dog_app

November 17, 2020

0.1 Machine Learning Engineer Nanodegree

0.2 Capstone Project

Shikher Srivastava

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

In this notebook, some template code has already been provided for you, and you will need to implement additional functionality to successfully complete this project. You will not need to modify the included code beyond what is requested. Sections that begin with **'(IMPLEMENTATION)'** in the header indicate that the following block of code will require additional functionality which you must provide. Instructions will be provided for each section, and the specifics of the implementation are marked in the code block with a 'TODO' statement. Please be sure to read the instructions carefully!

> **Note**: Once you have completed all of the code implementations, you need to finalize your work by exporting the Jupyter Notebook as an HTML document. Before exporting the notebook to html, all of the code cells need to have been run so that reviewers can see the final implementation and output. You can then export the notebook by using the menu above and navigating to **File -> Download as -> HTML (.html)**. Include the finished document along with this notebook as your submission.

In addition to implementing code, there will be questions that you must answer which relate to the project and your implementation. Each section where you will answer a question is preceded by a **'Question X'** header. Carefully read each question and provide thorough answers in the following text boxes that begin with **'Answer:'**. Your project submission will be evaluated based on your answers to each of the questions and the implementation you provide.

> **Note:** Code and Markdown cells can be executed using the **Shift + Enter** keyboard shortcut. Markdown cells can be edited by double-clicking the cell to enter edit mode.

The rubric contains *optional* "Stand Out Suggestions" for enhancing the project beyond the minimum requirements. If you decide to pursue the "Stand Out Suggestions", you should include the code in this Jupyter notebook.

Step 0: Import Datasets

Make sure that you've downloaded the required human and dog datasets:

Note: if you are using the Udacity workspace, you *DO NOT* need to re-download these - they can be found in the `/data` folder as noted in the cell below.

- Download the dog dataset. Unzip the folder and place it in this project's home directory, at the location `/dog_images`.
- Download the human dataset. Unzip the folder and place it in the home directory, at location `/lfw`.

Note: If you are using a Windows machine, you are encouraged to use 7zip to extract the folder.

In the code cell below, we save the file paths for both the human (LFW) dataset and dog dataset in the numpy arrays `human_files` and `dog_files`.

```
In [1]: 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.
```

Step 1: Detect Humans

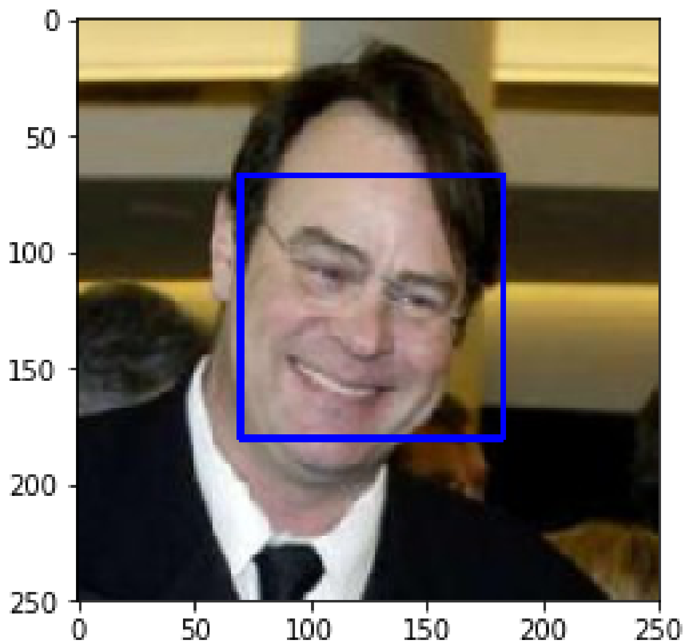
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

OpenCV provides many pre-trained face detectors, stored as XML files on github. We have downloaded one of these detectors and stored it in the `haarcascades` directory. In the next code cell, we demonstrate how to use this detector to find human faces in a sample image.

```
In [2]: 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)
```

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

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

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

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

# display the image, along with bounding box
plt.imshow(cv_rgb)
plt.show()
```

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



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

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

1.1.1 Write a Human Face Detector

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

```
In [3]: # 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
```

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

Question 1: Use the code cell below to test the performance of the `face_detector` function.
- What percentage of the first 100 images in `human_files` have a detected human face?
- What percentage of the first 100 images in `dog_files` have a detected human face?

Ideally, we would like 100% of human images with a detected face and 0% of dog images with a detected face. You will see that our algorithm falls short of this goal, but still gives acceptable performance. We extract the file paths for the first 100 images from each of the datasets and store them in the numpy arrays `human_files_short` and `dog_files_short`.

Answer: (You can print out your results and/or write your percentages in this cell)

