dog_app-jm

January 27, 2020

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: * Download the dog dataset. Unzip the folder and place it in this project's home directory, at the location `/dogImages`.

- 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 [23]: import numpy as np
         from glob import glob

         # load filenames for human and dog images
         human_files = np.array(glob("/content/drive/My Drive/Colab Notebooks/data/lfw/*/*"))
         dog_files = np.array(glob("/content/drive/My Drive/Colab Notebooks/data/dogImages/*/*/*

         # 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 7653 total human images.
There are 8351 total dog images.
```

Step 1: Detect Humans

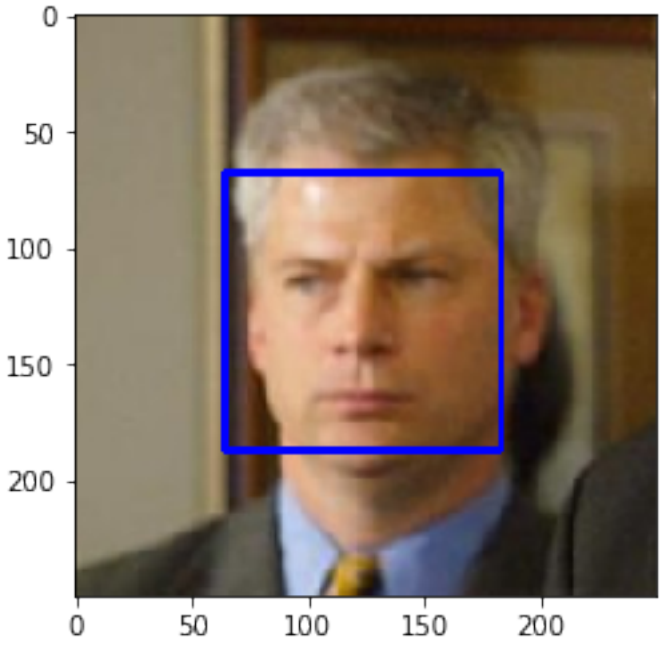
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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 [24]: import cv2
         import matplotlib.pyplot as plt
         %matplotlib inline

         # extract pre-trained face detector
         face_cascade = cv2.CascadeClassifier('/content/drive/My Drive/Colab Notebooks/data/haar

         # 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
```



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 [0]: # 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: The algorithm detected human face for 98% images of humans and 7% images of dogs.

```
In [0]: from tqdm import tqdm
        import collections

        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.

        counter = collections.Counter()

        for path in human_files_short:
            if face_detector(path):
                counter["human"] += 1

        for path in dog_files_short:
            if face_detector(path):
                counter["dog"] += 1

        print("The algorithm detected human face for {0}% images of humans and {1}% images of do
```

```
The algorithm detected human face for 98% images of humans and 7% images of dogs.
```

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 [0]: ### (Optional)
        ### TODO: Test performance of another face 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 [0]: 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()
```

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 [0]: from PIL import Image
        import torchvision.transforms as transforms
        from torch.autograd import Variable

        # Set PIL to be tolerant of image files that are truncated.
        from PIL import ImageFile
        ImageFile.LOAD_TRUNCATED_IMAGES = True
```

```
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

    image = Image.open(img_path)
    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

    ])

    image = transform(image)
    image = image.unsqueeze(0)
    if use_cuda:
        image = image.cuda()
    prediction = VGG16(image)

    return np.argmax(prediction.cpu().detach().numpy()) # 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 [0]: ### returns "True" if a dog is detected in the image stored at img_path
        def dog_detector(img_path):
            ## TODO: Complete the function.

            result = VGG16_predict(img_path)

            return result >= 151 and result <= 268
```

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: The algorithm detected dog for 0% images of humans and 97% images of dogs.

```
In [0]: ### TODO: Test the performance of the dog_detector function
        ### on the images in human_files_short and dog_files_short.

        counter = collections.Counter()

        for path in human_files_short:
            if dog_detector(path):
                counter["human"] += 1

        for path in dog_files_short:
            if dog_detector(path):
                counter["dog"] += 1

        print("The algorithm detected dog for {0}% images of humans and {1}% images of dogs.".fo
```

```
The algorithm detected dog for 0% images of humans and 97% images of dogs.
```

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 [0]: ### (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 `dogImages/train`, `dogImages/valid`, and `dogImages/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 [0]: import os
        from torchvision import datasets
        from torchvision import transforms
        from torch.utils import data

        import matplotlib.pyplot as plt

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

        data_transform = transforms.Compose([
            transforms.RandomResizedCrop(224),
            transforms.ToTensor(),
            transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
        ])

        data_transform_train = transforms.Compose([
            transforms.RandomResizedCrop(224),
            transforms.RandomRotation((0, 90)),
            transforms.ToTensor(),
```

```
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
    ])

    loaders_scratch = {}
    batch_size = 10
    num_workers = 2

    train_dataset = datasets.ImageFolder(root='/content/drive/My Drive/Colab Notebooks/data/
    loaders_scratch['train'] = data.DataLoader(train_dataset, batch_size=batch_size, shuffle

    valid_dataset = datasets.ImageFolder(root='/content/drive/My Drive/Colab Notebooks/data/
    loaders_scratch['valid'] = data.DataLoader(valid_dataset, batch_size=batch_size, shuffle

