Convolutional Neural Network (CNN) Project

Capstone Project Proposal : Dog Breed Classification

Yongyang Sun Udacity

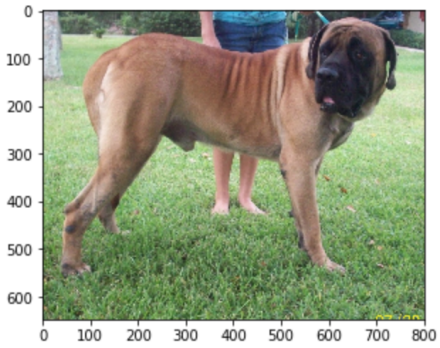
29th Dec 2020

I. Definition

Project Overview

Ready for exploring the secrets of the Convolutional Neural Network (CNN)? This fantastic project will give you a start. In this project, we will develop machine learning algorithms using the CNN to classify the dogs according to their breeds. Given an image, the algorithms will first judge whether it is a human or dog. If the supplied image is predicted as a dog, the algorithms will tell you which breed it belongs to. If the supplied image is predicted as a human, the algorithms will tell you which dog breed the human resembles.

Along with testing the performance of the CNN models compared to the benmark model, we will have a comprehensive understanding of the great power of the CNN algorithms. The main objective of this project is to show you how to create a CNN model from stratch and how to use a pre-trained model to help develop a CNN model.




```
Dog Detected!
It looks like a Mastiff
```

Problem Statement

The main objective is to create a CNN model for classifying dogs according to their breeds. The key steps involved are shown as follows:

1. Import, Explore and Visualize Datasets
2. Detect Humans
3. Detect Dogs
4. Create a CNN Model from Scratch to Classify Dog Breeds
5. Create a CNN Model by transfer learning to Classify Dog Breeds
6. Write your Algorithm
7. Test your Algorithm

The finally developed CNN model is expected to predict the dog breed if supplied a dog image. If supplied a human image, the model will show you the dog breed that the human resembles.

Metrics

The metrics that can be used to evaluate the performance of the trained CNN model are many, such as Accuracy, Recall, Precision, F1 score and so one. In our case, we use cross-entropy loss to optimize the trained model and accuracy to evaluate the performance of the models in classifying the dog breeds. Cross-entropy loss awards lower loss to predictions which are closer to the class label. The accuracy, on the other hand, is a binary true/false for a particular sample. That is, Loss here is a continuous variable i.e. it's best when predictions are close to 1 (for true labels) and close to 0 (for false ones). While accuracy is kind of discrete. In our case, there are 133 classes for dog breeds and the datasets are relatively balanced. Hence, accuracy can be used as a metric to evaluate the performance of the trained model. Accruacy is calculated by the ratio of the correct predictions to the total number:

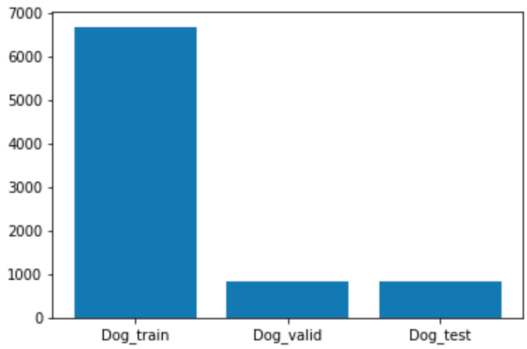
$Accuracy = \frac{TP}{TP+FP}$

II. Analysis

Data Exploration

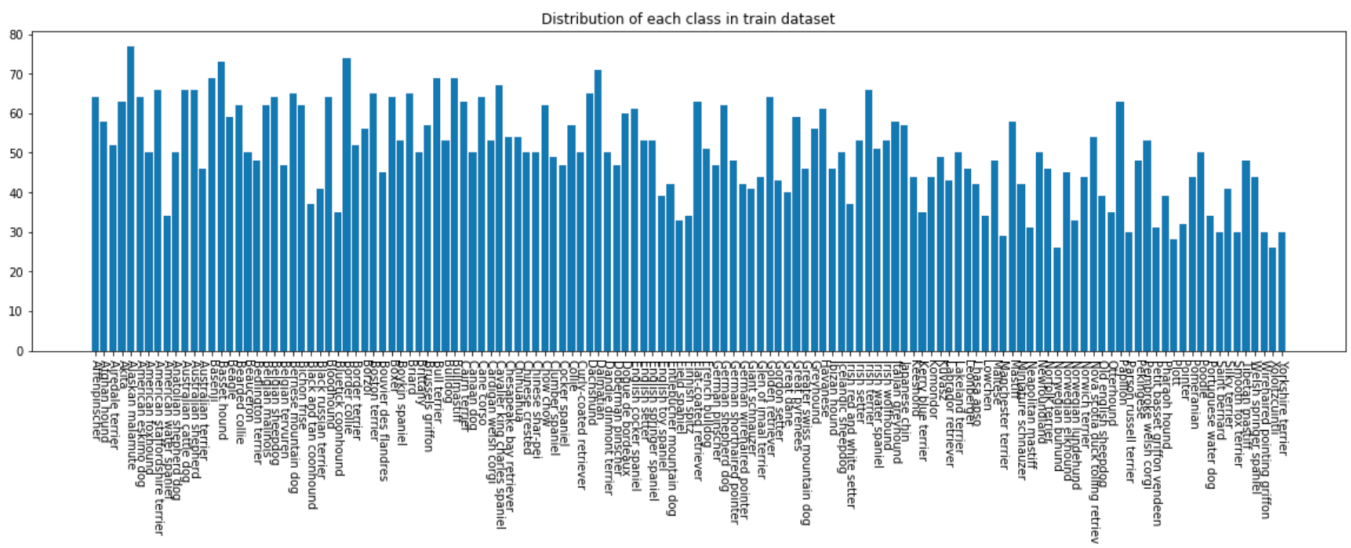
In this project, the human dataset has 13233 images stored 5749 folders while the dog dataset has 8351 dog images. In dog dataset, there are 133 dog breeds. The number of images for different dog breeds varies. Hence, the datasets are imbalanced. Besides, the dog images have different size and resolution, which requires preprocessing in order to be suitable as inputs into the designed model.

We observe that the dog images have already been split into "train", "validation" and "test" datasets. The percentages are 80% for "train", 10% for "validation" and 10% for "test". The split of dog images can be observed in the below graph. We can see that 6680 dog images will be used to train the model while 835 dog images will be used to tune the parameters. In the end, 836 dog images will be used to evaluate the performance of the model.



Balance Analysis

The study on the distribution of data that is used for training a model gives us the information about the balance of the data. Imbalanced data will significantly affect the predictive power of the trained model in application. If the dataset for training is relatively balanced or close to a threshold, it is good to move to the next step. If the dataset is extremely imbalanced or beyond a threshold significantly, approaches need to be taken to reduce the imbalance. Let's take a look at our case below. We could see that the number of images for each class is relatively balanced. The slight imbalance will not affect the model considerabely.



Algoritgms and Techniques

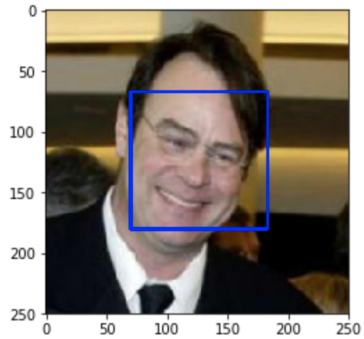
All the steps involved in this project are the following:

1. Import the datasets required (dog and human datasets);
2. Detect human using a pre-trained detector (here we will use the OpenCV's implementation of Haar feature-based cascade classifiers t to detect human faces from the image. Please refer to https://docs.opencv.org/master/db/d28/tutorial_cascade_classifier.html (https://docs.opencv.org/master/db/d28/tutorial_cascade_classifier.html) to get more information);
3. Detect dogs using several pre-trained detectors (here we will use VGG16, ResNet50 and ResNet18 to test their performance)
4. Since our data has been divided into "training", "validation" and "test" subsets, we can use the "training" dataset to create a Convolutional Neural Network (CNN) from scratch as a benmark model. We will test the model on the "test" dataset to get an accuracy of over 10% by adujusting the input parameters such as epoch value and the number of convolutional layersused. CrossEntropyLoss is used to evalute the quality of the created CNN model.
5. Create a CNN model by transfer learning using a pre-trained dog detector based on the same "training" dataset. By adjusting the parameters, we will develop a model with an expected acccuracy of over 60%. CrossEntropyLoss is used to evalute the quality of the created CNN model.

