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!).

Sample Dog Output

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

The Road Ahead

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

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

Step 0: Import Datasets

Import Dog Dataset

In the code cell below, we import a dataset of dog images. We populate a few variables through the use of the `load_files` function from the scikit-learn library:

- `train_files`, `valid_files`, `test_files` - numpy arrays containing file paths to images
- `train_targets`, `valid_targets`, `test_targets` - numpy arrays containing onehot-encoded classification labels
- `dog_names` - list of string-valued dog breed names for translating labels

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

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

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

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

# print statistics about the dataset
print('There are %d total dog categories.' % len(dog_names))
print('There are %s total dog images.\n' % len(np.hstack([train_files, 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 [2]:
```
import random
random.seed(8675309)

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

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

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

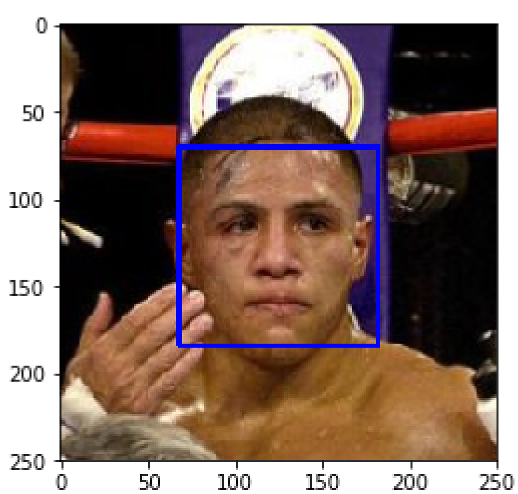
Step 1: Detect Humans

We use OpenCV's implementation of Haar feature-based cascade classifiers (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 [3]:
```
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 [4]:
```
# returns "True" if face is detected in image stored at img_path
def face_detector(img_path):
    img = cv2.imread(img_path)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    faces = face_cascade.detectMultiScale(gray)
    return len(faces) > 0
```

(IMPLEMENTATION) Assess the Human Face Detector

Question 1: Use the code cell below to test the performance of the `face_detector` function.

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

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

Answer: human_files = 98.0% dog_files = 11.0

In [5]:
```
human_files_short = human_files[:100]
dog_files_short = train_files[:100]
# Do NOT modify the code above this line.

## TODO: Test the performance of the face_detector algorithm
## on the images in human_files_short and dog_files_short.
def face_percentage_detector(files):
    num_humans = 0
    for f in files:
        if face_detector(f):
            num_humans += 1
    return num_humans/len(files)*100

print("Percentage of humans detected in 'human_files_short':", face_perc
entage_detector(human_files_short))
print("Percentage of humans detected in 'dog_files_short':", face_percen
tage_detector(dog_files_short))
```

```
Percentage of humans detected in 'human_files_short': 98.0
Percentage of humans detected in 'dog_files_short': 11.0
```

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

Answer: In my opinion, we should communicate to the user that we accept human images only when they provide a clear view of the face. But for the fun on it, I tried to raise the human detection percentage up via combining different opencv2 models. In this case, I've combined face_alt and upperbody opencv. I did manage to raise the human detection percentage for 'human_files_short' but the 'dog_files_short' data percentage went up as well unfortunately.

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

In [6]:

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

import os

# haarcascade_file = 'haarcascade_frontalface_default.xml'
haarcascade_file = 'haarcascade_upperbody.xml'
haarcascade_link = 'https://raw.githubusercontent.com/opencv/opencv/mast
er/data/haarcascades/%s' % haarcascade_file

# os.system('ls haarcascades')
os.system("wget %s -P haarcascades" % haarcascade_link)

# import subprocess
# print(subprocess.check_output(['ls','-l', 'haarcascades']))

haar_cascade = cv2.CascadeClassifier('haarcascades/%s' % haarcascade_fil
e)
def human_detector(img_path):
    img = cv2.imread(img_path)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    faces = haar_cascade.detectMultiScale(gray)
    return len(faces) > 0

def human_percentage_detector(files):
    num_humans = 0
    for f in files:
        # Use face and upperbody detection here
        if face_detector(f) or human_detector(f):
            num_humans += 1
    return num_humans/len(files)*100

print("Percentage of humans detected in 'human_files_short':", human_per
centage_detector(human_files_short))
print("Percentage of humans detected in 'dog_files_short':", human_perce
ntage_detector(dog_files_short))
```

```
Percentage of humans detected in 'human_files_short': 99.0
Percentage of humans detected in 'dog_files_short': 27.0
```

Step 2: Detect Dogs

In this section, we use a pre-trained [ResNet-50 (http://ethereon.github.io/netscope/#/gist/db945b393d40bfa26006)](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/)](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)](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 [7]:
```python
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 [8]:
```python
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 [9]:

```
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 [10]:

```
### 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: human_files_short = 1% dog_files_short = 100%

In [11]:
```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.

def dog_percentage_detector(files):
    num_dogs = 0
    for f in files:
        # Use face and upperbody detection here
        if dog_detector(f):
            num_dogs += 1
    return num_dogs/len(files)*100

print("Percentage of dog detected in 'human_files_short':", dog_percenta
ge_detector(human_files_short))
print("Percentage of dog detected in 'dog_files_short':", dog_percentage
_detector(dog_files_short))
```

```
Percentage of dog detected in 'human_files_short': 1.0
Percentage of dog detected in 'dog_files_short': 100.0
```

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

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 [12]:

```
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

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

```
100%|██████████| 6680/6680 [00:55<00:00, 121.23it/s]
100%|██████████| 835/835 [00:06<00:00, 134.11it/s]
100%|██████████| 836/836 [00:06<00:00, 135.42it/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:

Sample CNN

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: Initially, I've tried the architecture above, then, I tried the one provided by Alexis in Lesson 5-13 and compared the accuracy. It looks like the one provided by Alexis did better so I went with that. I then changed the Flatten layer to GlobalAveragePooling2D and that seemed to improve. Then I added a Dropout layer between the last 2 Dense layers and that seemed to improve the accuracy even more.

In [13]:
```python
from keras.layers import Conv2D, MaxPooling2D, GlobalAveragePooling2D
from keras.layers import Dropout, Flatten, Dense
from keras.models import Sequential

model = Sequential()

### TODO: Define your architecture.
# print(train_tensors[0].shape)
# print(valid_tensors.shape)
# print(valid_targets.shape)
# print(valid_targets[0])
# print(len(train_tensors[0][0]))

model.add(Conv2D(filters=16, kernel_size=2, padding='same', input_shape=
train_tensors[0].shape))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=32, kernel_size=2, padding='same'))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=64, kernel_size=2, padding='same'))
model.add(MaxPooling2D(pool_size=2))
model.add(GlobalAveragePooling2D(data_format=None))
# model.add(Dense(133, activation='softmax'))

