Data Scientist Nanodegree

Convolutional Neural Networks

Project: Write an Algorithm for a Dog Identification App

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

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

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

In this notebook, some template code has already been provided for you, and you will need to implement additional functionality to successfully complete this project. You will not need to modify the included code beyond what is requested. Sections that begin with **'(IMPLEMENTATION)'** in the header indicate that the following block of code will require additional functionality which you must provide. Instructions will be provided for each section, and the specifics of the implementation are marked in the code block with a 'TODO' statement. Please be sure to read the instructions carefully!

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('../../../data/dog_images/trai
n')
valid_files, valid_targets = load_dataset('../../../data/dog_images/vali
d')
test_files, test_targets = load_dataset('../../../data/dog_images/test')

# load list of dog names
dog_names = [item[20:-1] for item in sorted(glob("../../../data/dog_imag
es/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.
```

In [2]:
```
# Get dog breed labels
dog_names = [item[20:-1] for item in sorted(glob("../../../data/dog_imag
es/train/*/"))]
```

In [3]:
```
# Show dog breeds
dog_names
```

```
Out[3]: ['ages/train/001.Affenpinscher',
 'ages/train/002.Afghan_hound',
 'ages/train/003.Airedale_terrier',
 'ages/train/004.Akita',
 'ages/train/005.Alaskan_malamute',
 'ages/train/006.American_eskimo_dog',
 'ages/train/007.American_foxhound',
 'ages/train/008.American_staffordshire_terrier',
 'ages/train/009.American_water_spaniel',
 'ages/train/010.Anatolian_shepherd_dog',
 'ages/train/011.Australian_cattle_dog',
 'ages/train/012.Australian_shepherd',
 'ages/train/013.Australian_terrier',
 'ages/train/014.Basenji',
 'ages/train/015.Basset_hound',
 'ages/train/016.Beagle',
 'ages/train/017.Bearded_collie',
 'ages/train/018.Beauceron',
 'ages/train/019.Bedlington_terrier',
 'ages/train/020.Belgian_malinois',
 'ages/train/021.Belgian_sheepdog',
 'ages/train/022.Belgian_tervuren',
 'ages/train/023.Bernese_mountain_dog',
 'ages/train/024.Bichon_frise',
 'ages/train/025.Black_and_tan_coonhound',
 'ages/train/026.Black_russian_terrier',
 'ages/train/027.Bloodhound',
 'ages/train/028.Bluetick_coonhound',
 'ages/train/029.Border_collie',
 'ages/train/030.Border_terrier',
 'ages/train/031.Borzoi',
 'ages/train/032.Boston_terrier',
 'ages/train/033.Bouvier_des_flandres',
 'ages/train/034.Boxer',
 'ages/train/035.Boykin_spaniel',
 'ages/train/036.Briard',
 'ages/train/037.Brittany',
 'ages/train/038.Brussels_griffon',
 'ages/train/039.Bull_terrier',
 'ages/train/040.Bulldog',
 'ages/train/041.Bullmastiff',
 'ages/train/042.Cairn_terrier',
 'ages/train/043.Canaan_dog',
 'ages/train/044.Cane_corso',
 'ages/train/045.Cardigan_welsh_corgi',
 'ages/train/046.Cavalier_king_charles_spaniel',
 'ages/train/047.Chesapeake_bay_retriever',
 'ages/train/048.Chihuahua',
 'ages/train/049.Chinese_crested',
 'ages/train/050.Chinese_shar-pei',
 'ages/train/051.Chow_chow',
 'ages/train/052.Clumber_spaniel',
 'ages/train/053.Cocker_spaniel',
 'ages/train/054.Collie',
 'ages/train/055.Curly-coated_retriever',
 'ages/train/056.Dachshund',
 'ages/train/057.Dalmatian',
```

```
 'ages/train/058.Dandie_dinmont_terrier',
 'ages/train/059.Doberman_pinscher',
 'ages/train/060.Dogue_de_bordeaux',
 'ages/train/061.English_cocker_spaniel',
 'ages/train/062.English_setter',
 'ages/train/063.English_springer_spaniel',
 'ages/train/064.English_toy_spaniel',
 'ages/train/065.Entlebucher_mountain_dog',
 'ages/train/066.Field_spaniel',
 'ages/train/067.Finnish_spitz',
 'ages/train/068.Flat-coated_retriever',
 'ages/train/069.French_bulldog',
 'ages/train/070.German_pinscher',
 'ages/train/071.German_shepherd_dog',
 'ages/train/072.German_shorthaired_pointer',
 'ages/train/073.German_wirehaired_pointer',
 'ages/train/074.Giant_schnauzer',
 'ages/train/075.Glen_of_imaal_terrier',
 'ages/train/076.Golden_retriever',
 'ages/train/077.Gordon_setter',
 'ages/train/078.Great_dane',
 'ages/train/079.Great_pyrenees',
 'ages/train/080.Greater_swiss_mountain_dog',
 'ages/train/081.Greyhound',
 'ages/train/082.Havanese',
 'ages/train/083.Ibizan_hound',
 'ages/train/084.Icelandic_sheepdog',
 'ages/train/085.Irish_red_and_white_setter',
 'ages/train/086.Irish_setter',
 'ages/train/087.Irish_terrier',
 'ages/train/088.Irish_water_spaniel',
 'ages/train/089.Irish_wolfhound',
 'ages/train/090.Italian_greyhound',
 'ages/train/091.Japanese_chin',
 'ages/train/092.Keeshond',
 'ages/train/093.Kerry_blue_terrier',
 'ages/train/094.Komondor',
 'ages/train/095.Kuvasz',
 'ages/train/096.Labrador_retriever',
 'ages/train/097.Lakeland_terrier',
 'ages/train/098.Leonberger',
 'ages/train/099.Lhasa_apso',
 'ages/train/100.Lowchen',
 'ages/train/101.Maltese',
 'ages/train/102.Manchester_terrier',
 'ages/train/103.Mastiff',
 'ages/train/104.Miniature_schnauzer',
 'ages/train/105.Neapolitan_mastiff',
 'ages/train/106.Newfoundland',
 'ages/train/107.Norfolk_terrier',
 'ages/train/108.Norwegian_buhund',
 'ages/train/109.Norwegian_elkhound',
 'ages/train/110.Norwegian_lundehund',
 'ages/train/111.Norwich_terrier',
 'ages/train/112.Nova_scotia_duck_tolling_retriever',
 'ages/train/113.Old_english_sheepdog',
 'ages/train/114.Otterhound',
```