```
In [4]: from PIL import Image
        from tqdm import tqdm

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

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

        fd_human = [face_detector(fname) for fname in human_files_short]
        percentage_human = 100*sum([f>0 for f in fd_human])/len(fd_human)
        print(percentage_human,'% of the first 100 images in human_files had a human face detect

        fd_dog = [face_detector(fname) for fname in dog_files_short]
        percentage_dog = 100*sum([f>0 for f in fd_dog])/len(fd_dog)
        print(percentage_dog,'% of the first 100 images in dog_files had a human face detected i
```

```
98.0 % of the first 100 images in human_files had a human face detected in them.
17.0 % of the first 100 images in dog_files had a human face detected in them.
```

We suggest the face detector from OpenCV as a potential way to detect human images in your algorithm, but you are free to explore other approaches, especially approaches that make use of deep learning :). Please use the code cell below to design and test your own face detection algorithm. If you decide to pursue this *optional* task, report performance on `human_files_short` and `dog_files_short`.

```
In [5]: ### (Optional)
        ### TODO: Test performance of anotherface detection algorithm.
        ### Feel free to use as many code cells as needed.
```

Step 2: Detect Dogs
In this section, we use a pre-trained model to detect dogs in images.

1.1.3 Obtain Pre-trained VGG-16 Model

The code cell below downloads the VGG-16 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 [6]: 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()
```

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [01:48<00:00, 5082917.85it/s]
```

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.

1.1.4 (IMPLEMENTATION) Making Predictions with a Pre-trained Model

In the next code cell, you will write a function that accepts a path to an image (such as `'dogImages/train/001.Affenpinscher/Affenpinscher_00001.jpg'`) as input and returns the index corresponding to the ImageNet class that is predicted by the pre-trained VGG-16 model. The output should always be an integer between 0 and 999, inclusive.

Before writing the function, make sure that you take the time to learn how to appropriately pre-process tensors for pre-trained models in the PyTorch documentation.

```
In [7]: 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

            # from https://github.com/pytorch/examples/blob/42e5b996718797e45c46a25c55b031e6768f
            transform_pipeline = transforms.Compose([transforms.RandomResizedCrop(224),
                                                     transforms.RandomHorizontalFlip(),
                                                     transforms.ToTensor(),
                                                     transforms.Normalize(mean=[0.485, 0.456, 0.
            img = Image.open(img_path)
            img = transform_pipeline(img).unsqueeze(0)

            # solution to cuda issue found here https://discuss.pytorch.org/t/runtimeerror-expec
            if torch.cuda.is_available(): img = img.cuda()
            pred = VGG16(img)
            # used technique to get predictions from here: https://discuss.pytorch.org/t/making-
            if torch.cuda.is_available(): pred = pred.cpu().data.numpy().argmax()

            return pred  # predicted class index
```

1.1.5 (IMPLEMENTATION) Write a Dog Detector

While looking at the dictionary, you will notice that the categories corresponding to dogs appear in an uninterrupted sequence and correspond to dictionary keys 151-268, inclusive, to include all categories from `'Chihuahua'` to `'Mexican hairless'`. Thus, in order to check to see if an image is predicted to contain a dog by the pre-trained VGG-16 model, we need only check if the pre-trained model predicts an index between 151 and 268 (inclusive).

Use these ideas to complete the `dog_detector` function below, which returns `True` if a dog is detected in an image (and `False` if not).

```
In [8]: ### returns "True" if a dog is detected in the image stored at img_path
        def dog_detector(img_path):
            ## TODO: Complete the function.
```

```
        preds = VGG16_predict(img_path)

        return (preds >= 151) & (preds <= 268) # true/false
```

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

Question 2: Use the code cell below to test the performance of your `dog_detector` function.
- What percentage of the images in `human_files_short` have a detected dog?
- What percentage of the images in `dog_files_short` have a detected dog?
Answer:

```
In [9]: ### TODO: Test the performance of the dog_detector function
        ### on the images in human_files_short and dog_files_short.
        dog_found = []
        for img_path in dog_files_short:
            dog_found.append(dog_detector(img_path))
        print('In the dog images we detected dogs in {}% of images.'.format(sum(dog_found)/1))

        dog_found = []
        for img_path in human_files_short:
            dog_found.append(dog_detector(img_path))
        sum(dog_found)
        print('In the human images we detected dogs in {}% of images.'.format(sum(dog_found)/1))
```

```
In the dog images we detected dogs in 99.0% of images.
In the human images we detected dogs in 5.0% of images.
```

We suggest VGG-16 as a potential network to detect dog images in your algorithm, but you are free to explore other pre-trained networks (such as Inception-v3, ResNet-50, etc). Please use the code cell below to test other pre-trained PyTorch models. If you decide to pursue this *optional* task, report performance on `human_files_short` and `dog_files_short`.

