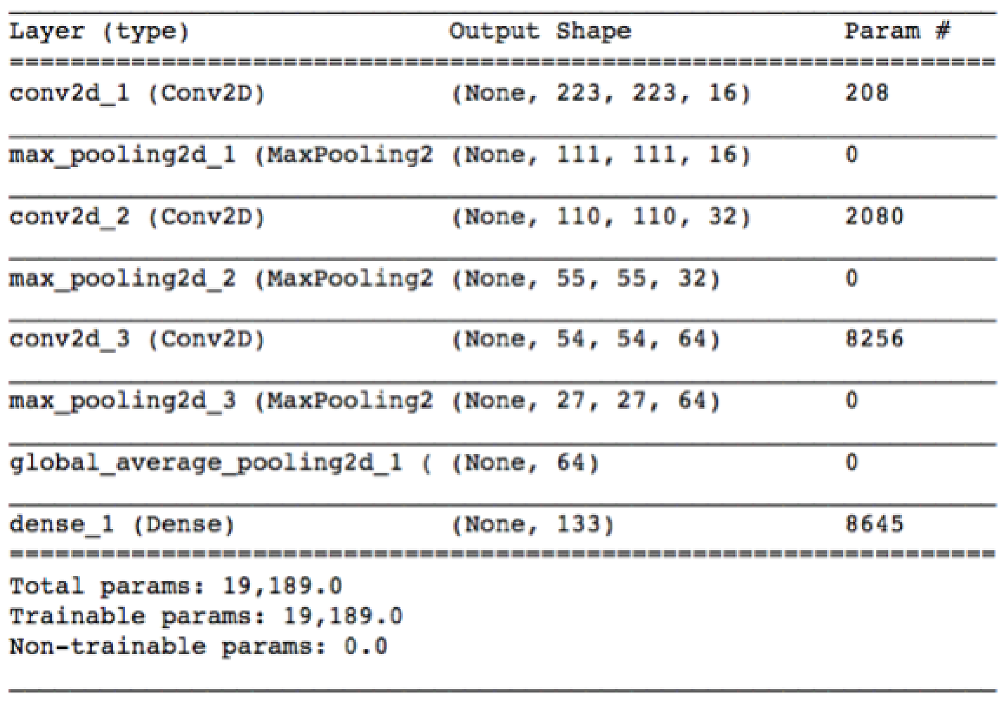
```
100%|██████████| 6680/6680 [00:53<00:00, 125.45it/s]
100%|██████████| 835/835 [00:05<00:00, 139.53it/s]
100%|██████████| 836/836 [00:05<00:00, 140.19it/s]
```

(IMPLEMENTATION) Model Architecture

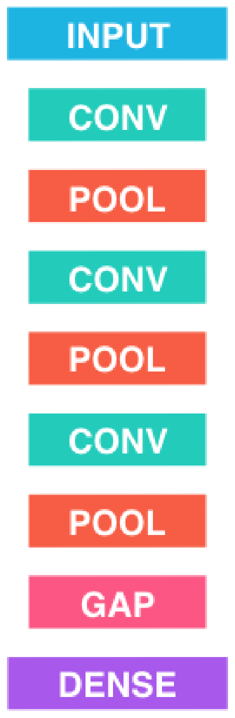
Create a CNN to classify dog breed. At the end of your code cell block, summarize the layers of your model by executing the line:

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



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

Answer:

I followed one of the most common form of a ConvNet architecture, in which a layer stack is built with one or two Conv layers with ELU activation, then a MaxPooling layer. Then the pattern repeats (more Conv layers then a MaxPooling layer), until the image is abstracted spatially into a small size. Then I connect it to fully connected / dense layer using RELU as activation layer. After which, a full connected layer with softmax activation outputs the final result.

I also added Dropout layers to battle overfitting issues.

Following the advice from my reviewer, I added batch normalization and switched to ELU from ReLU for activation. Though i only added batch normalization to the input layer, because my initial testing seemed slow with normalization applied to other hidden layers.

Detailed architecture reasoning and construction is as the following:

Stack #1:

1. Convolutional layer #1: Applies 16 3x3 filters with padding as 'Same', which specify that the output tensor should have the same width and height as the input tensor. The output tensor produced by this layer has a shape of [batch_size, 224, 224, 16]. It has the same width and height as the input layer, however it has 16 channel.
2. Batch normalization layer: Apply batch normalization to the input layer to speed up learning
3. ELU activation layer: apply ELU activation
4. Pooling Layer #1: Performs max pooling with a 2x2 filter. The output tensor produced by this layer has a shape of [batch_size, 112, 112, 16]; the 2x2 filter reduces width and height of the previous feature map by half.

Stack #2:

1. Convolutional layer #2: 32 filters of 3 * 3 regions, same padding, ELU activation. This is by the common pattern which with every additional stack, the channel of depth generally doubles from the previous stack.
2. Convolutional layer #3: 32 filters of 3 * 3 regions, same padding, ELU activation
3. Dropout layer #1: dropout layer with regularization rate of 0.25. By doing so, 25% of of the elements will be randomly dropped out.
4. Pooling Layer #2: Performs max pooling with a 2x2 filter. The output tensor produced by this layer has a shape of [batch_size, 56, 56, 32].

Stack #3:

1. Convolutional Layer #5: 64 filters of 3 * 3 regions, same padding, ELU activation
2. Convolutional Layer #6: 64 filters of 3 * 3 regions, same padding, ELU activation
3. Pooling Layer #3: here a gloabal average pooling layer is used to output the spatial averages of the tensors

Full connected layers to prepare the final predictions:

1. Dense Layer #1: , a fully connected layer of 1,024 neurons to connect with the output tensor from the preceeding layer, with dropout regularization rate set at 0.4.
2. Dropout layer #2: The final layer is the results layer. It is a fully connected dense layer with softmax activations and 133 neurons, one for each of the dog categories.

References:

CS231n Convolutional Neural Networks for Visual Recognition (https://cs231n.github.io/convolutional-networks/)

In [21]:
```
from keras.layers import Conv2D, MaxPooling2D, GlobalAveragePooling2D
from keras.layers import Dropout, Flatten, Dense, Activation
from keras.models import Sequential
from keras.layers.normalization import BatchNormalization

model = Sequential()

model.add(Conv2D(filters=16, kernel_size=3, padding='same', input_shape=
(224, 224, 3)))
model.add(BatchNormalization())
model.add(Activation('elu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=32, kernel_size=3, padding='same', activation=
'elu'))
model.add(Conv2D(filters=32, kernel_size=3, padding='same', activation=
'elu'))

model.add(Dropout(0.25))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=64, kernel_size=3, padding='same', activation=
'elu'))
model.add(Conv2D(filters=64, kernel_size=3, padding='same', activation=
'elu'))
model.add(GlobalAveragePooling2D())
model.add(Dropout(0.4))
model.add(Dense(1024, activation='elu'))
model.add(Dropout(0.5))
model.add(Dense(133, activation='softmax'))

### TODO: Define your architecture.

model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_5 (Conv2D)            (None, 224, 224, 16)      448       
_________________________________________________________________
batch_normalization_5 (Batch (None, 224, 224, 16)      64        
_________________________________________________________________
activation_50 (Activation)   (None, 224, 224, 16)      0         
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 112, 112, 16)      0         
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 112, 112, 32)      4640      
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 112, 112, 32)      9248      
_________________________________________________________________
dropout_1 (Dropout)          (None, 112, 112, 32)      0         
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 56, 56, 32)        0         
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 56, 56, 64)        18496     
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 56, 56, 64)        36928     
_________________________________________________________________
global_average_pooling2d_1 ( (None, 64)                0         
_________________________________________________________________
dropout_2 (Dropout)          (None, 64)                0         
_________________________________________________________________
dense_1 (Dense)              (None, 1024)              66560     
_________________________________________________________________
dropout_3 (Dropout)          (None, 1024)              0         
_________________________________________________________________
dense_2 (Dense)              (None, 133)               136325    
=================================================================
Total params: 272,709.0
Trainable params: 272,677.0
Non-trainable params: 32.0
_________________________________________________________________
```