    test_dataset = datasets.ImageFolder(root='/content/drive/My Drive/Colab Notebooks/data/d
    loaders_scratch['test'] = data.DataLoader(test_dataset, batch_size=batch_size, shuffle=T
```

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:

Each image was resized and randomly cropped to a square of 224x224 pixels. Following that the image was normalized according to library instructions. Such a size was selected because data loss could not be seen and image was not too big. I decided to augment training data because after few tries to learn the network detection accuracy was not so great. After that I could see imrovement in learning quality. As data augmentation I selected random rotation up to 90 degrees.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [0]: 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.layer1 = nn.Sequential(
                    nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1),
                    nn.ReLU(),
                    nn.MaxPool2d(kernel_size=2, stride=2))

                self.layer2 = nn.Sequential(
```

```
            nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))

        self.layer3 = nn.Sequential(
            nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))

        self.layer4 = nn.Sequential(
            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))

        self.dropout2 = nn.Dropout(0.2)
        self.dropout4 = nn.Dropout(0.4)

        self.full1 = nn.Sequential(
            nn.Linear(128 * 14 * 14, 512),
            nn.ReLU())

        self.fc2 = nn.Linear(512, 133)

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

        x = self.layer1(x)
        x = self.dropout2(x)
        x = self.layer2(x)
        x = self.dropout2(x)
        x = self.layer3(x)
        x = self.dropout2(x)
        x = self.layer4(x)

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

        x = self.dropout4(x)
        x = self.full1(x)
        x = self.dropout4(x)
        x = self.fc2(x)

        return x

#-#-# You do 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: In feature part of the network I decided to use 4 layers which consisted of: - convolutional layer with kernel size 3 increasing number of features - ReLu activation function - pooling layer to reduce image size by 2 - dropout mechanism with probability 0.2

In fully connected part I decided to use 2 layers. First reduced number of inputs from 25088 to 512 and the second one reduced number of inputs from 512 to number of classes to be detected - 133. There was dropout layer in beetwen with probability of 0.4.

Such a model was selected because similar approach was used during the classes to similar problem. In addition, some of the network models used in pytorch were similar although usually more complicated. I tried to select simple model due to small amount of data.

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 [0]: import torch.optim as optim

        learning_rate = 0.001

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

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

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 [0]: # the following import is required for training to be robust to truncated images
        import numpy as np
        from PIL import ImageFile
        ImageFile.LOAD_TRUNCATED_IMAGES = True

        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
            correct = 0.
            total = 0.

            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_los

    # clear the gradients of all optimized variables
    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 paramete
    loss.backward()
    # perform a single optimization step (parameter update)
    optimizer.step()
    # update training 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

    # forward pass: compute predicted outputs by passing inputs to the model
    output = model(data)
    # calculate the batch loss
    loss = criterion(output, target)
    # update average validation loss
    valid_loss += ((1 / (batch_idx + 1)) * (loss.data - valid_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 training/validation statistics
            print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f} \tValidation
                epoch,
                train_loss,
                valid_loss,
                100. * correct / total
                ))

            ## TODO: save the model if validation loss has decreased
            if valid_loss < valid_loss_min:
                torch.save(model.state_dict(), save_path)
                valid_loss_min = valid_loss

        # return trained model
        return model


    # train the model
    model_scratch = train(400, loaders_scratch, model_scratch, optimizer_scratch,
                          criterion_scratch, use_cuda, '/content/drive/My Drive/Colab Notebo

    # load the model that got the best validation accuracy
    model_scratch.load_state_dict(torch.load('/content/drive/My Drive/Colab Notebooks/model_
```

```
Epoch: 1         Training Loss: 4.889802         Validation Loss: 4.887720         Validation Ac
Epoch: 2         Training Loss: 4.880711         Validation Loss: 4.884031         Validation Ac
Epoch: 3         Training Loss: 4.874452         Validation Loss: 4.880095         Validation Ac
Epoch: 4         Training Loss: 4.872150         Validation Loss: 4.878139         Validation Ac
Epoch: 5         Training Loss: 4.866072         Validation Loss: 4.874239         Validation Ac
Epoch: 6         Training Loss: 4.861391         Validation Loss: 4.871423         Validation Ac
Epoch: 7         Training Loss: 4.860166         Validation Loss: 4.868813         Validation Ac
Epoch: 8         Training Loss: 4.853275         Validation Loss: 4.864301         Validation Ac
Epoch: 9         Training Loss: 4.844520         Validation Loss: 4.859428         Validation Ac
Epoch: 10         Training Loss: 4.831635         Validation Loss: 4.853657         Validation A
Epoch: 11         Training Loss: 4.819396         Validation Loss: 4.842750         Validation A
Epoch: 12         Training Loss: 4.812555         Validation Loss: 4.830974         Validation A
Epoch: 13         Training Loss: 4.792469         Validation Loss: 4.818999         Validation A
Epoch: 14         Training Loss: 4.779333         Validation Loss: 4.812827         Validation A
Epoch: 15         Training Loss: 4.778947         Validation Loss: 4.810057         Validation A
Epoch: 16         Training Loss: 4.770101         Validation Loss: 4.802031         Validation A
Epoch: 17         Training Loss: 4.765337         Validation Loss: 4.789282         Validation A
Epoch: 18         Training Loss: 4.761455         Validation Loss: 4.792146         Validation A
Epoch: 19         Training Loss: 4.737973         Validation Loss: 4.785418         Validation A
```

```
Epoch: 20         Training Loss: 4.739319         Validation Loss: 4.789217         Validation A
Epoch: 21         Training Loss: 4.726267         Validation Loss: 4.771544         Validation A
Epoch: 22         Training Loss: 4.737859         Validation Loss: 4.789311         Validation A
Epoch: 23         Training Loss: 4.719619         Validation Loss: 4.772806         Validation A
Epoch: 24         Training Loss: 4.714355         Validation Loss: 4.764539         Validation A
Epoch: 25         Training Loss: 4.707738         Validation Loss: 4.769601         Validation A
Epoch: 26         Training Loss: 4.702507         Validation Loss: 4.759590         Validation A
Epoch: 27         Training Loss: 4.686752         Validation Loss: 4.754711         Validation A
Epoch: 28         Training Loss: 4.681721         Validation Loss: 4.736724         Validation A
Epoch: 29         Training Loss: 4.665602         Validation Loss: 4.728213         Validation A
Epoch: 30         Training Loss: 4.651689         Validation Loss: 4.731114         Validation A
Epoch: 31         Training Loss: 4.639747         Validation Loss: 4.686635         Validation A
Epoch: 32         Training Loss: 4.614209         Validation Loss: 4.687821         Validation A
Epoch: 33         Training Loss: 4.612519         Validation Loss: 4.683485         Validation A
Epoch: 34         Training Loss: 4.588634         Validation Loss: 4.659617         Validation A
Epoch: 35         Training Loss: 4.589596         Validation Loss: 4.668017         Validation A
Epoch: 36         Training Loss: 4.579241         Validation Loss: 4.631346         Validation A
Epoch: 37         Training Loss: 4.573422         Validation Loss: 4.655549         Validation A
Epoch: 38         Training Loss: 4.551734         Validation Loss: 4.619094         Validation A
Epoch: 39         Training Loss: 4.558963         Validation Loss: 4.625632         Validation A
Epoch: 40         Training Loss: 4.529906         Validation Loss: 4.597682         Validation A
Epoch: 41         Training Loss: 4.534218         Validation Loss: 4.611000         Validation A
Epoch: 42         Training Loss: 4.524575         Validation Loss: 4.585309         Validation A
Epoch: 43         Training Loss: 4.523911         Validation Loss: 4.624469         Validation A
Epoch: 44         Training Loss: 4.513161         Validation Loss: 4.608538         Validation A
Epoch: 45         Training Loss: 4.510570         Validation Loss: 4.579090         Validation A
Epoch: 46         Training Loss: 4.492032         Validation Loss: 4.594042         Validation A
Epoch: 47         Training Loss: 4.486064         Validation Loss: 4.586446         Validation A
Epoch: 48         Training Loss: 4.479362         Validation Loss: 4.578300         Validation A
Epoch: 49         Training Loss: 4.474050         Validation Loss: 4.568787         Validation A
Epoch: 50         Training Loss: 4.477848         Validation Loss: 4.564169         Validation A
Epoch: 51         Training Loss: 4.461338         Validation Loss: 4.576610         Validation A
Epoch: 52         Training Loss: 4.446775         Validation Loss: 4.553746         Validation A
Epoch: 53         Training Loss: 4.451097         Validation Loss: 4.539131         Validation A
Epoch: 54         Training Loss: 4.445972         Validation Loss: 4.542055         Validation A
Epoch: 55         Training Loss: 4.439577         Validation Loss: 4.543793         Validation A
Epoch: 56         Training Loss: 4.432490         Validation Loss: 4.550891         Validation A
Epoch: 57         Training Loss: 4.423673         Validation Loss: 4.563600         Validation A
Epoch: 58         Training Loss: 4.411632         Validation Loss: 4.514838         Validation A
Epoch: 59         Training Loss: 4.408273         Validation Loss: 4.528983         Validation A
Epoch: 60         Training Loss: 4.414958         Validation Loss: 4.522494         Validation A
Epoch: 61         Training Loss: 4.376550         Validation Loss: 4.531257         Validation A
Epoch: 62         Training Loss: 4.381496         Validation Loss: 4.524548         Validation A
Epoch: 63         Training Loss: 4.383953         Validation Loss: 4.505863         Validation A
Epoch: 64         Training Loss: 4.377995         Validation Loss: 4.511341         Validation A
Epoch: 65         Training Loss: 4.372977         Validation Loss: 4.516416         Validation A
Epoch: 66         Training Loss: 4.371068         Validation Loss: 4.496193         Validation A
Epoch: 67         Training Loss: 4.357546         Validation Loss: 4.477891         Validation A
```

```
Epoch: 68         Training Loss: 4.359361         Validation Loss: 4.514221         Validation A
Epoch: 69         Training Loss: 4.358708         Validation Loss: 4.481921         Validation A
Epoch: 70         Training Loss: 4.342451         Validation Loss: 4.494312         Validation A
Epoch: 71         Training Loss: 4.336425         Validation Loss: 4.471353         Validation A
Epoch: 72         Training Loss: 4.337156         Validation Loss: 4.462639         Validation A
Epoch: 73         Training Loss: 4.331333         Validation Loss: 4.463763         Validation A
Epoch: 74         Training Loss: 4.324495         Validation Loss: 4.511412         Validation A
Epoch: 75         Training Loss: 4.330863         Validation Loss: 4.460423         Validation A
Epoch: 76         Training Loss: 4.310118         Validation Loss: 4.447963         Validation A
Epoch: 77         Training Loss: 4.295402         Validation Loss: 4.438911         Validation A
Epoch: 78         Training Loss: 4.304292         Validation Loss: 4.478023         Validation A
Epoch: 79         Training Loss: 4.298469         Validation Loss: 4.451281         Validation A
Epoch: 80         Training Loss: 4.302401         Validation Loss: 4.428257         Validation A
Epoch: 81         Training Loss: 4.288504         Validation Loss: 4.407689         Validation A
Epoch: 82         Training Loss: 4.292784         Validation Loss: 4.445201         Validation A
Epoch: 83         Training Loss: 4.284618         Validation Loss: 4.402153         Validation A
Epoch: 84         Training Loss: 4.265228         Validation Loss: 4.439430         Validation A
Epoch: 85         Training Loss: 4.280429         Validation Loss: 4.411247         Validation A
Epoch: 86         Training Loss: 4.265531         Validation Loss: 4.446830         Validation A
Epoch: 87         Training Loss: 4.277637         Validation Loss: 4.453113         Validation A
Epoch: 88         Training Loss: 4.246767         Validation Loss: 4.446455         Validation A
Epoch: 89         Training Loss: 4.250443         Validation Loss: 4.413422         Validation A
Epoch: 90         Training Loss: 4.256778         Validation Loss: 4.449054         Validation A
Epoch: 91         Training Loss: 4.248568         Validation Loss: 4.411288         Validation A
Epoch: 92         Training Loss: 4.236390         Validation Loss: 4.399438         Validation A
Epoch: 93         Training Loss: 4.231069         Validation Loss: 4.380850         Validation A
Epoch: 94         Training Loss: 4.226260         Validation Loss: 4.399861         Validation A
Epoch: 95         Training Loss: 4.241671         Validation Loss: 4.394013         Validation A
Epoch: 96         Training Loss: 4.235939         Validation Loss: 4.401415         Validation A
Epoch: 97         Training Loss: 4.207224         Validation Loss: 4.398232         Validation A
Epoch: 98         Training Loss: 4.210842         Validation Loss: 4.397831         Validation A
Epoch: 99         Training Loss: 4.203180         Validation Loss: 4.404888         Validation A
Epoch: 100         Training Loss: 4.204020         Validation Loss: 4.367791         Validation
Epoch: 101         Training Loss: 4.190968         Validation Loss: 4.377901         Validation
Epoch: 102         Training Loss: 4.181694         Validation Loss: 4.345314         Validation
