dog_app

January 4, 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:

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`.

1.1.1 Note:

This project was run in the udacity Dog Project Workspace. Below is some code that will help run outside of that environment. I ran this project in Amazon Sagemaker as well.

```
In [1]: # Uncomment below to get data and unzip files
        #!wget https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/dogImages.zip
        #!wget https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/lfw.zip
        #!unzip dogImages.zip
        #!unzip lfw.zip
```

```
In [ ]: # Uncomment below to clone the repo this project is from.
        # !git clone https://github.com/udacity/deep-learning-v2-pytorch.git
        # !cd deep-learning-v2-pytorch/project-dog-classification
```

```
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

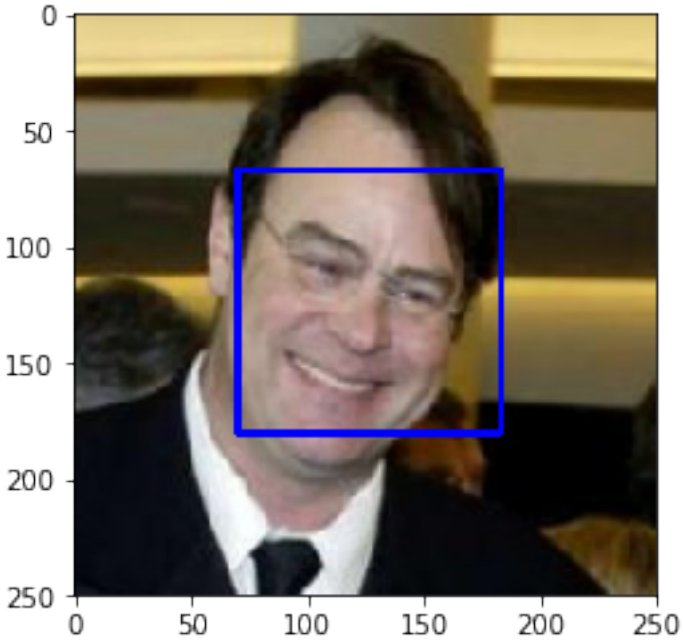
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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)

# 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.

1.1.2 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
```

1.1.3 (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 [7]: from tqdm import tqdm

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

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

        ## TODO: Test the performance of the face_detector algorithm
        ## on the images in human_files_short and dog_files_short.
        humanfaces = []
        for img_path in human_files_short:
            humanfaces.append(face_detector(img_path))
        print('We found human faces in {} of 100 human face photos: {}% found'.format(sum(humanf


        dogfaces = []
        for img_path in dog_files_short:
            dogfaces.append(face_detector(img_path))
        print('We found dog faces in {} of 100 dog face photos: {}% found'.format(sum(dogfaces),
```

```
We found human faces in 98 of 100 human face photos: 98.0% found
We found dog faces in 17 of 100 dog face photos: 17.0% found
```

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 [8]: ### (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.

1.1.4 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 [4]: 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 [00:28<00:00, 19439252.59it/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.

1.1.5 (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 [5]: 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
```

1.1.6 (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 [6]: ### 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
```

1.1.7 (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 [12]: ### 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 97.0% of images.
In the human images we detected dogs in 1.0% 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 [13]: ### (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!

1.1.8 (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 [142]: 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
          transform_pipeline = transforms.Compose([transforms.RandomResizedCrop(224),
                                                   transforms.RandomHorizontalFlip(),
                                                   transforms.RandomRotation(10),
                                                   transforms.ToTensor(),
                                                   transforms.Normalize(mean=[0.485, 0.456, 0.40
                                                  ])
          transform_pipeline_test = transforms.Compose([transforms.RandomResizedCrop(224),
                                                        transforms.ToTensor(),
                                                        transforms.Normalize(mean=[0.485, 0.456,
                                                       ])

          # https://pytorch.org/docs/stable/torchvision/datasets.html#imagefolder
          train_df = datasets.ImageFolder('dogImages/train', transform=transform_pipeline)
          val_df = datasets.ImageFolder('dogImages/valid', transform=transform_pipeline_test)
          test_df = datasets.ImageFolder('dogImages/test', transform=transform_pipeline_test)

          # batch size of 32 is a good starting point, and you should also try with 64, 128, and
          # https://stats.stackexchange.com/questions/164876/tradeoff-batch-size-vs-number-of-it
          bsize = 64

          train_dl = torch.utils.data.DataLoader(dataset = train_df, batch_size= bsize, shuffle
          val_dl = torch.utils.data.DataLoader(dataset = val_df, batch_size= 15)
          test_dl = torch.utils.data.DataLoader(dataset = test_df, batch_size= 15)

          loaders_scratch = {'train': train_dl, 'valid': val_dl, 'test': test_dl}
```