Compile the Model

In [33]:
```
import keras
opt = keras.optimizers.rmsprop(lr=0.001)
model.compile(optimizer=opt, 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 [34]:
```
from keras.callbacks import ModelCheckpoint

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

epochs = 100

### 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], verbos
e=1)
```

```
Train on 6680 samples, validate on 835 samples
Epoch 1/100
6660/6680 [============================>.] - ETA: 0s - loss: 4.8576 - a
cc: 0.0168Epoch 00000: val_loss improved from inf to 4.85398, saving mo
del to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 45s - loss: 4.8570 - acc:
0.0168 - val_loss: 4.8540 - val_acc: 0.0132
Epoch 2/100
6660/6680 [============================>.] - ETA: 0s - loss: 4.8116 - a
cc: 0.0221Epoch 00001: val_loss improved from 4.85398 to 4.81359, savin
g model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 44s - loss: 4.8116 - acc:
0.0220 - val_loss: 4.8136 - val_acc: 0.0192
Epoch 3/100
6660/6680 [============================>.] - ETA: 0s - loss: 4.7840 - a
cc: 0.0251Epoch 00002: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 4.7843 - acc:
0.0250 - val_loss: 4.9270 - val_acc: 0.0144
Epoch 4/100
6660/6680 [============================>.] - ETA: 0s - loss: 4.7440 - a
cc: 0.0282Epoch 00003: val_loss improved from 4.81359 to 4.79745, savin
g model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 44s - loss: 4.7443 - acc:
0.0281 - val_loss: 4.7974 - val_acc: 0.0156
Epoch 5/100
6660/6680 [============================>.] - ETA: 0s - loss: 4.7241 - a
cc: 0.0290Epoch 00004: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 4.7243 - acc:
0.0290 - val_loss: 5.3060 - val_acc: 0.0144
Epoch 6/100
6660/6680 [============================>.] - ETA: 0s - loss: 4.6849 - a
cc: 0.0312Epoch 00005: val_loss improved from 4.79745 to 4.75083, savin
g model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 44s - loss: 4.6846 - acc:
0.0313 - val_loss: 4.7508 - val_acc: 0.0216
Epoch 7/100
6660/6680 [============================>.] - ETA: 0s - loss: 4.6705 - a
cc: 0.0311Epoch 00006: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 4.6704 - acc:
0.0311 - val_loss: 5.2033 - val_acc: 0.0108
Epoch 8/100
6660/6680 [============================>.] - ETA: 0s - loss: 4.6292 - a
cc: 0.0389Epoch 00007: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 4.6287 - acc:
0.0389 - val_loss: 4.7997 - val_acc: 0.0371
Epoch 9/100
6660/6680 [============================>.] - ETA: 0s - loss: 4.6096 - a
cc: 0.0443Epoch 00008: val_loss improved from 4.75083 to 4.57325, savin
g model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 44s - loss: 4.6092 - acc:
0.0445 - val_loss: 4.5732 - val_acc: 0.0467
Epoch 10/100
6660/6680 [============================>.] - ETA: 0s - loss: 4.5706 - a
cc: 0.0422Epoch 00009: val_loss improved from 4.57325 to 4.57136, savin
g model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 44s - loss: 4.5711 - acc:
0.0421 - val_loss: 4.5714 - val_acc: 0.0311
```

```
Epoch 11/100
6660/6680 [============================>.] - ETA: 0s - loss: 4.5332 - a
cc: 0.0468Epoch 00010: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 4.5327 - acc:
0.0469 - val_loss: 5.4951 - val_acc: 0.0204
Epoch 12/100
6660/6680 [============================>.] - ETA: 0s - loss: 4.5125 - a
cc: 0.0527Epoch 00011: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 4.5124 - acc:
0.0528 - val_loss: 4.6971 - val_acc: 0.0479
Epoch 13/100
6660/6680 [============================>.] - ETA: 0s - loss: 4.4782 - a
cc: 0.0575Epoch 00012: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 4.4778 - acc:
0.0579 - val_loss: 4.6057 - val_acc: 0.0395
Epoch 14/100
6660/6680 [============================>.] - ETA: 0s - loss: 4.4460 - a
cc: 0.0613Epoch 00013: val_loss improved from 4.57136 to 4.34217, savin
g model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 44s - loss: 4.4456 - acc:
0.0614 - val_loss: 4.3422 - val_acc: 0.0539
Epoch 15/100
6660/6680 [============================>.] - ETA: 0s - loss: 4.4188 - a
cc: 0.0637Epoch 00014: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 4.4186 - acc:
0.0636 - val_loss: 4.6937 - val_acc: 0.0419
Epoch 16/100
6660/6680 [============================>.] - ETA: 0s - loss: 4.3790 - a
cc: 0.0647Epoch 00015: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 4.3785 - acc:
0.0648 - val_loss: 4.4890 - val_acc: 0.0623
Epoch 17/100
6660/6680 [============================>.] - ETA: 0s - loss: 4.3528 - a
cc: 0.0706Epoch 00016: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 4.3521 - acc:
0.0705 - val_loss: 4.3530 - val_acc: 0.0659
Epoch 18/100
6660/6680 [============================>.] - ETA: 0s - loss: 4.3165 - a
cc: 0.0748Epoch 00017: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 4.3160 - acc:
0.0746 - val_loss: 4.5639 - val_acc: 0.0515
Epoch 19/100
6660/6680 [============================>.] - ETA: 0s - loss: 4.2826 - a
cc: 0.0829Epoch 00018: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 4.2825 - acc:
0.0828 - val_loss: 4.8283 - val_acc: 0.0467
Epoch 20/100
6660/6680 [============================>.] - ETA: 0s - loss: 4.2330 - a
cc: 0.0898Epoch 00019: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 4.2349 - acc:
0.0895 - val_loss: 4.4825 - val_acc: 0.0527
Epoch 21/100
6660/6680 [============================>.] - ETA: 0s - loss: 4.2212 - a
cc: 0.0833Epoch 00020: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 4.2228 - acc:
0.0834 - val_loss: 4.6144 - val_acc: 0.0491
Epoch 22/100
```

