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

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

```
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.
```

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`.

```
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

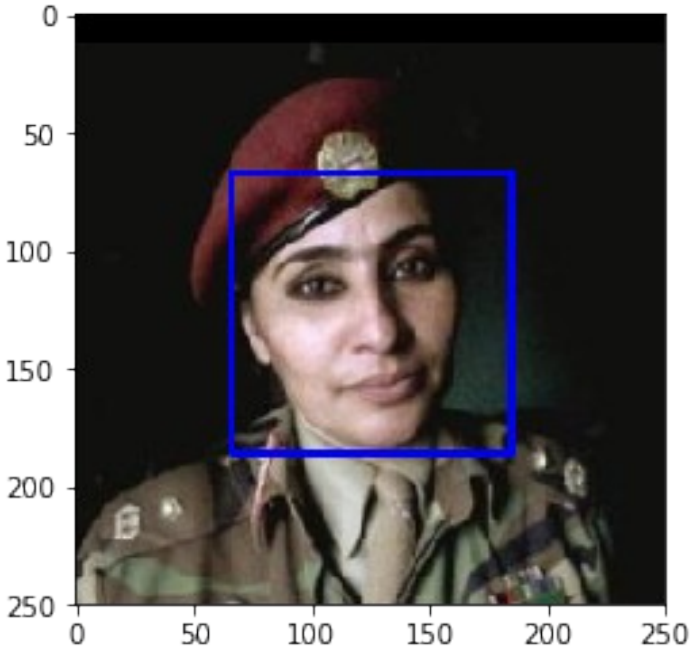
We use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images. OpenCV provides many pre-trained face detectors, stored as XML files on github. We have downloaded one of these detectors and stored it in the `haarcascades` directory.

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

```
import cv2
import matplotlib.pyplot as plt
%matplotlib inline

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

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

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

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

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

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

# display the image, along with bounding box
```

```
plt.imshow(cv_rgb)
plt.show()
```

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



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

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

Write a Human Face Detector

We can use this procedure to write a function that returns `True` if a human face is detected in an image and `False` otherwise. This function, aptly named `face_detector`, takes a string-valued file path to an image as input and appears in the code block below.

```
# 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:

percentage of the detected human faces from the the human files : 100%

percentage of the detected human faces from the the dog files : 11%

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

## TODO: Test the performance of the face_detector algorithm
## on the images in human_files_short and dog_files_short.
count_human = 0
count_dog = 0
for i,j in zip(human_files_short,dog_files_short):
    if(face_detector(i)):
        count_human+=1
    if(face_detector(j)):
        count_dog+=1
print('percentage of the detected human faces from the the human files
: ',100*(count_human/100),'%')
print('percentage of the detected human faces from the the dog files :
',100*(count_dog/100),'%')
```

```
percentage of the detected human faces from the the human files :
100.0 %
percentage of the detected human faces from the the dog files :  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 might choose a face detector that is trained with augemented images with different positions and angles.

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.

```
## (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.

```
from keras.applications.resnet50 import ResNet50

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

```
Downloading data from https://github.com/fchollet/deep-learning-
models/releases/download/v0.2/
resnet50_weights_tf_dim_ordering_tf_kernels.h5
102858752/102853048 [==============================] - 2s 0us/step
```

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!

```
from keras.preprocessing import image
from tqdm import tqdm

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

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

Making Predictions with ResNet-50

Getting the 4D tensor ready for ResNet-50, and for any other pre-trained model in Keras, requires some additional processing. First, the RGB image is converted to BGR by reordering the channels. All pre-trained models have the additional normalization step that the mean pixel (expressed in RGB as $[103.939, 116.779, 123.68]$ and calculated from all pixels in all images in ImageNet) must be subtracted from every pixel in each image. This is implemented in the imported function `preprocess_input`. If you're curious, you can check the code for `preprocess_input` here.

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.

```
from keras.applications.resnet50 import preprocess_input,
decode_predictions

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

Write a Dog Detector

While looking at the dictionary, 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).

```
### returns "True" if a dog is detected in the image stored at
img_path
def dog_detector(img_path):
    prediction = ResNet50_predict_labels(img_path)
    return ((prediction <= 268) & (prediction >= 151))
```

(IMPLEMENTATION) Assess the Dog Detector

Question 3: Use the code cell below to test the performance of your `dog_detector` function.

