# dog_app

November 24, 2019

## 1 Data Scientist Nanodegree

### 1.1 Convolutional Neural Networks

### 1.2 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.

## Step 0: Import Datasets

### 1.2.1 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

[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/train')
valid_files, valid_targets = load_dataset('../../../data/dog_images/valid')
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_images/
 ↪train/*/"))]

# print statistics about the dataset
print('There are %d total dog categories.' % len(dog_names))
print('There are %s total dog images.\n' % len(np.hstack([train_files,␣
 ↪valid_files, test_files])))
print('There are %d training dog images.' % len(train_files))
print('There are %d validation dog images.' % len(valid_files))
print('There are %d test dog images.'% len(test_files))
```

```
Using TensorFlow backend.

There are 133 total dog categories.
There are 8351 total dog images.

There are 6680 training dog images.
There are 835 validation dog images.
There are 836 test dog images.
```

### 1.2.2 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`.

[2]:
```
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

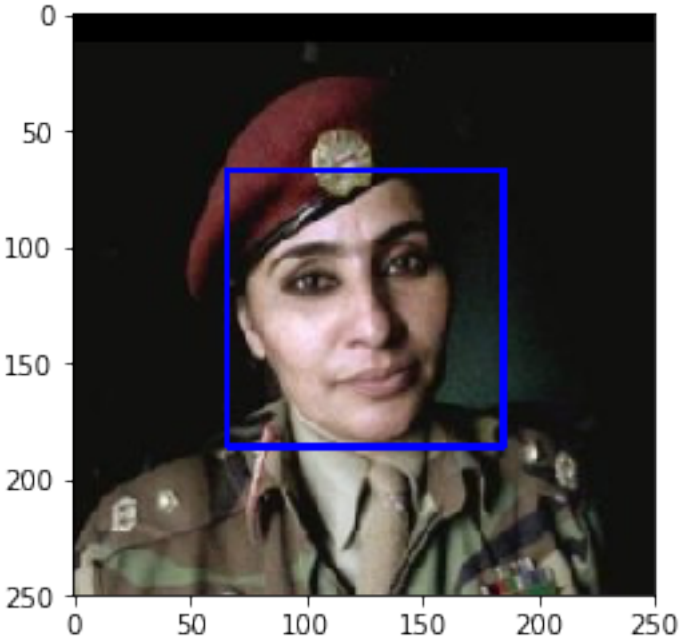
We use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images. OpenCV provides many pre-trained face detectors, stored as XML files on github. We have downloaded one of these detectors and stored it in the `haarcascades` directory.

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

[3]:
```
import cv2
import matplotlib.pyplot as plt
%matplotlib inline

# extract pre-trained face detector
face_cascade = cv2.CascadeClassifier('haarcascades/haarcascade_frontalface_alt.
 ↪xml')

# load color (BGR) image
img = cv2.imread(human_files[3])
# convert BGR image to grayscale
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# find faces in image
faces = face_cascade.detectMultiScale(gray)

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

# get bounding box for each detected face
for (x,y,w,h) in faces:
    # add bounding box to color image
    cv2.rectangle(img,(x,y),(x+w,y+h),(255,0,0),2)
```

```
# convert BGR image to RGB for plotting
cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

# display the image, along with bounding box
plt.imshow(cv_rgb)
plt.show()
```

```
Number of faces detected: 1
```




Before using any of the face detectors, it is standard procedure to convert the images to grayscale. The `detectMultiScale` function executes the classifier stored in `face_cascade` and takes the grayscale image as a parameter.

In the above code, `faces` is a numpy array of detected faces, where each row corresponds to a detected face. Each detected face is a 1D array with four entries that specifies the bounding box of the detected face. The first two entries in the array (extracted in the above code as `x` and `y`) specify the horizontal and vertical positions of the top left corner of the bounding box. The last two entries in the array (extracted here as `w` and `h`) specify the width and height of the box.

### 1.2.3 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.

[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
```

#### 1.2.4 (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:**

- What percentage of the first 100 images in `human_files` have a detected human face? 100%
- What percentage of the first 100 images in `dog_files` have a detected human face? 11%

[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 FaceDetector_Test(files) :
    total_cnt = len(files)
    detected_cnt = 0
    for img_path in files:
        detected_cnt += face_detector(img_path)
    perc = detected_cnt/total_cnt * 100
    return perc

perc_detected_human_face = FaceDetector_Test(human_files_short)
perc_detected_dog_face = FaceDetector_Test(dog_files_short)
```

[6]:
```
print("What percentage of the first 100 images in `human_files` have a detected␣
 ↪human face? {}%".format(perc_detected_human_face))
print("What percentage of the first 100 images in `dog_files` have a detected␣
 ↪human face? {}%".format(perc_detected_dog_face))
```

```
What percentage of the first 100 images in `human_files` have a detected human
face? 100.0%
What percentage of the first 100 images in `dog_files` have a detected human
face? 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:**

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.

[7]:
```
## (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 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, 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. 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.

[8]:
```
from keras.applications.resnet50 import ResNet50

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

### 1.2.5 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

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

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 $224 \times 224$ 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

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

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!

[9]:
```
from keras.preprocessing import image
from tqdm import tqdm

def path_to_tensor(img_path):
    # loads RGB image as PIL.Image.Image type
    img = image.load_img(img_path, target_size=(224, 224))
    # convert PIL.Image.Image type to 3D tensor with shape (224, 224, 3)
    x = image.img_to_array(img)
    # convert 3D tensor to 4D tensor with shape (1, 224, 224, 3) and return 4D␣
 ↪tensor
    return np.expand_dims(x, axis=0)

def paths_to_tensor(img_paths):
    list_of_tensors = [path_to_tensor(img_path) for img_path in tqdm(img_paths)]
    return np.vstack(list_of_tensors)
```

### 1.2.6 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.

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.

[10]:
```
from keras.applications.resnet50 import preprocess_input, decode_predictions

def ResNet50_predict_labels(img_path):
    # returns prediction vector for image located at img_path
    img = preprocess_input(path_to_tensor(img_path))
    return np.argmax(ResNet50_model.predict(img))
```

### 1.2.7 Write a Dog Detector

While looking at the dictionary, 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).

[11]:
```
### 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))
```

### 1.2.8 (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:**

- What percentage of the images in `human_files_short` have a detected dog? 0%
- What percentage of the images in `dog_files_short` have a detected dog? 100%

[12]:
```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.