```
6660/6680 [============================>.] - ETA: 0s - loss: 4.1994 - a
cc: 0.0878Epoch 00021: val_loss improved from 4.34217 to 4.18956, savin
g model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 44s - loss: 4.1994 - acc:
0.0877 - val_loss: 4.1896 - val_acc: 0.0826
Epoch 23/100
6660/6680 [============================>.] - ETA: 0s - loss: 4.1597 - a
cc: 0.0935Epoch 00022: val_loss improved from 4.18956 to 4.15940, savin
g model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 44s - loss: 4.1621 - acc:
0.0933 - val_loss: 4.1594 - val_acc: 0.0862
Epoch 24/100
6660/6680 [============================>.] - ETA: 0s - loss: 4.1500 - a
cc: 0.0937Epoch 00023: val_loss improved from 4.15940 to 4.09332, savin
g model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 44s - loss: 4.1489 - acc:
0.0937 - val_loss: 4.0933 - val_acc: 0.0934
Epoch 25/100
6660/6680 [============================>.] - ETA: 0s - loss: 4.1063 - a
cc: 0.0935Epoch 00024: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 4.1070 - acc:
0.0936 - val_loss: 5.4956 - val_acc: 0.0240
Epoch 26/100
6660/6680 [============================>.] - ETA: 0s - loss: 4.0814 - a
cc: 0.0991Epoch 00025: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 4.0820 - acc:
0.0994 - val_loss: 4.1824 - val_acc: 0.0934
Epoch 27/100
6660/6680 [============================>.] - ETA: 0s - loss: 4.0748 - a
cc: 0.1068Epoch 00026: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 4.0738 - acc:
0.1066 - val_loss: 4.3052 - val_acc: 0.0731
Epoch 28/100
6660/6680 [============================>.] - ETA: 0s - loss: 4.0613 - a
cc: 0.1087Epoch 00027: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 4.0610 - acc:
0.1088 - val_loss: 4.3298 - val_acc: 0.0695
Epoch 29/100
6660/6680 [============================>.] - ETA: 0s - loss: 4.0186 - a
cc: 0.1104Epoch 00028: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 4.0192 - acc:
0.1102 - val_loss: 4.1624 - val_acc: 0.0982
Epoch 30/100
6660/6680 [============================>.] - ETA: 0s - loss: 4.0150 - a
cc: 0.1197Epoch 00029: val_loss improved from 4.09332 to 3.88030, savin
g model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 44s - loss: 4.0147 - acc:
0.1196 - val_loss: 3.8803 - val_acc: 0.1126
Epoch 31/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.9748 - a
cc: 0.1155Epoch 00030: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.9737 - acc:
0.1157 - val_loss: 3.9244 - val_acc: 0.1246
Epoch 32/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.9467 - a
cc: 0.1248Epoch 00031: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.9477 - acc:
```

```
0.1249 - val_loss: 4.1426 - val_acc: 0.0970
Epoch 33/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.9547 - a
cc: 0.1209Epoch 00032: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.9535 - acc:
0.1211 - val_loss: 4.1140 - val_acc: 0.0790
Epoch 34/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.9021 - a
cc: 0.1243Epoch 00033: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.9000 - acc:
0.1244 - val_loss: 3.9744 - val_acc: 0.1042
Epoch 35/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.8968 - a
cc: 0.1342Epoch 00034: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.8963 - acc:
0.1340 - val_loss: 3.9554 - val_acc: 0.1222
Epoch 36/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.8536 - a
cc: 0.1339Epoch 00035: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.8533 - acc:
0.1338 - val_loss: 3.9685 - val_acc: 0.1281
Epoch 37/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.8689 - a
cc: 0.1353Epoch 00036: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.8678 - acc:
0.1353 - val_loss: 4.1747 - val_acc: 0.1030
Epoch 38/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.8501 - a
cc: 0.1354Epoch 00037: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.8513 - acc:
0.1353 - val_loss: 4.3532 - val_acc: 0.0826
Epoch 39/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.8256 - a
cc: 0.1363Epoch 00038: val_loss improved from 3.88030 to 3.86314, savin
g model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 44s - loss: 3.8245 - acc:
0.1367 - val_loss: 3.8631 - val_acc: 0.1377
Epoch 40/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.8122 - a
cc: 0.1432Epoch 00039: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.8123 - acc:
0.1431 - val_loss: 3.8966 - val_acc: 0.1186
Epoch 41/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.7894 - a
cc: 0.1389Epoch 00040: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.7893 - acc:
0.1389 - val_loss: 4.0227 - val_acc: 0.1138
Epoch 42/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.7819 - a
cc: 0.1488Epoch 00041: val_loss improved from 3.86314 to 3.76465, savin
g model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 44s - loss: 3.7814 - acc:
0.1487 - val_loss: 3.7646 - val_acc: 0.1617
Epoch 43/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.7623 - a
cc: 0.1411Epoch 00042: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.7637 - acc:
```

```
0.1409 - val_loss: 3.8124 - val_acc: 0.1293
Epoch 44/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.7509 - a
cc: 0.1495Epoch 00043: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.7506 - acc:
0.1494 - val_loss: 3.7893 - val_acc: 0.1210
Epoch 45/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.7349 - a
cc: 0.1533Epoch 00044: val_loss improved from 3.76465 to 3.68426, savin
g model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 44s - loss: 3.7340 - acc:
0.1531 - val_loss: 3.6843 - val_acc: 0.1473
Epoch 46/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.7284 - a
cc: 0.1569Epoch 00045: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.7282 - acc:
0.1567 - val_loss: 3.8777 - val_acc: 0.1293
Epoch 47/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.7055 - a
cc: 0.1629Epoch 00046: val_loss improved from 3.68426 to 3.63644, savin
g model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 44s - loss: 3.7040 - acc:
0.1632 - val_loss: 3.6364 - val_acc: 0.1545
Epoch 48/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.7232 - a
cc: 0.1575Epoch 00047: val_loss improved from 3.63644 to 3.48669, savin
g model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 44s - loss: 3.7225 - acc:
0.1576 - val_loss: 3.4867 - val_acc: 0.1832
Epoch 49/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.6919 - a
cc: 0.1674Epoch 00048: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.6923 - acc:
0.1672 - val_loss: 3.6216 - val_acc: 0.1725
Epoch 50/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.6875 - a
cc: 0.1661Epoch 00049: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.6878 - acc:
0.1657 - val_loss: 3.6169 - val_acc: 0.1725
Epoch 51/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.7096 - a
cc: 0.1604Epoch 00050: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.7089 - acc:
0.1603 - val_loss: 3.6689 - val_acc: 0.1461
Epoch 52/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.6826 - a
cc: 0.1643Epoch 00051: val_loss did not improve
6680/6680 [==============================] - 45s - loss: 3.6830 - acc:
0.1642 - val_loss: 3.8866 - val_acc: 0.1365
Epoch 53/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.6601 - a
cc: 0.1686Epoch 00052: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.6600 - acc:
0.1687 - val_loss: 3.6647 - val_acc: 0.1725
Epoch 54/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.6465 - a
cc: 0.1709Epoch 00053: val_loss did not improve
```

```
6680/6680 [==============================] - 44s - loss: 3.6470 - acc: 
0.1705 - val_loss: 3.5120 - val_acc: 0.2012
Epoch 55/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.6378 - a
cc: 0.1721Epoch 00054: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.6368 - acc: 
0.1723 - val_loss: 3.5837 - val_acc: 0.1832
Epoch 56/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.6224 - a
cc: 0.1755Epoch 00055: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.6235 - acc: 
0.1753 - val_loss: 4.4664 - val_acc: 0.1281
Epoch 57/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.6347 - a
cc: 0.1788Epoch 00056: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.6365 - acc: 
0.1786 - val_loss: 3.7279 - val_acc: 0.1772
Epoch 58/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.5862 - a
cc: 0.1794Epoch 00057: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.5868 - acc: 
0.1792 - val_loss: 3.6018 - val_acc: 0.1737
Epoch 59/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.6129 - a
cc: 0.1709Epoch 00058: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.6121 - acc: 
0.1710 - val_loss: 3.6955 - val_acc: 0.1832
Epoch 60/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.5929 - a
cc: 0.1751Epoch 00059: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.5934 - acc: 
0.1754 - val_loss: 3.6506 - val_acc: 0.1677
Epoch 61/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.5746 - a
cc: 0.1790Epoch 00060: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.5764 - acc: 
0.1789 - val_loss: 3.8951 - val_acc: 0.1281
Epoch 62/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.5489 - a
cc: 0.1881Epoch 00061: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.5505 - acc: 
0.1877 - val_loss: 3.5751 - val_acc: 0.1964
Epoch 63/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.5240 - a
cc: 0.1916Epoch 00062: val_loss improved from 3.48669 to 3.47870, savin
g model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 44s - loss: 3.5227 - acc: 
0.1916 - val_loss: 3.4787 - val_acc: 0.2036
Epoch 64/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.5541 - a
cc: 0.1862Epoch 00063: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.5556 - acc: 
0.1861 - val_loss: 3.5305 - val_acc: 0.1856
Epoch 65/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.5252 - a
cc: 0.1934Epoch 00064: val_loss improved from 3.47870 to 3.37811, savin
g model to saved_models/weights.best.from_scratch.hdf5
```

