Dog Breed Classification Using CNN

Project Overview

Being a Data Scientist, Deep Learning is one of the must-to-have knowledge and skills. This project is for the Udacity Data Scientist program to develop an algorithm and create a multi-class CNN (Convolutional Neural Networks) with the two tasks as:

1. Detect the object on a provided image as a dog, a human, or neither.
2. If the object was detected as a dog, based on the given classes of the world-wide dog breeds, the model would be able to identify the dog breed.

Why Convolutional Neural Networks model?

CNN is one of the most Artificial Neural Networks as regularized versions of MLP (Multilayer Perceptrons). So, A standard CNN is comprised of convolutional layers, sampling layers and then followed by fully connected layers (i.e. each neuron in one layer is connected to all neurons in the next layer).

CNN uses relatively little pre-processing compared to other image classification algorithms. It takes advantage of the hierarchical pattern in data and assembles more complex patterns using smaller and simpler patterns. With the help of CNN, we can use the large amount of data more effectively and accurately.

Therefore, CNN is typically applied for Image Classification from an input image with a single object and to an output as a class label from a list of object categories.

There are a few packages available to build the CNN model such as PyTorch, Keras, etc. In this project, we use Keras library for dog breed identification.

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

Metric

Based on our tasks, the CNN model should be able to classify the object on the image correctly, not only identifying if dog or human, but also identifying the dog breed. For the given images for testing, the more dog breed corrected classified, the better the model is.

What model is the best from training process?

In the project, we split the input images into training, validating and testing data sets. For each data set, the corresponding target data was also provided, which includes the onehot-encoded classification labels. With the defined architecture of the CNN, we will use the training and validating data (including the images and the given labels) to train the model under number of epochs and the best model will be saved.

Keras uses metric function to evaluate the performance of the model. The metric function returns a single tensor value as the mean of the output array across all data points. The most popular metrics used in CNN are "accuracy", "binary_accuracy", "categorical_accuracy", "sparse_categorical_accuracy", etc.

Accuracy is the ratio of number of correct predictions to the total number of input samples. However, it works well only if there are equal number of samples belonging to each class.

$$Accuracy = \frac{Number\ of\ Correct\ predictions}{Total\ number\ of\ predictions\ made}$$

In this project, there are total 133 dog breed as class labels. Based on the distribution of training/validation/testing selected (details see section "*Step 0: Import Datasets"*), the classes were approximately evenly distributed, except a couple of classes. Therefore, we should be able to use "accuracy" metric in training.

Once the model is trained, we apply it on the testing images for breed prediction. Same methodology as metric function used in model training, for each testing image, we will compare the model predicted breed class against the target class label, and calculate the ratio of number of images with the dog breed was accurately identified over the total number of images in the testing files. This ratio, named as "Test Accuracy", is set as the metric to measure the CNN model performance through the whole project.

Library Preparation

The following package/library will be used in this project for data loading, data processing, data/image visualization, model development, etc.

```
import panda as pd
import numpy as np
from glob import glob
import random
import cv2
import matplotlib.pyplot as plt
%matplotlib inline
from tqdm import tqdm

from sklearn.datasets import load_files
from keras.preprocessing import image
from keras.utils import np_utils
from keras.applications.resnet50 import preprocess_input, decode_predictions
from keras.applications.resnet50 import ResNet50
from keras.layers import Conv2D, MaxPooling2D, GlobalAveragePooling2D
from keras.layers import Dropout, Flatten, Dense
from keras.models import Sequential
from keras.callbacks import ModelCheckpoint

from extract_bottleneck_features import *
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True
from PIL import Image
import requests
```

Project Process

In addition to the dog classification, we also introduce the function to identify human face to see if the image is on a human, which is not from CNN .

Below is the flow to build the model.

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

In the following section, the details of the project process will be explained step by step. All the corresponding codes can be found in the github depository.

Step 0: Import Datasets

Two main data are used in this project: dog image and human image.