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

Answer:

percentage of the images in human_files_short have a detected dog : 0.0 %

percentage of the images in dog_files_short have a detected dog : 100.0 %

```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.
count_human = 0
```

```
count_dog = 0
for i,j in zip(human_files_short,dog_files_short):
    if(dog_detector(i)):
        count_human+=1
    if(dog_detector(j)):
        count_dog+=1
print('percentage of the images in human_files_short have a detected
dog : ',100*(count_human/100),'%')
print('percentage of the images in dog_files_short have a detected dog
: ',100*(count_dog/100),'%')
```

```
percentage of the images in human_files_short have a detected dog :
0.0 %
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

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

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

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.

```
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 [01:13<00:00, 91.47it/s]
100%|██████████| 835/835 [00:08<00:00, 102.09it/s]
100%|██████████| 836/836 [00:08<00:00, 102.65it/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:

I used the 32-64-128 to capture more patterns as much as possible from the original photo, And I used the 'relu' activation function because it gave better accuracy than other

activation functions as 'tanh' , I used global average pooling in the end to reduce number of parameters , Then used dense layer with 100 nodes to connect the patterns , In the end I added a dense layer with the number of the existent classes (133) with a softmax.

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

model = Sequential()

### TODO: Define your architecture.
model.add(Conv2D(filters=32, kernel_size=2, padding='same',
activation='relu',
                         input_shape=(224, 224, 3)))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=64, kernel_size=2, padding='same',
activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=128, kernel_size=2, padding='same',
activation='relu'))
model.add(GlobalAveragePooling2D())
model.add(Dropout(0.2))
#model.add(Flatten())
model.add(Dense(100, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(133, activation='softmax'))


model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 224, 224, 32)      416
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 112, 112, 32)      0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 112, 112, 64)      8256
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 56, 56, 64)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 56, 56, 128)       32896
_________________________________________________________________
global_average_pooling2d_1 ( (None, 128)               0
_________________________________________________________________
dropout_1 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 100)               12900
_________________________________________________________________
dropout_2 (Dropout)          (None, 100)               0
```

```
_________________________________________________________________
dense_2 (Dense)              (None, 133)               13433     
=================================================================
Total params: 67,901
Trainable params: 67,901
Non-trainable params: 0
_________________________________________________________________
```

Compile the Model

```
model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
```

(IMPLEMENTATION) Train the Model

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

You are welcome to augment the training data, but this is not a requirement.

```
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: 4.8847 -
acc: 0.0090Epoch 00001: val_loss improved from inf to 4.87213, saving
model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 34s 5ms/step - loss:
4.8846 - acc: 0.0091 - val_loss: 4.8721 - val_acc: 0.0108
Epoch 2/10
6660/6680 [============================>.] - ETA: 0s - loss: 4.8716 -
acc: 0.0104Epoch 00002: val_loss improved from 4.87213 to 4.86545,
saving model to saved_models/weights.best.from_scratch.hdf5
```