```
 'ages/train/115.Papillon',
 'ages/train/116.Parson_russell_terrier',
 'ages/train/117.Pekingese',
 'ages/train/118.Pembroke_welsh_corgi',
 'ages/train/119.Petit_basset_griffon_vendeen',
 'ages/train/120.Pharaoh_hound',
 'ages/train/121.Plott',
 'ages/train/122.Pointer',
 'ages/train/123.Pomeranian',
 'ages/train/124.Poodle',
 'ages/train/125.Portuguese_water_dog',
 'ages/train/126.Saint_bernard',
 'ages/train/127.Silky_terrier',
 'ages/train/128.Smooth_fox_terrier',
 'ages/train/129.Tibetan_mastiff',
 'ages/train/130.Welsh_springer_spaniel',
 'ages/train/131.Wirehaired_pointing_griffon',
 'ages/train/132.Xoloitzcuintli',
 'ages/train/133.Yorkshire_terrier']
```

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 [4]:
```
import random
random.seed(8675309)

# load filenames in shuffled human dataset
human_files = np.array(glob("../../../data/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

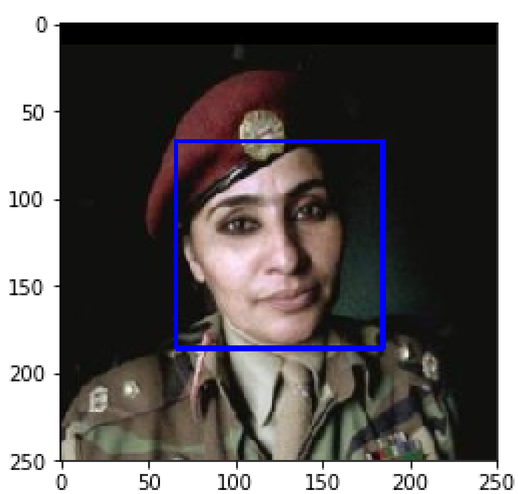
We use OpenCV's implementation of [Haar feature-based cascade classifiers (http://docs.opencv.org/trunk/d7/d8b/tutorial_py_face_detection.html)](http://docs.opencv.org/trunk/d7/d8b/tutorial_py_face_detection.html) to detect human faces in images. OpenCV provides many pre-trained face detectors, stored as XML files on [github (https://github.com/opencv/opencv/tree/master/data/haarcascades)](https://github.com/opencv/opencv/tree/master/data/haarcascades). We have downloaded one of these detectors and stored it in the `haarcascades` directory.

In the next code cell, we demonstrate how to use this detector to find human faces in a sample image.

In [5]:

```python
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 [6]:
```
# 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 faces were detected in 100% of the first 100 images in `human_files`

Human faces were detected in 11% of the first 100 images in `dog_files`

In [7]:
```
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_detect = [face_detector(i) for i in human_files_short]
dog_detect = [face_detector(i) for i in dog_files_short]
print(human_detect.count(True)/len(human_detect), dog_detect.count(True)
/len(dog_detect))
```

```
1.0 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:

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

In [8]:
```
## (Optional) TODO: Report the performance of another
## face detection algorithm on the LFW dataset
### Feel free to use as many code cells as needed.
```

Step 2: Detect Dogs

In this section, we use a pre-trained [ResNet-50 (http://ethereon.github.io/netscope/#/gist/db945b393d40bfa26006)](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 [9]:
```
# Import ResNet50
from keras.applications.resnet50 import ResNet50

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

Pre-process the Data

When using TensorFlow as backend, Keras CNNs require a 4D array (which we'll also refer to as a 4D tensor) as input, with shape

