Machine Learning Engineer Nanodegree

Capstone Project – Dog Breed Classifier

Ivan Tsvetkov
February 4th, 2020

I. Definition

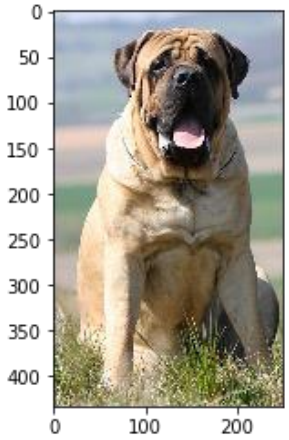
Project Overview

In this project, I combined several different machine learning models, to create a Dog Breed Classifier, which will be the basis for a web or mobile app. When you provide an image, the algorithm will detect if there is a human or a dog present in the image. If it detects a dog, it will give a breed prediction and if there is a human – it will give the breed that the human resembles, which may generate some funny situations.

The project is suggested by Udacity and has some provided template code. I used it as a guide and followed its suggestions.

This is what the finished algorithm outputs, when provided with an image of a dog:

```
You are a dog - a Bullmastiff
/data/dog_images/train/103.Mastiff/Mastiff_06871.jpg
```



Problem Statement

The problem that we are trying to solve is complicated – I needed 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 Convoluted Neural Network that infers dog breed. There are many points of possible failure, and no perfect algorithm exists.

Even a human would have trouble distinguishing between some breeds – for example a Brittany and a Welsh Springer Spaniel.

Random chance presents an exceptionally low bar - a random guess will provide a correct answer roughly 1 in 133 times (from 133 different dog breeds in the dataset), which corresponds to an accuracy of less than 1%.

There are several steps that I followed to complete the project:

Step 0: Import Datasets – The dataset that I used to train the model is provided by Udacity.

Step 1: A model to Detect Humans – I used OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

Step 2: A model to Detect Dogs – I used a pre-trained VGG-16 model to detect dogs in images, along with weights that have been trained on ImageNet.

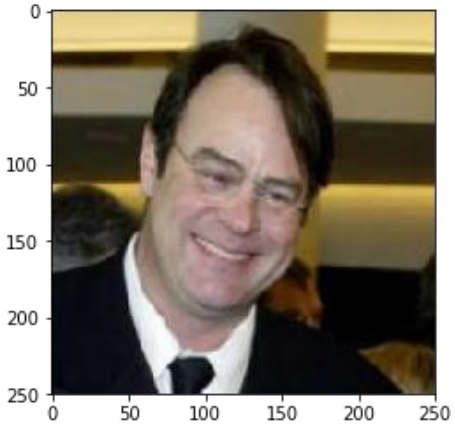
Step 3: Create a CNN to Classify Dog Breeds (from Scratch) – I created a CNN that classifies dog breeds by myself, aiming for at least 10% accuracy.

Step 4: Create a CNN to Classify Dog Breeds (using Transfer Learning) – I added an already trained ResNet50 model to mine, to improve the accuracy to at least 60%.

Step 5: Write an Algorithm – I wrote an algorithm that accepts a file path to an image and determines whether the image contains a human, dog, or neither.

Step 6: Test the Algorithm – I tested the algorithm with several photos to check if everything works as intended. As mentioned above, we are looking for the following results:

```
You are a human, but you look like a Basenji
/data/lfw/Dan_Ackroyd/Dan_Ackroyd_0001.jpg
```

