Capstone Project

Machine Learning Engineer
Nanodegree

Definition

Project Overview

Welcome to the Convolutional Neural Networks (CNN) project! In this project, I will show how to build a pipeline to process real-world, user-supplied images. Given an image of a dog, algorithm will identify an estimate of the canine's breed. If supplied an image of a human, the code will identify the resembling dog breed.

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

Problem Statement

Along with exploring state-of-the-art CNN models for classification, I will make important design decisions about the user experience for app. The goal is that by completing this lab to understand the challenges involved in piecing together a series of models designed to perform various tasks in a data processing pipeline. Each model has its strengths and weaknesses, and engineering a real-world application often involves solving many problems without a perfect answer.

Metrics

The quality of all models will be measured with accuracy.

Accuracy = (TP + TN)/(TP + TN + FP + FN)

Analysis

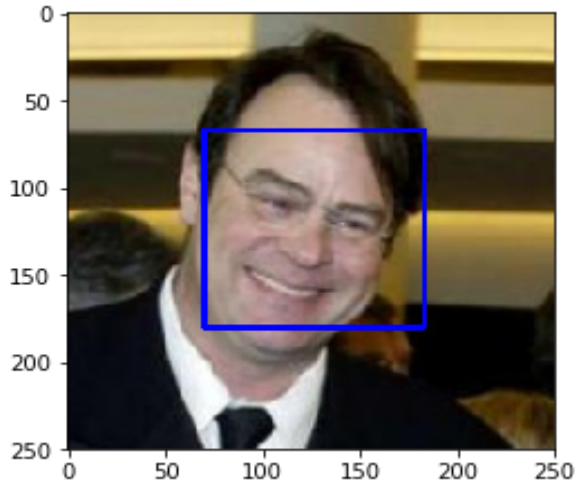
Data Exploration

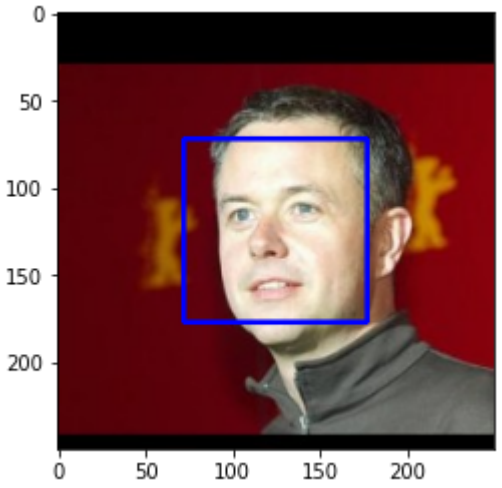
For our application was provided dog dataset and human dataset: 13233 images of humans and 8351 photos of dog images.

```
import numpy as np
from glob import glob

# load filenames for human and dog images
human_files = np.array(glob("/data/lfw/*/*"))
dog_files = np.array(glob("/data/dog_images/*/*/*"))

# print number of images in each dataset
print('There are %d total human images.' %
len(human_files))
print('There are %d total dog images.' % len(dog_files))

There are 13233 total human images.
There are 8351 total dog images.
```

Exploratory Visualization

For example I will provide some photos of people and dogs.





Algorithms and Techniques

Since my task is to use several approaches, I will use different algorithms accordingly. I 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.

For detect dogs I use a pre-trained model of **VGG-16**. This is a 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. In order to check to see if an image is predicted to contain a dog by the pre-trained VGG-16 model, we need only check if the pre-trained model predicts an index between 151 and 268 (inclusive).

Also I create a CNN to Classify Dog Breeds and then use transfer learning to create a CNN that can identify dog breed from images.

Methodology

Data Preprocessing

There was no need to perform any special data preprocessing for the task.
Implementation

Step 1: Detect Humans OpenCV's implementation of Haar feature-based cascade classifiers

```
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[0])
# 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()
```

We get such results:
Percentage of Humans detected as Humans : 98 %
Percentage of Dogs detected as Humans : 17 %

Step 2: Detect Dogs with pretrained model

```
import torch
import torchvision.models as models

# define VGG16 model
VGG16 = models.vgg16(pretrained=True)

# check if CUDA is available
use_cuda = torch.cuda.is_available()

# move model to GPU if CUDA is available
if use_cuda:
    VGG16 = VGG16.cuda()
```