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

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

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

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

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

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

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

In [10]:
```
# Import image processing and progress bar
from keras.preprocessing import image as kp_image
from tqdm import tqdm



def path_to_tensor(img_path):
    """
    Takes an image path input and returns the image as a 4d tensor



    Parameters:
        img_path(str): path/to/image



    Returns:
        4D tensor of image
    """


    # loads RGB image as PIL.Image.Image type
    img = kp_image.load_img(img_path, target_size=(224, 224))


    # convert PIL.Image.Image type to 3D tensor with shape (224, 224, 3)
    x = kp_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):
    """
    Takes img_paths and returns 4d tensors of all images stacked



    Parameters:
        img_paths(str): paths/to/images



    Returns:
        stack of 4d tensors
    """
```

```
    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 [11]:
```
# Imports
from keras.applications.resnet50 import preprocess_input, decode_predict
ions



def ResNet50_predict_labels(img_path):
    """
    Takes img_path and returns predicted label for the image



    Parameters:
        img_path(str): path/to/image



    Returns:
        ResNet50 model prediction
    """


    # returns prediction vector for image located at img_path
    img = preprocess_input(path_to_tensor(img_path))


    # Return prediction
    return np.argmax(ResNet50_model.predict(img))
```

Write a Dog Detector

While looking at the [dictionary (https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a)](https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a), you will notice that the categories corresponding to dogs appear in an uninterrupted sequence and correspond to dictionary keys 151-268, inclusive, to include all categories from `'Chihuahua'` to `'Mexican hairless'`. Thus, in order to check to see if an image is predicted to contain a dog by the pre-trained ResNet-50 model, we need only check if the `ResNet50_predict_labels` function above returns a value between 151 and 268 (inclusive).

We use these ideas to complete the `dog_detector` function below, which returns `True` if a dog is detected in an image (and `False` if not).

```
In [12]: def dog_detector(img_path):
             """
             Takes img_path and returns True/False based on if a dog is detected
          in the img



             Parameters:
                 img_path(str): path/to/image



             Returns:
                 ResNet50 model prediction
             """


             # Predict dog breed from image
             prediction = ResNet50_predict_labels(img_path)


             # Return True or False based on confidence of prediction
             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:

Dogs were detected in 0% of the images in `human_files_short`

Dogs were detected in 11% of the images in `dog_files_short`

In [13]:

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


# Apply dog_detector to each human image file
human_detect_2 = [dog_detector(i) for i in human_files_short]


# Apply dog_detector to each dog image file
dog_detect_2 = [dog_detector(i) for i in dog_files_short]


# Return percentage of dogs predicted from human files and dogs predicte
d from dog files
print(human_detect_2.count(True)/len(human_detect_2), dog_detect_2.count
(True)/len(dog_detect_2))
```

```
0.0 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

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 [14]:
```python
# Imports
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True


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

```
100%|██████████| 6680/6680 [01:09<00:00, 95.66it/s]
100%|██████████| 835/835 [00:07<00:00, 107.30it/s]
100%|██████████| 836/836 [00:07<00:00, 108.45it/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:

To get to my final CNN architecture, I gradually increased the depth of the data (to detect as many features and patterns as possible) while gradually decreasing the spatial dimensions of the data. This allowed for the content of the image be encoded in a representation almost absent of any spatial information left to extract. This data was then flattened and passed into a fully connected dense layer that calculates probabilities for each dog breed for the image.

More specifically, I used 4 convolutional layers, each increasing the depth of the data two-fold (8 filters to 64 filters), as well as 4 max-pooling layers, each decreasing height and width by a factor of two. For each convolitional layer, I used "same" padding (even though the convolutional layers output data with an even height and width, I wanted to preserve their shape). Relu activations functions were used in all of the convolutional layers. After the last pooling layer, I flattened the data and fed it into a dense layer with the same shape as the number of dog breed labels (133)

In [15]:
```python
# Look at train tensor shape
train_tensors.shape
```

Out[15]:
```
(6680, 224, 224, 3)
```

In [16]:

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


# Initiialize model
model = Sequential()


# Add convolutional layer with 8 filters
model.add(Conv2D(filters=8, kernel_size=2, strides=1, padding='same', ac
tivation='relu', input_shape=(224, 224, 3)))


# Further pool data
model.add(MaxPooling2D((2, 2)))


# Add convolutional layer with 16 filters
model.add(Conv2D(filters=16, kernel_size=2, strides=1, padding='same', a
ctivation='relu'))


# Further pool data
model.add(MaxPooling2D((2, 2)))


# Add convolutional layer with 32 filters
model.add(Conv2D(filters=32, kernel_size=2, strides=1, padding='same', a
ctivation='relu'))


# Further pool data
model.add(MaxPooling2D((2, 2)))


# Add convolutional layer with 64 filters
model.add(Conv2D(filters=64, kernel_size=2, strides=1, padding='same', a
ctivation='relu'))


# Further pool data
model.add(MaxPooling2D((2, 2)))


# Add convolutional layer with 64 filters
model.add(Conv2D(filters=64, kernel_size=2, strides=1, padding='same', a
ctivation='relu'))


# Further pool data
model.add(MaxPooling2D((2, 2)))


# Flatten into 1-D vector
```

```
model.add(Flatten())


# Add dense layer with shape equal to number of labels with softmax acti
vation
model.add(Dense(133,activation = 'softmax'))