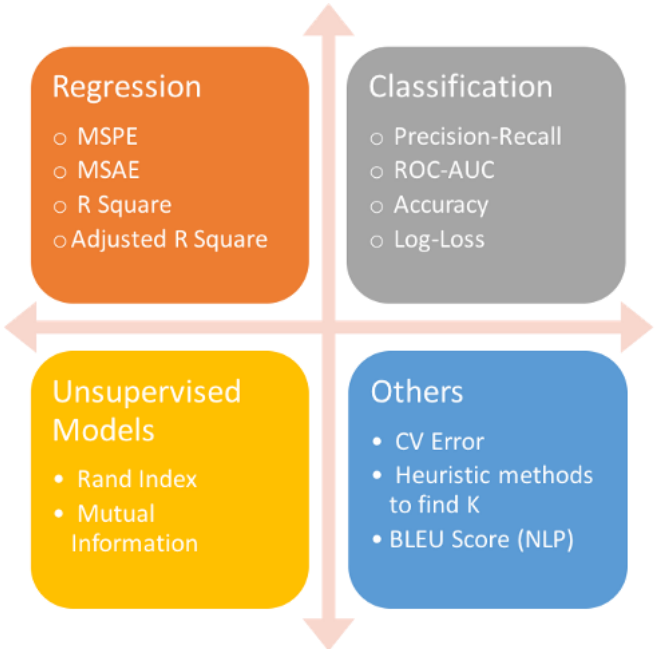



```
You are a dog - a Bullmastiff
/data/dog_images/train/103.Mastiff/Mastiff_06833.jpg
```

Metrics

There are several good choices for evaluating classification models:[1]

[1] https://www.kdnuggets.com/2018/04/right-metric-evaluating-machine-learning-models-1.html



Precision is a valid choice of evaluation metric when we want to be very sure of our prediction. Recall - when we want to capture as many positives as possible.[2]

I evaluated the models by their accuracy, since the data is diverse and balanced. Accuracy is one of the most convenient and easy to understand metrics – this is the percentage of correctly classified humans or dog breeds, out of all images provided.

$$accuracy = \frac{number\ of\ pictures\ classified\ correctly}{total\ number\ of\ pictures}$$

II. Analysis

Data Exploration and Visualization

Udacity provides a labeled dataset of 8351 total dog images in 133 categories (breeds). The data is prepped for the project and no significant changes or augmentation is required.

There are also 13233 total human images, that I will use to test the Human face detector model.

[2] https://towardsdatascience.com/the-5-classification-evaluation-metrics-you-must-know-aa97784ff226

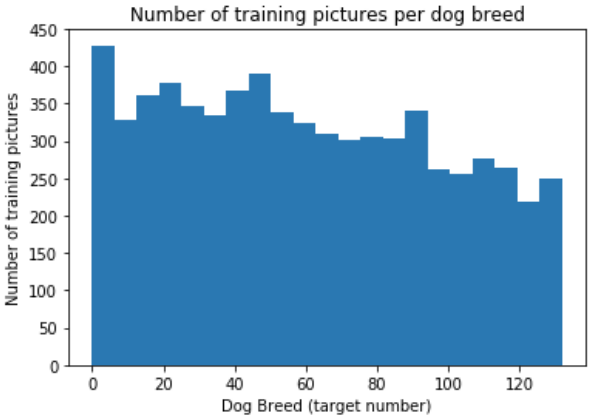
The data is divided as follows:

Training: 6680 training dog images.

Validation: 835 validation dog images.

Test: 836 test dog images.

This is a visualization of the number of images that we have for the different dog breeds (133). The data is slightly skewed but balanced enough for our purposes.



Algorithms, Techniques and Benchmark

As mentioned above, I used several different algorithms and techniques to solve the problem.

1. I used 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.

   Object Detection using Haar feature-based cascade classifiers is an effective object detection method proposed by Paul Viola and Michael Jones in their paper, "Rapid Object Detection using a Boosted Cascade of Simple Features" in 2001. It is a machine learning based approach where a cascade function is trained from a lot of positive and negative images. It is then used to detect objects in other images.[3]

2. I used a pre-trained, VGG-16 model to detect dogs in images. The model comes along with weights that have been trained on ImageNet, a very large, very popular dataset used for image classification and other vision tasks.