```
6680/6680 [==============================] - 33s 5ms/step - loss:
4.8719 - acc: 0.0103 - val_loss: 4.8654 - val_acc: 0.0132
Epoch 3/10
6660/6680 [============================>.] - ETA: 0s - loss: 4.8483 -
acc: 0.0135Epoch 00003: val_loss improved from 4.86545 to 4.81643,
saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 33s 5ms/step - loss:
4.8483 - acc: 0.0136 - val_loss: 4.8164 - val_acc: 0.0180
Epoch 4/10
6660/6680 [============================>.] - ETA: 0s - loss: 4.7819 -
acc: 0.0180Epoch 00004: val_loss improved from 4.81643 to 4.75950,
saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 33s 5ms/step - loss:
4.7819 - acc: 0.0180 - val_loss: 4.7595 - val_acc: 0.0228
Epoch 5/10
6660/6680 [============================>.] - ETA: 0s - loss: 4.7191 -
acc: 0.0186Epoch 00005: val_loss improved from 4.75950 to 4.70645,
saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 33s 5ms/step - loss:
4.7189 - acc: 0.0186 - val_loss: 4.7064 - val_acc: 0.0228
Epoch 6/10
6660/6680 [============================>.] - ETA: 0s - loss: 4.6798 -
acc: 0.0230Epoch 00006: val_loss improved from 4.70645 to 4.68975,
saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 33s 5ms/step - loss:
4.6797 - acc: 0.0231 - val_loss: 4.6898 - val_acc: 0.0240
Epoch 7/10
6660/6680 [============================>.] - ETA: 0s - loss: 4.6376 -
acc: 0.0248Epoch 00007: val_loss improved from 4.68975 to 4.64613,
saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 33s 5ms/step - loss:
4.6369 - acc: 0.0250 - val_loss: 4.6461 - val_acc: 0.0323
Epoch 8/10
6660/6680 [============================>.] - ETA: 0s - loss: 4.5982 -
acc: 0.0294Epoch 00008: val_loss improved from 4.64613 to 4.64502,
saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 33s 5ms/step - loss:
4.5981 - acc: 0.0295 - val_loss: 4.6450 - val_acc: 0.0323
Epoch 9/10
6660/6680 [============================>.] - ETA: 0s - loss: 4.5523 -
acc: 0.0366Epoch 00009: val_loss improved from 4.64502 to 4.59138,
saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 33s 5ms/step - loss:
4.5523 - acc: 0.0367 - val_loss: 4.5914 - val_acc: 0.0347
Epoch 10/10
6660/6680 [============================>.] - ETA: 0s - loss: 4.5055 -
acc: 0.0350Epoch 00010: val_loss improved from 4.59138 to 4.52847,
saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 33s 5ms/step - loss:
4.5059 - acc: 0.0350 - val_loss: 4.5285 - val_acc: 0.0467
```

```
<keras.callbacks.History at 0x7f336e9a26a0>
```

Load the Model with the Best Validation 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%.

```
# 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: 4.3062%
```

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

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

Model Architecture

The model uses the the pre-trained VGG-16 model as a fixed feature extractor, where the last convolutional output of VGG-16 is fed as input to our model. We only add a global average pooling layer and a fully connected layer, where the latter contains one node for each dog category and is equipped with a softmax.

```
VGG16_model = Sequential()
VGG16_model.add(GlobalAveragePooling2D(input_shape=train_VGG16.shape[1
:]))
VGG16_model.add(Dense(133, activation='softmax'))
```

```
VGG16_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
global_average_pooling2d_2 ( (None, 512)               0
_________________________________________________________________
dense_3 (Dense)              (None, 133)               68229
=================================================================
Total params: 68,229
Trainable params: 68,229
Non-trainable params: 0
_________________________________________________________________
```

Compile the Model

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

Train the Model

```
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
6600/6680 [============================>.] - ETA: 0s - loss: 12.3194 -
acc: 0.1164Epoch 00001: val_loss improved from inf to 10.74813, saving
model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 282us/step - loss:
12.3025 - acc: 0.1174 - val_loss: 10.7481 - val_acc: 0.1940
Epoch 2/20
6660/6680 [============================>.] - ETA: 0s - loss: 9.8909 -
acc: 0.2799Epoch 00002: val_loss improved from 10.74813 to 9.73336,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 239us/step - loss:
9.8887 - acc: 0.2801 - val_loss: 9.7334 - val_acc: 0.2790
Epoch 3/20
6560/6680 [============================>.] - ETA: 0s - loss: 9.0543 -
acc: 0.3537Epoch 00003: val_loss improved from 9.73336 to 9.07384,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 253us/step - loss:
9.0500 - acc: 0.3542 - val_loss: 9.0738 - val_acc: 0.3581
Epoch 4/20
6620/6680 [============================>.] - ETA: 0s - loss: 8.6981 -
acc: 0.4042Epoch 00004: val_loss improved from 9.07384 to 8.92764,
```