# model.add(Flatten())
model.add(Dense(500, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(133, activation='softmax'))
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_1 (Conv2D)            (None, 224, 224, 16)      208       
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 112, 112, 16)      0         
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 112, 112, 32)      2080      
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 56, 56, 32)        0         
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 56, 56, 64)        8256      
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 28, 28, 64)        0         
_________________________________________________________________
global_average_pooling2d_1 ( (None, 64)                0         
_________________________________________________________________
dense_1 (Dense)              (None, 500)               32500     
_________________________________________________________________
dropout_1 (Dropout)          (None, 500)               0         
_________________________________________________________________
dense_2 (Dense)              (None, 133)               66633     
=================================================================
Total params: 109,677
Trainable params: 109,677
Non-trainable params: 0
_________________________________________________________________
```

Compile the Model

In [14]:
```
model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metr
ics=['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 [15]:
```
from keras.callbacks import ModelCheckpoint

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

epochs = 50

### 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 1/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.8692 - a
cc: 0.0111Epoch 00000: val_loss improved from inf to 4.83040, saving mo
del to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 111s - loss: 4.8693 - acc:
0.0111 - val_loss: 4.8304 - val_acc: 0.0156
Epoch 2/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.8121 - a
cc: 0.0183Epoch 00001: val_loss improved from 4.83040 to 4.80889, savin
g model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 110s - loss: 4.8116 - acc:
0.0184 - val_loss: 4.8089 - val_acc: 0.0228
Epoch 3/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.7826 - a
cc: 0.0225Epoch 00002: val_loss improved from 4.80889 to 4.79582, savin
g model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 110s - loss: 4.7823 - acc:
0.0228 - val_loss: 4.7958 - val_acc: 0.0168
Epoch 4/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.7562 - a
cc: 0.0230Epoch 00003: val_loss improved from 4.79582 to 4.78940, savin
g model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 110s - loss: 4.7564 - acc:
0.0229 - val_loss: 4.7894 - val_acc: 0.0228
Epoch 5/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.7260 - a
cc: 0.0288Epoch 00004: val_loss improved from 4.78940 to 4.74742, savin
g model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 110s - loss: 4.7267 - acc:
0.0287 - val_loss: 4.7474 - val_acc: 0.0204
Epoch 6/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.6797 - a
cc: 0.0315Epoch 00005: val_loss improved from 4.74742 to 4.72437, savin
g model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 110s - loss: 4.6804 - acc:
0.0316 - val_loss: 4.7244 - val_acc: 0.0287
Epoch 7/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.6334 - a
cc: 0.0339Epoch 00006: val_loss did not improve
6680/6680 [==============================] - 110s - loss: 4.6331 - acc:
0.0338 - val_loss: 4.7309 - val_acc: 0.0287
Epoch 8/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.5963 - a
cc: 0.0369Epoch 00007: val_loss improved from 4.72437 to 4.70842, savin
g model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 110s - loss: 4.5958 - acc:
0.0370 - val_loss: 4.7084 - val_acc: 0.0311
Epoch 9/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.5652 - a
cc: 0.0380Epoch 00008: val_loss improved from 4.70842 to 4.64360, savin
g model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 110s - loss: 4.5659 - acc:
0.0379 - val_loss: 4.6436 - val_acc: 0.0383
Epoch 10/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.5301 - a
cc: 0.0423Epoch 00009: val_loss improved from 4.64360 to 4.61673, savin
```

```
g model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 110s - loss: 4.5305 - acc:
0.0424 - val_loss: 4.6167 - val_acc: 0.0419
Epoch 11/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.5055 - a
cc: 0.0437Epoch 00010: val_loss did not improve
6680/6680 [==============================] - 110s - loss: 4.5060 - acc:
0.0436 - val_loss: 4.6236 - val_acc: 0.0359
Epoch 12/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.4858 - a
cc: 0.0456Epoch 00011: val_loss did not improve
6680/6680 [==============================] - 110s - loss: 4.4841 - acc:
0.0461 - val_loss: 4.6369 - val_acc: 0.0395
Epoch 13/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.4580 - a
cc: 0.0517Epoch 00012: val_loss improved from 4.61673 to 4.58552, savin
g model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 110s - loss: 4.4579 - acc:
0.0516 - val_loss: 4.5855 - val_acc: 0.0407
Epoch 14/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.4407 - a
cc: 0.0505Epoch 00013: val_loss did not improve
6680/6680 [==============================] - 110s - loss: 4.4393 - acc:
0.0506 - val_loss: 4.6166 - val_acc: 0.0419
Epoch 15/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.4187 - a
cc: 0.0550Epoch 00014: val_loss did not improve
6680/6680 [==============================] - 110s - loss: 4.4188 - acc:
0.0549 - val_loss: 4.6108 - val_acc: 0.0359
Epoch 16/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.3989 - a
cc: 0.0536Epoch 00015: val_loss improved from 4.58552 to 4.55715, savin
g model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 110s - loss: 4.3991 - acc:
0.0536 - val_loss: 4.5571 - val_acc: 0.0407
Epoch 17/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.3700 - a
cc: 0.0583Epoch 00016: val_loss improved from 4.55715 to 4.51675, savin
g model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 110s - loss: 4.3693 - acc:
0.0582 - val_loss: 4.5167 - val_acc: 0.0515
Epoch 18/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.3505 - a
cc: 0.0628Epoch 00017: val_loss did not improve
6680/6680 [==============================] - 110s - loss: 4.3508 - acc:
0.0626 - val_loss: 4.5209 - val_acc: 0.0443
Epoch 19/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.3246 - a
cc: 0.0631Epoch 00018: val_loss did not improve
6680/6680 [==============================] - 110s - loss: 4.3237 - acc:
0.0630 - val_loss: 4.5888 - val_acc: 0.0515
Epoch 20/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.3000 - a
cc: 0.0710Epoch 00019: val_loss improved from 4.51675 to 4.51131, savin
g model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 110s - loss: 4.2998 - acc:
0.0710 - val_loss: 4.5113 - val_acc: 0.0539
```

```
Epoch 21/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.2765 - a
cc: 0.0679Epoch 00020: val_loss improved from 4.51131 to 4.49601, savin
g model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 110s - loss: 4.2758 - acc:
0.0677 - val_loss: 4.4960 - val_acc: 0.0587
Epoch 22/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.2555 - a
cc: 0.0710Epoch 00021: val_loss improved from 4.49601 to 4.48245, savin
g model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 110s - loss: 4.2568 - acc:
0.0708 - val_loss: 4.4825 - val_acc: 0.0503
Epoch 23/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.2351 - a
cc: 0.0718Epoch 00022: val_loss did not improve
6680/6680 [==============================] - 110s - loss: 4.2353 - acc:
0.0717 - val_loss: 4.5045 - val_acc: 0.0539
Epoch 24/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.2221 - a
cc: 0.0752Epoch 00023: val_loss did not improve
6680/6680 [==============================] - 110s - loss: 4.2221 - acc:
0.0751 - val_loss: 4.5114 - val_acc: 0.0551
Epoch 25/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.2182 - a
cc: 0.0757Epoch 00024: val_loss improved from 4.48245 to 4.45246, savin
g model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 110s - loss: 4.2185 - acc:
0.0757 - val_loss: 4.4525 - val_acc: 0.0491
Epoch 26/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.1967 - a
cc: 0.0751Epoch 00025: val_loss did not improve
6680/6680 [==============================] - 110s - loss: 4.1967 - acc:
0.0749 - val_loss: 4.4736 - val_acc: 0.0647
Epoch 27/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.1877 - a
cc: 0.0752Epoch 00026: val_loss did not improve
6680/6680 [==============================] - 110s - loss: 4.1877 - acc:
0.0750 - val_loss: 4.5105 - val_acc: 0.0635
Epoch 28/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.1754 - a
cc: 0.0802Epoch 00027: val_loss improved from 4.45246 to 4.44176, savin
g model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 110s - loss: 4.1758 - acc:
0.0801 - val_loss: 4.4418 - val_acc: 0.0623
Epoch 29/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.1665 - a
cc: 0.0787Epoch 00028: val_loss did not improve
6680/6680 [==============================] - 110s - loss: 4.1658 - acc:
0.0787 - val_loss: 4.5155 - val_acc: 0.0563
Epoch 30/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.1463 - a
cc: 0.0839Epoch 00029: val_loss did not improve
6680/6680 [==============================] - 110s - loss: 4.1450 - acc:
0.0843 - val_loss: 4.4731 - val_acc: 0.0635
Epoch 31/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.1497 - a
cc: 0.0839Epoch 00030: val_loss did not improve
```

```
6680/6680 [==============================] - 110s - loss: 4.1491 - acc:
0.0841 - val_loss: 4.4430 - val_acc: 0.0599
Epoch 32/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.1402 - a
cc: 0.0842Epoch 00031: val_loss improved from 4.44176 to 4.40812, savin
g model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 110s - loss: 4.1423 - acc:
0.0840 - val_loss: 4.4081 - val_acc: 0.0611
Epoch 33/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.1294 - a
cc: 0.0835Epoch 00032: val_loss did not improve
6680/6680 [==============================] - 110s - loss: 4.1296 - acc:
0.0832 - val_loss: 4.4174 - val_acc: 0.0635
Epoch 34/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.1196 - a
cc: 0.0830Epoch 00033: val_loss did not improve
6680/6680 [==============================] - 110s - loss: 4.1196 - acc:
0.0828 - val_loss: 4.4592 - val_acc: 0.0659
Epoch 35/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.1038 - a
cc: 0.0887Epoch 00034: val_loss did not improve
6680/6680 [==============================] - 110s - loss: 4.1029 - acc:
0.0889 - val_loss: 4.4873 - val_acc: 0.0635
Epoch 36/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.0973 - a
cc: 0.0920Epoch 00035: val_loss did not improve
6680/6680 [==============================] - 110s - loss: 4.0976 - acc:
0.0918 - val_loss: 4.4721 - val_acc: 0.0659
Epoch 37/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.0868 - a
cc: 0.0883Epoch 00036: val_loss did not improve
6680/6680 [==============================] - 110s - loss: 4.0885 - acc:
0.0885 - val_loss: 4.4786 - val_acc: 0.0575
Epoch 38/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.0836 - a
cc: 0.0896Epoch 00037: val_loss did not improve
6680/6680 [==============================] - 110s - loss: 4.0833 - acc:
0.0897 - val_loss: 4.4586 - val_acc: 0.0719
Epoch 39/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.0649 - a
cc: 0.0937Epoch 00038: val_loss improved from 4.40812 to 4.37075, savin
g model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 110s - loss: 4.0652 - acc:
0.0934 - val_loss: 4.3707 - val_acc: 0.0707
Epoch 40/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.0600 - a
cc: 0.0952Epoch 00039: val_loss did not improve
6680/6680 [==============================] - 110s - loss: 4.0588 - acc:
0.0951 - val_loss: 4.4293 - val_acc: 0.0683
Epoch 41/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.0479 - a
cc: 0.0967Epoch 00040: val_loss did not improve
6680/6680 [==============================] - 110s - loss: 4.0478 - acc:
0.0967 - val_loss: 4.3952 - val_acc: 0.0719
Epoch 42/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.0403 - a
cc: 0.0970Epoch 00041: val_loss did not improve
```

```
6680/6680 [==============================] - 110s - loss: 4.0403 - acc:
0.0970 - val_loss: 4.3749 - val_acc: 0.0707
Epoch 43/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.0304 - a
cc: 0.0959Epoch 00042: val_loss did not improve
6680/6680 [==============================] - 110s - loss: 4.0294 - acc:
0.0961 - val_loss: 4.3732 - val_acc: 0.0707
Epoch 44/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.0178 - a
cc: 0.0995Epoch 00043: val_loss did not improve
6680/6680 [==============================] - 110s - loss: 4.0154 - acc:
0.0999 - val_loss: 4.3880 - val_acc: 0.0778
Epoch 45/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.0082 - a
cc: 0.1024Epoch 00044: val_loss did not improve
6680/6680 [==============================] - 110s - loss: 4.0070 - acc:
0.1025 - val_loss: 4.4182 - val_acc: 0.0671
Epoch 46/50
6660/6680 [============================>.] - ETA: 0s - loss: 3.9971 - a
cc: 0.0980Epoch 00045: val_loss did not improve
6680/6680 [==============================] - 110s - loss: 3.9983 - acc:
0.0979 - val_loss: 4.3768 - val_acc: 0.0719
Epoch 47/50
6660/6680 [============================>.] - ETA: 0s - loss: 3.9918 - a
cc: 0.1047Epoch 00046: val_loss did not improve
6680/6680 [==============================] - 110s - loss: 3.9922 - acc:
0.1045 - val_loss: 4.3978 - val_acc: 0.0695
Epoch 48/50
6660/6680 [============================>.] - ETA: 0s - loss: 3.9864 - a
cc: 0.1023Epoch 00047: val_loss did not improve
6680/6680 [==============================] - 110s - loss: 3.9854 - acc:
0.1022 - val_loss: 4.4418 - val_acc: 0.0719
Epoch 49/50
6660/6680 [============================>.] - ETA: 0s - loss: 3.9745 - a
cc: 0.1062Epoch 00048: val_loss did not improve
6680/6680 [==============================] - 110s - loss: 3.9743 - acc:
0.1063 - val_loss: 4.3935 - val_acc: 0.0671
Epoch 50/50
6660/6680 [============================>.] - ETA: 0s - loss: 3.9590 - a
cc: 0.1020Epoch 00049: val_loss did not improve
6680/6680 [==============================] - 110s - loss: 3.9584 - acc:
0.1022 - val_loss: 4.4274 - val_acc: 0.0719
```