```
6680/6680 [==============================] - 44s - loss: 3.5253 - acc:
0.1933 - val_loss: 3.3781 - val_acc: 0.2024
Epoch 66/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.5324 - a
cc: 0.1916Epoch 00065: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.5319 - acc:
0.1919 - val_loss: 3.7579 - val_acc: 0.1413
Epoch 67/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.5146 - a
cc: 0.1988Epoch 00066: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.5166 - acc:
0.1990 - val_loss: 3.7626 - val_acc: 0.1533
Epoch 68/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.4891 - a
cc: 0.1977Epoch 00067: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.4893 - acc:
0.1979 - val_loss: 3.5424 - val_acc: 0.1928
Epoch 69/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.5017 - a
cc: 0.1982Epoch 00068: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.5031 - acc:
0.1979 - val_loss: 3.5446 - val_acc: 0.1832
Epoch 70/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.5036 - a
cc: 0.1967Epoch 00069: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.5046 - acc:
0.1967 - val_loss: 3.5539 - val_acc: 0.1868
Epoch 71/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.4912 - a
cc: 0.1997Epoch 00070: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.4895 - acc:
0.1997 - val_loss: 3.5194 - val_acc: 0.1988
Epoch 72/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.4690 - a
cc: 0.2084Epoch 00071: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.4681 - acc:
0.2082 - val_loss: 3.5963 - val_acc: 0.1880
Epoch 73/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.4740 - a
cc: 0.2110Epoch 00072: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.4741 - acc:
0.2111 - val_loss: 3.4708 - val_acc: 0.1952
Epoch 74/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.4642 - a
cc: 0.2102Epoch 00073: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.4628 - acc:
0.2105 - val_loss: 3.4938 - val_acc: 0.1928
Epoch 75/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.4417 - a
cc: 0.2128Epoch 00074: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.4435 - acc:
0.2124 - val_loss: 3.5365 - val_acc: 0.1904
Epoch 76/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.4282 - a
cc: 0.2071Epoch 00075: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.4287 - acc:
0.2070 - val_loss: 3.4916 - val_acc: 0.2108
```

```
Epoch 77/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.4634 - a
cc: 0.2053Epoch 00076: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.4628 - acc:
0.2054 - val_loss: 3.4759 - val_acc: 0.2012
Epoch 78/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.4429 - a
cc: 0.2072Epoch 00077: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.4435 - acc:
0.2069 - val_loss: 3.4411 - val_acc: 0.2060
Epoch 79/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.4161 - a
cc: 0.2099Epoch 00078: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.4176 - acc:
0.2100 - val_loss: 3.5466 - val_acc: 0.1844
Epoch 80/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.4263 - a
cc: 0.2167Epoch 00079: val_loss did not improve
6680/6680 [==============================] - 45s - loss: 3.4249 - acc:
0.2168 - val_loss: 4.2158 - val_acc: 0.1485
Epoch 81/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.4075 - a
cc: 0.2062Epoch 00080: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.4083 - acc:
0.2061 - val_loss: 3.4513 - val_acc: 0.1916
Epoch 82/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.3972 - a
cc: 0.2198Epoch 00081: val_loss improved from 3.37811 to 3.35560, savin
g model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 44s - loss: 3.3976 - acc:
0.2198 - val_loss: 3.3556 - val_acc: 0.2275
Epoch 83/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.3698 - a
cc: 0.2203Epoch 00082: val_loss improved from 3.35560 to 3.20756, savin
g model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 44s - loss: 3.3681 - acc:
0.2208 - val_loss: 3.2076 - val_acc: 0.2575
Epoch 84/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.3527 - a
cc: 0.2281Epoch 00083: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.3545 - acc:
0.2277 - val_loss: 3.5966 - val_acc: 0.1856
Epoch 85/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.3847 - a
cc: 0.2150Epoch 00084: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.3847 - acc:
0.2153 - val_loss: 3.3946 - val_acc: 0.2072
Epoch 86/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.3771 - a
cc: 0.2228Epoch 00085: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.3788 - acc:
0.2225 - val_loss: 3.5799 - val_acc: 0.1772
Epoch 87/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.3914 - a
cc: 0.2234Epoch 00086: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.3922 - acc:
0.2234 - val_loss: 3.3453 - val_acc: 0.2108
```

```
Epoch 88/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.3433 - a
cc: 0.2200Epoch 00087: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.3431 - acc:
0.2204 - val_loss: 3.2642 - val_acc: 0.2287
Epoch 89/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.3114 - a
cc: 0.2323Epoch 00088: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.3121 - acc:
0.2320 - val_loss: 3.3034 - val_acc: 0.2180
Epoch 90/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.3114 - a
cc: 0.2311Epoch 00089: val_loss improved from 3.20756 to 3.19948, savin
g model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 44s - loss: 3.3130 - acc:
0.2307 - val_loss: 3.1995 - val_acc: 0.2563
Epoch 91/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.3363 - a
cc: 0.2218Epoch 00090: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.3392 - acc:
0.2216 - val_loss: 3.4618 - val_acc: 0.2192
Epoch 92/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.3338 - a
cc: 0.2288Epoch 00091: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.3349 - acc:
0.2287 - val_loss: 3.2688 - val_acc: 0.2503
Epoch 93/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.2963 - a
cc: 0.2308Epoch 00092: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.2947 - acc:
0.2311 - val_loss: 3.3119 - val_acc: 0.2275
Epoch 94/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.3171 - a
cc: 0.2324Epoch 00093: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.3176 - acc:
0.2319 - val_loss: 3.3520 - val_acc: 0.2204
Epoch 95/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.3216 - a
cc: 0.2278Epoch 00094: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.3227 - acc:
0.2275 - val_loss: 3.2537 - val_acc: 0.2383
Epoch 96/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.3201 - a
cc: 0.2320Epoch 00095: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.3201 - acc:
0.2317 - val_loss: 3.3089 - val_acc: 0.2180
Epoch 97/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.2840 - a
cc: 0.2371Epoch 00096: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.2840 - acc:
0.2373 - val_loss: 3.2110 - val_acc: 0.2251
Epoch 98/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.2529 - a
cc: 0.2459Epoch 00097: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.2528 - acc:
0.2457 - val_loss: 3.2578 - val_acc: 0.2311
Epoch 99/100
```

```
6660/6680 [============================>.] - ETA: 0s - loss: 3.2832 - a
cc: 0.2368Epoch 00098: val_loss did not improve
6680/6680 [==============================] - 44s - loss: 3.2818 - acc:
0.2371 - val_loss: 3.2949 - val_acc: 0.2359
Epoch 100/100
6660/6680 [============================>.] - ETA: 0s - loss: 3.2815 - a
cc: 0.2434Epoch 00099: val_loss improved from 3.19948 to 3.17083, savin
g model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 44s - loss: 3.2818 - acc:
0.2437 - val_loss: 3.1708 - val_acc: 0.2623
```