```
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 250us/step - loss:
8.6828 - acc: 0.4051 - val_loss: 8.9276 - val_acc: 0.3581
Epoch 5/20
6560/6680 [============================>.] - ETA: 0s - loss: 8.5068 -
acc: 0.4256Epoch 00005: val_loss improved from 8.92764 to 8.85487,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 251us/step - loss:
8.5178 - acc: 0.4247 - val_loss: 8.8549 - val_acc: 0.3725
Epoch 6/20
6500/6680 [============================>.] - ETA: 0s - loss: 8.3539 -
acc: 0.4458Epoch 00006: val_loss improved from 8.85487 to 8.76129,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 252us/step - loss:
8.3415 - acc: 0.4470 - val_loss: 8.7613 - val_acc: 0.3832
Epoch 7/20
6480/6680 [============================>.] - ETA: 0s - loss: 8.2680 -
acc: 0.4596Epoch 00007: val_loss did not improve
6680/6680 [==============================] - 2s 253us/step - loss:
8.2713 - acc: 0.4590 - val_loss: 8.8089 - val_acc: 0.3844
Epoch 8/20
6540/6680 [============================>.] - ETA: 0s - loss: 8.1918 -
acc: 0.4696Epoch 00008: val_loss improved from 8.76129 to 8.67255,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 252us/step - loss:
8.1801 - acc: 0.4705 - val_loss: 8.6725 - val_acc: 0.3856
Epoch 9/20
6660/6680 [============================>.] - ETA: 0s - loss: 8.1390 -
acc: 0.4778Epoch 00009: val_loss did not improve
6680/6680 [==============================] - 2s 253us/step - loss:
8.1388 - acc: 0.4778 - val_loss: 8.7052 - val_acc: 0.3844
Epoch 10/20
6480/6680 [============================>.] - ETA: 0s - loss: 8.0875 -
acc: 0.4819Epoch 00010: val_loss improved from 8.67255 to 8.66342,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 254us/step - loss:
8.0958 - acc: 0.4816 - val_loss: 8.6634 - val_acc: 0.3880
Epoch 11/20
6660/6680 [============================>.] - ETA: 0s - loss: 8.0486 -
acc: 0.4884Epoch 00011: val_loss improved from 8.66342 to 8.61299,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 249us/step - loss:
8.0443 - acc: 0.4885 - val_loss: 8.6130 - val_acc: 0.4048
Epoch 12/20
6600/6680 [============================>.] - ETA: 0s - loss: 8.0015 -
acc: 0.4912Epoch 00012: val_loss improved from 8.61299 to 8.50809,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 250us/step - loss:
8.0032 - acc: 0.4909 - val_loss: 8.5081 - val_acc: 0.4024
Epoch 13/20
```

```
6620/6680 [============================>.] - ETA: 0s - loss: 7.9339 -
acc: 0.4958Epoch 00013: val_loss improved from 8.50809 to 8.39418,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 249us/step - loss:
7.9309 - acc: 0.4960 - val_loss: 8.3942 - val_acc: 0.4048
Epoch 14/20
6560/6680 [============================>.] - ETA: 0s - loss: 7.7822 -
acc: 0.5078Epoch 00014: val_loss did not improve
6680/6680 [==============================] - 2s 245us/step - loss:
7.7922 - acc: 0.5072 - val_loss: 8.4426 - val_acc: 0.4096
Epoch 15/20
6560/6680 [============================>.] - ETA: 0s - loss: 7.7519 -
acc: 0.5114Epoch 00015: val_loss improved from 8.39418 to 8.37347,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 247us/step - loss:
7.7698 - acc: 0.5102 - val_loss: 8.3735 - val_acc: 0.4192
Epoch 16/20
6520/6680 [============================>.] - ETA: 0s - loss: 7.7480 -
acc: 0.5129Epoch 00016: val_loss improved from 8.37347 to 8.37046,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 247us/step - loss:
7.7479 - acc: 0.5129 - val_loss: 8.3705 - val_acc: 0.4108
Epoch 17/20
6440/6680 [===========================>..] - ETA: 0s - loss: 7.7215 -
acc: 0.5126Epoch 00017: val_loss improved from 8.37046 to 8.29991,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 247us/step - loss:
7.7082 - acc: 0.5135 - val_loss: 8.2999 - val_acc: 0.4156
Epoch 18/20
6480/6680 [============================>.] - ETA: 0s - loss: 7.6819 -
acc: 0.5167Epoch 00018: val_loss improved from 8.29991 to 8.26222,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 248us/step - loss:
7.6787 - acc: 0.5171 - val_loss: 8.2622 - val_acc: 0.4251
Epoch 19/20
6620/6680 [============================>.] - ETA: 0s - loss: 7.6579 -
acc: 0.5201Epoch 00019: val_loss did not improve
6680/6680 [==============================] - 2s 244us/step - loss:
7.6664 - acc: 0.5196 - val_loss: 8.2805 - val_acc: 0.4287
Epoch 20/20
6500/6680 [============================>.] - ETA: 0s - loss: 7.6073 -
acc: 0.5188Epoch 00020: val_loss improved from 8.26222 to 8.22576,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 248us/step - loss:
7.5981 - acc: 0.5195 - val_loss: 8.2258 - val_acc: 0.4311