Making Predictions with a Pre-trained Model

```
from PIL import Image
import torchvision.transforms as transforms

def VGG16_predict(img_path):
    '''
    Use pre-trained VGG-16 model to obtain index corresponding to
    predicted ImageNet class for image at specified path

    Args:
    img_path: path to an image

    Returns:
```

```
    Index corresponding to VGG-16 model's prediction
    '''

    ## TODO: Complete the function.
    ## Load and pre-process an image from the given img_path
    ## Return the *index* of the predicted class for that image
    # get the image file
    file = Image.open(img_path)

    # establish the transform pipeline
    transform_pipeline =
transforms.Compose([transforms.Resize((224 , 224)) ,
transforms.ToTensor() , transforms.Normalize(mean = [0.485 , 0.456
, 0.406] , std = [0.229 , 0.224 , 0.225])])

    # transform the file and add dimension so that image is
suitable to pass to VGG16 model.
    img = transform_pipeline(file).unsqueeze(0)

    # Use GPU if available
    if use_cuda:
    img = img.to('cuda')

    # get the prediction
    pred = VGG16(img)

    # get the index
    idx = pred.argmax()

    return idx # predicted class index
```

Write a Dog Detector

```
def dog_detector(img_path):
    ## TODO: Complete the function.
    return ((VGG16_predict(img_path) >= 151) and
(VGG16_predict(img_path) <= 268))
```

We get good result:
Percentage of Dogs detected as Humans : 0 %
Percentage of Dogs detected as Dogs : 100 %

Step 3: Create a CNN to Classify Dog Breeds

```
import os
import sys
import torchvision
import torchvision.transforms as transforms
from torchvision import datasets
from torch.utils.data import DataLoader

### TODO: Write data loaders for training, validation, and test
sets
## Specify appropriate transforms, and batch_sizes
# build the transforms dictionary

data_dir = 'data/dog_images/'

transforms = transforms.Compose([
      transforms.Resize((224, 224)),
      transforms.RandomHorizontalFlip(),
      transforms.RandomRotation(degrees = (0 , 30)),
      transforms.ToTensor(),
      transforms.Normalize(mean=[0.485, 0.456, 0.406],
                           std=[0.229, 0.224, 0.225] )
      ])

# establish the paths to required folders
train_data_path = os.path.join(data_dir, 'train')
train_data =
torchvision.datasets.ImageFolder(root=train_data_path,
transform=transforms)

valid_data_path = os.path.join(data_dir, 'valid')
valid_data =
torchvision.datasets.ImageFolder(root=valid_data_path,
transform=transforms)

test_data_path = os.path.join(data_dir, 'test')
test_data = torchvision.datasets.ImageFolder(root=test_data_path,
```

```
transform=transforms)
# Put into a Dataloader using torch library
batch_size = 64
train_data_loader = DataLoader(train_data, batch_size=batch_size,
shuffle=True)
valid_data_loader = DataLoader(valid_data, batch_size=batch_size)
test_data_loader = DataLoader(test_data, batch_size=batch_size)
```