```
In [10]: ### (Optional)
         ### TODO: Report the performance of another pre-trained network.
         ### Feel free to use as many code cells as needed.
```

Step 3: Create a CNN to Classify Dog Breeds (from Scratch)
Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN that classifies dog breeds. You must create your CNN *from scratch* (so, you can't use transfer learning *yet*!), and you must attain a test accuracy of at least 10%. In Step 4 of this notebook, you will have the opportunity to use transfer learning to create a CNN that attains greatly improved accuracy.

We mention that the task of assigning breed to dogs from images is considered exceptionally challenging. To see why, consider that *even a human* would have trouble distinguishing between a Brittany and a Welsh Springer Spaniel.

Brittany	Welsh Springer Spaniel

It is not difficult to find other dog breed pairs with minimal inter-class variation (for instance, Curly-Coated Retrievers and American Water Spaniels).

Curly-Coated Retriever	American Water Spaniel

Likewise, recall that labradors come in yellow, chocolate, and black. Your vision-based algorithm will have to conquer this high intra-class variation to determine how to classify all of these different shades as the same breed.

Yellow Labrador	Chocolate Labrador

We also mention that random chance presents an exceptionally low bar: setting aside the fact that the classes are slightly imabalanced, a random guess will provide a correct answer roughly 1 in 133 times, which corresponds to an accuracy of less than 1%.

Remember that the practice is far ahead of the theory in deep learning. Experiment with many different architectures, and trust your intuition. And, of course, have fun!

1.1.7 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

Use the code cell below to write three separate data loaders for the training, validation, and test datasets of dog images (located at `dog_images/train`, `dog_images/valid`, and `dog_images/test`, respectively). You may find this documentation on custom datasets to be a useful resource. If you are interested in augmenting your training and/or validation data, check out the wide variety of transforms!

```
In [11]: import os
         import torch
         from torchvision import datasets
         import torchvision.transforms as transforms

         ### TODO: Write data loaders for training, validation, and test sets
         ## Specify appropriate transforms, and batch_sizes
         data_dir = '/data/dog_images'


         train_transforms = transforms.Compose([transforms.RandomRotation(30),
                                                transforms.Resize(256),
                                                transforms.CenterCrop(224),
                                                transforms.RandomHorizontalFlip(),
                                                transforms.ToTensor(),
                                                transforms.Normalize([0.5, 0.5, 0.5],
```

```
                                                             [0.5, 0.5, 0.5])])


transform = transforms.Compose([transforms.Resize(256),
                                transforms.CenterCrop(224),
                                transforms.ToTensor(),
                                transforms.Normalize([0.5, 0.5, 0.5],
                                                     [0.5, 0.5, 0.5])])

dataset_train = datasets.ImageFolder(data_dir + '/train', transform=train_transforms)
dataset_valid = datasets.ImageFolder(data_dir + '/valid', transform=transform)
dataset_test = datasets.ImageFolder(data_dir + '/test', transform=transform)


dataloader_train = torch.utils.data.DataLoader(dataset_train, batch_size=128, shuffle=T
dataloader_valid = torch.utils.data.DataLoader(dataset_valid, batch_size=128, shuffle=T
dataloader_test = torch.utils.data.DataLoader(dataset_test, batch_size=128, shuffle=Tru
```

Question 3: Describe your chosen procedure for preprocessing the data. - How does your code resize the images (by cropping, stretching, etc)? What size did you pick for the input tensor, and why?

- Did you decide to augment the dataset? If so, how (through translations, flips, rotations, etc)? If not, why not?

Answer:
I used transforms.Resize()
I also added random horizontal flips as it doesn't define a dog breed, but may help for the training

1.1.8 (IMPLEMENTATION) Model Architecture

Create a CNN to classify dog breed. Use the template in the code cell below.

```
In [12]: 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, 16, 3, padding = 1)
                 self.conv2 = nn.Conv2d(16, 32, 3, padding = 1)
                 self.conv3 = nn.Conv2d(32, 64, 3, padding = 1)
                 self.conv4 = nn.Conv2d(64, 128, 3, padding = 1)
```

```
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(128*14*14, 1024)
        self.fc2 = nn.Linear(1024, 512)
        self.fc3 = nn.Linear(512, 133)
        self.dropout = nn.Dropout(0.25)

    def forward(self, x):
        ## Define forward behavior

        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = self.pool(F.relu(self.conv4(x)))

        #print(x.shape)
        x = x.view(-1, 128*14*14)

        x = self.dropout(x)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(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()
```

Question 4: Outline the steps you took to get to your final CNN architecture and your reasoning at each step.

Answer:

I tried various options before arrive at this, My proces trial and error method to reach a model with a accuracy greater than 10 percent on test set

I tried using 2 convolution layer and 1 fully connected layer Tried more deep models , though was was concerned that having too many layers would be computationally expensive I tried 3 convolution layer and 2 fully connecgted layer, accuracy improved I tried using 4 convolution layer , output channel 14 and 3 fully connected layer Optimizer set to adam Learning rate set to 0.001 Batch size as 128

Reached at acccuracy of 11% on test set

1.1.9 (IMPLEMENTATION) Specify Loss Function and Optimizer

Use the next code cell to specify a loss function and optimizer. Save the chosen loss function as `criterion_scratch`, and the optimizer as `optimizer_scratch` below.