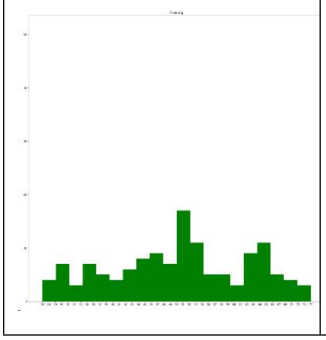
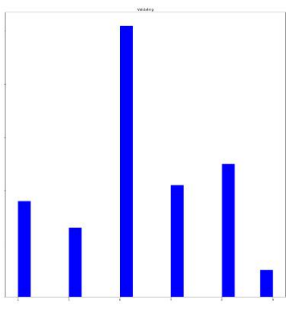
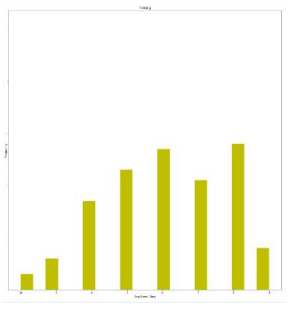
1. Dog image

The datasets were loaded via a pre-defined loading function. Three set of datasets, training, validation and testing data, are set set a ratio of 80%, 10%, 10% of total, respectively. Each set have a file of dog image and another target data with encoded dog classifications.

With the data, we have total 8,351 doc images and 133 dog categories (from dog_names) which will be used in final classification.

- *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

As mentioned in section "Metrics", we use "Accuracy" to measure the model performance, which required that the classes in the data are evenly distributed. Based on the chart, we can confirm the training dataset is mostly distributed evenly (except for a couple classes), while the validating and testing datasets are missing some classes due to the limitation of data size. Given the training data is good enough, we still can use this accuracy as measurement, but may enhance it with bigger validating and testing data in future model development.

Training data	Validating data	Testing data
		

2. Another data set of total 13,233 human images were imported and used for human face detector.

```
# 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/*/"))]
```

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

3. Additionally, there might be some other data needed to complete this project, if not available, including the bottleneck features and the sample image for final testing. The following codes will help download the files and save for next steps in the project.

```
url = "https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/DogInceptionV3Data.npz"
response = requests.get(url, stream=True)
open( 'bottleneck_features/DogInceptionV3Data.npz', 'wb').write(response.content)
```

```
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)
```

Step 1: Detect Human

In this step, OpenCV's implementation of Haar feature-based cascade classifiers is used to detect human faces from images. OpenCV provides many pre-trained face detectors, while we have downloaded one of these detectors for this project, "*Haarcascade_frontalface_alt.xml*".

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

Before using any of the face detectors, the standard procedure is to convert the images to grayscale. The detectMultiScale function executes the saved classifier/detector and takes the grayscale image as a parameter.

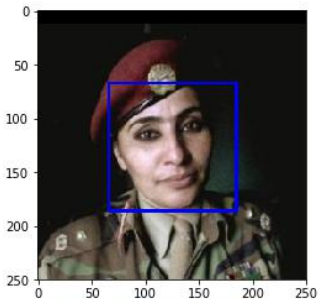
```
# load color (BGR) image
img = cv2.imread(human_files[3])
# convert BGR image to grayscale
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

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

After it, an array, faces, will be created with four entries, which specify various positions of the bounding box of the detected face. Using such an array, we can develop a face detector which takes a string-valued file path to an image as input and appears in the code below.

```python
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
```

The result of using this face detector for a given image is as below:

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



This detector was tested on 100 images of dog and human each, and we received 100% success in human images but only 11% in dog images. It implies that the fact detector works great for human face as it was designed for, but not for dog.

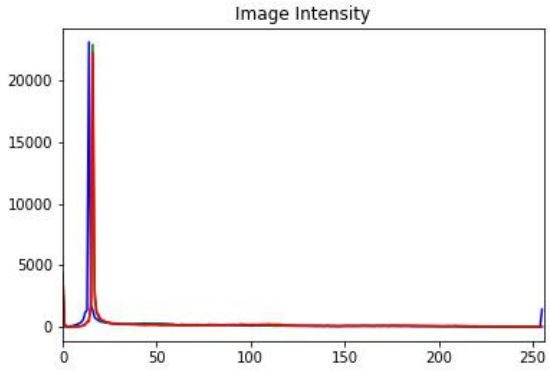
```
What percentage of the first 100 images in `human_files` have a detected human face? 100.0%
What percentage of the first 100 images in `dog_files` have a detected human face? 11.0%
```

So, what should we do to better identify dogs?

In order to better understand the image self and how the model processes each image, a plot of image intensity was created.

The intensity distribution of an image is a plot with pixel values (ranging from 0 to 255, not always) in X-axis and corresponding number of pixels in the image on Y-axis.