<keras.callbacks.History at 0x7f335d6ee748>
```

Load the Model with the Best Validation Loss

```
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.

```
# 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: 41.5072%
```

Predict Dog Breed with the Model

```
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.

(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']
```

```
### TODO: Obtain bottleneck features from another pre-trained CNN.
bottleneck_features =
np.load('bottleneck_features/DogResnet50Data.npz')
train_ResNet_50 = bottleneck_features['train']
valid_ResNet_50 = bottleneck_features['valid']
test_ResNet_50 = bottleneck_features['test']
```

(IMPLEMENTATION) Model Architecture

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

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

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

Answer:

I made a global average pooling to decrease number of parameters from the ResNet-50 model , Then made a dense layer with 150 nodes with 'relu' activation function to get higher accuracy and a dropout layer to also decrease overfitting in the model.

```
### TODO: Define your architecture.
resnet50_model = Sequential()
resnet50_model.add(GlobalAveragePooling2D(input_shape=train_ResNet_50.
shape[1:]))
resnet50_model.add(Dense(150,activation='relu'))
resnet50_model.add(Dropout(0.5))
resnet50_model.add(Dense(133, activation='softmax'))

resnet50_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
global_average_pooling2d_12  (None, 2048)              0
_________________________________________________________________
dense_22 (Dense)             (None, 150)               307350
_________________________________________________________________
dropout_12 (Dropout)         (None, 150)               0
_________________________________________________________________
dense_23 (Dense)             (None, 133)               20083
=================================================================
Total params: 327,433
Trainable params: 327,433
Non-trainable params: 0
_________________________________________________________________
```

(IMPLEMENTATION) Compile the Model

```
### TODO: Compile the model.
resnet50_model.compile(loss='categorical_crossentropy',
optimizer='rmsprop', metrics=['accuracy'])
```