Epoch: 103         Training Loss: 4.192420         Validation Loss: 4.383793         Validation
Epoch: 104         Training Loss: 4.183096         Validation Loss: 4.361443         Validation
Epoch: 105         Training Loss: 4.175605         Validation Loss: 4.361559         Validation
Epoch: 106         Training Loss: 4.178320         Validation Loss: 4.367634         Validation
Epoch: 107         Training Loss: 4.159990         Validation Loss: 4.353925         Validation
Epoch: 108         Training Loss: 4.147770         Validation Loss: 4.333640         Validation
Epoch: 109         Training Loss: 4.151306         Validation Loss: 4.317529         Validation
Epoch: 110         Training Loss: 4.148232         Validation Loss: 4.317950         Validation
Epoch: 111         Training Loss: 4.143281         Validation Loss: 4.364029         Validation
Epoch: 112         Training Loss: 4.126160         Validation Loss: 4.356215         Validation
Epoch: 113         Training Loss: 4.128595         Validation Loss: 4.335250         Validation
Epoch: 114         Training Loss: 4.115768         Validation Loss: 4.343791         Validation
Epoch: 115         Training Loss: 4.134300         Validation Loss: 4.289955         Validation
```

```
Epoch: 116        Training Loss: 4.124463        Validation Loss: 4.347414        Validation
Epoch: 117        Training Loss: 4.108920        Validation Loss: 4.339469        Validation
Epoch: 118        Training Loss: 4.091233        Validation Loss: 4.308765        Validation
Epoch: 119        Training Loss: 4.096913        Validation Loss: 4.282137        Validation
Epoch: 120        Training Loss: 4.072002        Validation Loss: 4.277309        Validation
Epoch: 121        Training Loss: 4.089199        Validation Loss: 4.335621        Validation
Epoch: 122        Training Loss: 4.070497        Validation Loss: 4.321894        Validation
Epoch: 123        Training Loss: 4.079008        Validation Loss: 4.284268        Validation
Epoch: 124        Training Loss: 4.052811        Validation Loss: 4.293931        Validation
Epoch: 125        Training Loss: 4.044912        Validation Loss: 4.251979        Validation
Epoch: 126        Training Loss: 4.063595        Validation Loss: 4.272233        Validation
Epoch: 127        Training Loss: 4.043044        Validation Loss: 4.296424        Validation
Epoch: 128        Training Loss: 4.037798        Validation Loss: 4.264029        Validation
Epoch: 129        Training Loss: 4.029923        Validation Loss: 4.291706        Validation
Epoch: 130        Training Loss: 4.035285        Validation Loss: 4.245619        Validation
Epoch: 131        Training Loss: 4.039813        Validation Loss: 4.212646        Validation
Epoch: 132        Training Loss: 4.009869        Validation Loss: 4.274029        Validation
Epoch: 133        Training Loss: 4.025888        Validation Loss: 4.260013        Validation
Epoch: 134        Training Loss: 4.016978        Validation Loss: 4.254373        Validation
Epoch: 135        Training Loss: 4.021310        Validation Loss: 4.253362        Validation
Epoch: 136        Training Loss: 4.003283        Validation Loss: 4.215093        Validation
Epoch: 137        Training Loss: 3.988268        Validation Loss: 4.256962        Validation
Epoch: 138        Training Loss: 3.976783        Validation Loss: 4.275323        Validation
Epoch: 139        Training Loss: 3.991629        Validation Loss: 4.244455        Validation
Epoch: 140        Training Loss: 3.977470        Validation Loss: 4.188062        Validation
Epoch: 141        Training Loss: 3.964296        Validation Loss: 4.218343        Validation
Epoch: 142        Training Loss: 3.949083        Validation Loss: 4.191419        Validation
Epoch: 143        Training Loss: 3.958943        Validation Loss: 4.218114        Validation
Epoch: 144        Training Loss: 3.951160        Validation Loss: 4.188188        Validation
Epoch: 145        Training Loss: 3.944432        Validation Loss: 4.183220        Validation
Epoch: 146        Training Loss: 3.936501        Validation Loss: 4.212515        Validation
Epoch: 147        Training Loss: 3.925274        Validation Loss: 4.181818        Validation
Epoch: 148        Training Loss: 3.931989        Validation Loss: 4.235027        Validation
Epoch: 149        Training Loss: 3.923272        Validation Loss: 4.209925        Validation
Epoch: 150        Training Loss: 3.911864        Validation Loss: 4.197152        Validation
Epoch: 151        Training Loss: 3.923867        Validation Loss: 4.176208        Validation
Epoch: 152        Training Loss: 3.914864        Validation Loss: 4.196151        Validation
Epoch: 153        Training Loss: 3.901565        Validation Loss: 4.166702        Validation
Epoch: 154        Training Loss: 3.898861        Validation Loss: 4.163126        Validation
Epoch: 155        Training Loss: 3.897008        Validation Loss: 4.158103        Validation
Epoch: 156        Training Loss: 3.882429        Validation Loss: 4.182452        Validation
Epoch: 157        Training Loss: 3.881769        Validation Loss: 4.127035        Validation
Epoch: 158        Training Loss: 3.856653        Validation Loss: 4.127237        Validation
Epoch: 159        Training Loss: 3.864453        Validation Loss: 4.138508        Validation
Epoch: 160        Training Loss: 3.857637        Validation Loss: 4.148266        Validation
Epoch: 161        Training Loss: 3.857973        Validation Loss: 4.145984        Validation
Epoch: 162        Training Loss: 3.830939        Validation Loss: 4.138818        Validation
Epoch: 163        Training Loss: 3.855467        Validation Loss: 4.147429        Validation
```

```
Epoch: 164        Training Loss: 3.837106        Validation Loss: 4.156875        Validation
Epoch: 165        Training Loss: 3.831790        Validation Loss: 4.119478        Validation
Epoch: 166        Training Loss: 3.820386        Validation Loss: 4.124229        Validation
Epoch: 167        Training Loss: 3.804905        Validation Loss: 4.135494        Validation
Epoch: 168        Training Loss: 3.816918        Validation Loss: 4.081820        Validation
Epoch: 169        Training Loss: 3.828218        Validation Loss: 4.104443        Validation
Epoch: 170        Training Loss: 3.802220        Validation Loss: 4.152456        Validation
Epoch: 171        Training Loss: 3.829246        Validation Loss: 4.092724        Validation
Epoch: 172        Training Loss: 3.786911        Validation Loss: 4.095738        Validation
Epoch: 173        Training Loss: 3.784518        Validation Loss: 4.118988        Validation
Epoch: 174        Training Loss: 3.771987        Validation Loss: 4.116381        Validation
Epoch: 175        Training Loss: 3.765835        Validation Loss: 4.147020        Validation
Epoch: 176        Training Loss: 3.770134        Validation Loss: 4.106996        Validation
Epoch: 177        Training Loss: 3.770224        Validation Loss: 4.116591        Validation
Epoch: 178        Training Loss: 3.728404        Validation Loss: 4.077125        Validation
Epoch: 179        Training Loss: 3.741863        Validation Loss: 4.124872        Validation
Epoch: 180        Training Loss: 3.759200        Validation Loss: 4.086443        Validation
Epoch: 181        Training Loss: 3.728881        Validation Loss: 4.139641        Validation
Epoch: 182        Training Loss: 3.719232        Validation Loss: 4.071069        Validation
Epoch: 183        Training Loss: 3.735832        Validation Loss: 4.134961        Validation
Epoch: 184        Training Loss: 3.694988        Validation Loss: 4.057529        Validation
Epoch: 185        Training Loss: 3.711285        Validation Loss: 4.045784        Validation
Epoch: 186        Training Loss: 3.700440        Validation Loss: 4.035214        Validation
Epoch: 187        Training Loss: 3.713903        Validation Loss: 4.069977        Validation
Epoch: 188        Training Loss: 3.708422        Validation Loss: 4.087810        Validation
Epoch: 189        Training Loss: 3.682284        Validation Loss: 4.072077        Validation
Epoch: 190        Training Loss: 3.701727        Validation Loss: 4.060647        Validation
Epoch: 191        Training Loss: 3.692966        Validation Loss: 4.067240        Validation
Epoch: 192        Training Loss: 3.676061        Validation Loss: 4.039976        Validation
Epoch: 193        Training Loss: 3.675046        Validation Loss: 4.082298        Validation
Epoch: 194        Training Loss: 3.656907        Validation Loss: 4.047762        Validation
Epoch: 195        Training Loss: 3.659859        Validation Loss: 4.073039        Validation
Epoch: 196        Training Loss: 3.669619        Validation Loss: 4.030258        Validation
Epoch: 197        Training Loss: 3.636013        Validation Loss: 4.021617        Validation
Epoch: 198        Training Loss: 3.607924        Validation Loss: 4.021764        Validation
Epoch: 199        Training Loss: 3.653014        Validation Loss: 4.027783        Validation
Epoch: 200        Training Loss: 3.598336        Validation Loss: 4.039530        Validation
Epoch: 201        Training Loss: 3.615581        Validation Loss: 4.020327        Validation
Epoch: 202        Training Loss: 3.592008        Validation Loss: 4.014199        Validation
Epoch: 203        Training Loss: 3.584552        Validation Loss: 4.053055        Validation
Epoch: 204        Training Loss: 3.605174        Validation Loss: 4.068951        Validation
Epoch: 205        Training Loss: 3.576563        Validation Loss: 3.976563        Validation
Epoch: 206        Training Loss: 3.595612        Validation Loss: 4.046586        Validation
Epoch: 207        Training Loss: 3.583784        Validation Loss: 3.976213        Validation
Epoch: 208        Training Loss: 3.582941        Validation Loss: 4.012356        Validation
Epoch: 209        Training Loss: 3.571632        Validation Loss: 3.970572        Validation
Epoch: 210        Training Loss: 3.559407        Validation Loss: 4.058072        Validation
Epoch: 211        Training Loss: 3.592970        Validation Loss: 4.000300        Validation
```

```
Epoch: 212        Training Loss: 3.589015        Validation Loss: 4.030813        Validation
Epoch: 213        Training Loss: 3.547368        Validation Loss: 4.051563        Validation
Epoch: 214        Training Loss: 3.559373        Validation Loss: 4.007505        Validation
Epoch: 215        Training Loss: 3.538662        Validation Loss: 3.983643        Validation
Epoch: 216        Training Loss: 3.528767        Validation Loss: 4.038404        Validation
Epoch: 217        Training Loss: 3.517386        Validation Loss: 3.977580        Validation
Epoch: 218        Training Loss: 3.531313        Validation Loss: 4.012578        Validation
Epoch: 219        Training Loss: 3.520049        Validation Loss: 4.008932        Validation
Epoch: 220        Training Loss: 3.495922        Validation Loss: 3.974050        Validation
Epoch: 221        Training Loss: 3.479124        Validation Loss: 4.022565        Validation
Epoch: 222        Training Loss: 3.506943        Validation Loss: 3.999432        Validation
Epoch: 223        Training Loss: 3.496641        Validation Loss: 3.949164        Validation
Epoch: 224        Training Loss: 3.492117        Validation Loss: 4.068584        Validation
Epoch: 225        Training Loss: 3.479025        Validation Loss: 4.010494        Validation
Epoch: 226        Training Loss: 3.479768        Validation Loss: 4.038422        Validation
Epoch: 227        Training Loss: 3.473467        Validation Loss: 4.005161        Validation
Epoch: 228        Training Loss: 3.475405        Validation Loss: 4.008047        Validation
Epoch: 229        Training Loss: 3.431635        Validation Loss: 4.037668        Validation
Epoch: 230        Training Loss: 3.432604        Validation Loss: 3.944960        Validation
Epoch: 231        Training Loss: 3.478246        Validation Loss: 4.001677        Validation
Epoch: 232        Training Loss: 3.448058        Validation Loss: 3.993002        Validation
Epoch: 233        Training Loss: 3.414895        Validation Loss: 3.961176        Validation
Epoch: 234        Training Loss: 3.423721        Validation Loss: 4.006955        Validation
Epoch: 235        Training Loss: 3.439458        Validation Loss: 3.960396        Validation
Epoch: 236        Training Loss: 3.393210        Validation Loss: 3.991027        Validation
Epoch: 237        Training Loss: 3.429129        Validation Loss: 3.977577        Validation
Epoch: 238        Training Loss: 3.393116        Validation Loss: 3.981567        Validation
Epoch: 239        Training Loss: 3.419396        Validation Loss: 4.003803        Validation
Epoch: 240        Training Loss: 3.413674        Validation Loss: 3.884477        Validation
Epoch: 241        Training Loss: 3.409231        Validation Loss: 4.008877        Validation
Epoch: 242        Training Loss: 3.380367        Validation Loss: 3.960272        Validation
Epoch: 243        Training Loss: 3.377679        Validation Loss: 3.981557        Validation
Epoch: 244        Training Loss: 3.351337        Validation Loss: 3.927392        Validation
Epoch: 245        Training Loss: 3.337668        Validation Loss: 4.004786        Validation
Epoch: 246        Training Loss: 3.342791        Validation Loss: 4.017981        Validation
Epoch: 247        Training Loss: 3.344606        Validation Loss: 3.989302        Validation
Epoch: 248        Training Loss: 3.356205        Validation Loss: 3.940933        Validation
Epoch: 249        Training Loss: 3.356083        Validation Loss: 3.938275        Validation
Epoch: 250        Training Loss: 3.333505        Validation Loss: 3.993502        Validation
Epoch: 251        Training Loss: 3.325341        Validation Loss: 4.013399        Validation
Epoch: 252        Training Loss: 3.319654        Validation Loss: 3.938750        Validation
Epoch: 253        Training Loss: 3.300279        Validation Loss: 3.995460        Validation
Epoch: 254        Training Loss: 3.315623        Validation Loss: 4.005507        Validation
Epoch: 255        Training Loss: 3.316379        Validation Loss: 3.943687        Validation
Epoch: 256        Training Loss: 3.312566        Validation Loss: 3.950633        Validation
Epoch: 257        Training Loss: 3.301498        Validation Loss: 4.038296        Validation
Epoch: 258        Training Loss: 3.296878        Validation Loss: 3.912035        Validation
Epoch: 259        Training Loss: 3.292913        Validation Loss: 4.019599        Validation
```

```
Epoch: 260         Training Loss: 3.299337         Validation Loss: 3.950137         Validation
Epoch: 261         Training Loss: 3.292271         Validation Loss: 3.933327         Validation
```