```python
# plot image density

def density_plot(image) :
    for i, col in enumerate(['b', 'g', 'r']):
        hist = cv2.calcHist([image], [i], None, [256], [0, 256])
        plt.plot(hist, color = col)
        plt.title("Image Density")
        plt.xlim([0, 256])
    plt.show()
```



Step 2: Detect Dogs

We use one of the most popular pre-trained model, ResNet-50, to detect dog from the images. ResNet-50 was pre-trained with more than 10 million images with 1000 categories.

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

Given that TensorFlow is used as backend, Keras CNNs require a 4D array (a.k.a."4D tensor") as input, while each dimension presents the number of rows, columns, and channels for each image.

A pre-defined function, path_to_tensor, receives the file path of a color image as input and returns a 4D tensor suitable for Keras CNN. The function first loads the image and resizes it to a square image at 224×224 pixels, and then convert the image to an array, which is then resized to a 4D tensor. In this case, the color images have three channels like (224,224,3) presenting the colors. This function takes a numpy array of string-valued image paths as input and returns a 4D tensor with shape (nb_samples,224,224,3), while nb_samples presents the number of images.

```
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)
```

In order to use ResNet-50, additional processing, such as converting RGB to RBG image and Normalization, is required as well. Normalization is to subtract the mean pixel calculated from all pixels in all images from every pixel in each image. It is implemented with the imported function, *preprocess_input*.

With this step done, we can start using ResNet-50 to detect dogs on the image.

```
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))
```

From the predicted probability vector for ResNet50, we obtain an integer corresponding to the projected object class, which is used to identify the category from a predefined dictionary. It is noticed that the categories corresponding to dogs appear in an uninterrupted sequence and correspond to dictionary keys 151-268. Therefore, checking if the ResNet50_predict_labels function returns a value between 151 and 268 (inclusive) will give us if the image is detected with a dog.

```
### 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))
```

With this dog detector tested over the same set of 100 human images and 100 dog images, we have the dog images 100% identified correctly, but nothing for human images.

```
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%
```

Now, we have the functions to identify the dog from the image.

Next, we will develop a method to classify the dog breed if a dog was detected on the image.

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

Not surprisingly speaking, assigning breed to dogs from images is exceptionally challenging. Many breeds look very alike or similar, and on the other hand, some dogs with same breed but different colors may look apart. Given that only 133 dog categories were provided, we would expect the accuracy at approximate 1%.

Pre-Process Data

Again, before developing the CNN model, we need to process the data to make it better fit the CNN. We will re-scale the image by dividing each pixel in each image by 255. Why?

For most image data, the pixel values are integers with values between 0 and 255. It is known that Neural networks process inputs using small weight values, and inputs with large integer values can slow down the learning process. Therefore, it is good to normalize the pixel values into values in the range 0-1 and then images can be viewed normally. So, dividing the pixel by 255 will return value from 0 to 1.