(IMPLEMENTATION) Train the Model

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

You are welcome to augment the training data, but this is not a requirement.

```
### TODO: Train the model.
checkpointer =
ModelCheckpoint(filepath='saved_models/weights.best.resnet50.hdf5',
                               verbose=1, save_best_only=True)

resnet50_model.fit(train_ResNet_50, train_targets,
          validation_data=(valid_ResNet_50, valid_targets),
          epochs=20, batch_size=50, callbacks=[checkpointer],
verbose=1)
```

```
Train on 6680 samples, validate on 835 samples
Epoch 1/20
6350/6680 [===========================>..] - ETA: 0s - loss: 3.8693 -
acc: 0.1660Epoch 00001: val_loss improved from inf to 2.08164, saving
model to saved_models/weights.best.resnet50.hdf5
6680/6680 [==============================] - 2s 262us/step - loss:
3.8247 - acc: 0.1717 - val_loss: 2.0816 - val_acc: 0.5653
Epoch 2/20
6350/6680 [===========================>..] - ETA: 0s - loss: 2.1086 -
acc: 0.4539Epoch 00002: val_loss improved from 2.08164 to 1.15489,
saving model to saved_models/weights.best.resnet50.hdf5
6680/6680 [==============================] - 1s 131us/step - loss:
2.0901 - acc: 0.4588 - val_loss: 1.1549 - val_acc: 0.7162
```

```
Epoch 3/20
6300/6680 [===========================>..] - ETA: 0s - loss: 1.4502 -
acc: 0.5803Epoch 00003: val_loss improved from 1.15489 to 0.84837,
saving model to saved_models/weights.best.resnet50.hdf5
6680/6680 [==============================] - 1s 130us/step - loss:
1.4459 - acc: 0.5828 - val_loss: 0.8484 - val_acc: 0.7437
Epoch 4/20
6350/6680 [===========================>..] - ETA: 0s - loss: 1.1242 -
acc: 0.6661Epoch 00004: val_loss improved from 0.84837 to 0.72960,
saving model to saved_models/weights.best.resnet50.hdf5
6680/6680 [==============================] - 1s 130us/step - loss:
1.1170 - acc: 0.6684 - val_loss: 0.7296 - val_acc: 0.7760
Epoch 5/20
6350/6680 [===========================>..] - ETA: 0s - loss: 0.9307 -
acc: 0.7227Epoch 00005: val_loss improved from 0.72960 to 0.69075,
saving model to saved_models/weights.best.resnet50.hdf5
6680/6680 [==============================] - 1s 132us/step - loss:
0.9298 - acc: 0.7216 - val_loss: 0.6907 - val_acc: 0.7856
Epoch 6/20
6300/6680 [===========================>..] - ETA: 0s - loss: 0.8383 -
acc: 0.7408Epoch 00006: val_loss improved from 0.69075 to 0.64383,
saving model to saved_models/weights.best.resnet50.hdf5
6680/6680 [==============================] - 1s 132us/step - loss:
0.8367 - acc: 0.7404 - val_loss: 0.6438 - val_acc: 0.7976
Epoch 7/20
6350/6680 [===========================>..] - ETA: 0s - loss: 0.7284 -
acc: 0.7712Epoch 00007: val_loss did not improve
6680/6680 [==============================] - 1s 129us/step - loss:
0.7331 - acc: 0.7702 - val_loss: 0.6600 - val_acc: 0.7928
Epoch 8/20
6300/6680 [===========================>..] - ETA: 0s - loss: 0.6519 -
acc: 0.7935Epoch 00008: val_loss improved from 0.64383 to 0.60920,
saving model to saved_models/weights.best.resnet50.hdf5
6680/6680 [==============================] - 1s 131us/step - loss:
0.6541 - acc: 0.7924 - val_loss: 0.6092 - val_acc: 0.8048
Epoch 9/20
6350/6680 [===========================>..] - ETA: 0s - loss: 0.6056 -
acc: 0.8022Epoch 00009: val_loss improved from 0.60920 to 0.59185,
saving model to saved_models/weights.best.resnet50.hdf5
6680/6680 [==============================] - 1s 130us/step - loss:
0.6068 - acc: 0.8019 - val_loss: 0.5919 - val_acc: 0.8156
Epoch 10/20
6250/6680 [===========================>..] - ETA: 0s - loss: 0.5494 -
acc: 0.8168Epoch 00010: val_loss did not improve
6680/6680 [==============================] - 1s 130us/step - loss:
0.5522 - acc: 0.8169 - val_loss: 0.5928 - val_acc: 0.8144
Epoch 11/20
6300/6680 [===========================>..] - ETA: 0s - loss: 0.5152 -
acc: 0.8302Epoch 00011: val_loss improved from 0.59185 to 0.58675,
saving model to saved_models/weights.best.resnet50.hdf5
```

```
6680/6680 [==============================] - 1s 132us/step - loss:
0.5202 - acc: 0.8287 - val_loss: 0.5868 - val_acc: 0.8192
Epoch 12/20
6350/6680 [===========================>..] - ETA: 0s - loss: 0.4795 -
acc: 0.8378Epoch 00012: val_loss did not improve
6680/6680 [==============================] - 1s 128us/step - loss:
0.4767 - acc: 0.8380 - val_loss: 0.5944 - val_acc: 0.8240
Epoch 13/20
6300/6680 [===========================>..] - ETA: 0s - loss: 0.4407 -
acc: 0.8511Epoch 00013: val_loss did not improve
6680/6680 [==============================] - 1s 128us/step - loss:
0.4409 - acc: 0.8509 - val_loss: 0.5930 - val_acc: 0.8240
Epoch 14/20
6350/6680 [===========================>..] - ETA: 0s - loss: 0.4362 -
acc: 0.8529Epoch 00014: val_loss did not improve
6680/6680 [==============================] - 1s 128us/step - loss:
0.4367 - acc: 0.8534 - val_loss: 0.6016 - val_acc: 0.8240
Epoch 15/20
6350/6680 [===========================>..] - ETA: 0s - loss: 0.3963 -
acc: 0.8685Epoch 00015: val_loss improved from 0.58675 to 0.57301,
saving model to saved_models/weights.best.resnet50.hdf5
6680/6680 [==============================] - 1s 130us/step - loss:
0.3955 - acc: 0.8689 - val_loss: 0.5730 - val_acc: 0.8371
Epoch 16/20
6300/6680 [===========================>..] - ETA: 0s - loss: 0.3789 -
acc: 0.8746Epoch 00016: val_loss did not improve
6680/6680 [==============================] - 1s 128us/step - loss:
0.3818 - acc: 0.8738 - val_loss: 0.6154 - val_acc: 0.8240
Epoch 17/20
6350/6680 [===========================>..] - ETA: 0s - loss: 0.3570 -
acc: 0.8841Epoch 00017: val_loss improved from 0.57301 to 0.57230,
saving model to saved_models/weights.best.resnet50.hdf5
6680/6680 [==============================] - 1s 132us/step - loss:
0.3581 - acc: 0.8832 - val_loss: 0.5723 - val_acc: 0.8323
Epoch 18/20
6350/6680 [===========================>..] - ETA: 0s - loss: 0.3471 -
acc: 0.8805Epoch 00018: val_loss did not improve
6680/6680 [==============================] - 1s 127us/step - loss:
0.3499 - acc: 0.8795 - val_loss: 0.6036 - val_acc: 0.8383
Epoch 19/20
6350/6680 [===========================>..] - ETA: 0s - loss: 0.3234 -
acc: 0.8890Epoch 00019: val_loss did not improve
6680/6680 [==============================] - 1s 128us/step - loss:
0.3279 - acc: 0.8880 - val_loss: 0.6249 - val_acc: 0.8275
Epoch 20/20
6350/6680 [===========================>..] - ETA: 0s - loss: 0.3102 -
acc: 0.8984Epoch 00020: val_loss did not improve
6680/6680 [==============================] - 1s 127us/step - loss:
0.3099 - acc: 0.8981 - val_loss: 0.6074 - val_acc: 0.8275
```

```
<keras.callbacks.History at 0x7f31db60cf98>
```