Step 1: Detect Humans

In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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



Then, we use the pre-trained face detector to test its performance in first 100 images of the datasets. The test result is as follows

```
from tqdm import tqdm

human_files_short = human_files[:100]
dog_files_short = dog_files[:100]

#-#-# Do NOT modify the code above this line. #-#-#

## TODO: Test the performance of the face_detector algorithm
## on the images in human_files_short and dog_files_short.
human_files_percentage = sum([face_detector(human) for human in human_files_short]) / len(human_files_short)
dog_files_percentage = sum([face_detector(dog) for dog in dog_files_short]) / len(dog_files_short)
print("The percentage of human detected in human files is", human_files_percentage * 100,"%")
print("The percentage of human detected in dog files is", dog_files_percentage * 100, "%")
```

```
The percentage of human detected in human files is 98.0 %
The percentage of human detected in dog files is 17.0 %
```

The percentage of human face detected in human files are 97% while the percentage of human face detected in dog files are 17%. The accuracy is not bad, right? The further work will test the perforamnce of more pre-trained face detectors, but this one is good for our case.

Step 2: Detect Dogs

Preprocess the data

In this section, we will use several pre-trained models to detect dogs and evaluate their performance. These models have some specific requirements for input data. Hence, we need to preprocess these data. In this step, images are resized, cropped and normalized to speed up the detection. Then, the images will be converted to tensors. First we resize the image to 256 px and then crop the image to 224 px. All the pre-trained models expect input images normalized in the same way, i.e. mini-batches of 3-channel RGB images of shape (3 x H x W), where H and W are expected to be at least 224. An example is like the following:

```
    img = Image.open(img_path)
    transform = transforms.Compose([transforms.Resize(256),
                                    transforms.CenterCrop(224),
                                    transforms.ToTensor(),
                                    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                         std=[0.229, 0.224, 0.225])
                                   ])


    img_t = transform(img)
    batch_t = img_t.unsqueeze(0)
    if use_cuda:
        batch_t = batch_t.cuda()

    # activate VGG16 model
    VGG16.eval()
    output = VGG16(batch_t)
    output = output.cpu().data.numpy().argmax()
    return output # predicted class index
```

Then, we develop several detectors based on pre-trained models, e.g., VGG16, ResNet50, ResNet18. An example is like the following:

```
### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    ## TODO: Complete the function.
    if VGG16_predict(img_path) in range(151,269):
        return True
    else:
        return False
```

Then we evaluate the performance of these models in detecting dogs. All these models work reasonably.

This is for VGG16

```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.
human_files_percentage = sum([dog_detector(human) for human in human_files_short]) / len(human_files_short)
dog_files_percentage = sum([dog_detector(dog) for dog in dog_files_short]) / len(dog_files_short)
print("The percentage of dog detected in human files is", human_files_percentage * 100,"%")
print("The percentage of dog detected in dog files is", dog_files_percentage * 100, "%")
```

```
The percentage of dog detected in human files is 0.0 %
The percentage of dog detected in dog files is 100.0 %
```

This is for ResNet50

```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.
human_files_percentage = sum([dog_detector2(human) for human in human_files_short]) / len(human_files_short)
dog_files_percentage = sum([dog_detector2(dog) for dog in dog_files_short]) / len(dog_files_short)
print("The percentage of dog detected in human files is", human_files_percentage * 100,"%")
print("The percentage of dog detected in dog files is", dog_files_percentage * 100, "%")
```

```
The percentage of dog detected in human files is 0.0 %
The percentage of dog detected in dog files is 100.0 %
```