# Show model summary
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_1 (Conv2D)            (None, 224, 224, 8)       104       
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 112, 112, 8)       0         
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 112, 112, 16)      528       
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 56, 56, 16)        0         
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 56, 56, 32)        2080      
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 28, 28, 32)        0         
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 28, 28, 64)        8256      
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 14, 14, 64)        0         
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 14, 14, 64)        16448     
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 7, 7, 64)          0         
_________________________________________________________________
flatten_2 (Flatten)          (None, 3136)              0         
_________________________________________________________________
dense_1 (Dense)              (None, 133)               417221    
=================================================================
Total params: 444,637
Trainable params: 444,637
Non-trainable params: 0
_________________________________________________________________
```

Compile the Model

In [17]:

```
# Compile model
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 [18]:
```
# Importmodel checkpoint module
from keras.callbacks import ModelCheckpoint


# Number of epochs
epochs = 5

### Do NOT modify the code below this line.

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

model.fit(train_tensors, train_targets,
          validation_data=(valid_tensors, valid_targets),
          epochs=epochs, batch_size=20, callbacks=[checkpointer], verbos
e=1)
```

```
Train on 6680 samples, validate on 835 samples
Epoch 1/5
6640/6680 [============================>.] - ETA: 0s - loss: 4.8392 - a
cc: 0.0163Epoch 00001: val_loss improved from inf to 4.69080, saving mo
del to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 18s 3ms/step - loss: 4.839
1 - acc: 0.0162 - val_loss: 4.6908 - val_acc: 0.0311
Epoch 2/5
6640/6680 [============================>.] - ETA: 0s - loss: 4.4853 - a
cc: 0.0505Epoch 00002: val_loss improved from 4.69080 to 4.47387, savin
g model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 17s 3ms/step - loss: 4.484
0 - acc: 0.0509 - val_loss: 4.4739 - val_acc: 0.0551
Epoch 3/5
6640/6680 [============================>.] - ETA: 0s - loss: 4.0452 - a
cc: 0.1101Epoch 00003: val_loss improved from 4.47387 to 4.42666, savin
g model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 17s 3ms/step - loss: 4.045
4 - acc: 0.1105 - val_loss: 4.4267 - val_acc: 0.0695
Epoch 4/5
6640/6680 [============================>.] - ETA: 0s - loss: 3.6031 - a
cc: 0.1795Epoch 00004: val_loss did not improve
6680/6680 [==============================] - 17s 3ms/step - loss: 3.600
8 - acc: 0.1798 - val_loss: 4.5382 - val_acc: 0.0743
Epoch 5/5
6660/6680 [============================>.] - ETA: 0s - loss: 3.1420 - a
cc: 0.2644Epoch 00005: val_loss did not improve
6680/6680 [==============================] - 17s 3ms/step - loss: 3.141
5 - acc: 0.2644 - val_loss: 4.5874 - val_acc: 0.0934
```

Out[18]: <keras.callbacks.History at 0x7f3cedf90400>

Load the Model with the Best Validation Loss

In [19]:
```
# Load model weights with best loss
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 [20]:
```
# get index of predicted dog breed for each image in test set
dog_breed_predictions = [np.argmax(model.predict(np.expand_dims(tensor, axis=0))) for tensor in test_tensors]

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

```
Test accuracy: 6.4593%
```

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 [21]:
```
# Load bottleneck feature data
bottleneck_features = np.load('bottleneck_features/DogVGG16Data.npz')
train_VGG16 = bottleneck_features['train']
valid_VGG16 = bottleneck_features['valid']
test_VGG16 = bottleneck_features['test']
```

In [22]:
```
# Look at training tensor shape for each image
train_VGG16.shape[1:]
```

Out[22]:
```
(7, 7, 512)
```

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 [23]: # Initialize sequential cnn
VGG16_model = Sequential()


# Add global pooling to make 1-D tensor
VGG16_model.add(GlobalAveragePooling2D(input_shape=train_VGG16.shape[1
:]))


# Dense layer to number of labels
VGG16_model.add(Dense(133, activation='softmax'))


# Look at model summary
VGG16_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
global_average_pooling2d_1 ( (None, 512)               0         
_________________________________________________________________
dense_2 (Dense)              (None, 133)               68229     
=================================================================
Total params: 68,229
Trainable params: 68,229
Non-trainable params: 0
_________________________________________________________________
```

Compile the Model

```
In [24]: # Compile model
VGG16_model.compile(loss='categorical_crossentropy', optimizer='rmsprop'
, metrics=['accuracy'])
```

Train the Model

In [25]:

```
# Set up model checkpointer
checkpointer = ModelCheckpoint(filepath='saved_models/weights.best.VGG1
6.hdf5',
                               verbose=1, save_best_only=True)


# Fit model
VGG16_model.fit(train_VGG16, train_targets,
          validation_data=(valid_VGG16, valid_targets),
          epochs=5, batch_size=20, callbacks=[checkpointer], verbose=1)
```