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 [0]:
```
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: 3.916158


Test Accuracy: 11% (100/836)
```

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 [0]: ## TODO: Specify data loaders
        import os
        from torchvision import datasets
        from torchvision import transforms
        from torch.utils import data

        import matplotlib.pyplot as plt

        data_transform = transforms.Compose([
            transforms.RandomResizedCrop(224),
            transforms.ToTensor(),
            transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
        ])

        data_transform_train = transforms.Compose([
            transforms.RandomResizedCrop(224),
            transforms.RandomRotation((0, 90)),
            transforms.ToTensor(),
            transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
        ])

        loaders_transfer = {}
        data_transfer = {}
        batch_size = 10
        num_workers = 2

        train_dataset = datasets.ImageFolder(root='/content/drive/My Drive/Colab Notebooks/data/
        data_transfer['train'] = train_dataset
        loaders_transfer['train'] = data.DataLoader(train_dataset, batch_size=batch_size, shuffl

        valid_dataset = datasets.ImageFolder(root='/content/drive/My Drive/Colab Notebooks/data/
        data_transfer['valid'] = valid_dataset
        loaders_transfer['valid'] = data.DataLoader(valid_dataset, batch_size=batch_size, shuffl

        test_dataset = datasets.ImageFolder(root='/content/drive/My Drive/Colab Notebooks/data/d
        data_transfer['test'] = test_dataset
```

```
loaders_transfer['test'] = data.DataLoader(test_dataset, batch_size=batch_size, shuffle=
```

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 [0]: import torchvision.models as models
        import torch.nn as nn

        ## TODO: Specify model architecture
        model_transfer = models.vgg16(pretrained=True)

        for param in model_transfer.features.parameters():
            param.requires_grad = False

        model_transfer.classifier[6] = nn.Linear(model_transfer.classifier[6].in_features, 133)

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

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: I looked on models available in PyTorch and decided to use VGG16 model. I selected that model because it is sutable for image recognition and among other things it recognizes dog's breeds as well. Therefore set of features it calculates should contain some dog breeds characteristics. I decided to modify only last fully connected layer, so it returns 133 classes, which correspond to dog's breeds. Then I initialized learning process only for fully connected layer.

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 [0]: import torch.optim as optim

        learning_rate = 0.001

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

        ### TODO: select optimizer
        optimizer_transfer = optim.SGD(model_transfer.parameters(), lr=learning_rate)
```

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 [0]: # the following import is required for training to be robust to truncated images
        import numpy as np
        from PIL import ImageFile
        ImageFile.LOAD_TRUNCATED_IMAGES = True

        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
            correct = 0.
            total = 0.