```
In [13]: import torch.optim as optim

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

         ### TODO: select optimizer
         optimizer_scratch = optim.Adam(model_scratch.parameters(), lr=0.001)
```

1.1.10 (IMPLEMENTATION) Train and Validate the Model

Train and validate your model in the code cell below. Save the final model parameters at filepath `'model_scratch.pt'`.

```
In [14]: 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

                     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


                with torch.no_grad():
                    output = model(data)

                #calculate batch loss - valid
                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:
                print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.fo
                valid_loss_min,
                valid_loss))
                torch.save(model.state_dict(), save_path)
                valid_loss_min = valid_loss

        # return trained model
        return model
```

In [15]:
```
loaders_scratch = {'train': dataloader_train,
                   'test': dataloader_test,
                   'valid': dataloader_valid}
```

In [17]:
```
# train the model
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True
```

```
model_scratch = train(20, 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'))
```

```
Epoch: 1 	Training Loss: 4.823985 	Validation Loss: 4.694880
Validation loss decreased (inf --> 4.694880).  Saving model ...
Epoch: 2 	Training Loss: 4.604432 	Validation Loss: 4.552750
Validation loss decreased (4.694880 --> 4.552750).  Saving model ...
Epoch: 3 	Training Loss: 4.466523 	Validation Loss: 4.449688
Validation loss decreased (4.552750 --> 4.449688).  Saving model ...
Epoch: 4 	Training Loss: 4.300224 	Validation Loss: 4.321225
Validation loss decreased (4.449688 --> 4.321225).  Saving model ...
Epoch: 5 	Training Loss: 4.115963 	Validation Loss: 4.163973
Validation loss decreased (4.321225 --> 4.163973).  Saving model ...
Epoch: 6 	Training Loss: 3.978287 	Validation Loss: 4.197807
Epoch: 7 	Training Loss: 3.872001 	Validation Loss: 4.070243
Validation loss decreased (4.163973 --> 4.070243).  Saving model ...
Epoch: 8 	Training Loss: 3.753901 	Validation Loss: 3.989006
Validation loss decreased (4.070243 --> 3.989006).  Saving model ...
Epoch: 9 	Training Loss: 3.645711 	Validation Loss: 3.895705
Validation loss decreased (3.989006 --> 3.895705).  Saving model ...
Epoch: 10 	Training Loss: 3.526536 	Validation Loss: 3.902088
Epoch: 11 	Training Loss: 3.409128 	Validation Loss: 3.823171
Validation loss decreased (3.895705 --> 3.823171).  Saving model ...
Epoch: 12 	Training Loss: 3.278212 	Validation Loss: 3.943214
Epoch: 13 	Training Loss: 3.169508 	Validation Loss: 3.908761
Epoch: 14 	Training Loss: 3.007832 	Validation Loss: 3.971845
Epoch: 15 	Training Loss: 2.905879 	Validation Loss: 3.936643
Epoch: 16 	Training Loss: 2.742069 	Validation Loss: 4.002162
Epoch: 17 	Training Loss: 2.568454 	Validation Loss: 4.207106
Epoch: 18 	Training Loss: 2.433034 	Validation Loss: 4.196708
Epoch: 19 	Training Loss: 2.215789 	Validation Loss: 4.294112
Epoch: 20 	Training Loss: 2.107224 	Validation Loss: 4.398666
```

1.1.11 (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 10%.

```
In [18]: model_scratch.load_state_dict(torch.load('model_scratch.pt'))
```

```
In [19]: 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))
```

```
In [20]: # call test function
         test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.845928


Test Accuracy: 11% (99/836)
```

Step 4: Create a CNN to Classify Dog Breeds (using Transfer Learning)
You will now use transfer learning to create a CNN that can identify dog breed from images. Your CNN must attain at least 60% accuracy on the test set.

1.1.12 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

Use the code cell below to write three separate data loaders for the training, validation, and test datasets of dog images (located at `dogImages/train`, `dogImages/valid`, and `dogImages/test`, respectively).

If you like, **you are welcome to use the same data loaders from the previous step**, when you created a CNN from scratch.