```
Train on 6680 samples, validate on 835 samples
Epoch 1/5
6500/6680 [============================>.] - ETA: 0s - loss: 13.0484 -
acc: 0.1111Epoch 00001: val_loss improved from inf to 11.55736, saving
model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 290us/step - loss: 13.0
196 - acc: 0.1126 - val_loss: 11.5574 - val_acc: 0.1868
Epoch 2/5
6660/6680 [============================>.] - ETA: 0s - loss: 11.1524 -
acc: 0.2482Epoch 00002: val_loss improved from 11.55736 to 11.18449, sa
ving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 250us/step - loss: 11.1
507 - acc: 0.2484 - val_loss: 11.1845 - val_acc: 0.2395
Epoch 3/5
6440/6680 [===========================>..] - ETA: 0s - loss: 10.8835 -
acc: 0.2935Epoch 00003: val_loss improved from 11.18449 to 11.10075, sa
ving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 249us/step - loss: 10.8
674 - acc: 0.2946 - val_loss: 11.1008 - val_acc: 0.2563
Epoch 4/5
6540/6680 [============================>.] - ETA: 0s - loss: 10.7293 -
acc: 0.3064Epoch 00004: val_loss improved from 11.10075 to 10.84065, sa
ving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 251us/step - loss: 10.7
049 - acc: 0.3081 - val_loss: 10.8407 - val_acc: 0.2814
Epoch 5/5
6520/6680 [============================>.] - ETA: 0s - loss: 10.4601 -
acc: 0.3278Epoch 00005: val_loss improved from 10.84065 to 10.58841, sa
ving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 251us/step - loss: 10.4
485 - acc: 0.3284 - val_loss: 10.5884 - val_acc: 0.2946
```

Out[25]:

```
<keras.callbacks.History at 0x7f3ce5a6e978>
```

Load the Model with the Best Validation Loss

In [26]:

```
# Load our best weights
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 [27]:
```
# 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: 29.4258%
```

Predict Dog Breed with the Model

In [28]:
```
# Load bottleneck_features functions
from extract_bottleneck_features import *


# Function to predict dog breed using transfer learning
def VGG16_predict_breed(img_path):
    """
    Loads image from path.

    Parameters:
        img_path (str): /path/to/image


    Returns:
        Predicted dog breed
    """


    # 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 [29]:

```
# Load bottleneck feature data
bottleneck_features = np.load('bottleneck_features/DogInceptionV3Data.np
z')
train_inception = bottleneck_features['train']
valid_inception= bottleneck_features['valid']
test_inception = bottleneck_features['test']
```

In [30]:
```
# Look at shape of train data images
train_inception.shape[1:]
```

Out[30]: (5, 5, 2048)

(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: Since the InceptionV3 CNN bottlenecked training data has dimensions (5,5,2048), I figured that most of the spatial information present in the images had already been extracted (the 2-D image data had already been reduced from (244,244) to (5,5)), I decided to pool the data down to a 1-D vector. I then used a 1024-sized dense layer with a relu activation function to halve the depth of the data. As a preemptive measure, I introduced a dropout layer just in case the additional transfer layers were causing overfitting. I then added a dense layer to reduce the depth of the data to the number of possibledog-breed labels and used a softmax activation function to retrieve the probabilistic values for each label in an image.

The plots of the accuracy and loss for the train and validation sets, show that the validation accuracy remains higher than that of the train accuracy and that the validation loss remains less than the training loss, which is an indicator that our model is not overfitting (or that the validation set contains much simpler, yet similar images to our training set). Overall, the test accuracy of the model cam out to ~82.7%, which is very good (Inceptionv3 transfer layers were originally trained to have ~80% accuracy on imagenet).

In [31]:

```
# Initialize CNN model
custom_inception_model = Sequential()


# Add pooling
custom_inception_model.add(GlobalAveragePooling2D( input_shape=train_inc
eption.shape[1:]))


# Add dense layer equsl to number of labels
custom_inception_model.add(Dense(1024, activation = 'relu'))


# Add dropout
custom_inception_model.add(Dropout(0.75))


# Add dense layer equsl to number of labels
custom_inception_model.add(Dense(133, activation = 'softmax'))


# Show model summary
custom_inception_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
global_average_pooling2d_2 ( (None, 2048)              0
_________________________________________________________________
dense_3 (Dense)              (None, 1024)              2098176
_________________________________________________________________
dropout_1 (Dropout)          (None, 1024)              0
_________________________________________________________________
dense_4 (Dense)              (None, 133)               136325
=================================================================
Total params: 2,234,501
Trainable params: 2,234,501
Non-trainable params: 0
_________________________________________________________________
```

(IMPLEMENTATION) Compile the Model

In [32]:

```
# Compile model
custom_inception_model.compile(loss='categorical_crossentropy', optimize
r='rmsprop',metrics=['accuracy'])
```

(IMPLEMENTATION) Train the Model

Train your model in the code cell below. Use model checkpointing to save the model that attains the best validation loss.

You are welcome to [augment the training data (https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html)](https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html), but this is not a requirement.

In [33]:

```
# Initialize checkpointer
checkpointer = ModelCheckpoint(filepath='saved_models/weights.best.incep
tion.hdf5',
                               verbose=1, save_best_only=False)


# Fit model
history=custom_inception_model.fit(train_inception, train_targets,
          validation_data=(valid_inception, valid_targets),
          epochs=20, batch_size=20, callbacks=[checkpointer], verbose=1)
```