[3] https://docs.opencv.org/trunk/db/d28/tutorial_cascade_classifier.html

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 VGG-16 model returns a prediction (derived from the 1000 possible categories in ImageNet) for the object that is contained in the image.

3. As a benchmark model, I created a Convolutional Neural Network model from scratch and aimed for at least 10% accuracy. As we mentioned, the task of assigning breed to dogs from images is considered exceptionally challenging.

4. I added an already trained ResNet50 model to mine, from the torchvision.models subpackage – it contains definitions of models for addressing different tasks, including: image classification, pixelwise semantic segmentation, object detection, instance segmentation, person keypoint detection and video classification. This improved the accuracy to 60% with a small amount of training.

5. I wrote and tested the algorithm, which uses the different models to read images and output humans and dog breeds.

III. Methodology

Data Preprocessing

The data is already divided into train, validation and test set; however, the images are of different sizes.

I used RandomResizedCrop to resize the images to 224 x 224, since this seems to be a common choice in similar problems.

Then converted them into a 3D tensor with shape 224 x 224 x 3. The 3 comes from the fact that we are using color images, which have 3 layers.

I created data loaders for the train, validation and test sets.

Further augmentation and randomization of the data is not needed for this project.

Implementation

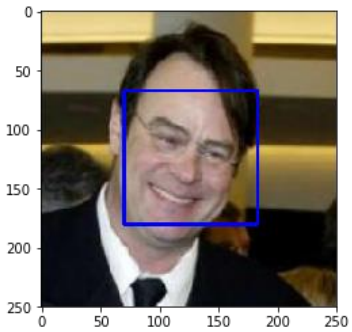
I implemented the process described above:

Step 1: Detect Humans

In this section, I used OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images – the images are loaded, converted to greyscale and used in the classifier – it returns True if a face is detected.

```
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()
```

Number of faces detected: 1



```python
# 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
```

Step 2: Detect Dogs

In this section, we use a pre-trained model to detect dogs in images. Given an image, this pre-trained VGG-16 model returns a prediction (derived from the 1000 possible categories in ImageNet) for the object that is contained in the image.

Looking at the dictionary[4], we can see 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 VGG-16 model, we need only check if the pre-trained model predicts an index between 151 and 268 (inclusive).

```python
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()
```

[4] https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a

```python
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

    img = Image.open(img_path)

    transform_pipeline = transforms.Compose([transforms.RandomResizedCrop(250),transforms.ToTensor()])

    tensor = transform_pipeline(img)

    tensor = tensor.unsqueeze(0)

    if torch.cuda.is_available():
        tensor = tensor.cuda()

    prediction = VGG16(tensor)

    if torch.cuda.is_available():
        prediction = prediction.cpu()

    ## Return the *index* of the predicted class for that image

    index = prediction.data.numpy().argmax()

    return index # predicted class index
```

```python
### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    ## TODO: Complete the function.
    index = VGG16_predict(img_path)

    return (151 <= index and index <= 268) # true/false
```

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

Specify Data Loaders for the Dog Dataset:

```
import os
from torchvision import datasets
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True
### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes


transform1 = transforms.Compose([transforms.RandomResizedCrop(224),transforms.ToTensor()])

train_data = datasets.ImageFolder('/data/dog_images/train', transform=transform1)
valid_data = datasets.ImageFolder('/data/dog_images/valid', transform=transform1)
test_data = datasets.ImageFolder('/data/dog_images/test', transform=transform1)

batch_size = 10

train_loader = torch.utils.data.DataLoader(train_data,
                                           batch_size=batch_size,
                                           shuffle=True)
valid_loader = torch.utils.data.DataLoader(valid_data,
                                           batch_size=batch_size,
                                           shuffle=False)
test_loader = torch.utils.data.DataLoader(test_data,
                                           batch_size=batch_size,
                                           shuffle=False)
loaders_scratch = {
    'train': train_loader,
    'valid': valid_loader,
    'test': test_loader
}
```

Create a CNN to classify dog breed:

I created 3 convolutional layers with kernel size 3 and padding 1. Each layer has double the depth of the previous one.