(IMPLEMENTATION) Load the Model with the Best Validation Loss

```
### TODO: Load the model weights with the best validation loss.
resnet50_model.load_weights('saved_models/weights.best.resnet50.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%.

```
### TODO: Calculate classification accuracy on the test dataset.
# get index of predicted dog breed for each image in test set
resnet50_predictions =
[np.argmax(resnet50_model.predict(np.expand_dims(feature, axis=0)))
for feature in test_ResNet_50]

# report test accuracy
test_accuracy =
100*np.sum(np.array(resnet50_predictions)==np.argmax(test_targets,
axis=1))/len(resnet50_predictions)
print('Test accuracy: %.4f%%' % test_accuracy)
```

```
Test accuracy: 82.8947%
```

(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`.

```
### 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 resnet50_predict_breed(img_path):
    # extract bottleneck features
    bottleneck_feature = extract_Resnet50(path_to_tensor(img_path))
    # obtain predicted vector
    predicted_vector = resnet50_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

```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.
def dog_human_preds(img_path):
    if(face_detector(img_path)):
        print('That is a human')
        img = cv2.imread(img_path)
        # convert BGR image to grayscale

        # 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()
        bottleneck_feature =
extract_Resnet50(path_to_tensor(img_path))
        # obtain predicted vector
        predicted_vector = resnet50_model.predict(bottleneck_feature)
```

```
        # return dog breed that is predicted by the model
        return dog_names[np.argmax(predicted_vector)]

    if(dog_detector(img_path)):
        print('That is a dog')
        img = cv2.imread(img_path)
        # convert BGR image to grayscale

        # 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()
        bottleneck_feature =
extract_Resnet50(path_to_tensor(img_path))
        # obtain predicted vector
        predicted_vector = resnet50_model.predict(bottleneck_feature)
        # return dog breed that is predicted by the model
        return dog_names[np.argmax(predicted_vector)]
    img = cv2.imread(img_path)
        # convert BGR image to grayscale

        # 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()

    return 'invalid input,This neither a human nor a dog'
```

Step 7: Test Your Algorithm

In this section, you will take your new algorithm for a spin! What kind of dog does the algorithm think that **you** look like? If you have a dog, does it predict your dog's breed accurately? If you have a cat, does it mistakenly think that your cat is a dog?

(IMPLEMENTATION) Test Your Algorithm on Sample Images!

Test your algorithm at least six images on your computer. Feel free to use any images you like. Use at least two human and two dog images.

Question 6: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

Answer:

The output is better than expected,

- The accuracy can be improved by adding more layers .
- The algorthim can provide a picture of the most resembling breed when the input is a human .
- A face detector based on deep learning with better accuracy could be added .

```
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.
dog_human_preds('dog_2.jpg')
```

```
that is a dog
```


```
'ages/train/050.Chinese_shar-pei'
```