```
# 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
```

Build the model architecture with mix of layers

CNN as regularized versions of Multilayer Perceptrons is comprised of convolution layer, pooling layer, dense layer, etc.

In this architecture, we enhanced the CNN model from the structure provided by the program, by adding some other layer in order to develop the lower loss and better accuracy.

I tested a few different layers in the model.

Test model #1, replace GlobalAveragePooling() layer with Flatten(): It broke the spatial structure, required much high parameters and then was very time consuming, but produced much higher loss and lower accuracy.

Test model #2, add Dense() layer with activation of 'relu' in the first three sets of convolution and max pooling layer and adjust the array dimension to >300: It connected the front layers and flatten it. The earlier Dense layer did not help improve model.

Test model #3, add BatchNormorlization() layer: It helped a little on loss improvement, but very time consuming. Comparing to other more effective improvement, it is not implemented in the model for this project.

The final model architecture is based on the three set of convolution and max pooling layers but add two Dropout layer to reduce the possible overfitting and add one more Dense layer after GAP layer to connect all neurons.

The final Dense layer as full connected layer is acting as classifier with 133 nodes to identify the dog breed based on the provided 133 categories.

The total trainable parameters or weights for this model is 130K+.

```
#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
model.add(Conv2D(filters=16, kernel_size=2, padding='same', activation='relu',
                 input_shape= train_input_shape ))
#3. First pooling
model.add(MaxPooling2D(pool_size=(2, 2)))
#4. Second layer
model.add(Conv2D(filters=32, kernel_size=2, padding='same',  activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dense(100, activation='relu'))
#4. Third layer
model.add(Conv2D(filters=64, kernel_size=2, padding='same',  activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
#5. GAP
model.add(Dropout(0.2))
model.add(GlobalAveragePooling2D())
model.add(Dense(500, activation='relu'))
#6. Fully connected layers
model.add(Dropout(0.3))
model.add(Dense(133, activation='softmax'))
```

```
_________________________________________________________________
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
```

The model was compiled with the optimizer ‘RMSprop’, which divides the learning rate by an exponentially decaying average of squared gradients. It also uses the loss function of ‘categorical_crossentropy’.

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

The model was trained with 10 epochs. I also tested training the model with 5 epochs, which presents lower accuracy and higher loss.

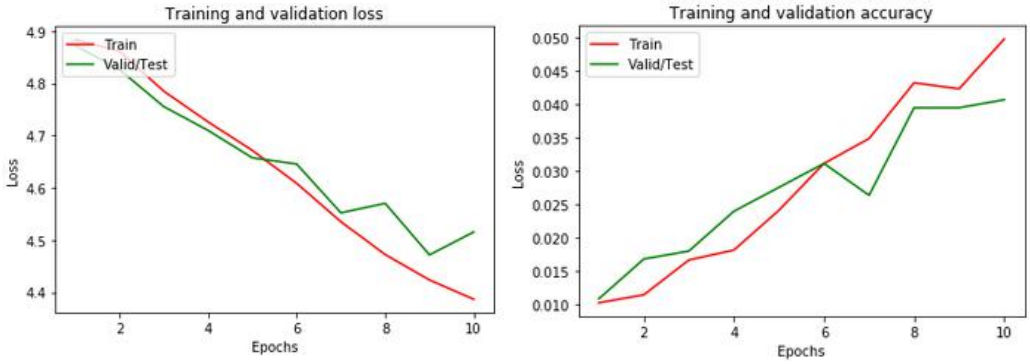
```
epochs = 10

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

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

After loading with best validation loss, the model was tested over the testing data and returned **5.5%** accuracy without any fine-tuning of parameters and without any augmentation on the data.

But, I want to know, how the model performed at each of epoch?

In Keras, the *fit* method used above returns a *history* object. The *history.history* attribute is a dictionary recording training loss values and metrics values at successive epochs, as well as other metrics values if applicable. So, I output the model history and plot the loss and accuracy curve for the training process, as below.

We can see that along each epoch, the accuracy was improved constantly.



Step 4: Use a CNN to Classify Dog Breeds

During the prior model architecture development, the long model training was top concern. To reduce the training time but still maintain the performance, a popular approach in deep learning, Transfer learning, was introduced.

Transfer learning involves taking a pre-trained neural network developed for a task and adapting the neural network to a new data set as the starting point for a model on a second task. Having this, the second model will utilize knowledge from the learned task and be more efficient without repeating the training.