I used MaxPool layers to halve the height and width of the CLs. The purpose of pooling layers in CNN is to reduce or downsample the dimensionality of the input image. Pooling layers make feature detection independent of noise and small changes like image rotation or tilting. This property is known as "spatial variance."[5]

The CLs are applied first, a MaxPooling layer after each one, activated by a ReLU function. The rectified linear activation function is a piecewise linear function that will output the input directly if is positive, otherwise, it will output zero. It has become the default activation function for many types of neural networks because a model that uses it is easier to train and often achieves better performance.[6]

The output is flattened and passed to the 2 fully connected layers, which output the 133 dog categories.

[5] https://missinglink.ai/guides/tensorflow/tensorflow-maxpool-working-cnn-max-pooling-layers-tensorflow/
[6] https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/

```python
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
        self.conv1 = nn.Conv2d(3, 32, 3, padding=1)
        self.norm2d1 = nn.BatchNorm2d(32)
        self.conv2 = nn.Conv2d(32, 64, 3, padding=1)
        self.conv3 = nn.Conv2d(64, 128, 3, padding=1)

        self.pool = nn.MaxPool2d(2, 2)

        size_linear_layer = 500

        self.fc1 = nn.Linear(128 * 28 * 28, size_linear_layer)
        self.fc2 = nn.Linear(size_linear_layer, 133)

    def forward(self, x):
        ## Define forward behavior
        x = self.pool(F.relu(self.norm2d1(self.conv1(x))))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))

        # flatten image input
        x = x.view(-1, 128 * 28 * 28)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

#-#-# 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()
```

Specify Loss Function and Optimizer:

```
import torch.optim as optim

### TODO: select loss function
criterion_scratch = nn.CrossEntropyLoss()

### TODO: select optimizer
optimizer_scratch = optim.SGD(model_scratch.parameters(), lr=0.01)

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

Refinement

To drastically improve the performance of the model – we can use transfer learning.

I used a pre-trained ResNet50 model, since it has achieved very high results for image classification – it is trained on the ImageNet data set for classifying images into one of 1000 categories or classes.

Our dog dataset is relatively small and similar (ImageNet has dog breeds), so I froze the parameters for the pretrained model and changed the final layer to output 133 classes – the dog breeds that we are trying to predict.

I specified the loss function, the optimizer and trained and validated the model.

Model Architecture:

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

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

for par in model_transfer.parameters():
    par.requires_grad = False

model_transfer.fc = nn.Linear(2048, 133, bias=True)

fc_parameters = model_transfer.fc.parameters()

for param in fc_parameters:
    param.requires_grad = True

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

```
Downloading: "https://download.pytorch.org/models/resnet50-19c8e357.pth"
100%|██████████| 102502400/102502400 [00:01<00:00, 92303513.04it/s]
```

Predicting Dog Breed with the Model:

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

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

data_transfer = loaders_transfer.copy()

class_names = [item[4:].replace("_", " ") for item in data_transfer['train'].dataset.classes]

def predict_breed_transfer(img_path):
    global model_transfer
    transform_pipeline = transforms.Compose([transforms.Resize(size=(224, 224)),
                                     transforms.ToTensor()])

    image = Image.open(img_path).convert('RGB')

    image = transform_pipeline(image)[:3,:,:].unsqueeze(0)

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

    model_transfer.eval()
    idx = torch.argmax(model_transfer(image))
    return class_names[idx]
```

Writing the Algorithm:

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

def run_app(img_path):
    ## handle cases for a human face, dog, and neither
    if face_detector(img_path) > 0:
        breed = predict_breed_transfer(img_path)
        print('You are a human, but you look like a ' + breed)
    elif dog_detector(img_path):
        breed = predict_breed_transfer(img_path)
        print('You are a dog - a ' + breed)
    else:
        print('Not Dog, Neither Human')
```

IV. Results

Model Evaluation and Validation

Assessing the Human Face Detector:

98% of human images have a face detected. 17% of dog images have a face detected.