```
Out[34]: <keras.callbacks.History at 0x7f573a1e8fd0>
```

Load the Model with the Best Validation Loss

```
In [35]: 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 [36]: # 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(te
         st_targets, axis=1))/len(dog_breed_predictions)
         print('Test accuracy: %.4f%%' % test_accuracy)
```

```
Test accuracy: 25.7177%
```

```
In [99]: 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 [100]: 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_35  (None, 512)               0         
_________________________________________________________________
dense_52 (Dense)             (None, 133)               68229     
=================================================================
Total params: 68,229.0
Trainable params: 68,229.0
Non-trainable params: 0.0
_________________________________________________________________
```

Compile the Model

```
In [101]: VGG16_model.compile(loss='categorical_crossentropy', optimizer='rmsprop'
          , metrics=['accuracy'])
```

Train the Model

In [102]:
```
checkpointer = ModelCheckpoint(filepath='saved_models/weights.best.VGG1
6.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
6580/6680 [============================>.] - ETA: 0s - loss: 12.6723 -
acc: 0.0933Epoch 00000: val_loss improved from inf to 10.90994, saving
model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s - loss: 12.6382 - acc:
0.0954 - val_loss: 10.9099 - val_acc: 0.1868
Epoch 2/20
6440/6680 [===========================>..] - ETA: 0s - loss: 10.3011 -
acc: 0.2599Epoch 00001: val_loss improved from 10.90994 to 10.12594, sa
ving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 10.3093 - acc:
0.2608 - val_loss: 10.1259 - val_acc: 0.2814
Epoch 3/20
6560/6680 [============================>.] - ETA: 0s - loss: 9.7783 - a
cc: 0.3248Epoch 00002: val_loss improved from 10.12594 to 9.83853, savi
ng model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 9.7906 - acc:
0.3243 - val_loss: 9.8385 - val_acc: 0.3102
Epoch 4/20
6460/6680 [============================>.] - ETA: 0s - loss: 9.5733 - a
cc: 0.3610Epoch 00003: val_loss improved from 9.83853 to 9.78336, savin
g model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 9.5603 - acc:
0.3614 - val_loss: 9.7834 - val_acc: 0.3293
Epoch 5/20
6460/6680 [============================>.] - ETA: 0s - loss: 9.4759 - a
cc: 0.3754Epoch 00004: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 9.4593 - acc:
0.3763 - val_loss: 9.8329 - val_acc: 0.3269
Epoch 6/20
6620/6680 [============================>.] - ETA: 0s - loss: 9.3755 - a
cc: 0.3915Epoch 00005: val_loss improved from 9.78336 to 9.72854, savin
g model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 9.3775 - acc:
0.3913 - val_loss: 9.7285 - val_acc: 0.3413
Epoch 7/20
6620/6680 [============================>.] - ETA: 0s - loss: 9.2546 - a
cc: 0.4035Epoch 00006: val_loss improved from 9.72854 to 9.63104, savin
g model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 9.2415 - acc:
0.4045 - val_loss: 9.6310 - val_acc: 0.3473
Epoch 8/20
6480/6680 [============================>.] - ETA: 0s - loss: 9.1598 - a
cc: 0.4116Epoch 00007: val_loss improved from 9.63104 to 9.59767, savin
g model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 9.1355 - acc:
0.4129 - val_loss: 9.5977 - val_acc: 0.3449
Epoch 9/20
6600/6680 [============================>.] - ETA: 0s - loss: 9.0696 - a
cc: 0.4202Epoch 00008: val_loss improved from 9.59767 to 9.53648, savin
g model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 9.0652 - acc:
0.4204 - val_loss: 9.5365 - val_acc: 0.3545
Epoch 10/20
6440/6680 [===========================>..] - ETA: 0s - loss: 9.0046 - a
cc: 0.4272Epoch 00009: val_loss did not improve
```

```
6680/6680 [==============================] - 1s - loss: 9.0175 - acc:
0.4260 - val_loss: 9.6292 - val_acc: 0.3485
Epoch 11/20
6600/6680 [============================>.] - ETA: 0s - loss: 8.8852 - a
cc: 0.4311Epoch 00010: val_loss improved from 9.53648 to 9.38393, savin
g model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.8895 - acc:
0.4308 - val_loss: 9.3839 - val_acc: 0.3473
Epoch 12/20
6620/6680 [============================>.] - ETA: 0s - loss: 8.7524 - a
cc: 0.4400Epoch 00011: val_loss improved from 9.38393 to 9.27037, savin
g model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.7488 - acc:
0.4403 - val_loss: 9.2704 - val_acc: 0.3581
Epoch 13/20
6560/6680 [============================>.] - ETA: 0s - loss: 8.6110 - a
cc: 0.4457Epoch 00012: val_loss improved from 9.27037 to 9.01565, savin
g model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.6072 - acc:
0.4463 - val_loss: 9.0157 - val_acc: 0.3832
Epoch 14/20
6660/6680 [============================>.] - ETA: 0s - loss: 8.4342 - a
cc: 0.4640Epoch 00013: val_loss improved from 9.01565 to 8.97419, savin
g model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.4241 - acc:
0.4644 - val_loss: 8.9742 - val_acc: 0.3844
Epoch 15/20
6460/6680 [============================>.] - ETA: 0s - loss: 8.3682 - a
cc: 0.4724Epoch 00014: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 8.3937 - acc:
0.4704 - val_loss: 9.1012 - val_acc: 0.3725
Epoch 16/20
6600/6680 [============================>.] - ETA: 0s - loss: 8.3716 - a
cc: 0.4712Epoch 00015: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 8.3684 - acc:
0.4713 - val_loss: 9.0751 - val_acc: 0.3749
Epoch 17/20
6460/6680 [============================>.] - ETA: 0s - loss: 8.3154 - a
cc: 0.4768Epoch 00016: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 8.3356 - acc:
0.4751 - val_loss: 9.0500 - val_acc: 0.3772
Epoch 18/20
6640/6680 [============================>.] - ETA: 0s - loss: 8.2525 - a
cc: 0.4770Epoch 00017: val_loss improved from 8.97419 to 8.94185, savin
g model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.2564 - acc:
0.4766 - val_loss: 8.9419 - val_acc: 0.3916
Epoch 19/20
6640/6680 [============================>.] - ETA: 0s - loss: 8.1683 - a
cc: 0.4854Epoch 00018: val_loss improved from 8.94185 to 8.87122, savin
g model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.1676 - acc:
0.4855 - val_loss: 8.8712 - val_acc: 0.3892
Epoch 20/20
6580/6680 [============================>.] - ETA: 0s - loss: 8.1627 - a
cc: 0.4883Epoch 00019: val_loss did not improve
```

```
         6680/6680 [==============================] - 1s - loss: 8.1515 - acc:
         0.4891 - val_loss: 8.8935 - val_acc: 0.3880

Out[102]: <keras.callbacks.History at 0x7f3a7e869e10>
```

Load the Model with the Best Validation Loss

```
In [103]: 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 [104]: # get index of predicted dog breed for each image in test set
          VGG16_predictions = [np.argmax(VGG16_model.predict(np.expand_dims(featur
          e, axis=0))) for feature in test_VGG16]