I resized the image to 224x224 as Most existing architectures use 224 × 224 or 299 × 299 for their image inputs. In general, the larger the input size, the more data for the network to learn from. The flip side is that you can often fit a smaller batch of images within the GPU's memory. I first used CenterCrop to Crop the given image at the center. I resized the image to 224x224 and performed RandomHorizontalFlip with a probability of 0.5 I used RandomRotation with a minimum degree of 0 and a max degree of 30. Finally, I converted the augmented image to a tensor and then normalized it.

```
import torch.nn as nn
import torch.nn.functional as F

# 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

      #Convolutional Layers
      self.conv1 = nn.Conv2d(in_channels = 3 , out_channels = 32 ,
kernel_size = 3 , stride = 1 , padding = 1) # Takes in (224*224*3
image tensor)
      self.conv2 = nn.Conv2d(in_channels = 32 , out_channels = 64 ,
kernel_size = 3 , stride = 1 , padding = 1) # Takes in (112*112*32
image tensor)
      self.conv3 = nn.Conv2d(in_channels = 64 , out_channels = 128
, kernel_size = 3 , stride = 1 , padding = 1) # Takes in (56*56*64
image tensor)
      self.conv4 = nn.Conv2d(in_channels = 128 , out_channels = 256
, kernel_size = 3 , stride = 1 , padding = 1) # Takes in
(28*28*128  image tensor)
      self.conv5 = nn.Conv2d(in_channels = 256 , out_channels = 512
, kernel_size = 3 , stride = 1 , padding = 1) # Takes in
```

```
(14*14*256  image tensor)

      # dropout layer (p=0.3)
      self.dropout = nn.Dropout(0.3)

      # Linear Layers
      self.fc1 = nn.Linear(in_features = 512*7*7 , out_features =
512)
      self.fc2 = nn.Linear(in_features = 512 , out_features = 256)
      self.out = nn.Linear(in_features = 256 , out_features = 133)



      def forward(self, x):
      ## Define forward behavior
      #input layer
      x = x

      # hidden conv layer 1
      x = F.relu(self.conv1(x))
      x = F.max_pool2d(x , kernel_size = 2 , stride = 2)

      # hidden conv layer 2
      x = F.relu(self.conv2(x))
      x = F.max_pool2d(x , kernel_size = 2 , stride = 2)

      # hidden conv layer 3
      x = F.relu(self.conv3(x))
      x = F.max_pool2d(x , kernel_size = 2 , stride = 2)

      # hidden conv layer 4
      x = F.relu(self.conv4(x))
      x = F.max_pool2d(x , kernel_size = 2 , stride = 2)

      # hidden conv layer 5
      x = F.relu(self.conv5(x))
      x = F.max_pool2d(x , kernel_size = 2 , stride = 2)

      # hidden linear layer 1
      x = x.reshape(-1 , 512*7*7) #must be flattened for 1st linear
layer
```

```
        x = F.relu(self.fc1(x))
        x = self.dropout(x)

        # hidden linear layer 2
        x = F.relu(self.fc2(x))
        x = self.dropout(x)

        # outpul layer
        x = self.out(x)

        return x

        def __repr__(self): # Signature function
        return "Neural net"

#-#-# You so NOT have to modify the code below this line. #-#-#

# instantiate the CNN
model_scratch = Net()

# move tensors to GPU if CUDA is available
if use_cuda:
        model_scratch.cuda()
```

I have choosed CNN architecture consists of 5 convolutional layers and 3 Fully Connected Linear Layers. Each convolutional layer is followed by a max_pool2d function, which ha a kernel size = 2 and stride = 2, similarly each Linear layer except the output layer is followed by a dropout layer with a dropout probability of 0.3.
For the first conv layers, there are three in-channels and 32 out channels, after that, I kept increasing the channels by a factor of 2. Each conv layer has a kernel size = 3 , stride = 1 and padding = 1.
After the tensor passes through the conv layers it reaches the 1st linear layer, there I flatten the tensor(the logic of flattening is described below) and pass it through the 1st linear layer, after that the tensor passes through a dropout layer and then again to a successive layer and finally to the output layer. The number of output classes is 133 (i.e., the number of dog breeds in the dataset). Hence, the output layer has 133 out-features. Along with this, I have included a signature function which prints "Abhishek's net" when print(model_scratch) is used, and in order to provide a summary of my model, I have used torchsummary module which is very similar to Keras's model.summary()
After all the accuracy isn't that good: Test Accuracy: 21% (181/836)

Step 4: Create a CNN to Classify Dog Breeds (using Transfer Learning)

At this stage, I have used a pre-trained ResNet50 model to make predictions using Transfer Learning. I am using the already defined data loaders in loders_scratch. ResNet50 is a deep residual network which is mostly used for image classification problems. It is a subclass of the Convolutional Neural Network. The main innovation of resnet is the skip connection. I preferred it over VGG-16 and AlexNet as ResNet is way more profound than either of those two and has better chances of predicting correct labels.
I first downloaded the pre-trained ResNet model, then froze all the layers of this model and finally replaced the last FC layer with a custom Linear Layer which ha 133 out_features correspond to 133 dogs breeds in our dataset.

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

## TODO: Specify model architecture
model_transfer = models.resnet50(pretrained = True)

# freeze the layers of already trained resnet50 model to avoid
furthur training, parameters of newly constructed modules have
requires_grad = True by default
for param in model_transfer.parameters():
      param.requires_grad = False

# remove the last fc layer and add a new one insted which has 133
out_features corresponding to 133 dog breeds in dataset
num_ftrs = model_transfer.fc.in_features
model_transfer.fc = nn.Linear(num_ftrs , 133)

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

Test Accuracy: 78% (653/836)

Results

The best result I have get with pre-trained ResNet50 model to make predictions using Transfer Learning.

There are a few opportunities for increasing quality:
1) Play around with the learning rate.
2) Getting more data.
3) Using more epochs
4) Using more augmentations.

The result of this project is an application that defines people and dogs, as well as their breed. This app would be great for use in veterinary clinics.