```
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.
human1 = np.average([face_detector(i) for i in tqdm(human_files_short)])
dog1 = np.average([face_detector(i) for i in tqdm(dog_files_short)])
print(human1)
print(dog1)
```

```
100%|██████████| 100/100 [00:02<00:00, 35.10it/s]
100%|██████████| 100/100 [00:29<00:00,  7.32it/s]
0.98
0.17
```

Assessing the Dog Detector:

0 Humans detected as dogs. 76% Accurate dog detections.

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

human2 = np.average([dog_detector(img) for img in human_files_short])
dog2 = np.average([dog_detector(img) for img in dog_files_short])
print(human2,dog2)
```

```
0.0 0.76
```

Training and Validation for the CNN model:

```
def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
    """returns trained model"""
    # initialize tracker for minimum validation loss
    valid_loss_min = np.Inf

    for epoch in range(1, n_epochs+1):
        # initialize variables to monitor training and validation loss
        train_loss = 0.0
        valid_loss = 0.0

        ###################
        # train the model #
        ###################
        model.train()
        for batch_idx, (data, target) in enumerate(loaders['train']):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## find the loss and update the model parameters accordingly
            ## record the average training loss, using something like
            ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss))
            optimizer.zero_grad()
            output = model(data)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()
            train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss))
```

```
        ######################
        # validate the model #
        ######################
        model.eval()
        for batch_idx, (data, target) in enumerate(loaders['valid']):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## update the average validation loss
            output = model(data)
            loss = criterion(output, target)
            valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data - valid_loss))


        # print training/validation statistics
        print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
            epoch,
            train_loss,
            valid_loss
            ))

        ## TODO: save the model if validation loss has decreased
        if valid_loss < valid_loss_min:
            torch.save(model.state_dict(), save_path)
            print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'
                  .format(valid_loss_min, valid_loss))
            valid_loss_min = valid_loss

    # return trained model
    return model


# train the model
model_scratch = train(30, loaders_scratch, model_scratch, optimizer_scratch,
                      criterion_scratch, use_cuda, 'model_scratch.pt')

# load the model that got the best validation accuracy
model_scratch.load_state_dict(torch.load('model_scratch.pt'))
```

Testing the CNN – aiming for 10% accuracy or more:

```python
def test(loaders, model, criterion, use_cuda):

    # monitor test loss and accuracy
    test_loss = 0.
    correct = 0.
    total = 0.

    model.eval()
    for batch_idx, (data, target) in enumerate(loaders['test']):
        # move to GPU
        if use_cuda:
            data, target = data.cuda(), target.cuda()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the loss
        loss = criterion(output, target)
        # update average test loss
        test_loss = test_loss + ((1 / (batch_idx + 1)) * (loss.data - test_loss))
        # convert output probabilities to predicted class
        pred = output.data.max(1, keepdim=True)[1]
        # compare predictions to true label
        correct += np.sum(np.squeeze(pred.eq(target.data.view_as(pred))).cpu().numpy())
        total += data.size(0)

    print('Test Loss: {:.6f}\n'.format(test_loss))

    print('\nTest Accuracy: %2d%% (%2d/%2d)' % (
        100. * correct / total, correct, total))

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

```
Test Loss: 4.140642


Test Accuracy: 11% (94/836)
```

The above 10% accuracy that we aimed for was achieved with around 30-40 epochs.

Training and validating the transferred ResNet model with our data:

```python
def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
    """returns trained model"""
    # initialize tracker for minimum validation loss
    valid_loss_min = np.Inf

    for epoch in range(1, n_epochs+1):
        # initialize variables to monitor training and validation loss
        train_loss = 0.0
        valid_loss = 0.0