```
In [21]: ## TODO: Specify data loaders
         # using the same as in the previous step

         import os
         import torch
         from torchvision import datasets
         import torchvision.transforms as transforms

         ### TODO: Write data loaders for training, validation, and test sets
         ## Specify appropriate transforms, and batch_sizes
         data_dir = '/data/dog_images'

         train_transforms_transfer = transforms.Compose([transforms.RandomRotation(30),
                                                         transforms.Resize(255),
                                                         transforms.CenterCrop(224),
                                                         transforms.RandomHorizontalFlip(),
                                                         transforms.ToTensor(),
                                                         transforms.Normalize([0.485, 0.456, 0.4
                                                                               [0.229, 0.224, 0.22

         transform_transfer = transforms.Compose([transforms.Resize(255),
                                                  transforms.CenterCrop(224),
                                                  transforms.ToTensor(),
                                                  transforms.Normalize([0.485, 0.456, 0.406],
                                                                        [0.229, 0.224, 0.225])])

         dataset_train_transfer = datasets.ImageFolder(data_dir + '/train', transform=train_tran
         dataset_valid_transfer = datasets.ImageFolder(data_dir + '/valid', transform=transform_
         dataset_test_transfer = datasets.ImageFolder(data_dir + '/test', transform=transform_tr


         dataloader_train_transfer = torch.utils.data.DataLoader(dataset_train_transfer, batch_s
         dataloader_valid_transfer = torch.utils.data.DataLoader(dataset_valid_transfer, batch_s
         dataloader_test_transfer = torch.utils.data.DataLoader(dataset_test_transfer, batch_siz

In [22]: loaders_transfer = {'train': dataloader_train_transfer,
                             'test': dataloader_test_transfer,
                             'valid': dataloader_valid_transfer}
```

1.1.13 (IMPLEMENTATION) Model Architecture

Use transfer learning to create a CNN to classify dog breed. Use the code cell below, and save your initialized model as the variable `model_transfer`.

```
In [23]: import torchvision.models as models
         import torch.nn as nn

         ## TODO: Specify model architecture
```

```python
model_transfer = models.vgg16(pretrained=True) # using the model with low error accordi
# https://pytorch.org/docs/stable/torchvision/models.html
# reference: https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html
print(model_transfer)
```

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace)
    (2): Dropout(p=0.5)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace)
    (5): Dropout(p=0.5)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
```

)

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

Answer:

It is very efficient to use pre-trained networks to solve challenging problems in computer vision. Once trained, these models work very well as feature detectors for images they weren't trained on. Here we'll use transfer learning to train a network that can classify our dog photos.

Here, I chose the VGG16 model.

i thought the VGG16 is sutable for the current problem. Because it already trained large dataset.and it performed really well (came 2nd in imagenet classification competition) The fully connected layer was trained on the ImageNet dataset, so it won't work for the dog classification specific problem. That means we need to replace the classifier (133 classes), but the features will work perfectly on their own.

Hence, I selected VGG16 pre-trained model for transfer learning. I froze all feature parameters. I just changed the last fully connected layer output as 133 and trained classifier again. In this model, cross entropy loss criteria was selected because of making classification and Adam optimizer was selected.

```
In [24]: for param in model_transfer.features.parameters():
             param.requires_grad=False


         # I tried reconstruct the classifier but rather than do that, i decided to change just
         new_layer = nn.Linear(4096, 133)
         model_transfer.classifier[6] = new_layer

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

```
In [25]: print(model_transfer.classifier)
```

```
Sequential(
  (0): Linear(in_features=25088, out_features=4096, bias=True)
  (1): ReLU(inplace)
  (2): Dropout(p=0.5)
  (3): Linear(in_features=4096, out_features=4096, bias=True)
  (4): ReLU(inplace)
  (5): Dropout(p=0.5)
  (6): Linear(in_features=4096, out_features=133, bias=True)
)
```

```
In [26]: print(model_transfer)
```

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

```
    (1): ReLU(inplace)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace)
    (2): Dropout(p=0.5)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace)
    (5): Dropout(p=0.5)
    (6): Linear(in_features=4096, out_features=133, bias=True)
  )
)
```

1.1.14 (IMPLEMENTATION) Specify Loss Function and Optimizer

Use the next code cell to specify a loss function and optimizer. Save the chosen loss function as `criterion_transfer`, and the optimizer as `optimizer_transfer` below.

```
In [27]: import torch.optim as optim

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

         ### TODO: select optimizer
         # https://github.com/amdegroot/ssd.pytorch/issues/109
         optimizer_transfer = optim.Adam(model_transfer.classifier.parameters(), lr=0.001)
```

1.1.15 (IMPLEMENTATION) Train and Validate the Model

Train and validate your model in the code cell below. Save the final model parameters at filepath 'model_transfer.pt'.