def DogDetector_Test(files) :
    total_cnt = len(files)
    detected_cnt = 0
    for img_path in files:
        detected_cnt += dog_detector(img_path)
    perc = detected_cnt/total_cnt * 100
    return perc

perc_human_detected_dog_pic = DogDetector_Test(human_files_short)
perc_dog_detected_dog_pic = DogDetector_Test(dog_files_short)
```

[13]:
```
print("What percentage of the images in human_files_short have a detected dog?␣
 ↪{}%".format(perc_human_detected_dog_pic))
print("What percentage of the images in dog_files_short have a detected dog?␣
 ↪{}%".format(perc_dog_detected_dog_pic))
```

```
What percentage of the images in human_files_short have a detected dog? 0.0%
What percentage of the images in dog_files_short have a detected dog? 100.0%
```

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

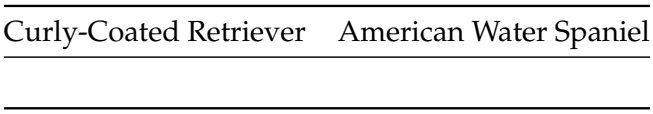
Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN that classifies dog breeds. You must create your CNN *from scratch* (so, you can't use transfer learning *yet*!), and you must attain a test accuracy of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

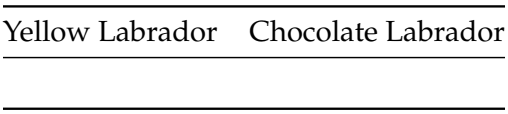
Brittany Welsh Springer Spaniel

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

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!

### 1.2.9 Pre-process the Data

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

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

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

```
test_tensors = paths_to_tensor(test_files).astype('float32')/255
```

```
100%|| 6680/6680 [01:10<00:00, 94.15it/s]
100%|| 835/835 [00:08<00:00, 104.22it/s]
100%|| 836/836 [00:07<00:00, 105.85it/s]
```

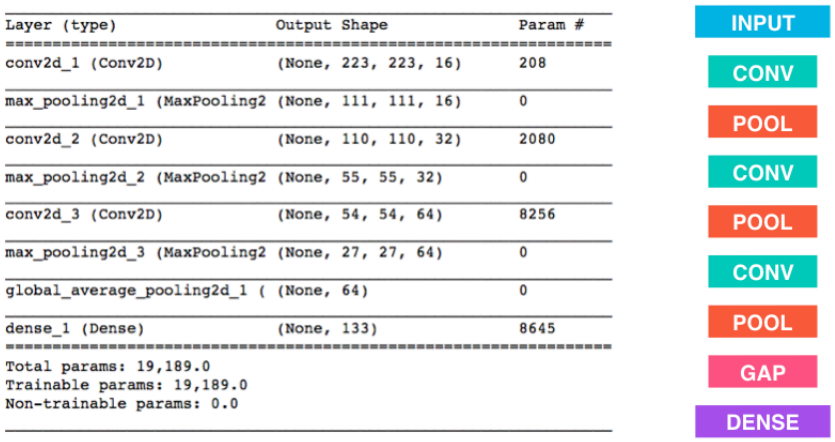
[15]:
```
train_tensors_backup = train_tensors
valid_tensors_backup = valid_tensors
test_tensors_backup = test_tensors
```

### 1.2.10 (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:**

(reference of https://keras.io/getting-started/sequential-model-guide/).

Steps: 1. First feed the image with convolutional layer to pick up edge/curve and then to detect shapes 2. Add first Max Pooling layer to remove the spacial dimention, but pass the valid info 3. Add the second layer of convolutional and max pooling 4. Add the third layer of convolutional and max pooling 5. Add the first Dropout layer to minimize overfitting 6. Add GAP and generate 2D tensor with shape 7. Add first Dense layer 8. Add another Dropout layer 9. Add final Dense/fully connected layer (with same dimention of dog classes) for classification

[16]:
```
# dog_classes = len(dog_names)
# print(dog_classes)#model.add(Dense(200, activation='relu'))
# # 133
# train_input_shape = train_tensors.shape[1:]
# print(train_input_shape)
# # (224, 224, 3)
```

[17]:
```
# Test a few models:

# model2: remove first Dense at second layer and remove strides in MaxPooling
# Note Dense layer in first two layers detoriated loss
# loss: 4, accuracy 0.09
# model3: add Dropout at each layer
#       : try Batch Normalization: Accelerating Deep Network Training by␣
 ↪Reducing Internal Covariate Shift (https://arxiv.org/abs/1502.03167)
#       : given the time consuming of Batch Normalization, only apply to first␣
 ↪Conv layer
# bad result

# model4: Keep one Dropout at last layer only. Use Flattern instead of GAP
# too slow, and loss is too high

# Final: model3
# no Batch Normalization, use GAP, keep last two Dense, keep last two Dropout␣
 ↪(put each layer underfit)
# 10 epochs
```

[18]:
```
from keras.layers import Conv2D, MaxPooling2D, GlobalAveragePooling2D
from keras.layers import Dropout, Flatten, Dense
from keras.models import Sequential
from keras.layers.normalization import BatchNormalization

model = Sequential()

### TODO: Define your architecture.


#1. Get the total number of dog classes and the image dimention
dog_classes = len(dog_names)
train_input_shape = train_tensors.shape[1:]

#2. First Convolution layer
# - creates a convolution kernel that is convolved with the layer input to␣
 ↪produce a tensor of outputs
# As the first layer in a model, provide the keyword argument input_shape,␣
 ↪(tuple of integers, does not include the batch axis)
# filters: dimensionality of the number of output filters in the convolution
```

```
# kernel_size: 2-digit, specifying the height and width of the 2D convolution␣
 ↪window. A single integer to specify the same value for all spatial␣
 ↪dimensions.
# padding: "same" results in padding the input such that the output has the␣
 ↪same length as the original input.
# activation function: Non-Linearity (RELU) - replacing all negative pixel␣
 ↪values in feature map by zero.
model.add(Conv2D(filters=16, kernel_size=2, padding='same', activation='relu',␣
 ↪input_shape= train_input_shape ))

# Batch normalization layer Normalize the activations of the previous layer at␣
 ↪each batch,
# applies a transformation that maintains the mean activation close to 0 and␣
 ↪the activation standard deviation close to 1.
# model.add(BatchNormalization(axis=2, scale=False))