        ###################
        # train the model #
        ###################
        model.train()
        for batch_idx, (data, target) in enumerate(loaders['train']):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## find the loss and update the model parameters accordingly
            ## record the average training loss, using something like
            ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss))

            optimizer.zero_grad()
            # forward pass: compute predicted outputs by passing inputs to the model
            output = model(data)
            # calculate the batch loss
            loss = criterion(output, target)
            # backward pass: compute gradient of the loss with respect to model parameters
            loss.backward()
            # perform a single optimization step (parameter update)
            optimizer.step()

            train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss))

            #if batch_idx % 100 == 0:
            #    print('Epoch %d, Batch %d loss: %.6f' % (epoch, batch_idx + 1, train_loss))
```

```
        ######################
        # validate the model #
        ######################
        model.eval()
        for batch_idx, (data, target) in enumerate(loaders['valid']):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## update the average validation loss
            output = model(data)
            loss = criterion(output, target)
            valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data - valid_loss))

        # print training/validation statistics
        print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
            epoch,
            train_loss,
            valid_loss
            ))

        ## TODO: save the model if validation loss has decreased
        if valid_loss < valid_loss_min:
            torch.save(model.state_dict(), save_path)
            print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'
                  .format(valid_loss_min, valid_loss))
            valid_loss_min = valid_loss

    # return trained model
    return model

n_epochs = 20

# train the model
model_transfer = train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer, criterion_transfer,
                       use_cuda, 'model_transfer.pt')

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

The training is progressing, and validation loss is decreasing steadily after 20 epochs:

```
Epoch: 18       Training Loss: 2.027824         Validation Loss: 1.636924
Epoch: 19       Training Loss: 1.980940         Validation Loss: 1.569956
Validation loss decreased (1.633794 --> 1.569956).  Saving model ...
Epoch: 20       Training Loss: 1.917797         Validation Loss: 1.511585
Validation loss decreased (1.569956 --> 1.511585).  Saving model ...
```

The transferred model seems to be doing well on the new data.

Justification

Testing the transferred ResNet model:

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

```
Test Loss: 1.547791


Test Accuracy: 69% (579/836)
```

The 60 percent accuracy that we aimed for are achieved – we can consider those results good enough for a consumer web or mobile app, that is intended only for fun.

As we can see, the transferred model achieves much better accuracy, compared to the simple CNN, even with less training.

That is normal and expected, since the ResNet model is much deeper and had much more training.