```
In [28]: # train the model

         model_transfer = train(20, loaders_transfer, model_transfer, optimizer_transfer, criter

         # 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: 2.812788         Validation Loss: 1.231931
Validation loss decreased (inf --> 1.231931).  Saving model ...
Epoch: 2         Training Loss: 2.088157         Validation Loss: 1.193147
Validation loss decreased (1.231931 --> 1.193147).  Saving model ...
Epoch: 3         Training Loss: 2.062088         Validation Loss: 1.202094
Epoch: 4         Training Loss: 2.039228         Validation Loss: 1.129060
Validation loss decreased (1.193147 --> 1.129060).  Saving model ...
Epoch: 5         Training Loss: 1.899985         Validation Loss: 1.031925
Validation loss decreased (1.129060 --> 1.031925).  Saving model ...
Epoch: 6         Training Loss: 2.006771         Validation Loss: 0.992616
Validation loss decreased (1.031925 --> 0.992616).  Saving model ...
Epoch: 7         Training Loss: 1.938812         Validation Loss: 1.012870
Epoch: 8         Training Loss: 1.958124         Validation Loss: 0.984010
Validation loss decreased (0.992616 --> 0.984010).  Saving model ...
Epoch: 9         Training Loss: 1.823942         Validation Loss: 1.042436
Epoch: 10         Training Loss: 1.905179         Validation Loss: 1.091561
Epoch: 11         Training Loss: 1.896088         Validation Loss: 0.943560
Validation loss decreased (0.984010 --> 0.943560).  Saving model ...
Epoch: 12         Training Loss: 1.907542         Validation Loss: 0.969853
Epoch: 13         Training Loss: 1.829268         Validation Loss: 1.029986
Epoch: 14         Training Loss: 2.013943         Validation Loss: 1.023226
Epoch: 15         Training Loss: 1.937138         Validation Loss: 1.122746
Epoch: 16         Training Loss: 1.910178         Validation Loss: 0.986596
Epoch: 17         Training Loss: 1.768022         Validation Loss: 1.006211
Epoch: 18         Training Loss: 1.799760         Validation Loss: 1.195073
Epoch: 19         Training Loss: 1.842298         Validation Loss: 1.008879
Epoch: 20         Training Loss: 1.849954         Validation Loss: 1.056640
```

1.1.16 (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%.

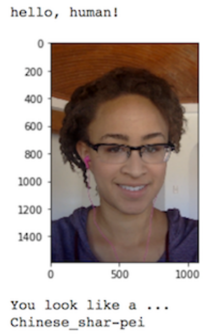
```
In [29]: model_transfer.load_state_dict(torch.load('model_transfer.pt'))
         test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 1.030367


Test Accuracy: 73% (613/836)
```

1.1.17 (IMPLEMENTATION) Predict Dog Breed with the Model

Write a function that takes an image path as input and returns the dog breed (`Affenpinscher`, `Afghan hound`, etc) that is predicted by your model.

```
In [30]: ### TODO: Write a function that takes a path to an image as input
         ### and returns the dog breed that is predicted by the model.


         # list of class names by index, i.e. a name can be accessed like class_names[0]
         class_names = [item[4:].replace("_", " ") for item in dataset_train_transfer.classes]

         def predict_breed_transfer(img_path):
             # load the image and return the predicted breed

             img_path = Image.open(img_path)

             transform = transforms.Compose([transforms.Resize(255),
                                             transforms.CenterCrop(224),
                                             transforms.ToTensor(),
                                             transforms.Normalize(mean=[0.485, 0.456, 0.406], st

             img_path = transform(img_path)

             img_path.unsqueeze_(0)
             # When i did not add this line,
             # i got "https://discuss.pytorch.org/t/expected-stride-to-be-a-single-integer-value
             # I found the solution on above link.

             if use_cuda:
                 img_path = img_path.cuda()

             with torch.no_grad():
                 model_transfer.eval()
                 output = model_transfer(img_path)
```



Sample Human Output

```
        output = output.argmax()

    return class_names[output] # predicted class index
```

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!

1.1.18 (IMPLEMENTATION) Write your Algorithm

```
In [37]: ### Feel free to use as many code cells as needed.

         def run_app(img_path):
             ## handle cases for a human face, dog, and neither
             img = cv2.imread(img_path)
             cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
             human_found = face_detector(img_path)
             dog_found = dog_detector(img_path)

             if human_found:
                 plt.imshow(cv_rgb)
                 plt.show()
                 print('Hello Human! Your doggy-doppelganger is a: ' + predict_breed_transfer(im
             elif dog_found:
                 plt.imshow(cv_rgb)
                 plt.show()
                 print('Good Doggy! You look like a: ' +  predict_breed_transfer(img_path))
```