#3. First pooling
# pool_size: Integer, size of the max pooling windows.
# strides: Integer, or None. Factor by which to downscale.
#model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2), padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

#4. Second layer
# keep all other arguments same but the filters doubled
# Dense: regular densely-connected NN layer.
model.add(Conv2D(filters=32, kernel_size=2, padding='same',  activation='relu'))
#model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2), padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2)))
#model.add(Dropout(0.2))
model.add(Dense(100, activation='relu'))

#4. Third layer
# keep all other arguments same but the filters doubled
model.add(Conv2D(filters=64, kernel_size=2, padding='same',  activation='relu'))
# model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2), padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2)))
#model.add(Dense(200, activation='relu'))

#5. Flatten
# Dropout consists in randomly setting a fraction rate of input units to 0 at␣
 ↪each update during training time, which helps prevent overfitting.
model.add(Dropout(0.2))
model.add(GlobalAveragePooling2D())  # usa GAP instead of Flattern() to reduce␣
 ↪the nb of parameter and run time
# Flattens the input. Does not affect the batch size.
# model.add(Flatten())
```

```
model.add(Dense(500, activation='relu'))

#6. Fully connect layers
model.add(Dropout(0.3))
model.add(Dense(133, activation='softmax'))

# End: Define your architecture.


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
_________________________________________________________________
dense_1 (Dense)              (None, 56, 56, 100)       3300
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 56, 56, 64)        25664
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 28, 28, 64)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 28, 28, 64)        0
_________________________________________________________________
global_average_pooling2d_1 ( (None, 64)                0
_________________________________________________________________
dense_2 (Dense)              (None, 500)               32500
_________________________________________________________________
dropout_2 (Dropout)          (None, 500)               0
_________________________________________________________________
dense_3 (Dense)              (None, 133)               66633
=================================================================
Total params: 130,385
Trainable params: 130,385
Non-trainable params: 0
_________________________________________________________________
```

### 1.2.11 Compile the Model

[19]:
```
model.compile(optimizer='rmsprop', loss='categorical_crossentropy',␣
↪metrics=['accuracy'])
```

### 1.2.12 (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, but this is not a requirement.

[20]:
```
from keras.callbacks import ModelCheckpoint

### TODO: specify the number of epochs that you would like to use to train the␣
↪model.

epochs = 10

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

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

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

```
Train on 6680 samples, validate on 835 samples
Epoch 1/10
6660/6680 [============================>.] - ETA: 0s - loss: 4.8831 - acc:
0.0086Epoch 00001: val_loss improved from inf to 4.87142, saving model to
saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 28s 4ms/step - loss: 4.8831 - acc:
0.0085 - val_loss: 4.8714 - val_acc: 0.0096
Epoch 2/10
6660/6680 [============================>.] - ETA: 0s - loss: 4.8714 - acc:
0.0113Epoch 00002: val_loss improved from 4.87142 to 4.86985, saving model to
saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 27s 4ms/step - loss: 4.8715 - acc:
0.0112 - val_loss: 4.8698 - val_acc: 0.0108
Epoch 3/10
6660/6680 [============================>.] - ETA: 0s - loss: 4.8630 - acc:
0.0126Epoch 00003: val_loss improved from 4.86985 to 4.84003, saving model to
saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 27s 4ms/step - loss: 4.8630 - acc:
0.0126 - val_loss: 4.8400 - val_acc: 0.0192
Epoch 4/10
```

```
6660/6680 [============================>.] - ETA: 0s - loss: 4.8050 - acc:
0.0177Epoch 00004: val_loss improved from 4.84003 to 4.79481, saving model to
saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 27s 4ms/step - loss: 4.8053 - acc:
0.0178 - val_loss: 4.7948 - val_acc: 0.0180
Epoch 5/10
6660/6680 [============================>.] - ETA: 0s - loss: 4.7292 - acc:
0.0219Epoch 00005: val_loss improved from 4.79481 to 4.73945, saving model to
saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 28s 4ms/step - loss: 4.7290 - acc:
0.0219 - val_loss: 4.7394 - val_acc: 0.0228
Epoch 6/10
6660/6680 [============================>.] - ETA: 0s - loss: 4.6560 - acc:
0.0273Epoch 00006: val_loss improved from 4.73945 to 4.63965, saving model to
saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 27s 4ms/step - loss: 4.6549 - acc:
0.0274 - val_loss: 4.6396 - val_acc: 0.0407
Epoch 7/10
6660/6680 [============================>.] - ETA: 0s - loss: 4.5710 - acc:
0.0345Epoch 00007: val_loss improved from 4.63965 to 4.56629, saving model to
saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 27s 4ms/step - loss: 4.5717 - acc:
0.0344 - val_loss: 4.5663 - val_acc: 0.0371
Epoch 8/10
6660/6680 [============================>.] - ETA: 0s - loss: 4.4899 - acc:
0.0422Epoch 00008: val_loss improved from 4.56629 to 4.52400, saving model to
saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 27s 4ms/step - loss: 4.4897 - acc:
0.0421 - val_loss: 4.5240 - val_acc: 0.0407
Epoch 9/10
6660/6680 [============================>.] - ETA: 0s - loss: 4.4161 - acc:
0.0462Epoch 00009: val_loss improved from 4.52400 to 4.47793, saving model to
saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 28s 4ms/step - loss: 4.4158 - acc:
0.0463 - val_loss: 4.4779 - val_acc: 0.0575
Epoch 10/10
6660/6680 [============================>.] - ETA: 0s - loss: 4.3583 - acc:
0.0521Epoch 00010: val_loss improved from 4.47793 to 4.40929, saving model to
saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 28s 4ms/step - loss: 4.3575 - acc:
0.0522 - val_loss: 4.4093 - val_acc: 0.0587
```

[20]: `<keras.callbacks.History at 0x7f87ccebc080>`

### 1.2.13 Load the Model with the Best Validation Loss

[21]:
```
model.load_weights('saved_models/weights.best.from_scratch.hdf5')
```

### 1.2.14 Test the Model

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

[22]:
```
# 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)

# 3.852% if epochs = 5
# 4.426% if epochs = 10
```

```
Test accuracy: 5.5024%
```

## 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.

### 1.2.15 Obtain Bottleneck Features

[23]:
```
bottleneck_features = np.load('bottleneck_features/DogVGG16Data.npz')
train_VGG16 = bottleneck_features['train']
valid_VGG16 = bottleneck_features['valid']
test_VGG16 = bottleneck_features['test']
```