This is for ResNet18

```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.
human_files_percentage = sum([dog_detector3(human) for human in human_files_short]) / len(human_files_short)
dog_files_percentage = sum([dog_detector3(dog) for dog in dog_files_short]) / len(dog_files_short)
print("The percentage of dog files detected in human files is", human_files_percentage * 100,"%")
print("The percentage of dog files detected in dog files is", dog_files_percentage * 100, "%")
```

```
The percentage of dog files detected in human files is 0.0 %
The percentage of dog files detected in dog files is 99.0 %
```

Step 3: Create a CNN model from scratch

From the test work above, we have the detectors for detecting humans and dogs in any input images. Now comes to the work of creating a model to predict the dog breed from images. In this step, a CNN model from scratch will be designed. We will use 5 convolutional layers and 2 fully connected layers. 5 convolutional layers are created with max-pooling layer in between them to reduce the dimansionality. Since we have 133 breed classes, the number of nodes in the last fully connected layer is set to 133. Relu activation is used for most layers.

The implementation of creating a CNN model from scratch can be summerized as follows:

- Load the "train", "validation" and "test" datasets into memory, preprocessing them as instructed above;
- Define the CNN model architecture and training parameters;
- Define the loss function to optimize the model and the accuracy to evaluate the model
- Train the network and test the accuracy to see if parameter tuning are required;

First, as we did in the dog detector function, we define three tranform to preprocess the data as intructed. Especially for the trainning set, we augment the data by flipping the images horizontally and rotating slightly, which will add more variations for the trainning set. Then, we create data loaders to load the three datasets into memory.

This is how the images are loaded and preprocessed

```
ImageFile.LOAD_TRUNCATED_IMAGES = True
transform = {
    'train' : transforms.Compose([transforms.Resize(256),
                                  transforms.RandomResizedCrop(224),
                                  transforms.RandomHorizontalFlip(),
                                  transforms.RandomRotation(10),
                                  transforms.ToTensor(),
                                  transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                       std=[0.229, 0.224, 0.225])]),
    'valid' : transforms.Compose([transforms.Resize(256),
                                  transforms.CenterCrop(224),
                                  transforms.ToTensor(),
                                  transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                       std=[0.229, 0.224, 0.225])]),

    'test' : transforms.Compose([transforms.Resize(256),
                                 transforms.CenterCrop(224),
                                 transforms.ToTensor(),
                                 transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                      std=[0.229, 0.224, 0.225])])
}
num_workers = 0
batch_size = 20
image_datasets = {x: datasets.ImageFolder(os.path.join('/data/dog_images/', x), transform[x])
                  for x in ['train', 'valid', 'test']}
# Create data loaders (train, valid, test)
loaders_scratch = {x: torch.utils.data.DataLoader(image_datasets[x], batch_size=batch_size,
                                                  shuffle=True, num_workers=num_workers)
                   for x in ['train', 'valid', 'test']}
```

Second, we defin the CNN architecture and training parameters.

This is the CNN architecture

```
# define the CNN architecture
class Net(nn.Module):
    ### TODO: choose an architecture, and complete the class
    def __init__(self):
        super(Net, self).__init__()
        ## Define layers of a CNN
         ## Define layers of a CNN
        self.conv1 = nn.Conv2d( 3, 16, 3)
        self.conv2 = nn.Conv2d(16, 32, 3)
        self.conv3 = nn.Conv2d(32, 64, 3)
        self.conv4 = nn.Conv2d(64, 128, 3)
        self.conv5 = nn.Conv2d(128, 256, 3)
        # max pooling layer
        self.pool = nn.MaxPool2d(2, 2)
        # linear layer (256 * 5* 5 -> 500)
        self.fc1 = nn.Linear( 256 * 5 * 5, 500)
        self.fc2 = nn.Linear(500, 133)

    def forward(self, x):
        ## Define forward behavior
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
```

The steps that are used to develop the CNN architecture is summerized as follows: first, we need to know how to calculate the spatial dimensions after a convolutional layer. The expression is like (W_in - F + 2P)/S + 1 (P=0 and S=1 by default).