```
Train on 6680 samples, validate on 835 samples
Epoch 1/20
6560/6680 [============================>.] - ETA: 0s - loss: 2.9215 - a
cc: 0.4096Epoch 00001: saving model to saved_models/weights.best.incept
ion.hdf5
6680/6680 [==============================] - 3s 506us/step - loss: 2.90
58 - acc: 0.4115 - val_loss: 0.8396 - val_acc: 0.7521
Epoch 2/20
6600/6680 [============================>.] - ETA: 0s - loss: 1.5602 - a
cc: 0.6258Epoch 00002: saving model to saved_models/weights.best.incept
ion.hdf5
6680/6680 [==============================] - 3s 455us/step - loss: 1.56
30 - acc: 0.6254 - val_loss: 0.7016 - val_acc: 0.7940
Epoch 3/20
6620/6680 [============================>.] - ETA: 0s - loss: 1.3627 - a
cc: 0.6725Epoch 00003: saving model to saved_models/weights.best.incept
ion.hdf5
6680/6680 [==============================] - 3s 456us/step - loss: 1.36
83 - acc: 0.6716 - val_loss: 0.7009 - val_acc: 0.8132
Epoch 4/20
6640/6680 [============================>.] - ETA: 0s - loss: 1.3000 - a
cc: 0.6962Epoch 00004: saving model to saved_models/weights.best.incept
ion.hdf5
6680/6680 [==============================] - 3s 455us/step - loss: 1.30
50 - acc: 0.6963 - val_loss: 0.6377 - val_acc: 0.8060
Epoch 5/20
6580/6680 [============================>.] - ETA: 0s - loss: 1.2519 - a
cc: 0.7216Epoch 00005: saving model to saved_models/weights.best.incept
ion.hdf5
6680/6680 [==============================] - 3s 455us/step - loss: 1.24
63 - acc: 0.7225 - val_loss: 0.6217 - val_acc: 0.8311
Epoch 6/20
6580/6680 [============================>.] - ETA: 0s - loss: 1.2271 - a
cc: 0.7315Epoch 00006: saving model to saved_models/weights.best.incept
ion.hdf5
6680/6680 [==============================] - 3s 458us/step - loss: 1.22
86 - acc: 0.7313 - val_loss: 0.6577 - val_acc: 0.8335
Epoch 7/20
6600/6680 [============================>.] - ETA: 0s - loss: 1.2021 - a
cc: 0.7373Epoch 00007: saving model to saved_models/weights.best.incept
ion.hdf5
6680/6680 [==============================] - 3s 453us/step - loss: 1.20
14 - acc: 0.7370 - val_loss: 0.7178 - val_acc: 0.8335
Epoch 8/20
6660/6680 [============================>.] - ETA: 0s - loss: 1.1534 - a
cc: 0.7452Epoch 00008: saving model to saved_models/weights.best.incept
ion.hdf5
6680/6680 [==============================] - 3s 457us/step - loss: 1.15
60 - acc: 0.7446 - val_loss: 0.6361 - val_acc: 0.8539
Epoch 9/20
6660/6680 [============================>.] - ETA: 0s - loss: 1.1716 - a
cc: 0.7616Epoch 00009: saving model to saved_models/weights.best.incept
ion.hdf5
6680/6680 [==============================] - 3s 458us/step - loss: 1.17
06 - acc: 0.7617 - val_loss: 0.7195 - val_acc: 0.8419
Epoch 10/20
6560/6680 [============================>.] - ETA: 0s - loss: 1.1529 - a
```