### 1.2.16 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.

[24]:
```
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_4 (Dense)              (None, 133)               68229
=================================================================
Total params: 68,229
Trainable params: 68,229
Non-trainable params: 0
_________________________________________________________________
```

### 1.2.17 Compile the Model

[25]:
```
VGG16_model.compile(loss='categorical_crossentropy', optimizer='rmsprop',␣
↪metrics=['accuracy'])
```

### 1.2.18 Train the Model

[26]:
```
checkpointer = ModelCheckpoint(filepath='saved_models/weights.best.VGG16.hdf5',
                               verbose=1, save_best_only=True)

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

```
Train on 6680 samples, validate on 835 samples
Epoch 1/20
6620/6680 [============================>.] - ETA: 0s - loss: 11.7599 - acc:
0.1411Epoch 00001: val_loss improved from inf to 9.99362, saving model to
saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 335us/step - loss: 11.7417 -
acc: 0.1424 - val_loss: 9.9936 - val_acc: 0.2443
Epoch 2/20
6580/6680 [============================>.] - ETA: 0s - loss: 9.1982 - acc:
0.3242Epoch 00002: val_loss improved from 9.99362 to 9.14447, saving model to
saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 290us/step - loss: 9.1865 - acc:
0.3241 - val_loss: 9.1445 - val_acc: 0.3210
Epoch 3/20
6640/6680 [============================>.] - ETA: 0s - loss: 8.5234 - acc:
0.4006Epoch 00003: val_loss improved from 9.14447 to 8.91702, saving model to
saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 286us/step - loss: 8.5269 - acc:
0.4004 - val_loss: 8.9170 - val_acc: 0.3497
Epoch 4/20
6660/6680 [============================>.] - ETA: 0s - loss: 8.2633 - acc:
```

```
0.4375Epoch 00004: val_loss improved from 8.91702 to 8.66561, saving model to
saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 289us/step - loss: 8.2559 - acc:
0.4379 - val_loss: 8.6656 - val_acc: 0.3856
Epoch 5/20
6600/6680 [============================>.] - ETA: 0s - loss: 8.0298 - acc:
0.4630Epoch 00005: val_loss improved from 8.66561 to 8.55124, saving model to
saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 278us/step - loss: 8.0141 - acc:
0.4632 - val_loss: 8.5512 - val_acc: 0.3713
Epoch 6/20
6620/6680 [============================>.] - ETA: 0s - loss: 7.7245 - acc:
0.4831Epoch 00006: val_loss improved from 8.55124 to 8.37696, saving model to
saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 261us/step - loss: 7.7306 - acc:
0.4825 - val_loss: 8.3770 - val_acc: 0.3988
Epoch 7/20
6580/6680 [============================>.] - ETA: 0s - loss: 7.5875 - acc:
0.5008Epoch 00007: val_loss improved from 8.37696 to 8.27379, saving model to
saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 264us/step - loss: 7.5873 - acc:
0.5009 - val_loss: 8.2738 - val_acc: 0.4048
Epoch 8/20
6600/6680 [============================>.] - ETA: 0s - loss: 7.4331 - acc:
0.5167Epoch 00008: val_loss improved from 8.27379 to 8.24346, saving model to
saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 264us/step - loss: 7.4428 - acc:
0.5162 - val_loss: 8.2435 - val_acc: 0.4000
Epoch 9/20
6600/6680 [============================>.] - ETA: 0s - loss: 7.3581 - acc:
0.5229Epoch 00009: val_loss improved from 8.24346 to 8.07605, saving model to
saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 262us/step - loss: 7.3571 - acc:
0.5232 - val_loss: 8.0760 - val_acc: 0.4180
Epoch 10/20
6620/6680 [============================>.] - ETA: 0s - loss: 7.2478 - acc:
0.5343Epoch 00010: val_loss improved from 8.07605 to 8.02403, saving model to
saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 261us/step - loss: 7.2624 - acc:
0.5335 - val_loss: 8.0240 - val_acc: 0.4275
Epoch 11/20
6620/6680 [============================>.] - ETA: 0s - loss: 7.2465 - acc:
0.5408Epoch 00011: val_loss did not improve
6680/6680 [==============================] - 2s 260us/step - loss: 7.2321 - acc:
0.5418 - val_loss: 8.0323 - val_acc: 0.4228
Epoch 12/20
6620/6680 [============================>.] - ETA: 0s - loss: 7.2348 - acc:
0.5434Epoch 00012: val_loss improved from 8.02403 to 7.98977, saving model to
```

```
saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 264us/step - loss: 7.2257 - acc:
0.5439 - val_loss: 7.9898 - val_acc: 0.4275
Epoch 13/20
6660/6680 [============================>.] - ETA: 0s - loss: 7.2091 - acc:
0.5467Epoch 00013: val_loss did not improve
6680/6680 [==============================] - 2s 264us/step - loss: 7.2117 - acc:
0.5466 - val_loss: 7.9914 - val_acc: 0.4371
Epoch 14/20
6600/6680 [============================>.] - ETA: 0s - loss: 7.2019 - acc:
0.5461Epoch 00014: val_loss improved from 7.98977 to 7.96191, saving model to
saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 262us/step - loss: 7.2074 - acc:
0.5458 - val_loss: 7.9619 - val_acc: 0.4443
Epoch 15/20
6520/6680 [============================>.] - ETA: 0s - loss: 7.0724 - acc:
0.5535Epoch 00015: val_loss improved from 7.96191 to 7.92616, saving model to
saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 265us/step - loss: 7.1013 - acc:
0.5516 - val_loss: 7.9262 - val_acc: 0.4359
Epoch 16/20
6600/6680 [============================>.] - ETA: 0s - loss: 7.0396 - acc:
0.5585Epoch 00016: val_loss improved from 7.92616 to 7.90985, saving model to
saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 264us/step - loss: 7.0569 - acc:
0.5573 - val_loss: 7.9098 - val_acc: 0.4407
Epoch 17/20
6620/6680 [============================>.] - ETA: 0s - loss: 7.0472 - acc:
0.5582Epoch 00017: val_loss improved from 7.90985 to 7.85776, saving model to
saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 262us/step - loss: 7.0442 - acc:
0.5584 - val_loss: 7.8578 - val_acc: 0.4431
Epoch 18/20
6580/6680 [============================>.] - ETA: 0s - loss: 6.9563 - acc:
0.5602Epoch 00018: val_loss improved from 7.85776 to 7.70701, saving model to
saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 264us/step - loss: 6.9633 - acc:
0.5597 - val_loss: 7.7070 - val_acc: 0.4683
Epoch 19/20
6580/6680 [============================>.] - ETA: 0s - loss: 6.8235 - acc:
0.5693Epoch 00019: val_loss did not improve
6680/6680 [==============================] - 2s 261us/step - loss: 6.8408 - acc:
0.5683 - val_loss: 7.7111 - val_acc: 0.4515
Epoch 20/20
6620/6680 [============================>.] - ETA: 0s - loss: 6.8100 - acc:
0.5739Epoch 00020: val_loss improved from 7.70701 to 7.70114, saving model to
saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 264us/step - loss: 6.8171 - acc:
```