C layer 1: load in tensor with the size of 224 *224* 3 and convert to the one with the size of 222 *222* 16; M layer 1: convert the output from C layer 1 to the one with the size of 111 *111* 16; C layer 2: convert the output from M layer 1 to the one with the size of 109 *109* 32; M layer 2: convert the output from C layer 2 to the one with the size of 54 *54* 32; C layer 3: convert the output from M layer 2 to the one with the size of 52 *52* 64; M layer 3: convert the output from C layer 3 to the one with the size of 26 *26* 64; C layer 4: convert the output from M layer 3 to the one with the size of 24 *24* 128; M layer 4: convert the output from C layer 4 to the one with the size of 12 *12* 128; C layer 5: convert the output from M layer 4 to the one with the size of 10 *10* 256; M layer 5: convert the output from C layer 5 to the one with the size of 5 *5* 256; Then, flatten the image input and make the prediction for different dog breeds. Note that there are 133 labels for dog breeds.

Let's have a look at the results. The accuracy is 18%, relatively low. Note that we set the epochs =20. If we increase the number of the epochs, for example, 100. The accuracy will increase dramatically. Some other parameters can also be tuned to give a satisfactory accuracy, but it will cost much longer time to get the results. Since we have already met the least accuracy requirement (10%). We will move to the next step.

```
# call test function
test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.367710


Test Accuracy: 18% (155/836)
```

Step 4: Create a CNN model by transfer learning

Step 3 shows the model predictions based on a CNN model created from scratch. The model accuracy is 18%, which beyonds the least accuracy required but not enough for dog breed classification. Adjusting parameters will increase the accuracy significantly. However, it will take much longer time to appraoch, which is out of the scope of this work. An alternative is to create a CNN model by transfer learnig. We could use pre-trained models such as ImageNet, ResNet, VGG16, etc to classify the dog breeds. We have already tested the performance of three models in dog image detection above, we will choose one of them to create the CNN model by transfer learning. We use the pre-trained models to define the architecture of the CNN model and freeze the tuned parameters. We just need to replace the last fully connected layer with 133 nodes.

This is an example of how we define the model

```
import torchvision.models as models
import torch.nn as nn

## TODO: Specify model architecture
model_transfer = ResNet50

# Freeze parameters so we don't backprop through them
for param in model_transfer.parameters():
    param.requires_grad = False

# Replace the last fully connected layer with a linear layer with 133 out features
model_transfer.fc = nn.Linear(2048, 133)

if use_cuda:
    model_transfer = model_transfer.cuda()
```

Let's see the model results. The accuracy is 83%. Very good, right? Compared to the benchmark model above, the accuracy increases considerably. Note that the number of epochs for this model is 10, much smaller than the one we set for the benchmark model. If we increase the number of epochs, the accuracy will also increase.

```
# train the model
model_transfer = train(10, loaders_transfer, model_transfer, optimizer_transfer, criterion_transfer, use_cuda, 'model_transfer.p

# load the model that got the best validation accuracy (uncomment the line below)
model_transfer.load_state_dict(torch.load('model_transfer.pt'))
```

```
Epoch: 1        Training Loss: 2.839314         Validation Loss: 0.952225
Validation loss decreased (inf --> 0.952225).    Saving model...
Epoch: 2        Training Loss: 1.516055         Validation Loss: 0.672417
Validation loss decreased (0.952225 --> 0.672417).    Saving model...
Epoch: 3        Training Loss: 1.340298         Validation Loss: 0.631558
Validation loss decreased (0.672417 --> 0.631558).    Saving model...
Epoch: 4        Training Loss: 1.277721         Validation Loss: 0.559298
Validation loss decreased (0.631558 --> 0.559298).    Saving model...
Epoch: 5        Training Loss: 1.202081         Validation Loss: 0.498856
Validation loss decreased (0.559298 --> 0.498856).    Saving model...
Epoch: 6        Training Loss: 1.180721         Validation Loss: 0.543812
Epoch: 7        Training Loss: 1.145859         Validation Loss: 0.596924
Epoch: 8        Training Loss: 1.156151         Validation Loss: 0.587403
Epoch: 9        Training Loss: 1.126089         Validation Loss: 0.500400
Epoch: 10       Training Loss: 1.140203         Validation Loss: 0.603168
```