```
cc: 0.7651Epoch 00010: saving model to saved_models/weights.best.incept
ion.hdf5
6680/6680 [==============================] - 3s 460us/step - loss: 1.14
99 - acc: 0.7654 - val_loss: 0.7758 - val_acc: 0.8335
Epoch 11/20
6620/6680 [============================>.] - ETA: 0s - loss: 1.1003 - a
cc: 0.7715Epoch 00011: saving model to saved_models/weights.best.incept
ion.hdf5
6680/6680 [==============================] - 3s 459us/step - loss: 1.10
20 - acc: 0.7716 - val_loss: 0.8044 - val_acc: 0.8311
Epoch 12/20
6640/6680 [============================>.] - ETA: 0s - loss: 1.1262 - a
cc: 0.7768Epoch 00012: saving model to saved_models/weights.best.incept
ion.hdf5
6680/6680 [==============================] - 3s 456us/step - loss: 1.12
67 - acc: 0.7766 - val_loss: 0.7426 - val_acc: 0.8407
Epoch 13/20
6580/6680 [============================>.] - ETA: 0s - loss: 1.1331 - a
cc: 0.7737Epoch 00013: saving model to saved_models/weights.best.incept
ion.hdf5
6680/6680 [==============================] - 3s 457us/step - loss: 1.13
49 - acc: 0.7738 - val_loss: 0.7040 - val_acc: 0.8503
Epoch 14/20
6560/6680 [============================>.] - ETA: 0s - loss: 1.0911 - a
cc: 0.7787Epoch 00014: saving model to saved_models/weights.best.incept
ion.hdf5
6680/6680 [==============================] - 3s 456us/step - loss: 1.10
42 - acc: 0.7778 - val_loss: 0.7750 - val_acc: 0.8479
Epoch 15/20
6600/6680 [============================>.] - ETA: 0s - loss: 1.0813 - a
cc: 0.7912Epoch 00015: saving model to saved_models/weights.best.incept
ion.hdf5
6680/6680 [==============================] - 3s 456us/step - loss: 1.08
62 - acc: 0.7907 - val_loss: 0.8453 - val_acc: 0.8551
Epoch 16/20
6560/6680 [============================>.] - ETA: 0s - loss: 1.1084 - a
cc: 0.7945Epoch 00016: saving model to saved_models/weights.best.incept
ion.hdf5
6680/6680 [==============================] - 3s 453us/step - loss: 1.10
76 - acc: 0.7940 - val_loss: 0.8314 - val_acc: 0.8359
Epoch 17/20
6640/6680 [============================>.] - ETA: 0s - loss: 1.1697 - a
cc: 0.7869Epoch 00017: saving model to saved_models/weights.best.incept
ion.hdf5
6680/6680 [==============================] - 3s 455us/step - loss: 1.16
86 - acc: 0.7871 - val_loss: 0.8226 - val_acc: 0.8395
Epoch 18/20
6560/6680 [============================>.] - ETA: 0s - loss: 1.1474 - a
cc: 0.7971Epoch 00018: saving model to saved_models/weights.best.incept
ion.hdf5
6680/6680 [==============================] - 3s 454us/step - loss: 1.15
06 - acc: 0.7966 - val_loss: 0.8446 - val_acc: 0.8491
Epoch 19/20
6580/6680 [============================>.] - ETA: 0s - loss: 1.1042 - a
cc: 0.7983Epoch 00019: saving model to saved_models/weights.best.incept
ion.hdf5
6680/6680 [==============================] - 3s 456us/step - loss: 1.10
```

```
02 - acc: 0.7985 - val_loss: 0.8792 - val_acc: 0.8371
Epoch 20/20
6580/6680 [============================>.] - ETA: 0s - loss: 1.0823 - a
cc: 0.8024Epoch 00020: saving model to saved_models/weights.best.incept
ion.hdf5
6680/6680 [==============================] - 3s 455us/step - loss: 1.08
01 - acc: 0.8030 - val_loss: 0.8824 - val_acc: 0.8599
```

(IMPLEMENTATION) Load the Model with the Best Validation Loss

In [34]:

```
### TODO: Load the model weights with the best validation loss.
custom_inception_model.load_weights('saved_models/weights.best.inceptio
n.hdf5')
```

(IMPLEMENTATION) Test the Model

Try out your model on the test dataset of dog images. Ensure that your test accuracy is greater than 60%.

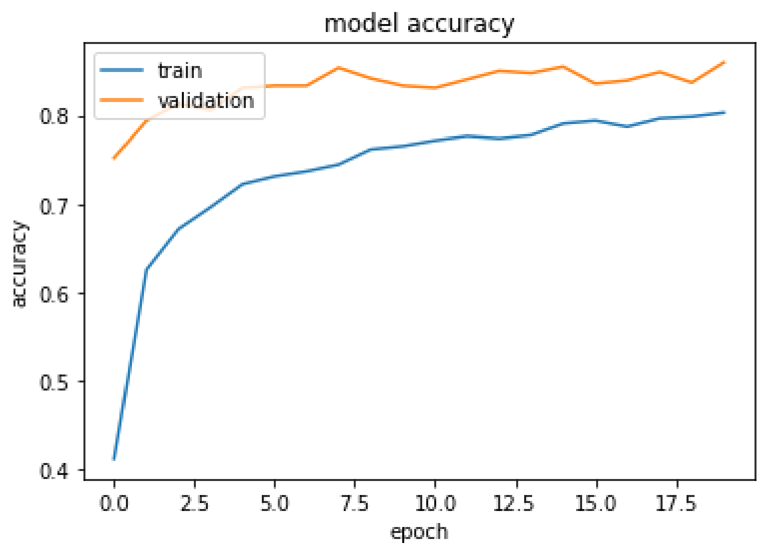
In [35]:

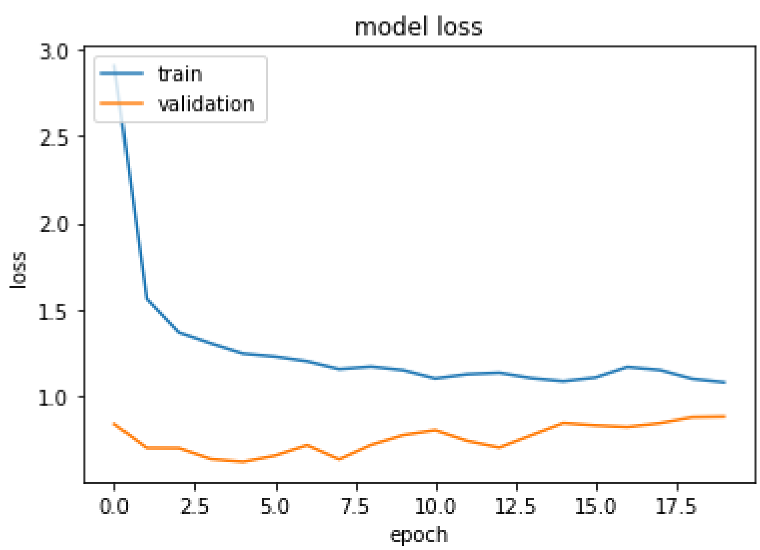
```
### TODO: Calculate classification accuracy on the test dataset.
inception_predictions = [np.argmax(custom_inception_model.predict(np.exp
and_dims(feature, axis=0))) for feature in test_inception]