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

          Test accuracy: 36.6029%
```

Predict Dog Breed with the Model

```
In [105]: 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 [42]:

```
### TODO: Obtain bottleneck features from another pre-trained CNN.
network = 'Xception'
checkpointer_path = 'saved_models/weights.best.{0}.hdf5'.format(network)
feature_path = 'bottleneck_features/Dog{0}Data.npz'.format(network)
bottleneck_features = np.load(feature_path)
train_cnn = bottleneck_features['train']
valid_cnn = bottleneck_features['valid']
test_cnn = 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:

For this task, the most important step is choose the right model. Because we have a relatively small dataset, and I believe the new data set is similar to the dog images that are currently in the ImageNet, which is used in obtaining the advanced models such as VGG, resNet, Xception and Inception.

Choosing the model

I have tried all of the four pre-trained models specified for this project. Results quickly emerged that the VGG-19 model is greatly out performed by Resnet-50, Xception and Inception. VGG-19 model does perform better than the VGG-16 used earlier in the project.

Among Resnet-50, Xception and Inception, with a configuration of 20 epochs, batch size of 20, a global average pooling layer and a fully connected layer, they all perform fast and fairly accurate(80%); However, Xception seems to perform slightly better. So I decided to use Xception.

Theoretically (in terms of CNN development) this makes sense. VGG networks is one of the earliest CNN networks that achieved astounding success. However, VGG networks have a few drawbacks: 1) it is slow; 2) the network architecture weights themselves are quite large (in terms of disk/bandwidth; More importantly the networks suffered from the problem of vanishing gradients, in which gradient signals from the error function decreased exponentially as they backpropogated to earlier layers. As a result, after a certain threshold, as the depth of network increase, the performance degrades.

Resnet tries a different approach. Instead of trying to learn an underlying mapping from x to H(x), learn the difference between the two, or the “residual.” Resnet performs very well and it solves the vanishing gradients problem. Because that, a lot more layers can be added to the model and can still continously learn from the data.

Inception model employes multiple configurations in parallel in each step. For example, at each layer, Inception uses a set of transformation layers (one 2 by 2, 3 by 3, 5 by 5) with a pooling layer, then let the model pick the "winner". To solve the enormous computation cost, Inception also uses 1x1 convolutions to perform dimensionality reduction.

Xception is Inception on steroids (extreme inception). It also maps out the spatial correlations for each output channel separately and then use a 1x1 depthwise convolution to capture cross-channel correlation.

"cross-channel correlations and spatial correlations are sufficiently decoupled that it is preferable not to map them jointly."

So Xceptons "wins"!

Fine-tuning

Once the model is chosen, I simply stripped out the top layer of the trained model and added a global average pooling layer and a dense layer of 133 neurons. To combat overfitting issue, I added dropout layer with a small regularization rate of 0.2 (when i made it bigger, the performance degraded).

Also thanks to my reviewer, I used `adam` optimizer instead of `rmsprop`. It did work better.

The test accuracy using this model with only 50 epochs is ######86.1244%######. I think it is not bad. It improved .5% from my last submission.

Dog breed identifier web app

After the model is trained and tested, I saved the model in .h5 format which contains the architecture and the weights of the model. Then I used keras.load_model to load the model into a flask application. Together with gunicorn, I am ablt to run the dog breed identifier online. Please try it out :)

Dog Breed Identifier (https://dog-breed-identifier.herokuapp.com/)

References:

An Intuitive Guide to Deep Network Architectures (https://towardsdatascience.com/an-intuitive-guide-to-deep-network-architectures-65fdc477db41)

ResNet, AlexNet, VGGNet, Inception: Understanding various architectures of Convolutional Networks (http://cv-tricks.com/cnn/understand-resnet-alexnet-vgg-inception/)

In [43]:
```
### TODO: Define your architecture.
cnn_model = Sequential()
cnn_model.add(GlobalAveragePooling2D(input_shape=train_cnn.shape[1:]))
cnn_model.add(Dropout(.2))
cnn_model.add(Dense(133, activation='softmax'))

cnn_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
global_average_pooling2d_3 ( (None, 2048)              0         
_________________________________________________________________
dropout_5 (Dropout)          (None, 2048)              0         
_________________________________________________________________
dense_4 (Dense)              (None, 133)               272517    
=================================================================
Total params: 272,517.0
Trainable params: 272,517.0
Non-trainable params: 0.0
_________________________________________________________________
```

(IMPLEMENTATION) Compile the Model