```
0.5734 - val_loss: 7.7011 - val_acc: 0.4515
```

[26]:
```
<keras.callbacks.History at 0x7f87ae3f6978>
```

#### 1.2.19 Load the Model with the Best Validation Loss

[27]:
```
VGG16_model.load_weights('saved_models/weights.best.VGG16.hdf5')
```

#### 1.2.20 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.

[28]:
```
# get index of predicted dog breed for each image in test set
VGG16_predictions = [np.argmax(VGG16_model.predict(np.expand_dims(feature,␣
 ↪axis=0))) for feature in test_VGG16]

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

```
Test accuracy: 47.6077%
```

#### 1.2.21 Predict Dog Breed with the Model

[29]:
```
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 bottleneck features - ResNet-50 bottleneck features - Inception bottleneck features - Xception 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.

#### 1.2.22 (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']
```

[30]:
```
# The npz file is missing in Udacity location
# Load at the first time only

# import requests
# url = "https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/
 ↪DogInceptionV3Data.npz"
# response = requests.get(url, stream=True)
# print(len(response.content))
# 1710285390

# open( 'bottleneck_features/DogInceptionV3Data.npz', 'wb').write(response.
 ↪content)
# 1710285390
```

[31]:
```
### TODO: Obtain bottleneck features from another pre-trained CNN.

# use InceptionV3 CNN
bottleneck_features = np.load(r'bottleneck_features/DogInceptionV3Data.npz')

train_InceptionV3 = bottleneck_features['train']
valid_InceptionV3 = bottleneck_features['valid']
test_InceptionV3 = bottleneck_features['test']
```

#### 1.2.23 (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 leveraged the VGG16 architecture that tested in piror sections, but applied InceptionV3 model. This architecture is suitable for this problem because InceptionV3 model was already pre-trained and weighted well and the model output will be used as input for my CNN. It will need to get 133 ending nodes to represent the 133 dog breeds provided.
In the last step, I added the GAP layer o help extend the temporal data and also added another dense perceptron layer to make full connection for the appropriate classifications.

[32]:
```
### TODO: Define your architecture.
InceptionV3_model = Sequential()
InceptionV3_model.add(GlobalAveragePooling2D(input_shape=train_InceptionV3.
 ↪shape[1:]))
InceptionV3_model.add(Dense(133, activation='softmax'))

InceptionV3_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
global_average_pooling2d_3 ( (None, 2048)              0
_________________________________________________________________
dense_5 (Dense)              (None, 133)               272517
=================================================================
Total params: 272,517
Trainable params: 272,517
Non-trainable params: 0
_________________________________________________________________
```

#### 1.2.24 (IMPLEMENTATION) Compile the Model

[33]:
```
### TODO: Compile the model.
InceptionV3_model.compile(loss='categorical_crossentropy', optimizer='rmsprop',␣
 ↪metrics=['accuracy'])
```

#### 1.2.25 (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, but this is not a requirement.

[34]:
```
### TODO: Train the model.

checkpointer = ModelCheckpoint(filepath='saved_models/weights.best.InceptionV3.
 ↪hdf5',
                               verbose=1, save_best_only=True)

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

```
Train on 6680 samples, validate on 835 samples
Epoch 1/20
6500/6680 [============================>.] - ETA: 0s - loss: 1.1627 - acc:
0.7057Epoch 00001: val_loss improved from inf to 0.62109, saving model to
saved_models/weights.best.InceptionV3.hdf5
6680/6680 [==============================] - 2s 367us/step - loss: 1.1492 - acc:
0.7087 - val_loss: 0.6211 - val_acc: 0.8180
Epoch 2/20
6660/6680 [============================>.] - ETA: 0s - loss: 0.4706 - acc:
0.8544Epoch 00002: val_loss did not improve
6680/6680 [==============================] - 2s 315us/step - loss: 0.4699 - acc:
0.8546 - val_loss: 0.6562 - val_acc: 0.8204
Epoch 3/20
6500/6680 [============================>.] - ETA: 0s - loss: 0.3547 - acc:
0.8940Epoch 00003: val_loss did not improve
6680/6680 [==============================] - 2s 312us/step - loss: 0.3567 - acc:
0.8939 - val_loss: 0.6503 - val_acc: 0.8407
Epoch 4/20
6500/6680 [============================>.] - ETA: 0s - loss: 0.2822 - acc:
0.9078Epoch 00004: val_loss did not improve
6680/6680 [==============================] - 2s 313us/step - loss: 0.2823 - acc:
0.9078 - val_loss: 0.6713 - val_acc: 0.8443
Epoch 5/20
6540/6680 [============================>.] - ETA: 0s - loss: 0.2360 - acc:
0.9235Epoch 00005: val_loss did not improve
6680/6680 [==============================] - 2s 312us/step - loss: 0.2356 - acc:
0.9238 - val_loss: 0.7139 - val_acc: 0.8419
Epoch 6/20
6520/6680 [============================>.] - ETA: 0s - loss: 0.2008 - acc:
0.9348Epoch 00006: val_loss did not improve
6680/6680 [==============================] - 2s 311us/step - loss: 0.2005 - acc:
0.9352 - val_loss: 0.7084 - val_acc: 0.8515
Epoch 7/20
6520/6680 [============================>.] - ETA: 0s - loss: 0.1670 - acc:
0.9486Epoch 00007: val_loss did not improve
6680/6680 [==============================] - 2s 313us/step - loss: 0.1674 - acc:
0.9481 - val_loss: 0.7416 - val_acc: 0.8419
Epoch 8/20
6520/6680 [============================>.] - ETA: 0s - loss: 0.1453 - acc:
0.9561Epoch 00008: val_loss did not improve
6680/6680 [==============================] - 2s 311us/step - loss: 0.1459 - acc:
0.9557 - val_loss: 0.7662 - val_acc: 0.8479
Epoch 9/20
6500/6680 [============================>.] - ETA: 0s - loss: 0.1292 - acc:
0.9600Epoch 00009: val_loss did not improve
6680/6680 [==============================] - 2s 313us/step - loss: 0.1285 - acc:
0.9602 - val_loss: 0.7486 - val_acc: 0.8575
Epoch 10/20
```