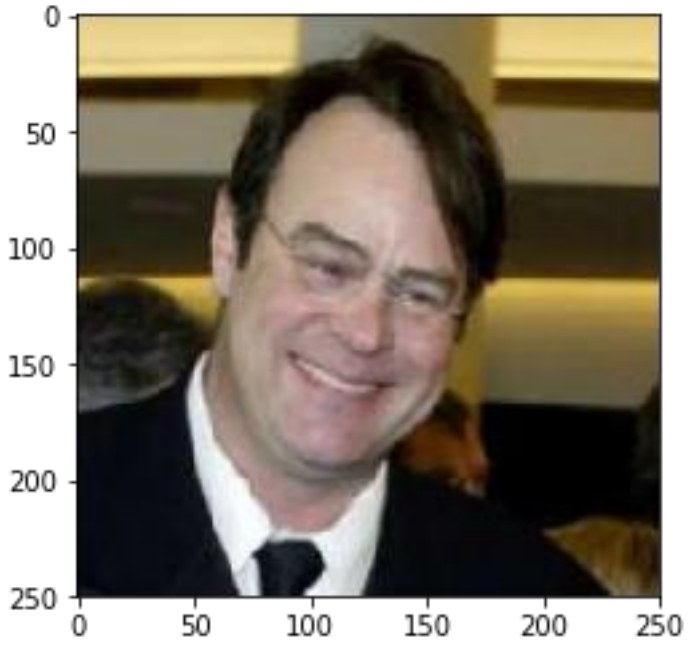
V. Conclusion

Free-Form Visualization

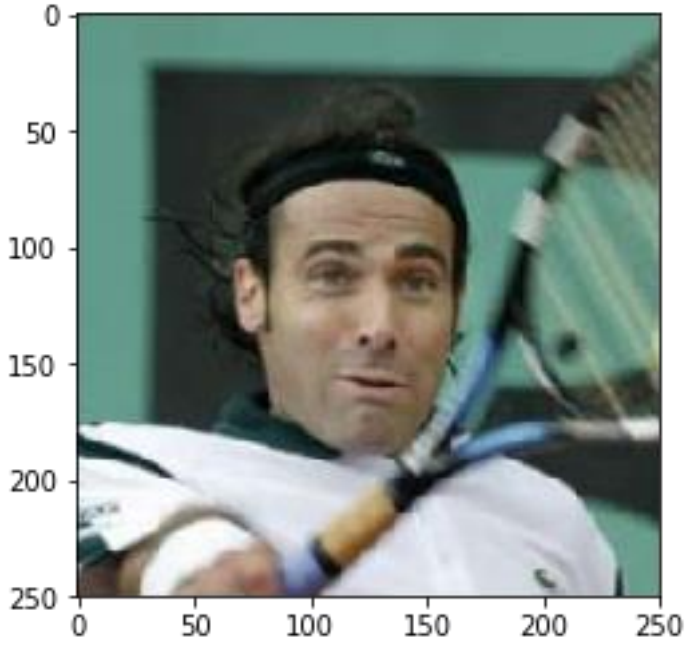
Testing the algorithm:

```
## suggested code, below
for file in np.hstack((human_files[:3], dog_files[:3])):
    run_app(file)
    print(file)
    imag = Image.open(file)
    plt.imshow(imag)
    plt.show()
```

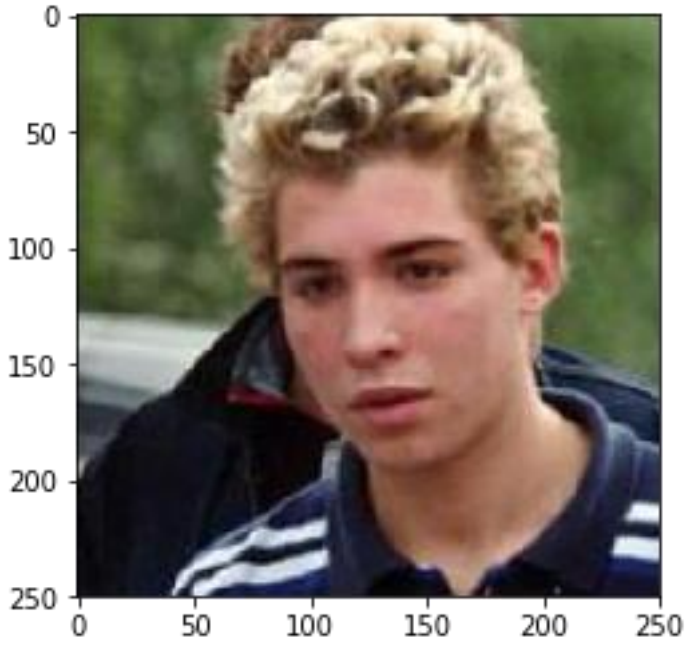
```
You are a human, but you look like a Basenji
/data/lfw/Dan_Ackroyd/Dan_Ackroyd_0001.jpg
```




```
You are a human, but you look like a Papillon
/data/lfw/Alex_Corretja/Alex_Corretja_0001.jpg
```