            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_los

                    # clear the gradients of all optimized variables
                    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 paramete
                    loss.backward()
                    # perform a single optimization step (parameter update)
                    optimizer.step()
                    # update training 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

            # forward pass: compute predicted outputs by passing inputs to the model
            output = model(data)
            # calculate the batch loss
            loss = criterion(output, target)
            # update average validation loss
            valid_loss += ((1 / (batch_idx + 1)) * (loss.data - valid_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 training/validation statistics
        print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f} \tValidation
            epoch,
            train_loss,
            valid_loss,
            100. * correct / total
            ))

        ## TODO: save the model if validation loss has decreased
        if valid_loss < valid_loss_min:
            torch.save(model.state_dict(), save_path)
            valid_loss_min = valid_loss

    # return trained model
    return model

# train the model
model_transfer = train(200, 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('/content/drive/My Drive/Colab Notebooks/model
```

```
Epoch: 1         Training Loss: 4.350549         Validation Loss: 3.048523         Validation Ac
Epoch: 2         Training Loss: 3.022008         Validation Loss: 1.852071         Validation Ac
Epoch: 3         Training Loss: 2.436795         Validation Loss: 1.395277         Validation Ac
Epoch: 4         Training Loss: 2.190438         Validation Loss: 1.250793         Validation Ac
Epoch: 5         Training Loss: 2.009026         Validation Loss: 1.243276         Validation Ac
Epoch: 6         Training Loss: 1.988504         Validation Loss: 1.092259         Validation Ac
Epoch: 7         Training Loss: 1.854950         Validation Loss: 1.067003         Validation Ac
Epoch: 8         Training Loss: 1.820828         Validation Loss: 1.048521         Validation Ac
```

```
Epoch: 9         Training Loss: 1.793996         Validation Loss: 0.986793         Validation Ac
Epoch: 10         Training Loss: 1.756487         Validation Loss: 1.014400         Validation A
Epoch: 11         Training Loss: 1.709455         Validation Loss: 0.993467         Validation A
Epoch: 12         Training Loss: 1.725079         Validation Loss: 0.972734         Validation A
Epoch: 13         Training Loss: 1.709674         Validation Loss: 0.978867         Validation A
Epoch: 14         Training Loss: 1.660839         Validation Loss: 0.950839         Validation A
Epoch: 15         Training Loss: 1.631993         Validation Loss: 0.916922         Validation A
Epoch: 16         Training Loss: 1.590955         Validation Loss: 0.944802         Validation A
Epoch: 17         Training Loss: 1.574732         Validation Loss: 0.954425         Validation A
Epoch: 18         Training Loss: 1.574804         Validation Loss: 1.034662         Validation A
Epoch: 19         Training Loss: 1.569586         Validation Loss: 0.959205         Validation A
Epoch: 20         Training Loss: 1.528988         Validation Loss: 0.866728         Validation A
Epoch: 21         Training Loss: 1.524658         Validation Loss: 0.936819         Validation A
Epoch: 22         Training Loss: 1.513015         Validation Loss: 0.875037         Validation A
Epoch: 23         Training Loss: 1.481418         Validation Loss: 0.883922         Validation A
Epoch: 24         Training Loss: 1.487957         Validation Loss: 0.950757         Validation A
Epoch: 25         Training Loss: 1.473754         Validation Loss: 0.922161         Validation A
Epoch: 26         Training Loss: 1.455523         Validation Loss: 0.979282         Validation A
Epoch: 27         Training Loss: 1.466012         Validation Loss: 0.876578         Validation A
Epoch: 28         Training Loss: 1.459650         Validation Loss: 0.919370         Validation A
Epoch: 29         Training Loss: 1.423060         Validation Loss: 0.860250         Validation A
Epoch: 30         Training Loss: 1.405744         Validation Loss: 0.899574         Validation A
Epoch: 31         Training Loss: 1.432563         Validation Loss: 0.856633         Validation A
Epoch: 32         Training Loss: 1.400478         Validation Loss: 0.878073         Validation A
Epoch: 33         Training Loss: 1.383176         Validation Loss: 0.868487         Validation A
Epoch: 34         Training Loss: 1.372401         Validation Loss: 0.911573         Validation A
Epoch: 35         Training Loss: 1.379999         Validation Loss: 0.862773         Validation A
Epoch: 36         Training Loss: 1.382046         Validation Loss: 0.867083         Validation A
Epoch: 37         Training Loss: 1.365462         Validation Loss: 0.864638         Validation A
Epoch: 38         Training Loss: 1.363595         Validation Loss: 0.969619         Validation A
Epoch: 39         Training Loss: 1.348352         Validation Loss: 0.888406         Validation A
Epoch: 40         Training Loss: 1.358488         Validation Loss: 0.828097         Validation A
Epoch: 41         Training Loss: 1.341664         Validation Loss: 0.826655         Validation A
Epoch: 42         Training Loss: 1.343980         Validation Loss: 0.858087         Validation A
Epoch: 43         Training Loss: 1.320033         Validation Loss: 0.906909         Validation A
Epoch: 44         Training Loss: 1.342481         Validation Loss: 0.932043         Validation A
Epoch: 45         Training Loss: 1.288690         Validation Loss: 0.900238         Validation A
Epoch: 46         Training Loss: 1.321352         Validation Loss: 0.915504         Validation A
Epoch: 47         Training Loss: 1.320359         Validation Loss: 0.858901         Validation A
Epoch: 48         Training Loss: 1.293133         Validation Loss: 0.871253         Validation A
Epoch: 49         Training Loss: 1.280491         Validation Loss: 0.811483         Validation A
Epoch: 50         Training Loss: 1.277678         Validation Loss: 0.842175         Validation A
Epoch: 51         Training Loss: 1.272234         Validation Loss: 0.918432         Validation A
Epoch: 52         Training Loss: 1.247608         Validation Loss: 0.861602         Validation A
Epoch: 53         Training Loss: 1.294649         Validation Loss: 0.875606         Validation A
Epoch: 54         Training Loss: 1.243491         Validation Loss: 0.786732         Validation A
Epoch: 55         Training Loss: 1.243947         Validation Loss: 0.847915         Validation A
Epoch: 56         Training Loss: 1.225638         Validation Loss: 0.910021         Validation A
```