```
In [44]: ### TODO: Compile the model.
         import keras
         opt = keras.optimizers.Adam(lr=0.0005)
         cnn_model.compile(loss='categorical_crossentropy', optimizer=opt, metric
         s=['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 [45]:
```
### TODO: Train the model.
from keras.callbacks import ModelCheckpoint
checkpointer = ModelCheckpoint(filepath=checkpointer_path,
                               verbose=1, save_best_only=True)
epochs_max = 20

cnn_model.fit(train_cnn, train_targets,
          validation_data=(valid_cnn, valid_targets),
          epochs=epochs_max, batch_size=20, callbacks=[checkpointer], ve
rbose=1)
```

```
Train on 6680 samples, validate on 835 samples
Epoch 1/20
6600/6680 [============================>.] - ETA: 0s - loss: 1.5504 - a
cc: 0.6774Epoch 00000: val_loss improved from inf to 0.64606, saving mo
del to saved_models/weights.best.Xception.hdf5
6680/6680 [==============================] - 12s - loss: 1.5402 - acc:
0.6789 - val_loss: 0.6461 - val_acc: 0.8287
Epoch 2/20
6600/6680 [============================>.] - ETA: 0s - loss: 0.4789 - a
cc: 0.8745Epoch 00001: val_loss improved from 0.64606 to 0.51027, savin
g model to saved_models/weights.best.Xception.hdf5
6680/6680 [==============================] - 4s - loss: 0.4776 - acc:
0.8747 - val_loss: 0.5103 - val_acc: 0.8479
Epoch 3/20
6620/6680 [============================>.] - ETA: 0s - loss: 0.3470 - a
cc: 0.9020Epoch 00002: val_loss improved from 0.51027 to 0.46808, savin
g model to saved_models/weights.best.Xception.hdf5
6680/6680 [==============================] - 4s - loss: 0.3474 - acc:
0.9018 - val_loss: 0.4681 - val_acc: 0.8551
Epoch 4/20
6580/6680 [============================>.] - ETA: 0s - loss: 0.2736 - a
cc: 0.9263Epoch 00003: val_loss improved from 0.46808 to 0.46105, savin
g model to saved_models/weights.best.Xception.hdf5
6680/6680 [==============================] - 4s - loss: 0.2736 - acc:
0.9263 - val_loss: 0.4611 - val_acc: 0.8467
Epoch 5/20
6640/6680 [============================>.] - ETA: 0s - loss: 0.2169 - a
cc: 0.9455Epoch 00004: val_loss improved from 0.46105 to 0.44970, savin
g model to saved_models/weights.best.Xception.hdf5
6680/6680 [==============================] - 4s - loss: 0.2174 - acc:
0.9452 - val_loss: 0.4497 - val_acc: 0.8551
Epoch 6/20
6580/6680 [============================>.] - ETA: 0s - loss: 0.1853 - a
cc: 0.9540Epoch 00005: val_loss improved from 0.44970 to 0.44067, savin
g model to saved_models/weights.best.Xception.hdf5
6680/6680 [==============================] - 4s - loss: 0.1846 - acc:
0.9545 - val_loss: 0.4407 - val_acc: 0.8635
Epoch 7/20
6640/6680 [============================>.] - ETA: 0s - loss: 0.1487 - a
cc: 0.9688Epoch 00006: val_loss improved from 0.44067 to 0.44032, savin
g model to saved_models/weights.best.Xception.hdf5
6680/6680 [==============================] - 4s - loss: 0.1492 - acc:
0.9686 - val_loss: 0.4403 - val_acc: 0.8611
Epoch 8/20
6620/6680 [============================>.] - ETA: 0s - loss: 0.1272 - a
cc: 0.9761Epoch 00007: val_loss improved from 0.44032 to 0.43736, savin
g model to saved_models/weights.best.Xception.hdf5
6680/6680 [==============================] - 4s - loss: 0.1269 - acc:
0.9763 - val_loss: 0.4374 - val_acc: 0.8527
Epoch 9/20
6660/6680 [============================>.] - ETA: 0s - loss: 0.1105 - a
cc: 0.9788Epoch 00008: val_loss did not improve
6680/6680 [==============================] - 4s - loss: 0.1104 - acc:
0.9789 - val_loss: 0.4388 - val_acc: 0.8647
Epoch 10/20
6600/6680 [============================>.] - ETA: 0s - loss: 0.0951 - a
cc: 0.9842Epoch 00009: val_loss did not improve
```

```
        6680/6680 [==============================] - 4s - loss: 0.0948 - acc:
        0.9841 - val_loss: 0.4377 - val_acc: 0.8623
        Epoch 11/20
        6640/6680 [============================>.] - ETA: 0s - loss: 0.0828 - a
        cc: 0.9875Epoch 00010: val_loss improved from 0.43736 to 0.43232, savin
        g model to saved_models/weights.best.Xception.hdf5
        6680/6680 [==============================] - 4s - loss: 0.0828 - acc:
        0.9874 - val_loss: 0.4323 - val_acc: 0.8611
        Epoch 12/20
        6640/6680 [============================>.] - ETA: 0s - loss: 0.0722 - a
        cc: 0.9904Epoch 00011: val_loss did not improve
        6680/6680 [==============================] - 4s - loss: 0.0725 - acc:
        0.9901 - val_loss: 0.4437 - val_acc: 0.8563
        Epoch 13/20
        6660/6680 [============================>.] - ETA: 0s - loss: 0.0653 - a
        cc: 0.9914Epoch 00012: val_loss did not improve
        6680/6680 [==============================] - 4s - loss: 0.0651 - acc:
        0.9915 - val_loss: 0.4463 - val_acc: 0.8659
        Epoch 14/20
        6640/6680 [============================>.] - ETA: 0s - loss: 0.0579 - a
        cc: 0.9932Epoch 00013: val_loss did not improve
        6680/6680 [==============================] - 4s - loss: 0.0583 - acc:
        0.9930 - val_loss: 0.4478 - val_acc: 0.8647
        Epoch 15/20
        6660/6680 [============================>.] - ETA: 0s - loss: 0.0512 - a
        cc: 0.9941Epoch 00014: val_loss did not improve
        6680/6680 [==============================] - 5s - loss: 0.0512 - acc:
        0.9942 - val_loss: 0.4619 - val_acc: 0.8599
        Epoch 16/20
        6660/6680 [============================>.] - ETA: 0s - loss: 0.0476 - a
        cc: 0.9940Epoch 00015: val_loss did not improve
        6680/6680 [==============================] - 5s - loss: 0.0476 - acc:
        0.9940 - val_loss: 0.4685 - val_acc: 0.8527
        Epoch 17/20
        6660/6680 [============================>.] - ETA: 0s - loss: 0.0432 - a
        cc: 0.9953Epoch 00016: val_loss did not improve
        6680/6680 [==============================] - 5s - loss: 0.0433 - acc:
        0.9954 - val_loss: 0.4592 - val_acc: 0.8635
        Epoch 18/20
        6620/6680 [============================>.] - ETA: 0s - loss: 0.0389 - a
        cc: 0.9943Epoch 00017: val_loss did not improve
        6680/6680 [==============================] - 4s - loss: 0.0387 - acc:
        0.9943 - val_loss: 0.4770 - val_acc: 0.8611
        Epoch 19/20
        6600/6680 [============================>.] - ETA: 0s - loss: 0.0349 - a
        cc: 0.9956Epoch 00018: val_loss did not improve
        6680/6680 [==============================] - 4s - loss: 0.0351 - acc:
        0.9955 - val_loss: 0.4646 - val_acc: 0.8647
        Epoch 20/20
        6600/6680 [============================>.] - ETA: 0s - loss: 0.0317 - a
        cc: 0.9958Epoch 00019: val_loss did not improve
        6680/6680 [==============================] - 4s - loss: 0.0317 - acc:
        0.9958 - val_loss: 0.4746 - val_acc: 0.8587

Out[45]: <keras.callbacks.History at 0x7f621838fc50>
```

(IMPLEMENTATION) Load the Model with the Best Validation Loss

```
In [46]: ### TODO: Load the model weights with the best validation loss.
         cnn_model.load_weights(checkpointer_path)
```

(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 [47]: ### TODO: Calculate classification accuracy on the test dataset.
         # get index of predicted dog breed for each image in test set
         cnn_predictions = [np.argmax(cnn_model.predict(np.expand_dims(feature, a
         xis=0))) for feature in test_cnn]

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

         Test accuracy: 86.1244%
```

(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 [48]:
```
### TODO: Write a function that takes a path to an image as input
### and returns the dog breed that is predicted by the model.
from extract_bottleneck_features import *
def Xception_predict_breed(img_path):
    # extract bottleneck features
    bottleneck_feature = extract_Xception(path_to_tensor(img_path))
    # obtain predicted vector
    predicted_vector = cnn_model.predict(bottleneck_feature)
    # return dog breed that is predicted by the model
    return np.max(predicted_vector), dog_names[np.argmax(predicted_vecto
r)]
```

Step 6: Write your Algorithm

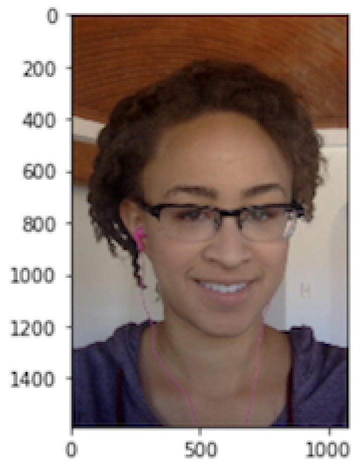
Write an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then,

- 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 [49]:
```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.
def final_predict(img_path):
    dog_face_detected = 1 if dog_detector(img_path) else 0
    human_face_detected = 1 if face_detector(img_path) else 0
    if dog_face_detected:
        res = Xception_predict_breed(img_path)
        return 'dog', res
    if human_face_detected:
        res = Xception_predict_breed(img_path)
        return 'human', res  #'This seems like a human face, but may I s
ay it resembles a {0}? I think I am {1}% confident'.format(res[1], res
[0] )
    else:
        return 'error', None  #
```

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 better than I expected, though I am a little disappointed that I am not able to improve it further. I was hoping i could make it to 90%.

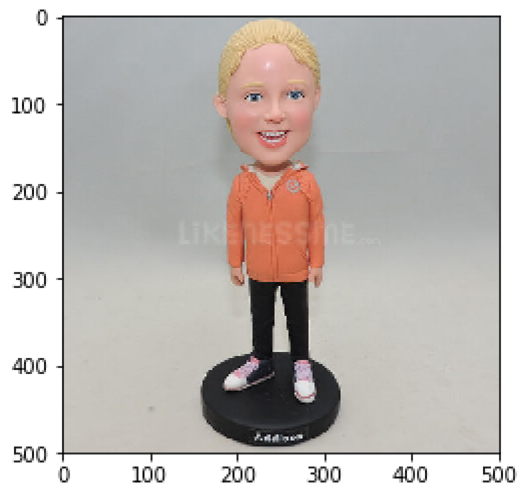
Three possible points of improvement for my algorithm could be:

1) Better handling of overfitting, better dropout, regularization;

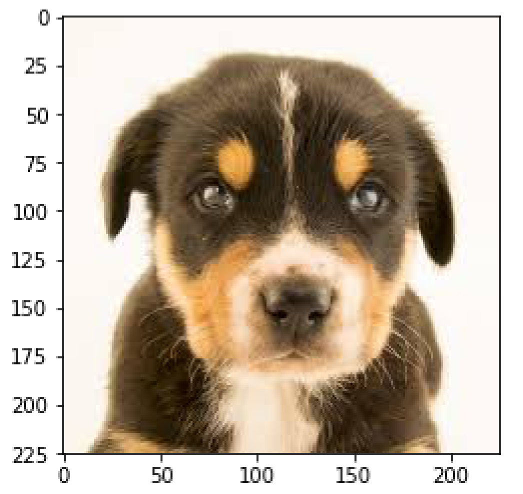
2) Obtain more training images;

3) Augument the training images by applying scaling, distortion, rotationm etc.

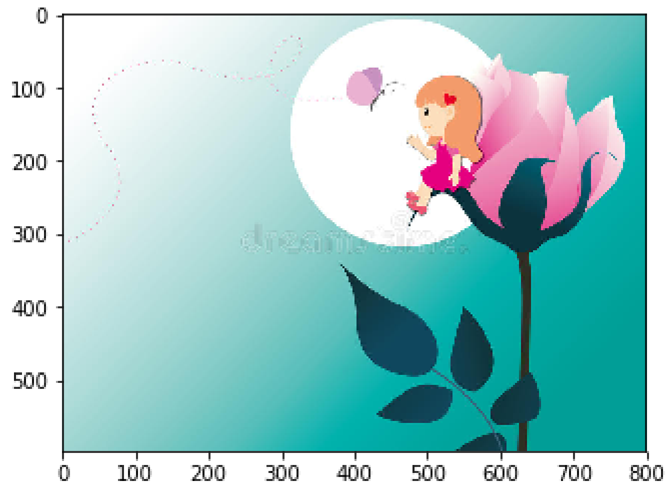
In [50]:
```
from os import listdir