The last activation feature map in the VGG16 model gives us the bottleneck features, which can then be fed to our model for a classifier. The bottleneck features can be downloaded from the resource as shown in "Library Preparation" section.

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

In this step, VGG-16 model was used as a fixed feature extractor and the last convolutional output will be fed as input to our CNN model for dog breed identification. Only one GAP layer and a fully connected layer (with softmax) were added as classifier.

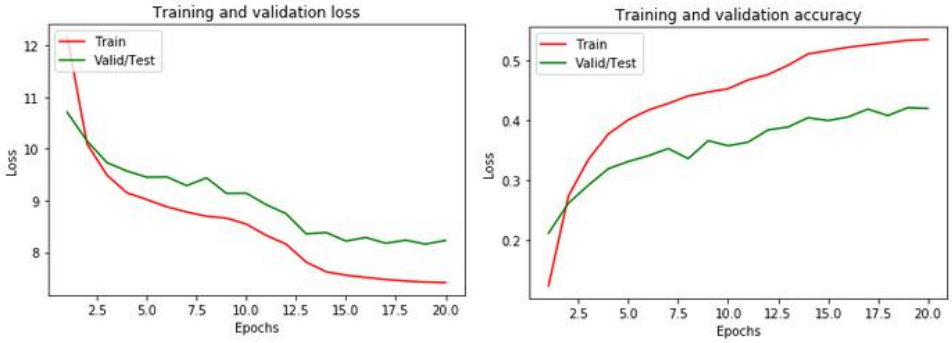
```
VGG16_model = Sequential()
VGG16_model.add(GlobalAveragePooling2D(input_shape=train_VGG16.shape[1:]))
VGG16_model.add(Dense(133, activation='softmax'))
```

```
_________________________________________________________________
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
```

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

checkpointer = ModelCheckpoint(filepath='saved_models/weights.best.VGG16.hdf5',
                               verbose=1, save_best_only=True)

his_VGG16 = VGG16_model.fit(train_VGG16, train_targets,
          validation_data=(valid_VGG16, valid_targets),
          epochs=20, batch_size=20, callbacks=[checkpointer], verbose=1)
```

VGG16 model produced the test accuracy of **42.6%**, much higher than the result from the previous CNN model from scratch, but still not as desired 60% from Udacity. .

Here is the loss and accuracy curve for the VGG16 model, showing how the model was trained. Apparently, the model did not perform in the validating data well as in training.



Step 5: Create a CNN to Classify Dog Breeds (using Transfer Learning)

In this step, similar as how the VGG16 model was used, I will apply Transfer Learning to create a CNN using another pre-trained model available in Keras, InceptionV3.

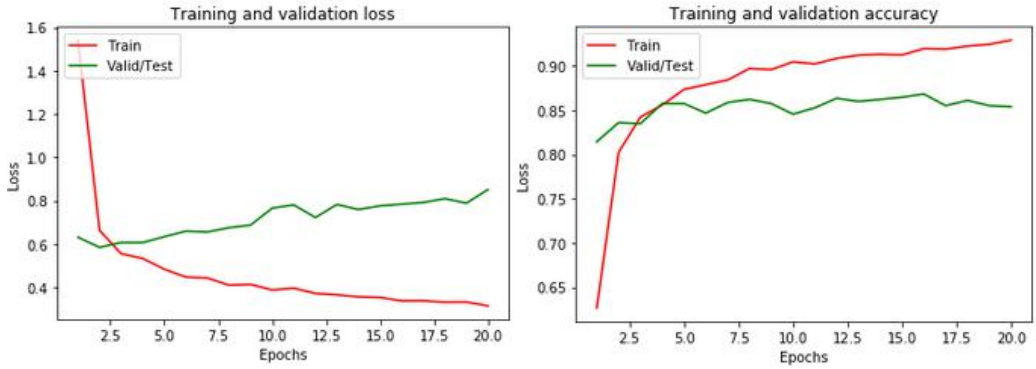
```
InceptionV3_model = Sequential()
InceptionV3_model.add(GlobalAveragePooling2D(input_shape=train_InceptionV3.shape[1:]))
InceptionV3_model.add(Dropout(0.5))
InceptionV3_model.add(Dense(133, activation='softmax'))

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
```

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

his_InceptionV3_model = InceptionV3_model.fit(train_InceptionV3, train_targets,
          validation_data=(valid_InceptionV3, valid_targets),
          epochs=20, batch_size=20, callbacks=[checkpointer], verbose=1)
```

After compiling the model and loading with the best Validation loss, we test the model on the test data of dog images and received the target accuracy of **80%**, much higher than the targeted of 60% from Udacity .