```
Epoch: 57         Training Loss: 1.229640         Validation Loss: 0.833956         Validation A
Epoch: 58         Training Loss: 1.246513         Validation Loss: 0.822229         Validation A
Epoch: 59         Training Loss: 1.233181         Validation Loss: 0.874539         Validation A
Epoch: 60         Training Loss: 1.251836         Validation Loss: 0.839039         Validation A
Epoch: 61         Training Loss: 1.228976         Validation Loss: 0.916061         Validation A
Epoch: 62         Training Loss: 1.205238         Validation Loss: 0.814595         Validation A
Epoch: 63         Training Loss: 1.219204         Validation Loss: 0.877125         Validation A
Epoch: 64         Training Loss: 1.184932         Validation Loss: 0.833166         Validation A
Epoch: 65         Training Loss: 1.179863         Validation Loss: 0.882737         Validation A
Epoch: 66         Training Loss: 1.178546         Validation Loss: 0.894260         Validation A
Epoch: 67         Training Loss: 1.179698         Validation Loss: 0.821752         Validation A
Epoch: 68         Training Loss: 1.175331         Validation Loss: 0.906829         Validation A
Epoch: 69         Training Loss: 1.188597         Validation Loss: 0.785788         Validation A
Epoch: 70         Training Loss: 1.166398         Validation Loss: 0.923929         Validation A
Epoch: 71         Training Loss: 1.155743         Validation Loss: 0.892187         Validation A
Epoch: 72         Training Loss: 1.179028         Validation Loss: 0.877222         Validation A
Epoch: 73         Training Loss: 1.167311         Validation Loss: 0.913024         Validation A
Epoch: 74         Training Loss: 1.169264         Validation Loss: 0.902663         Validation A
Epoch: 75         Training Loss: 1.161647         Validation Loss: 0.804652         Validation A
Epoch: 76         Training Loss: 1.155036         Validation Loss: 0.885070         Validation A
Epoch: 77         Training Loss: 1.164256         Validation Loss: 0.869039         Validation A
Epoch: 78         Training Loss: 1.151888         Validation Loss: 0.904351         Validation A
Epoch: 79         Training Loss: 1.120381         Validation Loss: 0.879150         Validation A
Epoch: 80         Training Loss: 1.137585         Validation Loss: 0.887680         Validation A
Epoch: 81         Training Loss: 1.133347         Validation Loss: 0.887100         Validation A
Epoch: 82         Training Loss: 1.124412         Validation Loss: 0.897556         Validation A
Epoch: 83         Training Loss: 1.128219         Validation Loss: 0.837869         Validation A
Epoch: 84         Training Loss: 1.129614         Validation Loss: 0.828445         Validation A
Epoch: 85         Training Loss: 1.113989         Validation Loss: 0.865107         Validation A
Epoch: 86         Training Loss: 1.141778         Validation Loss: 0.871239         Validation A
Epoch: 87         Training Loss: 1.121677         Validation Loss: 0.856440         Validation A
Epoch: 88         Training Loss: 1.124460         Validation Loss: 0.818129         Validation A
Epoch: 89         Training Loss: 1.119911         Validation Loss: 0.821620         Validation A
Epoch: 90         Training Loss: 1.127928         Validation Loss: 0.914855         Validation A
Epoch: 91         Training Loss: 1.074187         Validation Loss: 0.828309         Validation A
Epoch: 92         Training Loss: 1.111570         Validation Loss: 0.864336         Validation A
Epoch: 93         Training Loss: 1.090127         Validation Loss: 0.852935         Validation A
Epoch: 94         Training Loss: 1.093921         Validation Loss: 0.890560         Validation A
Epoch: 95         Training Loss: 1.114071         Validation Loss: 0.818762         Validation A
Epoch: 96         Training Loss: 1.110500         Validation Loss: 0.871621         Validation A
Epoch: 97         Training Loss: 1.071164         Validation Loss: 0.906091         Validation A
Epoch: 98         Training Loss: 1.078476         Validation Loss: 0.847184         Validation A
Epoch: 99         Training Loss: 1.090338         Validation Loss: 0.873513         Validation A
Epoch: 100         Training Loss: 1.096459         Validation Loss: 0.825962         Validation
Epoch: 101         Training Loss: 1.051832         Validation Loss: 0.798784         Validation
Epoch: 102         Training Loss: 1.058251         Validation Loss: 0.874341         Validation
Epoch: 103         Training Loss: 1.065672         Validation Loss: 0.838769         Validation
Epoch: 104         Training Loss: 1.067207         Validation Loss: 0.787508         Validation
```

```
Epoch: 105        Training Loss: 1.069818        Validation Loss: 0.852506        Validation
Epoch: 106        Training Loss: 1.033501        Validation Loss: 0.891365        Validation
Epoch: 107        Training Loss: 1.056100        Validation Loss: 0.869576        Validation
Epoch: 108        Training Loss: 1.053541        Validation Loss: 0.835647        Validation
Epoch: 109        Training Loss: 1.058026        Validation Loss: 0.851428        Validation
Epoch: 110        Training Loss: 1.049820        Validation Loss: 0.872159        Validation
Epoch: 111        Training Loss: 1.029159        Validation Loss: 0.801336        Validation
Epoch: 112        Training Loss: 1.052759        Validation Loss: 0.829640        Validation
Epoch: 113        Training Loss: 1.048656        Validation Loss: 0.884582        Validation
Epoch: 114        Training Loss: 1.037304        Validation Loss: 0.784785        Validation
Epoch: 115        Training Loss: 1.029194        Validation Loss: 0.825595        Validation
Epoch: 116        Training Loss: 1.021951        Validation Loss: 0.835324        Validation
Epoch: 117        Training Loss: 1.037267        Validation Loss: 0.909042        Validation
Epoch: 118        Training Loss: 1.034168        Validation Loss: 0.842677        Validation
Epoch: 119        Training Loss: 1.011630        Validation Loss: 0.863215        Validation
Epoch: 120        Training Loss: 1.053097        Validation Loss: 0.802961        Validation
Epoch: 121        Training Loss: 1.020529        Validation Loss: 0.907416        Validation
Epoch: 122        Training Loss: 1.041613        Validation Loss: 0.964690        Validation
Epoch: 123        Training Loss: 1.024317        Validation Loss: 0.855272        Validation
Epoch: 124        Training Loss: 1.022575        Validation Loss: 0.839153        Validation
Epoch: 125        Training Loss: 1.032861        Validation Loss: 0.913314        Validation
Epoch: 126        Training Loss: 0.984614        Validation Loss: 0.869581        Validation
Epoch: 127        Training Loss: 1.035380        Validation Loss: 0.849176        Validation
Epoch: 128        Training Loss: 0.980110        Validation Loss: 0.872390        Validation
Epoch: 129        Training Loss: 0.976327        Validation Loss: 0.876102        Validation
Epoch: 130        Training Loss: 0.987447        Validation Loss: 0.766225        Validation
Epoch: 131        Training Loss: 1.022517        Validation Loss: 0.827656        Validation
Epoch: 132        Training Loss: 0.971149        Validation Loss: 0.798763        Validation
Epoch: 133        Training Loss: 1.009829        Validation Loss: 0.876827        Validation
Epoch: 134        Training Loss: 0.960093        Validation Loss: 0.937074        Validation
Epoch: 135        Training Loss: 0.977952        Validation Loss: 0.805697        Validation
Epoch: 136        Training Loss: 0.945256        Validation Loss: 0.848456        Validation
Epoch: 137        Training Loss: 0.984060        Validation Loss: 0.827487        Validation
Epoch: 138        Training Loss: 0.967931        Validation Loss: 0.863880        Validation
Epoch: 139        Training Loss: 0.993942        Validation Loss: 0.838410        Validation
Epoch: 140        Training Loss: 0.958190        Validation Loss: 0.831148        Validation
Epoch: 141        Training Loss: 0.958597        Validation Loss: 0.880443        Validation
Epoch: 142        Training Loss: 0.958433        Validation Loss: 0.770863        Validation
Epoch: 143        Training Loss: 0.945799        Validation Loss: 0.839779        Validation
Epoch: 144        Training Loss: 0.973686        Validation Loss: 0.836786        Validation
Epoch: 145        Training Loss: 0.967472        Validation Loss: 0.922112        Validation
Epoch: 146        Training Loss: 0.984392        Validation Loss: 0.864949        Validation
Epoch: 147        Training Loss: 0.984810        Validation Loss: 0.745232        Validation
Epoch: 148        Training Loss: 0.966439        Validation Loss: 0.841016        Validation
Epoch: 149        Training Loss: 0.942397        Validation Loss: 0.824959        Validation
Epoch: 150        Training Loss: 0.966647        Validation Loss: 0.848369        Validation
Epoch: 151        Training Loss: 0.957443        Validation Loss: 0.834324        Validation
Epoch: 152        Training Loss: 0.959414        Validation Loss: 0.818835        Validation
```