imgdir = 'my-images'

for img in listdir(imgdir):
    img_path= imgdir + '/' + img
    img=mpimg.imread(img_path)
    imgplot = plt.imshow(img)
    plt.show()
    predictName, res = final_predict(img_path)
    if predictName == 'error':
        print('Hmmm, something is wrong. This does not look anything lik
e a dog or a human')
    else:
        if predictName == 'dog':
            print('This looks like {0}, I am {1:.2f}% confident'.format(
res[1], res[0]*100 ))
        else:
            print('This seems like a human, but may I say it resembles a
 {0}? I am {1:.2f}% confident'.format(res[1], res[0]*100))




## 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.
```



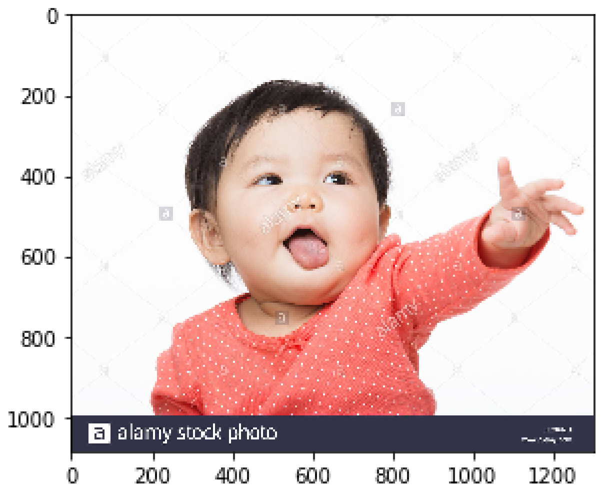
```
This seems like a human, but may I say it resembles a Chinese_crested?
I am 18.74% confident
```



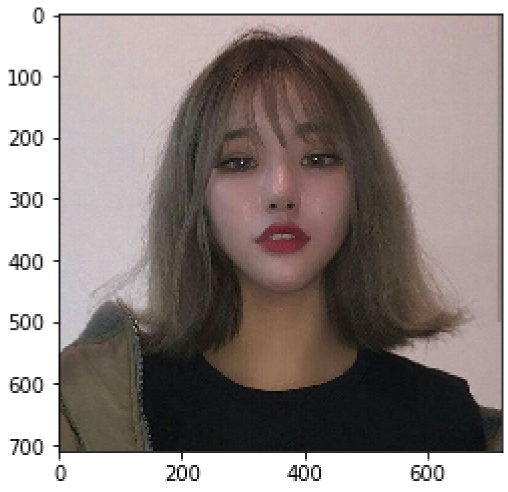
```
This looks like Greater_swiss_mountain_dog, I am 97.40% confident
```



```
Hmmm, something is wrong. This does not look anything like a dog or a h
uman
```

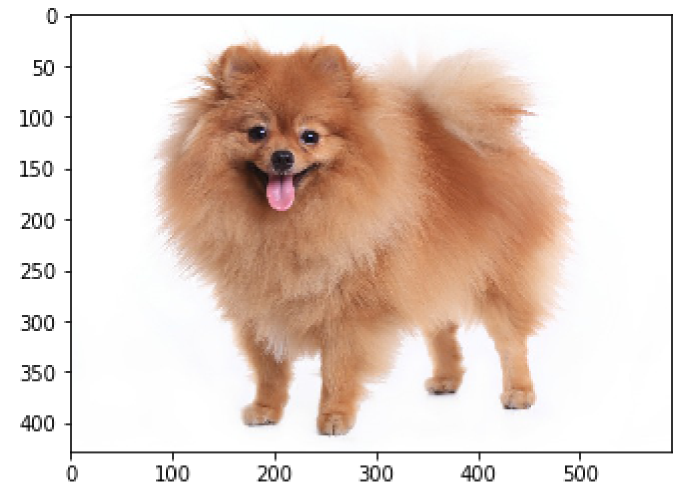


```
This seems like a human, but may I say it resembles a Dachshund? I am 1
7.14% confident
```

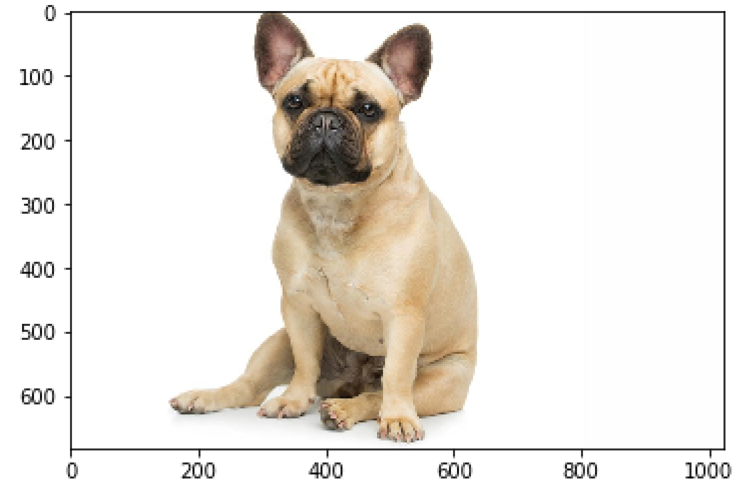


```
This seems like a human, but may I say it resembles a Afghan_hound? I a
m 10.71% confident
```

```
This looks like Golden_retriever, I am 99.10% confident
```



```
This looks like Pomeranian, I am 99.73% confident
```



```
This looks like French_bulldog, I am 99.26% confident
```