```
        else:
            plt.imshow(cv_rgb)
            plt.show()
            print('Error: No Dog or Human found in picture. Is there either in picture?')
```

Step 6: Test Your Algorithm

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

1.1.19 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

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

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

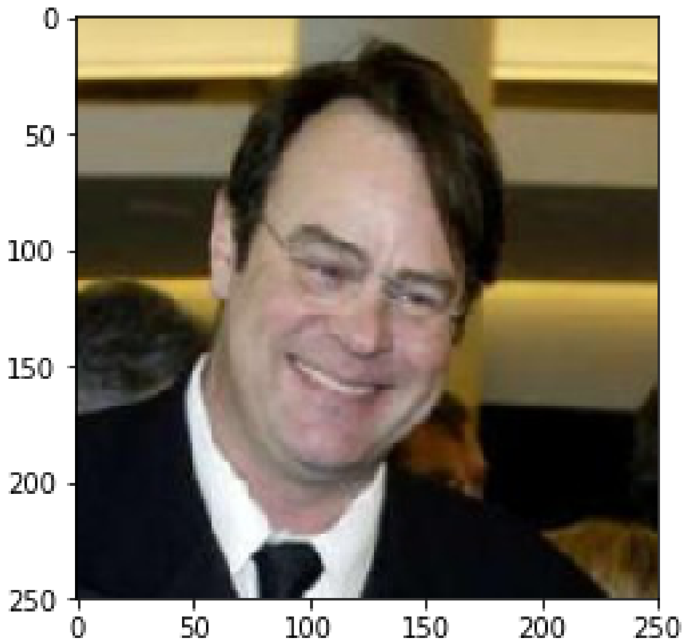
Answer: (Three possible points for improvement)

I think the output is better than I expected.

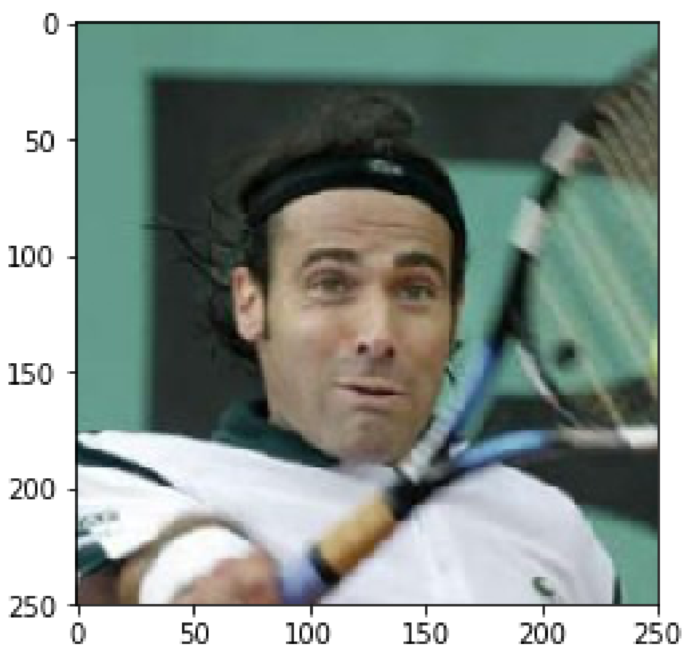
Things to improvement for my algorithm

- Fine tune parameters the model to give a better accuracy
- In terms of user experience , Serve this function on a website using API (Flask, AWS, etc.)
- Handle better the case when there are multiple dogs/humans or dogs and humans in an image
- Trying different models, optimizers and loss functions, input datasets

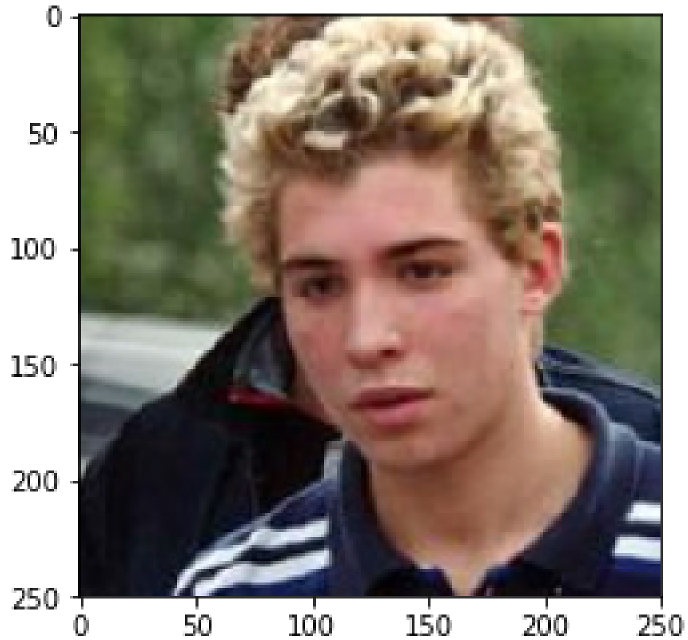
```
In [38]: ## TODO: Execute your algorithm from Step 6 on
         ## at least 6 images on your computer.
         ## Feel free to use as many code cells as needed.