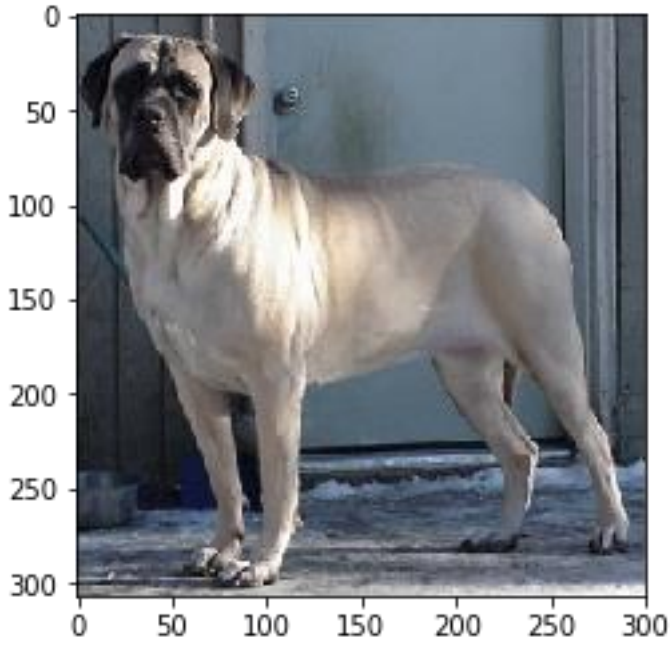



```
You are a human, but you look like a Cane corso
/data/lfw/Daniele_Bergamin/Daniele_Bergamin_0001.jpg
```

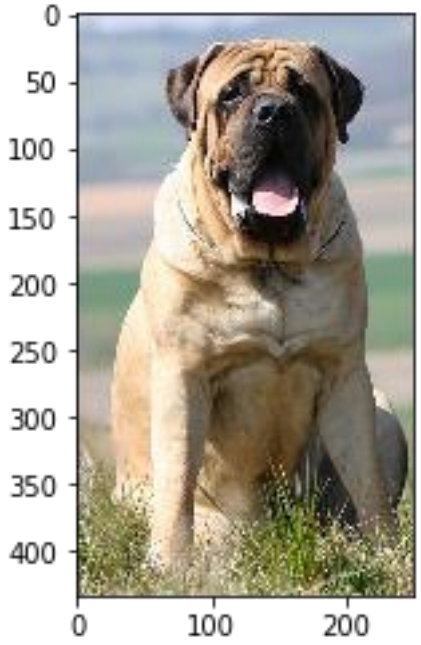


You are a dog - a Bullmastiff
/data/dog_images/train/103.Mastiff/Mastiff_06833.jpg

You are a dog - a Mastiff
/data/dog_images/train/103.Mastiff/Mastiff_06826.jpg




```
You are a dog - a Bullmastiff
/data/dog_images/train/103.Mastiff/Mastiff_06871.jpg
```



We can see that the algorithm works as intended – identifies dog breeds correctly and gives a breed estimation for humans, which may make the app more fun.

Reflection

Project summary:

1. Clearly define a question that I want to answer
2. Obtain data for training and testing a model
3. Select the appropriate models and metrics to evaluate them
4. Process the data and train the models
5. Evaluate the results

I found the whole project very interesting, since this is my first one with image recognition and it is good representation of tackling it.

The steps are basic, and even with the template code, the project was difficult for me and I had to refer to Udacity's additional materials for CNNs to complete it. Since they reviewed similar problems, I was able to use the information for this project, build a CNN and use an already trained model.

The final solution works as expected and seems to be a good choice for problems like these – it can easily be used to create consumer apps and games that don't require very accurate results.

Improvement

Every step of this project can further be improved to obtain better results for the classification, since this is only a basic overview of the image recognition workflow.

There are several easy ways to improve the models:

1. More training for the ResNet model - the validation error decreases steadily, but for the project purposes, 20 epochs are enough.
2. Additional data or augmenting our dataset.
3. Improve or select other human and dog detection models.

I will explore those points further in future projects.