```
Epoch: 153        Training Loss: 0.927498        Validation Loss: 0.851661        Validation
Epoch: 154        Training Loss: 0.979422        Validation Loss: 0.913471        Validation
Epoch: 155        Training Loss: 0.962420        Validation Loss: 0.890850        Validation
Epoch: 156        Training Loss: 0.925692        Validation Loss: 0.879071        Validation
Epoch: 157        Training Loss: 0.965130        Validation Loss: 0.847183        Validation
Epoch: 158        Training Loss: 0.950044        Validation Loss: 0.808831        Validation
Epoch: 159        Training Loss: 0.943548        Validation Loss: 0.830338        Validation
Epoch: 160        Training Loss: 0.905804        Validation Loss: 0.882916        Validation
Epoch: 161        Training Loss: 0.937819        Validation Loss: 0.866552        Validation
Epoch: 162        Training Loss: 0.894047        Validation Loss: 0.833897        Validation
Epoch: 163        Training Loss: 0.926091        Validation Loss: 0.858849        Validation
Epoch: 164        Training Loss: 0.944304        Validation Loss: 0.800117        Validation
Epoch: 165        Training Loss: 0.945319        Validation Loss: 0.855845        Validation
Epoch: 166        Training Loss: 0.943876        Validation Loss: 0.896114        Validation
Epoch: 167        Training Loss: 0.919474        Validation Loss: 0.840434        Validation
Epoch: 168        Training Loss: 0.948620        Validation Loss: 0.936812        Validation
Epoch: 169        Training Loss: 0.930282        Validation Loss: 0.911792        Validation
Epoch: 170        Training Loss: 0.910861        Validation Loss: 0.853065        Validation
Epoch: 171        Training Loss: 0.947704        Validation Loss: 0.871225        Validation
Epoch: 172        Training Loss: 0.912299        Validation Loss: 0.870394        Validation
Epoch: 173        Training Loss: 0.927816        Validation Loss: 0.837104        Validation
Epoch: 174        Training Loss: 0.922887        Validation Loss: 0.953049        Validation
Epoch: 175        Training Loss: 0.917136        Validation Loss: 0.885836        Validation
Epoch: 176        Training Loss: 0.933738        Validation Loss: 0.881380        Validation
Epoch: 177        Training Loss: 0.895659        Validation Loss: 0.781877        Validation
Epoch: 178        Training Loss: 0.901494        Validation Loss: 0.782590        Validation
Epoch: 179        Training Loss: 0.959918        Validation Loss: 0.861145        Validation
Epoch: 180        Training Loss: 0.881766        Validation Loss: 0.875878        Validation
Epoch: 181        Training Loss: 0.879401        Validation Loss: 0.841445        Validation
Epoch: 182        Training Loss: 0.895581        Validation Loss: 0.872405        Validation
Epoch: 183        Training Loss: 0.922558        Validation Loss: 0.861642        Validation
Epoch: 184        Training Loss: 0.901537        Validation Loss: 0.935880        Validation
Epoch: 185        Training Loss: 0.869326        Validation Loss: 0.809489        Validation
Epoch: 186        Training Loss: 0.865229        Validation Loss: 0.878388        Validation
Epoch: 187        Training Loss: 0.912063        Validation Loss: 0.826799        Validation
Epoch: 188        Training Loss: 0.904545        Validation Loss: 0.866770        Validation
Epoch: 189        Training Loss: 0.892413        Validation Loss: 0.809744        Validation
Epoch: 190        Training Loss: 0.885405        Validation Loss: 0.875728        Validation
Epoch: 191        Training Loss: 0.885403        Validation Loss: 0.869741        Validation
Epoch: 192        Training Loss: 0.899547        Validation Loss: 0.801121        Validation
Epoch: 193        Training Loss: 0.871854        Validation Loss: 0.894483        Validation
Epoch: 194        Training Loss: 0.882282        Validation Loss: 0.823136        Validation
Epoch: 195        Training Loss: 0.885834        Validation Loss: 0.916837        Validation
Epoch: 196        Training Loss: 0.886272        Validation Loss: 0.815705        Validation
Epoch: 197        Training Loss: 0.829622        Validation Loss: 0.820222        Validation
Epoch: 198        Training Loss: 0.874619        Validation Loss: 0.852219        Validation
Epoch: 199        Training Loss: 0.871274        Validation Loss: 0.875762        Validation
Epoch: 200        Training Loss: 0.882420        Validation Loss: 0.890313        Validation
```

```
Out[0]: <All keys matched successfully>
```

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 [0]: 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))

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

```
Test Loss: 0.835999


Test Accuracy: 77% (650/836)
```

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.

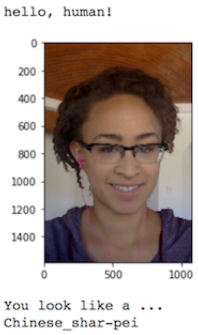
```
In [0]: ### TODO: Write a function that takes a path to an image as input
        ### and returns the dog breed that is predicted by the model.

        from PIL import Image
        import torchvision.transforms as transforms
        from torch.autograd import Variable

        # Set PIL to be tolerant of image files that are truncated.
        from PIL import ImageFile
        ImageFile.LOAD_TRUNCATED_IMAGES = True

        # 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 data_transfer['train'].classes]

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

            image = Image.open(img_path)
            transform = transforms.Compose([
                transforms.ToTensor(),
                transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

            ])

            image = transform(image)
            image = image.unsqueeze(0)
            if use_cuda:
                image = image.cuda()
            prediction = model_transfer(image)
            prediction_class = np.argmax(prediction.cpu().detach().numpy()) # predicted class in

            return class_names[prediction_class]
```

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

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



Sample Human Output

1.1.18 (IMPLEMENTATION) Write your Algorithm

```
In [0]: ### 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

            isFace = face_detector(img_path)
            isDog = dog_detector(img_path)
            if isDog:
              breed = predict_breed_transfer(img_path)
              return "It is a dog. Detected breed is: " + breed
            elif isFace:
              breed = predict_breed_transfer(img_path)
              return "It is a human. Detected breed is: " + breed
            else:
              return "It is neither dog nor human."
```

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.

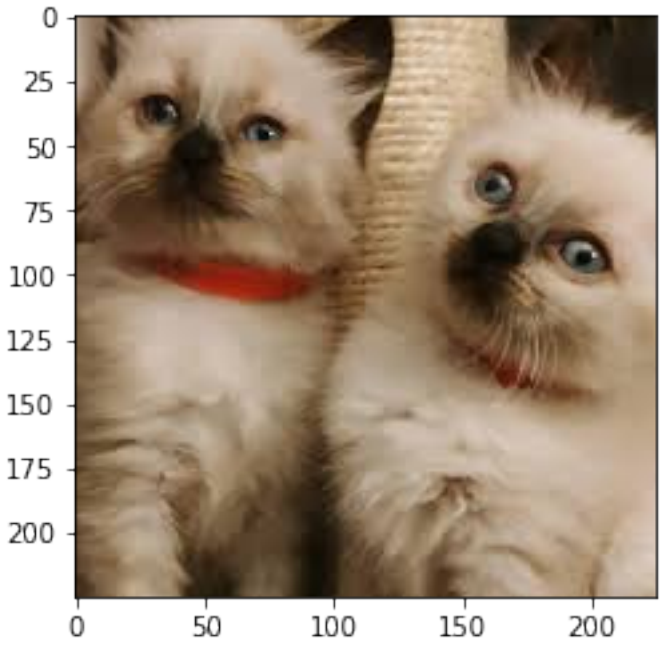
Answer: I tested network on some cats, dogs, and human images. The network did not detect cats neither as dogs nor as human, which is very good result. The most of dog breeds were recognized correctly. On the other hand humans were not really similar to detected dog's breed.

To improve algorightm we could try the following: - increase number of samples used for learning the network - increase number of features (convolutional layer) detected by the network - add additional fully connected layer - use technologies of model ensembling

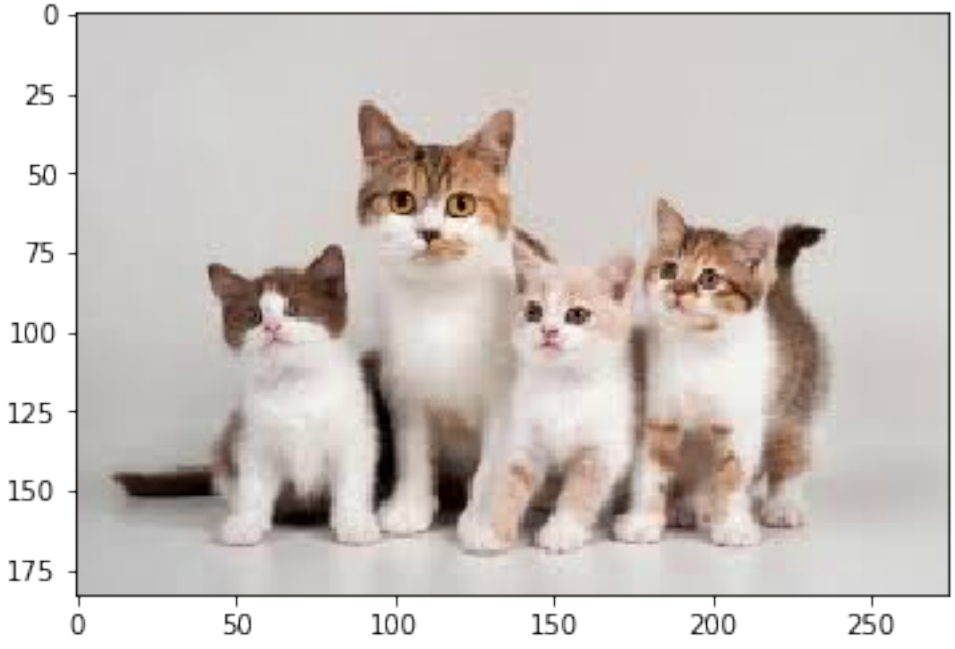
In [3]:
```
import numpy as np
from glob import glob