Based on the loss curve, the model did not work well for the validation data. And, the model started overfitting after the 10th epoch.



A predictor is created for next step.

```
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 & 7: Write and Test your own Algorithm

In this step, I will develop an algorithm that accepts the path of the image.

The first task is to determine if the given image contains a dog, or a human, or neither.

The next task is

- if a **dog** is detected in the image, return the predicted breed, using the predictor created from last step.
- 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.

The function of classifier is built as below.

First of all, an image as input will be loaded, and then the downloaded bottleneck features for the selected pre-trained model will be applied to the image, this is then processed through our trained fully-connected model to give a prediction vector.

To this we apply the numpy argmax function to extract the highest probability class/index and use our labels right from the beginning of the notebook to get the name of the dog breed.

```
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.")
```

```
def identify_dog_image(img_path):
    dog_breed_identification(img_path)
    img = cv2.imread(img_path)
    img_name = img_path[7:len(img_path)-4]
    intensity_plot(img, img_name )
```

Next, I will test the defined algorithm on various images, including samples selected from internet. How is the model?

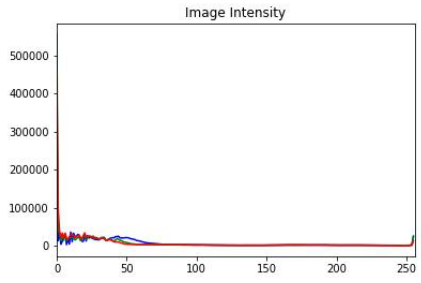
Model Results and Refinement

Given that 78% of test accuracy, the model was not as good as expected. Actually, based on the result testing on the images I had, the model was able to differentiate dogs, human and non-dog objects excellently, but when classified the dog breed, it did not work well as expected.

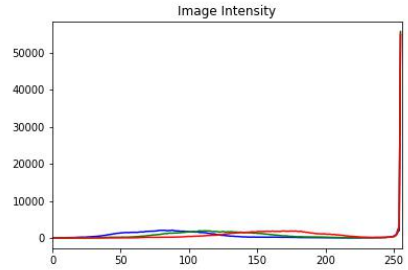
- Image #.1 and #.2 were correctly identified as dog and their breeds.
- However, the #.3 was a rottweiler, but was incorrectly predicted as bordueaux as the #.2.
- Image #.4 and #.7 were correctly classified.
- The rest of image including human pictures, carton neither dog nor human images were all successfully identified from dogs.
- The last image is a masked/covered human face, but the model still worked. It implied that the model was able to grasp the key feature and characters of the image.

There are a few ways to improve the model, including but not limited to the following:

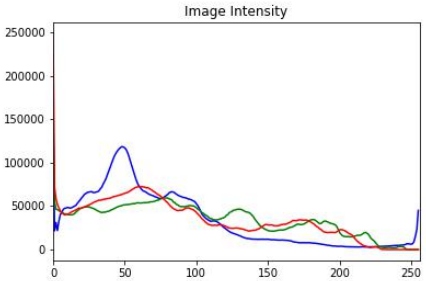