         ## suggested code, below
         for file in np.hstack((human_files[:3], dog_files[:3])):
             run_app(file)
```



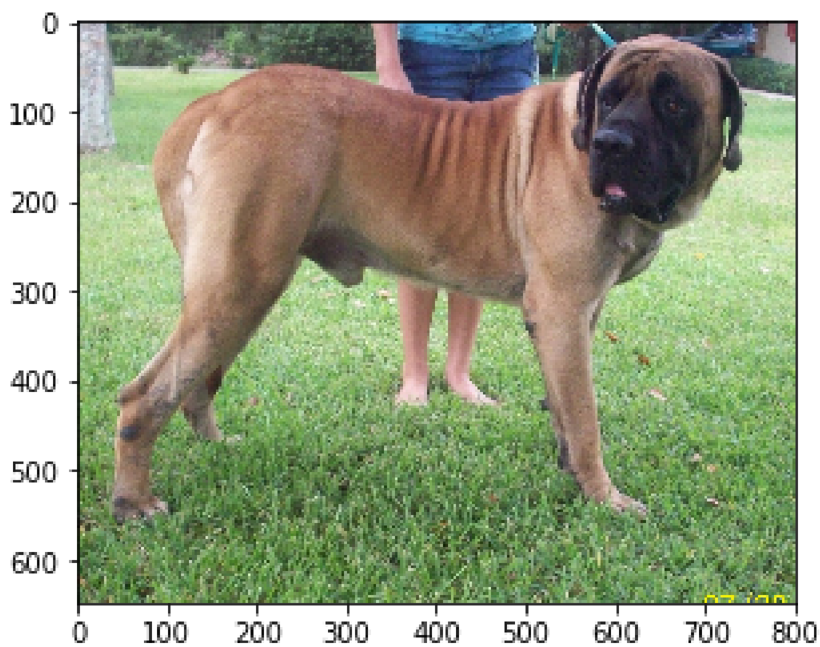
Hello Human! Your doggy-doppelganger is a: American staffordshire terrier



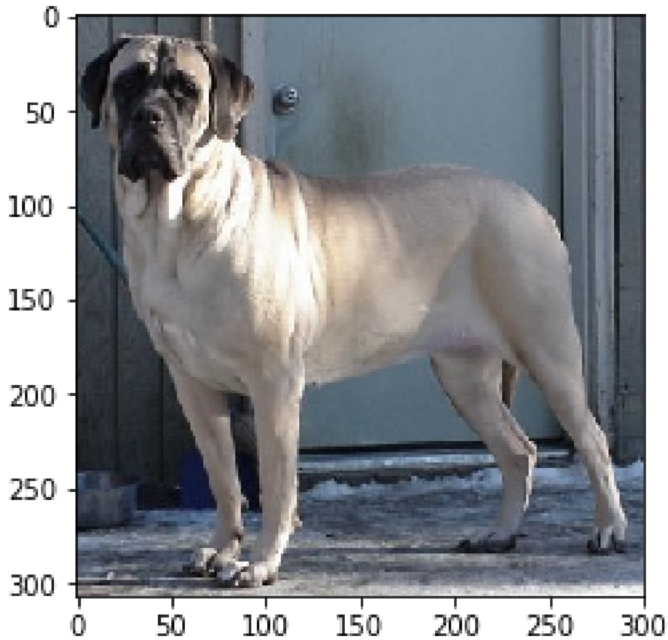
```
Hello Human! Your doggy-doppelganger is a: Chinese crested
```



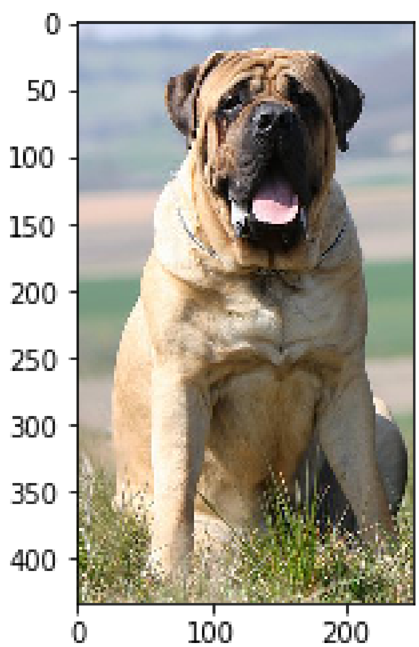

```
Hello Human! Your doggy-doppelganger is a: Airedale terrier
```



```
Good Doggy! You look like a: Mastiff
```




```
Good Doggy! You look like a: Chinese shar-pei
```



```
Good Doggy! You look like a: Cane corso
```

```
In [ ]:
```