```
6500/6680 [============================>.] - ETA: 0s - loss: 0.1079 - acc:
0.9649Epoch 00010: val_loss did not improve
6680/6680 [==============================] - 2s 316us/step - loss: 0.1081 - acc:
0.9651 - val_loss: 0.8012 - val_acc: 0.8563
Epoch 11/20
6500/6680 [============================>.] - ETA: 0s - loss: 0.0899 - acc:
0.9712Epoch 00011: val_loss did not improve
6680/6680 [==============================] - 2s 314us/step - loss: 0.0891 - acc:
0.9716 - val_loss: 0.8400 - val_acc: 0.8467
Epoch 12/20
6500/6680 [============================>.] - ETA: 0s - loss: 0.0778 - acc:
0.9752Epoch 00012: val_loss did not improve
6680/6680 [==============================] - 2s 315us/step - loss: 0.0783 - acc:
0.9750 - val_loss: 0.8686 - val_acc: 0.8383
Epoch 13/20
6500/6680 [============================>.] - ETA: 0s - loss: 0.0667 - acc:
0.9780Epoch 00013: val_loss did not improve
6680/6680 [==============================] - 2s 313us/step - loss: 0.0686 - acc:
0.9777 - val_loss: 0.8809 - val_acc: 0.8515
Epoch 14/20
6500/6680 [============================>.] - ETA: 0s - loss: 0.0644 - acc:
0.9800Epoch 00014: val_loss did not improve
6680/6680 [==============================] - 2s 314us/step - loss: 0.0661 - acc:
0.9799 - val_loss: 0.8472 - val_acc: 0.8503
Epoch 15/20
6500/6680 [============================>.] - ETA: 0s - loss: 0.0543 - acc:
0.9829Epoch 00015: val_loss did not improve
6680/6680 [==============================] - 2s 314us/step - loss: 0.0565 - acc:
0.9822 - val_loss: 0.9102 - val_acc: 0.8455
Epoch 16/20
6520/6680 [============================>.] - ETA: 0s - loss: 0.0471 - acc:
0.9856Epoch 00016: val_loss did not improve
6680/6680 [==============================] - 2s 314us/step - loss: 0.0470 - acc:
0.9856 - val_loss: 0.9076 - val_acc: 0.8467
Epoch 17/20
6500/6680 [============================>.] - ETA: 0s - loss: 0.0445 - acc:
0.9843Epoch 00017: val_loss did not improve
6680/6680 [==============================] - 2s 313us/step - loss: 0.0437 - acc:
0.9846 - val_loss: 0.8894 - val_acc: 0.8587
Epoch 18/20
6500/6680 [============================>.] - ETA: 0s - loss: 0.0352 - acc:
0.9888Epoch 00018: val_loss did not improve
6680/6680 [==============================] - 2s 311us/step - loss: 0.0352 - acc:
0.9888 - val_loss: 0.8917 - val_acc: 0.8515
Epoch 19/20
6620/6680 [============================>.] - ETA: 0s - loss: 0.0322 - acc:
0.9903Epoch 00019: val_loss did not improve
6680/6680 [==============================] - 2s 320us/step - loss: 0.0320 - acc:
```

```
0.9904 - val_loss: 0.9665 - val_acc: 0.8503
Epoch 20/20
6520/6680 [============================>.] - ETA: 0s - loss: 0.0302 - acc:
0.9897Epoch 00020: val_loss did not improve
6680/6680 [==============================] - 2s 311us/step - loss: 0.0307 - acc:
0.9897 - val_loss: 0.9601 - val_acc: 0.8491
```

[34]:
```
<keras.callbacks.History at 0x7f87ae1a6748>
```

#### 1.2.26 (IMPLEMENTATION) Load the Model with the Best Validation Loss

[35]:
```
### TODO: Load the model weights with the best validation loss.

InceptionV3_model.load_weights('saved_models/weights.best.InceptionV3.hdf5')
```

#### 1.2.27 (IMPLEMENTATION) Test the Model

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

[36]:
```
### TODO: Calculate classification accuracy on the test dataset.

# get index of predicted dog breed for each image in test set
InceptionV3_predictions = [np.argmax(InceptionV3_model.predict(np.
 ↪expand_dims(feature, axis=0))) for feature in test_InceptionV3]

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

```
Test accuracy: 78.3493%
```

#### 1.2.28 (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`.

[37]:
```
### TODO: Write a function that takes a path to an image as input
### and returns the dog breed that is predicted by the model.

def InceptionV3_predict_breed(img_path):
    # extract bottleneck features
    bottleneck_feature = extract_InceptionV3(path_to_tensor(img_path))
    # obtain predicted vector
    predicted_vector = InceptionV3_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. ### (IMPLEMENTATION) Write your Algorithm

[38]:
```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.

from PIL import Image

def dog_breed_identification(img_path):

    img = Image.open(img_path)
    plt.imshow(img)
```

```
    plt.show()

    dog_detected = dog_detector(img_path)
    human_detected = face_detector(img_path)

    if dog_detected:
        dog_breed = InceptionV3_predict_breed(img_path)
        print("A dog is detected and its breed is {}".format(dog_breed))
    elif human_detected:
        dog_breed = InceptionV3_predict_breed(img_path)
        print("A human is detected, but if this was a dog, its breed would be␣
↪{}".format(dog_breed))
    else:
        print("Um, it can't identify as a dog or human. Try another image.")
```

## 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?

### 1.2.29 (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:**

Ways to improve the mode:

```
Increase number of epoches (proved in my CNN architechture)
Augmentate training data
Tune some of the model parameters (tested in my CNN Architecture)
```

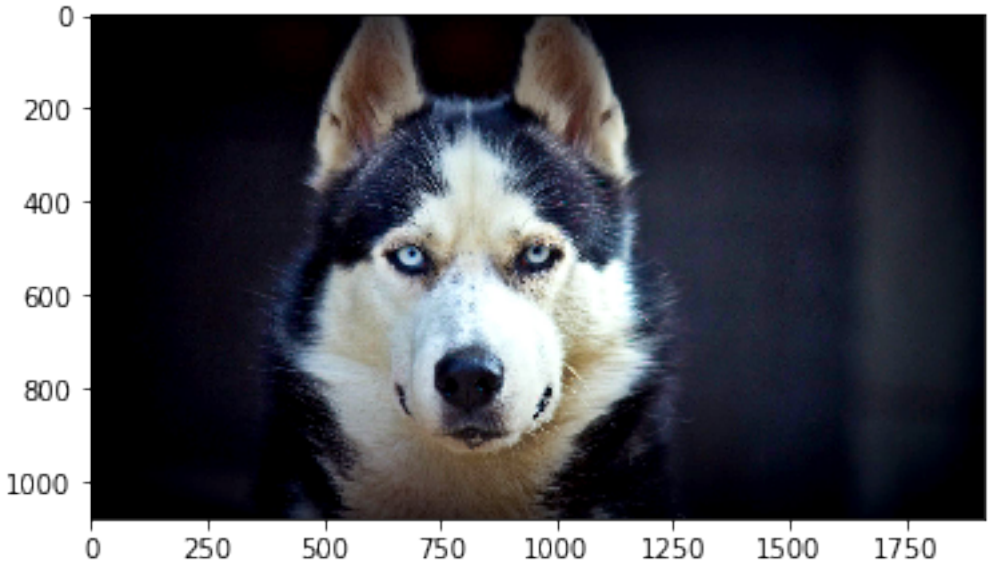
[39]:
```
# Load my sample pictures. - once only

# import requests

# # husky
# url = "https://www.elsetge.cat/myimg/f/63-633907_huskies-wolves.jpg"
# samplepic = requests.get(url )
# open( 'images/mysamplepic_dog1.jpg', 'wb').write(samplepic.content)
# #
# url = "https://images.all-free-download.com/images/graphiclarge/
↪cute_dog_03_hd_pictures_168930.jpg"
# samplepic = requests.get(url )
```

```
# open( 'images/mysamplepic_dog2.jpg', 'wb').write(samplepic.content)
# # Rottweiler
# url = "https://www.hdwallpaper.nu/wp-content/uploads/2015/12/
 ↪rottweiler_wallpaper_1610.jpg"
# samplepic = requests.get(url )
# open( 'images/mysamplepic_dog3.jpg', 'wb').write(samplepic.content)


# # human1
# url = "https://t1.thpservices.com/previewimage/gallil/
 ↪72b39b1a593dcc39c83d971123198025/hez-1629865.jpg"
# samplepic = requests.get(url )
# open( 'images/mysamplepic_human1.jpg', 'wb').write(samplepic.content)
# # human2
# url = "https://www.ajc.com/rf/image_large/Pub/p9/AJC/2018/02/13/Images/
 ↪newsEngin.20864894_RUBY-DEE.jpg"
# samplepic = requests.get(url )
# open( 'images/mysamplepic_human2.jpg', 'wb').write(samplepic.content)


# # otter
# url = "https://blog.sigmaphoto.com/wp-content/uploads/2019/09/
 ↪David-FitzSimmons-Asian-Small-Clawed-Otter-2-CLE-Zoo-100-pct-Detail.jpg"
# samplepic = requests.get(url )
# open( 'images/mysamplepic_other1.jpg', 'wb').write(samplepic.content)
# # Jedi
# url = "https://pngriver.com/wp-content/uploads/2018/04/
 ↪Download-Star-Wars-PNG-Pic-For-Designing-Projects.png"
# samplepic = requests.get(url )
# open( 'images/mysamplepic_other2.jpg', 'wb').write(samplepic.content)
# # mask
# url = "https://www.elsetge.cat/myimg/f/
 ↪8-83418_popular-dual-monitors-backgrounds-star-wars.jpg"
# samplepic = requests.get(url )
# open( 'images/mysamplepic_other3.jpg', 'wb').write(samplepic.content)
```

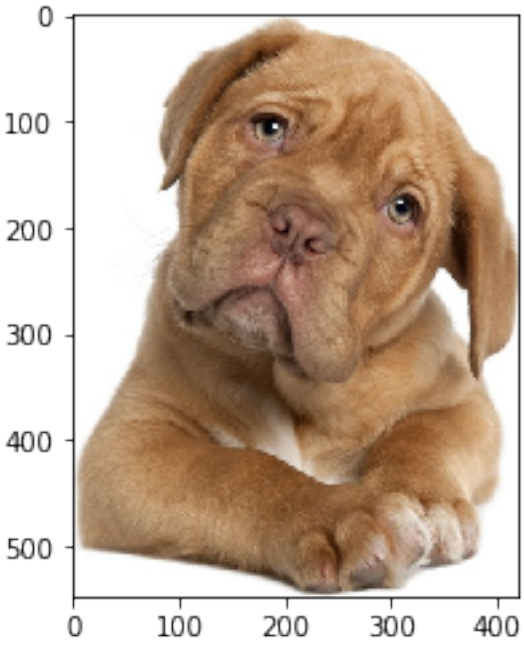
[40]:
```
# Testing
# dog_breed_identification("images/mysamplepic_dog1.jpg")
```

[41]:
```
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.

# Identify dogs
dog_breed_identification("images/mysamplepic_dog1.jpg")
dog_breed_identification("images/mysamplepic_dog2.jpg")
```

```
dog_breed_identification("images/mysamplepic_dog3.jpg")
dog_breed_identification("images/Curly-coated_retriever_03896.jpg")

# Identify human
dog_breed_identification("images/mysamplepic_human1.jpg")
dog_breed_identification("images/mysamplepic_human2.jpg")
dog_breed_identification("images/Brittany_02625.jpg")