- Increase the breeds and train more images
- Increase number of epochs
- Augmentation of training data
- Tune some of the model parameters



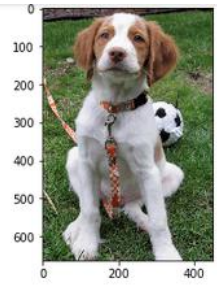
A dog is detected and its breed is ages/train/005.Alaskan_malamute

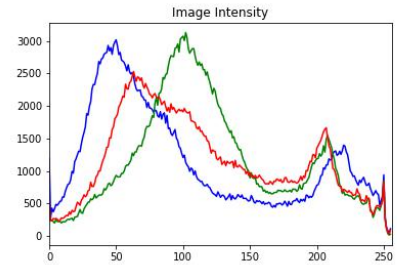


A dog is detected and its breed is ages/train/060.Dogue_de_bordeaux

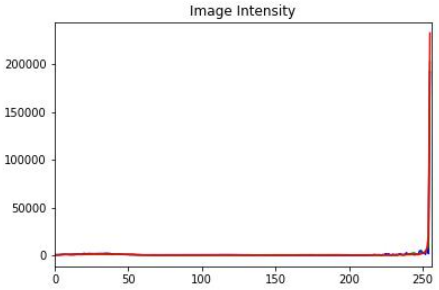


A dog is detected and its breed is ages/train/060.Dogue_de_bordeaux

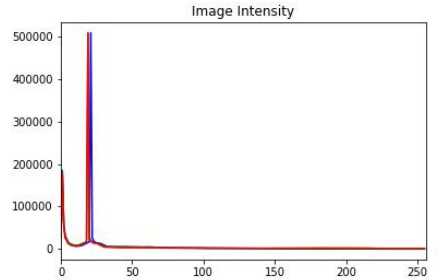




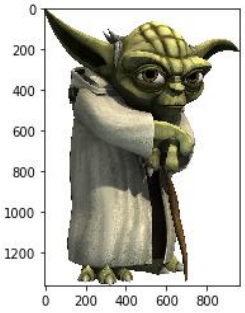
A dog is detected and its breed is ages/train/037.Brittany

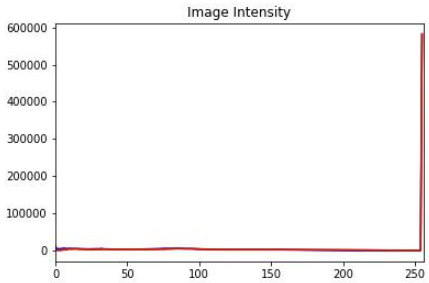


A human is detected, but if this was a dog, its breed would be ages/train/082.Havanese

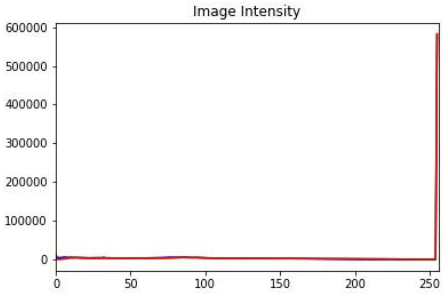


A human is detected, but if this was a dog, its breed would be ages/train/074.Giant_schnauzer






```
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.
```

Reflection

From the actions done through this project, it is approved that CNN is very effective and efficient for image classification, especially for multi-class classification, such as the dog classifier I try to develop in this project.

In this project, I learned how to build the detector to identify dog or human face from the image and subsequently identify the dog breed if applicable. The task could be done from different ways, such as by e defined detector, the CNN built from scratch and the CNN built from Transfer Learning (i.e. leveraging the pre-trained model as input and develop the fully connected layer for the output).

Compared to the pre-trained models used applied in the process, such as VGG16 and InceptionV3, my own CNN architecture does not work so well. It is approved that the more images the model was trained and the more categories to classify to, the more the model is learning and the better the model predicts.

There are also some items I will test on the CNN model in future to improve the performance, including:

1. Train more images, especially more clear images.
2. Optimizer, loss function and other augments used in model compiling
3. Expand the size of dog breed classifications

Reference:

1. https://en.wikipedia.org/wiki/Convolutional_neural_network
2. http://deeplearning.net/tutorial/lenet.html
3. http://deeplearning.stanford.edu/tutorial/supervised/ConvolutionalNeuralNetwork/

Publication link:

https://medium.com/@yanzhang14689/dog-breed-classification-using-cnn-e9883d473dc9