(IMPLEMENTATION) Test the Model

Try out your model on the test dataset of dog images. Use the code cell below to calculate and print the test loss and accuracy. Ensure that your test accuracy is greater than 60%.

```
test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.552045


Test Accuracy: 83% (695/836)
```

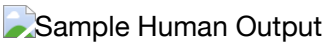
Step 5: Write your Algorithm

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 `human_detector` functions developed above. You are **required** to use your CNN from Step 4 to predict dog breed.

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

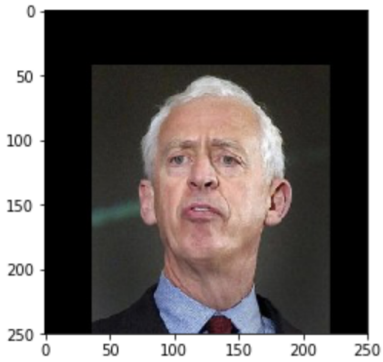
 Sample Human Output

In [1]:

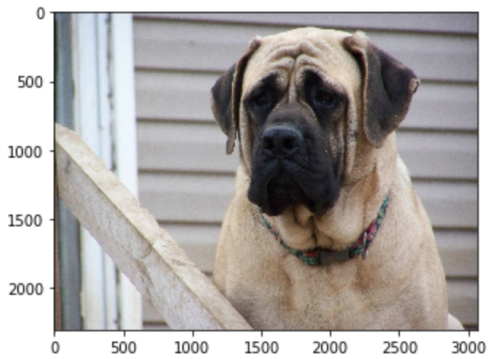
```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.

def run_app1(img_path):
    ## handle cases for a human face, dog, and neither
    img = Image.open(img_path)
    plt.imshow(img)
    plt.show()
    if dog_detector(img_path):
        prediction = predict_breed_transfer(img_path)
        print("Dog Detected!\nIt looks like a {0}".format(prediction))
    elif face_detector(img_path) > 0:
        prediction = predict_breed_transfer(img_path)
        print("Hello, human!\nIf you were a dog... You look like a {0}".format(p
rediction))
    else:
        print("Neither human nor dog detected, input error")
```

Step 6: Test your Algorithm

Let's see some interesting predictions.



```
Hello, human!
If you were a dog... You look like a Cocker spaniel
```



```
Dog Detected!
It looks like a Mastiff
```

Step 7: Potential Improvements

- More dataset augmentation (e.g., higher rotation, flipping vertically) can be tested to see whether this would improve the model accuracy
- Image processing is repeated, e.g., transform defined in many functions, which can be defined as a function.
- More parameters can be tested, e.g., increasing the value of epochs, employing different pre-trained models, and increasing the number of layers for the CNN architecture.

Results

In this section, we observe the performance of our final model (the CNN model by tranfer learning). The accuracy is 83%, which is much higher than the one of the CNN model created from scratch, 18%. Both have met the requirement of this project for leat accuracy of 10%.

In the implementation phase of the pre-trained model ResNet_50, we test its performance on the dog detection and we see that in the first 100 images of the dogs, it gives the right prediction value of the dog, 100%. Although it does not mean it is a perfect model (we only use accracy as a metric to evaluate the performance and this metric might be misleading in the case of imbalanced data), it works fine for our case.

Note that we only set the number of epochs to 10, much smaller than the value (20) set for the customized CNN model. The accuracy has improved by 360% from 18% to 83%. If we continue tuning the parameters, for example, increasing the number of epochs to 20, the accuracy will increase.

III. Conclusion

Finally, I make it. Although I had experienced lots of problems such as the version mismatch, Udacity working environment differs from the local environment, pre-trained models give different results, I arrived at the end. Thanks everybody who have helped me go through these difficulties. This project prvides with me a valuable opportunity to explore the arts of deep learning and make a great start. I will continue contributing my efforts to deep learning in next few years.