Out[15]:
```
<keras.callbacks.History at 0x7f9d1b5a44a8>
```

Load the Model with the Best Validation Loss

In [16]:
```
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 [17]:
```
# 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: 7.4163%
```

Step 4: Use a CNN to Classify Dog Breeds

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

Obtain Bottleneck Features

In [18]:
```
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 [19]:
```
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_2 ( (None, 512)               0
_________________________________________________________________
dense_3 (Dense)              (None, 133)               68229
=================================================================
Total params: 68,229
Trainable params: 68,229
Non-trainable params: 0
_________________________________________________________________
```

Compile the Model

In [20]:
```
VGG16_model.compile(loss='categorical_crossentropy', optimizer='rmsprop'
, metrics=['accuracy'])
```

Train the Model

In [21]:
```
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
6360/6680 [===========================>..] - ETA: 0s - loss: 12.8537 -
acc: 0.1083Epoch 00000: val_loss improved from inf to 11.82751, saving
model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 12.8095 - acc:
0.1118 - val_loss: 11.8275 - val_acc: 0.1760
Epoch 2/20
6500/6680 [============================>.] - ETA: 0s - loss: 11.1639 -
acc: 0.2408Epoch 00001: val_loss improved from 11.82751 to 11.17661, sa
ving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 11.1422 - acc:
0.2424 - val_loss: 11.1766 - val_acc: 0.2443
Epoch 3/20
6540/6680 [============================>.] - ETA: 0s - loss: 10.7475 -
acc: 0.2875Epoch 00002: val_loss improved from 11.17661 to 10.92860, sa
ving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 10.7530 - acc:
0.2871 - val_loss: 10.9286 - val_acc: 0.2623
Epoch 4/20
6480/6680 [============================>.] - ETA: 0s - loss: 10.4907 -
acc: 0.3131Epoch 00003: val_loss improved from 10.92860 to 10.67808, sa
ving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 10.4881 - acc:
0.3136 - val_loss: 10.6781 - val_acc: 0.2874
Epoch 5/20
6500/6680 [============================>.] - ETA: 0s - loss: 10.3435 -
acc: 0.3326Epoch 00004: val_loss improved from 10.67808 to 10.62943, sa
ving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 10.3541 - acc:
0.3320 - val_loss: 10.6294 - val_acc: 0.2970
Epoch 6/20
6440/6680 [===========================>..] - ETA: 0s - loss: 10.2394 -
acc: 0.3452Epoch 00005: val_loss improved from 10.62943 to 10.44075, sa
ving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 10.2577 - acc:
0.3434 - val_loss: 10.4408 - val_acc: 0.2994
Epoch 7/20
6500/6680 [============================>.] - ETA: 0s - loss: 10.0520 -
acc: 0.3571Epoch 00006: val_loss improved from 10.44075 to 10.38227, sa
ving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 10.0494 - acc:
0.3575 - val_loss: 10.3823 - val_acc: 0.3162
Epoch 8/20
6560/6680 [============================>.] - ETA: 0s - loss: 10.0152 -
acc: 0.3659Epoch 00007: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 10.0073 - acc:
0.3663 - val_loss: 10.4043 - val_acc: 0.3090
Epoch 9/20
6520/6680 [============================>.] - ETA: 0s - loss: 9.9685 - a
cc: 0.3698Epoch 00008: val_loss improved from 10.38227 to 10.38168, sav
ing model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 9.9666 - acc:
0.3699 - val_loss: 10.3817 - val_acc: 0.3090
Epoch 10/20
6540/6680 [============================>.] - ETA: 0s - loss: 9.7444 - a
cc: 0.3798Epoch 00009: val_loss improved from 10.38168 to 10.18406, sav
```

```
ing model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 9.7307 - acc:
0.3805 - val_loss: 10.1841 - val_acc: 0.3186
Epoch 11/20
6540/6680 [============================>.] - ETA: 0s - loss: 9.6187 - a
cc: 0.3910Epoch 00010: val_loss improved from 10.18406 to 10.07069, sav
ing model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 9.6125 - acc:
0.3913 - val_loss: 10.0707 - val_acc: 0.3257
Epoch 12/20
6600/6680 [============================>.] - ETA: 0s - loss: 9.5457 - a
cc: 0.3945Epoch 00011: val_loss improved from 10.07069 to 9.87900, savi
ng model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 9.5427 - acc:
0.3948 - val_loss: 9.8790 - val_acc: 0.3341
Epoch 13/20
6520/6680 [============================>.] - ETA: 0s - loss: 9.3058 - a
cc: 0.4049Epoch 00012: val_loss improved from 9.87900 to 9.82443, savin
g model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 9.3174 - acc:
0.4045 - val_loss: 9.8244 - val_acc: 0.3317
Epoch 14/20
6480/6680 [============================>.] - ETA: 0s - loss: 9.2405 - a
cc: 0.4122Epoch 00013: val_loss improved from 9.82443 to 9.65819, savin
g model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 9.2303 - acc:
0.4130 - val_loss: 9.6582 - val_acc: 0.3365
Epoch 15/20
6500/6680 [============================>.] - ETA: 0s - loss: 9.1690 - a
cc: 0.4191Epoch 00014: val_loss improved from 9.65819 to 9.64144, savin
g model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 9.1515 - acc:
0.4204 - val_loss: 9.6414 - val_acc: 0.3425
Epoch 16/20
6560/6680 [============================>.] - ETA: 0s - loss: 9.0894 - a
cc: 0.4247Epoch 00015: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 9.0867 - acc:
0.4247 - val_loss: 9.7774 - val_acc: 0.3246
Epoch 17/20
6420/6680 [===========================>..] - ETA: 0s - loss: 8.8121 - a
cc: 0.4330Epoch 00016: val_loss improved from 9.64144 to 9.48288, savin
g model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.8365 - acc:
0.4313 - val_loss: 9.4829 - val_acc: 0.3341
Epoch 18/20
6540/6680 [============================>.] - ETA: 0s - loss: 8.6531 - a
cc: 0.4465Epoch 00017: val_loss improved from 9.48288 to 9.28213, savin
g model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.6645 - acc:
0.4460 - val_loss: 9.2821 - val_acc: 0.3677
Epoch 19/20
6600/6680 [============================>.] - ETA: 0s - loss: 8.6187 - a
cc: 0.4545Epoch 00018: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 8.6194 - acc:
0.4545 - val_loss: 9.3049 - val_acc: 0.3677
Epoch 20/20
6560/6680 [============================>.] - ETA: 0s - loss: 8.5946 - a
```

```
cc: 0.4599Epoch 00019: val_loss improved from 9.28213 to 9.24015, savin
g model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.5948 - acc:
0.4600 - val_loss: 9.2402 - val_acc: 0.3808
```

Out[21]: <keras.callbacks.History at 0x7f9d1aefebe0>

Load the Model with the Best Validation Loss

In [22]:

```
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 [23]:

```
# 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: 38.2775%
```

Predict Dog Breed with the Model

In [24]:

```
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 [25]: ### TODO: Obtain bottleneck features from another pre-trained CNN.
# architecture = 'VGG19'
# architecture = 'Resnet50'
# architecture = 'InceptionV3'
architecture = 'Xception'
bottleneck_features = np.load('bottleneck_features/Dog%sData.npz' % arch
itecture)
train_Xception = bottleneck_features['train']
valid_Xception = bottleneck_features['valid']
test_Xception = bottleneck_features['test']
```

(IMPLEMENTATION) Model Architecture

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

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

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

Answer: I've used the same architecture as class 4 because this is seemed to yeild a accuracy rate of sub 80%. I tried modifying this using another dense layer and adding a dropout layer, but the accuracy wasn't as good.