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

```
Test accuracy: 82.6555%
```

In [36]:

```
# Plot model loss and accuracy
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper left')
plt.show()
# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper left')
plt.show()
```





(IMPLEMENTATION) Predict Dog Breed with the Model

Write a function that takes an image path as input and returns the dog breed (`Affenpinscher` , `Afghan_hound` , etc) that is predicted by your model.

Similar to the analogous function in Step 5, your function should have three steps:

1. Extract the bottleneck features corresponding to the chosen CNN model.
2. Supply the bottleneck features as input to the model to return the predicted vector. Note that the argmax of this prediction vector gives the index of the predicted dog breed.
3. Use the `dog_names` array defined in Step 0 of this notebook to return the corresponding breed.

The functions to extract the bottleneck features can be found in `extract_bottleneck_features.py` , and they have been imported in an earlier code cell. To obtain the bottleneck features corresponding to your chosen CNN architecture, you need to use the function

```
extract_{network}
```

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

In [37]:

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




def inception_predict_breed(img_path):
    """
    Loads image from path.

    Parameters:
        img_path (str): /path/to/image


    Returns:
        Predicted dog breed
    """


    # extract bottleneck features
    bottleneck_feature = extract_InceptionV3(path_to_tensor(img_path))


    # obtain predicted vector
    predicted_vector = custom_inception_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.

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

Sample Human Output

This photo looks like an Afghan Hound.

In [38]:

```
# Import Image
from IPython.display import Image



def display_image(img_path):
    """
    Function to display image from image path


    Parameters:
        img_path(str): /path/to/image


    Returns:
        im(obj): image object for display
    """


    # Create image object
    im = Image(filename=img_path)


    # Return object
    return im
```

In [39]:

```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.
def human_dog_predict(img_path):
    """
    Loads image from path processes it using path_to_tensor,
    Detects whether or not a human or dog is present,
    then uses the custom inception model to predict dog breed.
    Displays image human/dog detections  dog breed prediction .

    Parameters:
        img_path (str): /path/to/image


    Returns:
        None
    """





    # Process image for display
    image = display_image(img_path)


    # Display image
    display(image)


    # Use custom inception model to predict dog breed
    predicted_dog_breed  = inception_predict_breed(img_path)


    # Clean up the predicted dog breed image name
    if predicted_dog_breed is not None:
        predicted_dog_breed = ' '.join(predicted_dog_breed.split('.')[-1
].split('_')).title()


    # Whether or not dog is detected
    is_dog = dog_detector(img_path)


    # Whether or not human is detected
    is_human = face_detector(img_path)


    # Initialize string to print
    s0 = 'The most similar dog breed is:'


    # In case both human and dog are detected, add that to string
    if is_dog and is_human:
        ps = 'Detected a human or a dog. {} {}'.format(s0,predicted_dog_
breed)
```

```
    # If dog is detected and not human, add that to string
    elif is_dog and not is_human:
        ps = 'Detected a dog. {} {}'.format(s0, predicted_dog_breed)


    # If human is detected and not dog, add that to string
    else:
        ps = 'Detected a human. {} {}'.format(s0, predicted_dog_breed)


    # Print string with predictions and image detection
    print(ps)
```

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 definitely better than I expected! The model was able to correctly predict breeds for rotated and side-viewed dog images. The easiest and most straightforward way we could improve our model is by adding more dog images to our dataset. We could also use different optimizers for our loss function when training our model (The Adam and Adamax optimizers). We could also tune our batch size, optimizer learning rate, and dropout rate.

In [40]:
```
## 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.
human_dog_predict('images/gshepherd.jpg')
```

```
Detected a dog. The most similar dog breed is: German Shepherd Dog
```

In [41]:
```
human_dog_predict('images/gshepherd2.jpg')
```

```
Detected a human or a dog. The most similar dog breed is: German Shephe
rd Dog
```

In [42]:
```python
human_dog_predict('images/gshepherd3.jpg')
```

Detected a dog. The most similar dog breed is: German Shepherd Dog

In [43]:
```python
human_dog_predict('images/multiple.jpg')
```

Detected a dog. The most similar dog breed is: American Foxhound

In [44]:
```python
human_dog_predict('images/chewbacca.jpg')
```

Detected a dog. The most similar dog breed is: Lhasa Apso

In [45]:
```
## LOL EPIC
image = display_image('images/lhasaapso.jpg')



display(image)
```

In [47]:
```
human_dog_predict('images/sample_human_2.png')
```

Detected a human. The most similar dog breed is: Anatolian Shepherd Dog