# Identify other
dog_breed_identification("images/mysamplepic_other1.jpg")
dog_breed_identification("images/mysamplepic_other2.jpg")
dog_breed_identification("images/mysamplepic_other3.jpg")
```

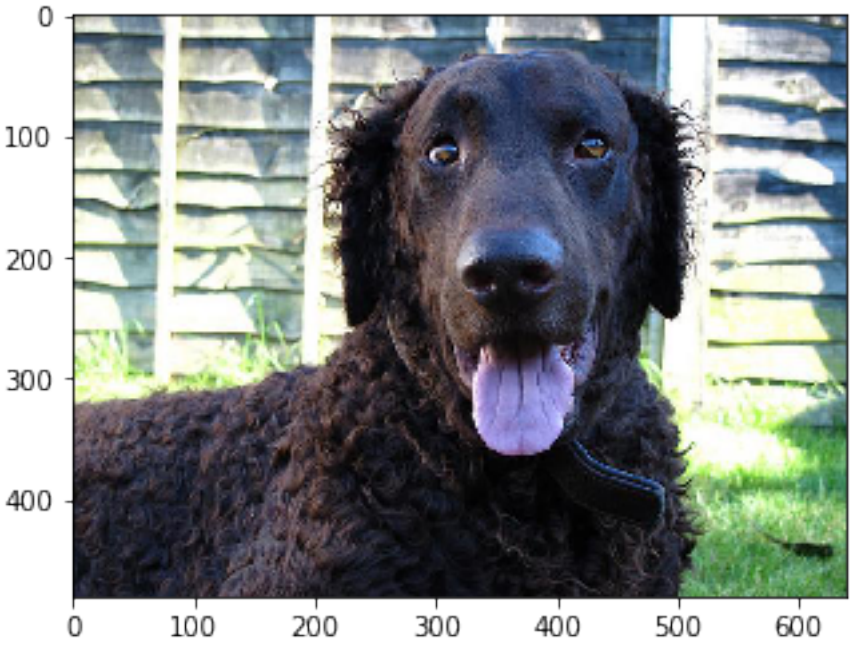



```
A dog is detected and its breed is ages/train/005.Alaskan_malamute
```




```
A dog is detected and its breed is ages/train/060.Dogue_de_bordeaux
```

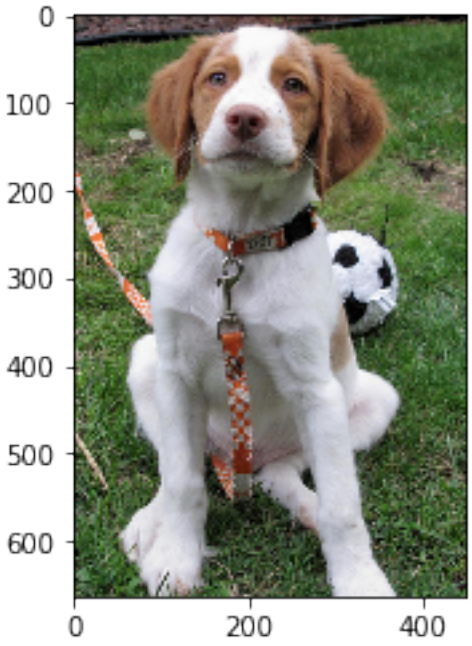

```
A dog is detected and its breed is ages/train/060.Dogue_de_bordeaux
```




A dog is detected and its breed is ages/train/055.Curly-coated_retriever

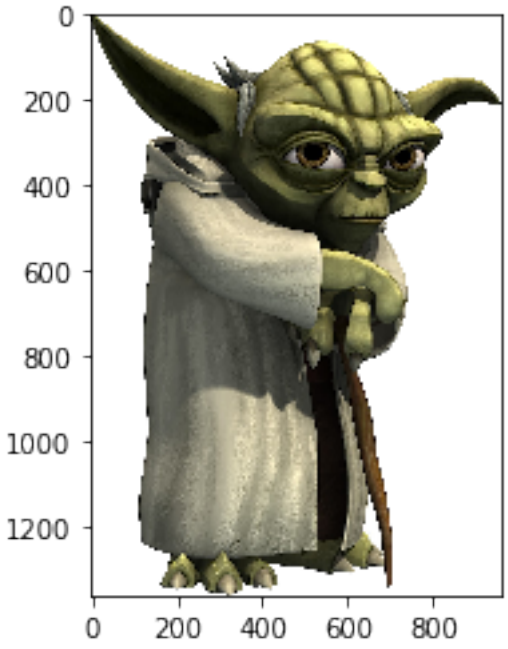

A human is detected, but if this was a dog, its breed would be ages/train/128.Smooth_fox_terrier


A human is detected, but if this was a dog, its breed would be ages/train/082.Havanese

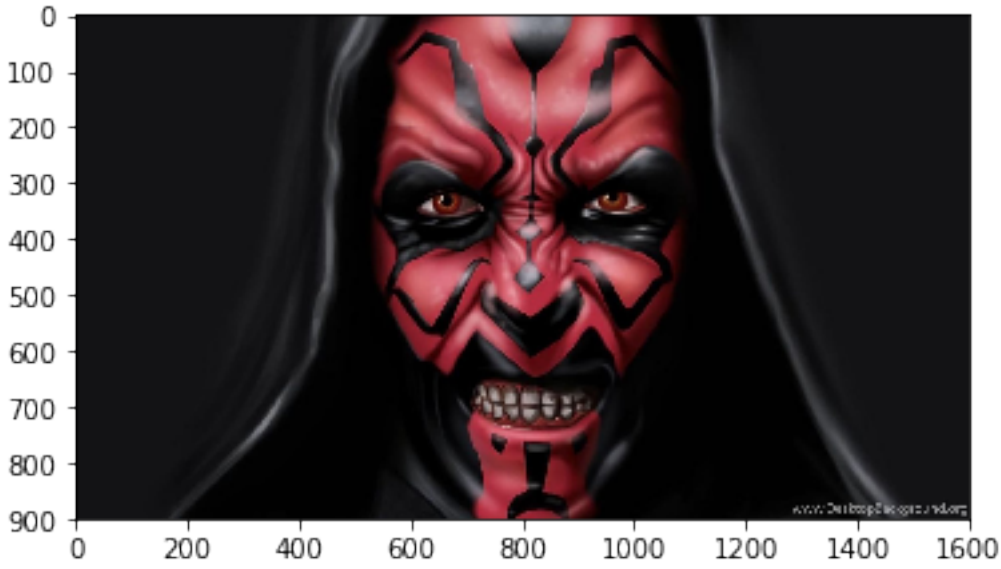



A dog is detected and its breed is ages/train/037.Brittany


```
Um, it can't identify as a dog or human. Try another image.
```




```
Um, it can't identify as a dog or human. Try another image.
```




```
A human is detected, but if this was a dog, its breed would be
ages/train/074.Giant_schnauzer
```

[ ]:
```
# Detection accuracy
# 100% !
# Well done, CNN!
```