```
In [26]: ### TODO: Define your architecture.
Xception_model = Sequential()
Xception_model.add(GlobalAveragePooling2D(input_shape=train_Xception.sha
pe[1:]))
# Xception_model.add(Dense(500, activation='relu'))
# Xception_model.add(Dropout(0.2))
Xception_model.add(Dense(133, activation='softmax'))

Xception_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
global_average_pooling2d_3 ( (None, 2048)              0
_________________________________________________________________
dense_4 (Dense)              (None, 133)               272517
=================================================================
Total params: 272,517
Trainable params: 272,517
Non-trainable params: 0
_________________________________________________________________
```

(IMPLEMENTATION) Compile the Model

In [27]:
```
### TODO: Compile the model.
Xception_model.compile(loss='categorical_crossentropy', optimizer='rmspr
op', 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 [28]:
```
### TODO: Train the model.
checkpointer = ModelCheckpoint(filepath='saved_models/weights.best.%s.hd
f5' % architecture,
                               verbose=1, save_best_only=True)

Xception_model.fit(train_Xception, train_targets,
          validation_data=(valid_Xception, valid_targets),
          epochs=20, batch_size=20, callbacks=[checkpointer], verbose=1)
```

```
Train on 6680 samples, validate on 835 samples
Epoch 1/20
6620/6680 [============================>.] - ETA: 0s - loss: 1.0600 - a
cc: 0.7338Epoch 00000: val_loss improved from inf to 0.52311, saving mo
del to saved_models/weights.best.Xception.hdf5
6680/6680 [==============================] - 4s - loss: 1.0556 - acc:
0.7344 - val_loss: 0.5231 - val_acc: 0.8311
Epoch 2/20
6620/6680 [============================>.] - ETA: 0s - loss: 0.3995 - a
cc: 0.8725Epoch 00001: val_loss improved from 0.52311 to 0.48425, savin
g model to saved_models/weights.best.Xception.hdf5
6680/6680 [==============================] - 4s - loss: 0.3990 - acc:
0.8725 - val_loss: 0.4843 - val_acc: 0.8503
Epoch 3/20
6620/6680 [============================>.] - ETA: 0s - loss: 0.3252 - a
cc: 0.8991Epoch 00002: val_loss did not improve
6680/6680 [==============================] - 4s - loss: 0.3255 - acc:
0.8990 - val_loss: 0.5039 - val_acc: 0.8527
Epoch 4/20
6620/6680 [============================>.] - ETA: 0s - loss: 0.2814 - a
cc: 0.9127Epoch 00003: val_loss did not improve
6680/6680 [==============================] - 4s - loss: 0.2800 - acc:
0.9130 - val_loss: 0.4907 - val_acc: 0.8491
Epoch 5/20
6620/6680 [============================>.] - ETA: 0s - loss: 0.2455 - a
cc: 0.9254Epoch 00004: val_loss did not improve
6680/6680 [==============================] - 4s - loss: 0.2457 - acc:
0.9253 - val_loss: 0.5147 - val_acc: 0.8551
Epoch 6/20
6620/6680 [============================>.] - ETA: 0s - loss: 0.2210 - a
cc: 0.9341Epoch 00005: val_loss did not improve
6680/6680 [==============================] - 4s - loss: 0.2219 - acc:
0.9338 - val_loss: 0.5263 - val_acc: 0.8575
Epoch 7/20
6620/6680 [============================>.] - ETA: 0s - loss: 0.2006 - a
cc: 0.9384Epoch 00006: val_loss did not improve
6680/6680 [==============================] - 4s - loss: 0.2009 - acc:
0.9383 - val_loss: 0.5451 - val_acc: 0.8587
Epoch 8/20
6620/6680 [============================>.] - ETA: 0s - loss: 0.1800 - a
cc: 0.9447Epoch 00007: val_loss did not improve
6680/6680 [==============================] - 4s - loss: 0.1790 - acc:
0.9451 - val_loss: 0.5634 - val_acc: 0.8563
Epoch 9/20
6620/6680 [============================>.] - ETA: 0s - loss: 0.1632 - a
cc: 0.9503Epoch 00008: val_loss did not improve
6680/6680 [==============================] - 4s - loss: 0.1626 - acc:
0.9503 - val_loss: 0.5553 - val_acc: 0.8539
Epoch 10/20
6620/6680 [============================>.] - ETA: 0s - loss: 0.1474 - a
cc: 0.9542Epoch 00009: val_loss did not improve
6680/6680 [==============================] - 4s - loss: 0.1505 - acc:
0.9540 - val_loss: 0.5955 - val_acc: 0.8443
Epoch 11/20
6620/6680 [============================>.] - ETA: 0s - loss: 0.1347 - a
cc: 0.9609Epoch 00010: val_loss did not improve
6680/6680 [==============================] - 4s - loss: 0.1351 - acc:
```

```
0.9609 - val_loss: 0.5960 - val_acc: 0.8491
Epoch 12/20
6620/6680 [============================>.] - ETA: 0s - loss: 0.1269 - a
cc: 0.9619Epoch 00011: val_loss did not improve
6680/6680 [==============================] - 4s - loss: 0.1284 - acc:
0.9618 - val_loss: 0.6113 - val_acc: 0.8599
Epoch 13/20
6620/6680 [============================>.] - ETA: 0s - loss: 0.1150 - a
cc: 0.9650Epoch 00012: val_loss did not improve
6680/6680 [==============================] - 4s - loss: 0.1141 - acc:
0.9653 - val_loss: 0.6289 - val_acc: 0.8587
Epoch 14/20
6620/6680 [============================>.] - ETA: 0s - loss: 0.1059 - a
cc: 0.9695Epoch 00013: val_loss did not improve
6680/6680 [==============================] - 4s - loss: 0.1062 - acc:
0.9693 - val_loss: 0.6195 - val_acc: 0.8587
Epoch 15/20
6620/6680 [============================>.] - ETA: 0s - loss: 0.1000 - a
cc: 0.9718Epoch 00014: val_loss did not improve
6680/6680 [==============================] - 4s - loss: 0.1014 - acc:
0.9719 - val_loss: 0.6826 - val_acc: 0.8527
Epoch 16/20
6620/6680 [============================>.] - ETA: 0s - loss: 0.0947 - a
cc: 0.9730Epoch 00015: val_loss did not improve
6680/6680 [==============================] - 4s - loss: 0.0950 - acc:
0.9726 - val_loss: 0.6861 - val_acc: 0.8551
Epoch 17/20
6620/6680 [============================>.] - ETA: 0s - loss: 0.0868 - a
cc: 0.9728Epoch 00016: val_loss did not improve
6680/6680 [==============================] - 4s - loss: 0.0863 - acc:
0.9729 - val_loss: 0.6512 - val_acc: 0.8635
Epoch 18/20
6620/6680 [============================>.] - ETA: 0s - loss: 0.0803 - a
cc: 0.9757Epoch 00017: val_loss did not improve
6680/6680 [==============================] - 4s - loss: 0.0810 - acc:
0.9753 - val_loss: 0.6640 - val_acc: 0.8443
Epoch 19/20
6620/6680 [============================>.] - ETA: 0s - loss: 0.0781 - a
cc: 0.9770Epoch 00018: val_loss did not improve
6680/6680 [==============================] - 4s - loss: 0.0779 - acc:
0.9768 - val_loss: 0.7045 - val_acc: 0.8551
Epoch 20/20
6600/6680 [============================>.] - ETA: 0s - loss: 0.0723 - a
cc: 0.9782Epoch 00019: val_loss did not improve
6680/6680 [==============================] - 4s - loss: 0.0719 - acc:
0.9783 - val_loss: 0.7087 - val_acc: 0.8611
```

```
Out[28]: <keras.callbacks.History at 0x7f9d1acddf28>
```