# load filenames for human and dog images
cat_files = np.array(glob("/content/drive/My Drive/Colab Notebooks/data/cats/*"))
human_files = np.array(glob("/content/drive/My Drive/Colab Notebooks/data/humans/*"))
dog_files = np.array(glob("/content/drive/My Drive/Colab Notebooks/data/dogs/*"))

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

```
There are 4 total cat images.
There are 4 total human images.
There are 4 total dog images.
```

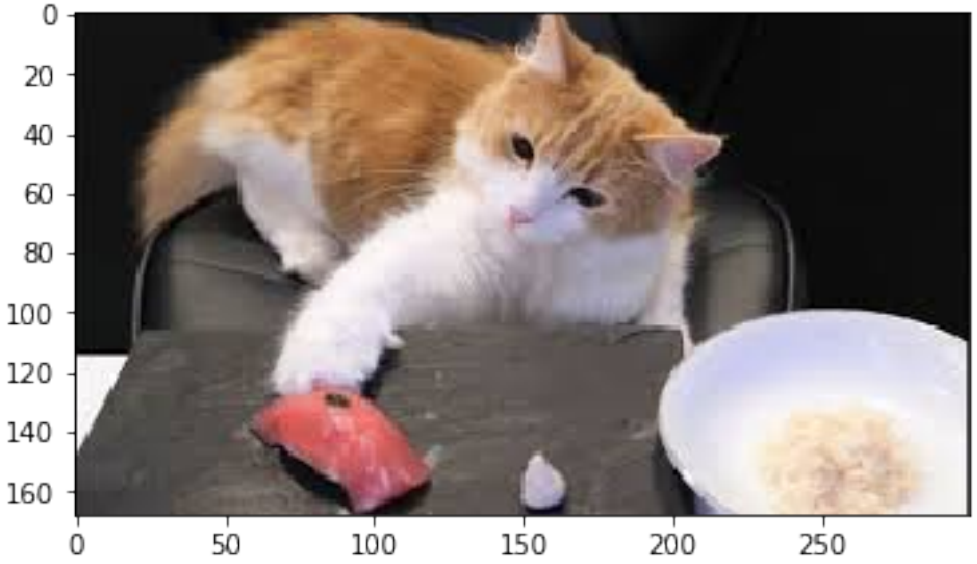
In [22]:
```
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.
import cv2
import matplotlib.pyplot as plt
%matplotlib inline

## suggested code, below
for file in np.hstack((cat_files, human_files, dog_files)):
    img = cv2.imread(file)
    cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    plt.imshow(cv_rgb)
    plt.show()
    result = run_app(file)
    print(result)
```




```
It is neither dog nor human.
```



```
It is neither dog nor human.
```

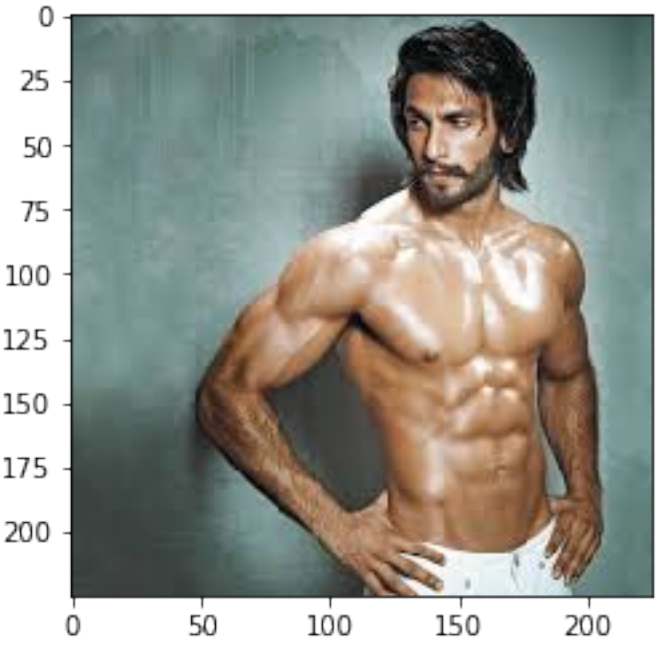



```
It is neither dog nor human.
```

```
It is neither dog nor human.
```

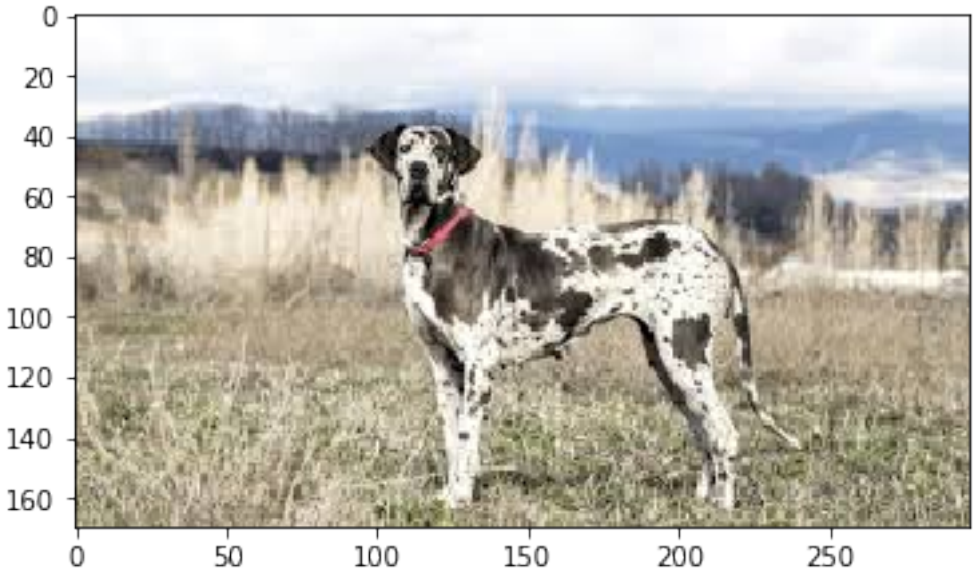

```
It is a human. Detected breed is: Akita
```

```
It is a human. Detected breed is: Akita
```

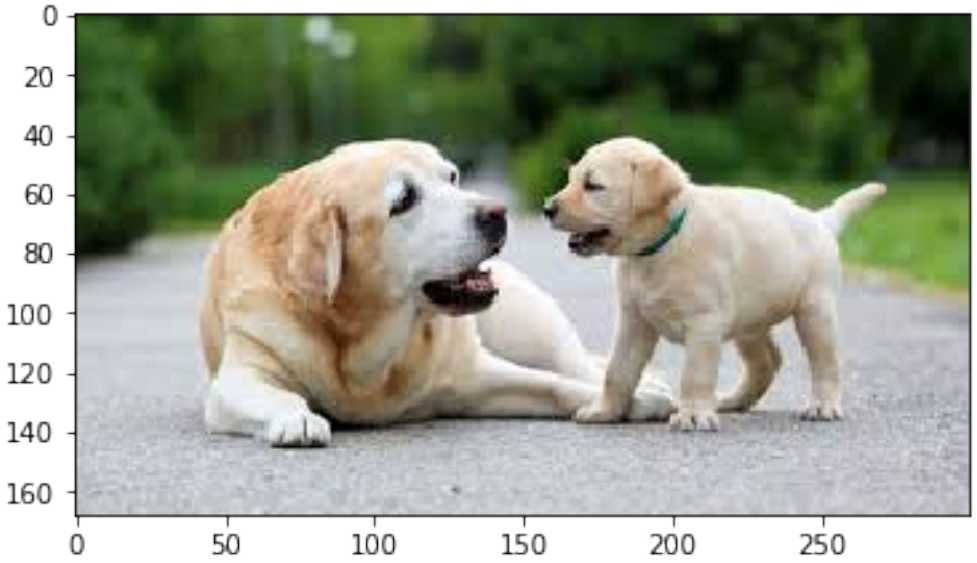



```
It is a human. Detected breed is: Chinese shar-pei
```


```
It is a human. Detected breed is: Chinese crested
```




```
It is a dog. Detected breed is: Great dane
```




```
It is a dog. Detected breed is: Anatolian shepherd dog
```


```
It is a dog. Detected breed is: Otterhound
```

```
It is a dog. Detected breed is: German shepherd dog
```