(IMPLEMENTATION) Load the Model with the Best Validation Loss

In [29]:
```
### TODO: Load the model weights with the best validation loss.
Xception_model.load_weights('saved_models/weights.best.%s.hdf5' % archit
ecture)
```

(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 [30]:
```
### TODO: Calculate classification accuracy on the test dataset.
# get index of predicted dog breed for each image in test set
Xception_predictions = [np.argmax(Xception_model.predict(np.expand_dims(
feature, axis=0))) for feature in test_Xception]

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

```
Test accuracy: 84.2105%
```

(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 [31]:
```
### TODO: Write a function that takes a path to an image as input
### and returns the dog breed that is predicted by the model.
def Xception_predict_breed(img_path):
    # extract bottleneck features
    bottleneck_feature = extract_Xception(path_to_tensor(img_path))
    # obtain predicted vector
    predicted_vector = Xception_model.predict(bottleneck_feature)
    # return dog breed that is predicted by the model
    return dog_names[np.argmax(predicted_vector)]
```

Step 6: Write your Algorithm

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

- if a **dog** is detected in the image, return the predicted breed.
- if a **human** is detected in the image, return the resembling dog breed.
- if **neither** is detected in the image, provide output that indicates an error.

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

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

Sample Human Output

(IMPLEMENTATION) Write your Algorithm

In [32]:

```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.

def output_img(img_path):
    img = cv2.imread(img_path)
    # convert BGR image to RGB for plotting
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    plt.imshow(img)
    plt.show()


def output(img_path, img_type):
    print("hello,", img_type)

    output_img(img_path)

    # Read image for displaying
    print("you look like a ...")
    breed = Xception_predict_breed(img_path)
    print(breed)
    return breed

def dog_breed_detector(img_path):
    if dog_detector(img_path):
        return output(img_path, "doggy")
    elif face_detector(img_path):
        return output(img_path, "human")
    error_message = "ERROR: Did not detect human or doggy"
    print(error_message)
    output_img(img_path)
    return error_message
```

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 didn't do well in my opinion.

For first two images, the model predicted that my daughter is an icelandic sheep dog and a daschund. I think the 'hello kitty' faces on her dress in the 1st image and the 'crown' thing she's wearing in her head (I don't know what that is) in the 2nd picture could've confused the model. One way I could possibly fix this is by filtering her head from the image then submit that to the model. Also, I when I told my daughter what breeds she was, she didn't seem to appreciate it. She reminded me that she's not a dog. :D

Another problem is that for the 3rd images, the model did not recognize my son as a human or dog. I think this might be cause by him being sideways or his eyes closed. I could possibly fix this by adding training data of people with their eyes shut and weird angles.

The next problem is not the model's problem, but it was mine. I thought that dog was a Japanese Spitz. After spending some time comparing pictures of Japanese Spitz and American Eskimo Dogs, I'm convinced that that puppy is an American Eskimo Dog!

The 4th picture is a cartoon picture of a dog. I was hoping the model would detect it, but I was asking for too much I guess. :(

For the 5th (cat) and 6th (wolf in suit) images, the model did a great job detecting that those aren't dogs/humans.

Improvements

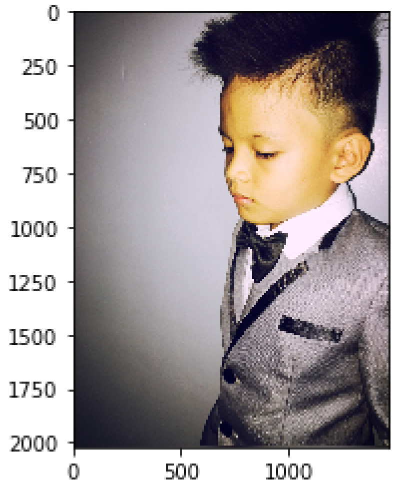
1. Try out tranfer learning from different architectures besides xception
2. Try adding more dense layers
3. Try dropout layers
4. Try BatchNormalization
5. Improve human detection using combination of different architecture
6. Improve human detection by adding human images with closed eyes and sideways angle

In [33]:
```
## 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.

image_names = ["daughter_sad.png",
               "daughter_silly.png",
               "son_dont_like_suit.png",
               "japanese_spitz.png",
               "japanese_spitz_cartoon.png",
               "smelly_cat.png",
               "wolf_in_a_suit.png"
              ]

my_images = np.array(['my_images/' + img for img in image_names])After

print(my_images.__class__)
for img_path in my_images:
    print("-" * 100)
    print("FILE:", img_path)
    dog_breed_detector(img_path)
```

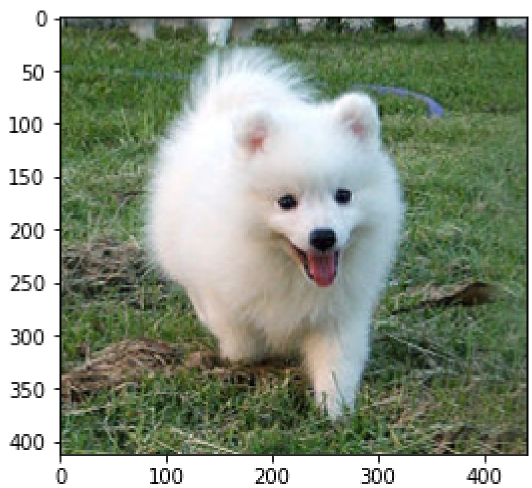
```
<class 'numpy.ndarray'>
-----------------------------------------------------------------------
-----------------------------
FILE: my_images/daughter_sad.png
hello, human
```

```
you look like a ...
Icelandic_sheepdog
-----------------------------------------------------------------------
-----------------------------
FILE: my_images/daughter_silly.png
hello, human
```

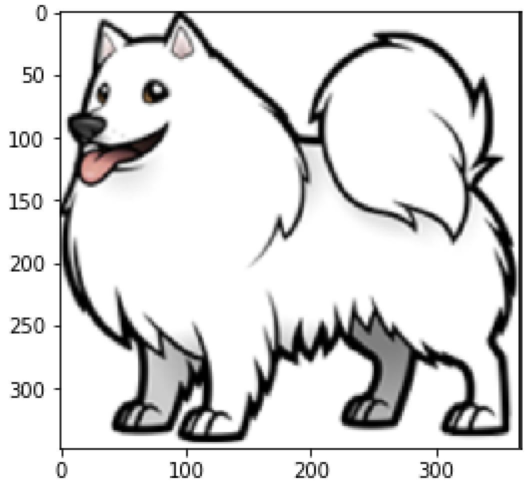
```
you look like a ...
Dachshund
-----------------------------------------------------------------------
-----------------------------
FILE: my_images/son_dont_like_suit.png
ERROR: Did not detect human or doggy
```



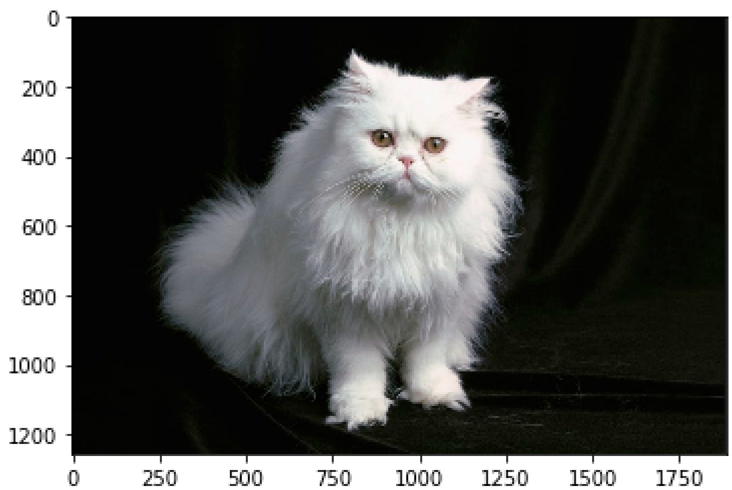

```
----------------------------------------------------------------------------------------------------
FILE: my_images/japanese_spitz.png
hello, doggy
```




```
you look like a ...
American_eskimo_dog
----------------------------------------------------------------------------------------------------
FILE: my_images/japanese_spitz_cartoon.png
ERROR: Did not detect human or doggy
```




```
-----------------------------------------------------------------------------------------------------------
FILE: my_images/smelly_cat.png
ERROR: Did not detect human or doggy
```




```
-----------------------------------------------------------------------------------------------------------
FILE: my_images/wolf_in_a_suit.png
ERROR: Did not detect human or doggy
```