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: How does your code resize the images (by cropping, stretching, etc)? What size did you pick for the input tensor, and why? - I used RandomResizedCrop which cropes an image and extracts between 80-100% of the image, while also distorting the aspect ratio anywhere between 0.75 and 1.333. The croped image is then resized to 224 x 224. I chose this size to keep the input tensor smaller to avoid longer training time if possible.

Did you decide to augment the dataset? If so, how (through translations, flips, rotations, etc)? If not, why not? - I decided to include a horizontal flip as this flip would still make the image a valid training example of a dog. I also included a slight rotation of 10 degrees because i thought this could help augment that dataset. I decided to try to the normalization as well as implemented in the VGG16. I figure these transformations would offer a certain amount of distortion to the images that would make the model more generalizable. I did not include the flip or the slight rotation in the testing or validation transformations.

```
In [143]: import torch.nn as nn
          import torch.nn.functional as F

          # define the CNN architecture
          class Net(nn.Module):
              ### TODO: choose an architecture, and complete the class
              def __init__(self):
                  super(Net, self).__init__()
                  ## Define layers of a CNN
                  # convolutional layer size:
                      #https://medium.com/@iamvarman/how-to-calculate-the-number-of-parameters-i
                      #https://towardsdatascience.com/intuitively-understanding-convolutions-for
                  dep1 = 16
                  dep2 = dep1 * 2
                  dep3 = dep2 * 2
                  dep4 = dep3 * 2
                  dep5 = dep4 * 2

                  # Dropout Layer
                  self.drop = nn.Dropout(0.25)

                  # Batch normalization to help each layer to learn on a more stable distributio
                  # and would accelerate the training of the network -
                  # https://mlexplained.com/2018/01/10/an-intuitive-explanation-of-why-batch-nor
                  self.norm1 = nn.BatchNorm2d(dep1)
                  self.norm2 = nn.BatchNorm2d(dep2)
                  self.norm3 = nn.BatchNorm2d(dep3)
                  self.norm4 = nn.BatchNorm2d(dep4)
                  self.norm5 = nn.BatchNorm2d(dep5)

                  # pooling layer
                  self.pool = nn.MaxPool2d(kernel_size=2)

                  # convolutional layers. Tried 3-7 covnets, but landed on 5 after trial and err
```

```
        self.con1 = nn.Conv2d(in_channels=3, out_channels=dep1, kernel_size=3,stride =
        self.con2 = nn.Conv2d(in_channels=dep1, out_channels=dep2, kernel_size=3,strid
        self.con3 = nn.Conv2d(in_channels=dep2, out_channels=dep3, kernel_size=3, stri
        self.con4 = nn.Conv2d(in_channels=dep3, out_channels=dep4, kernel_size=3,strid
        self.con5 = nn.Conv2d(in_channels=dep4, out_channels=dep5, kernel_size=3, stri

        # activation function: Relu - choosen as default because it is the best known
        self.relu = nn.ReLU()

        # fully connected layer to give dog breed prediction
        self.fullconn = nn.Linear(in_features=dep5 * 5 * 5, out_features=133)

    def forward(self, x):
    ## Define forward behavior
        dep1 = 16
        dep2 = dep1 * 2
        dep3 = dep2 * 2
        dep4 = dep3 * 2
        dep5 = dep4 * 2

        x = self.con1(x)
        x = self.relu(x)
        x = self.norm1(x)
        x = self.pool(x)

        x = self.con2(x)
        x = self.relu(x)
        x = self.norm2(x)
        x = self.pool(x)

        x = self.con3(x)
        x = self.relu(x)
        x = self.norm3(x)
        x = self.pool(x)

        x = self.con4(x)
        x = self.relu(x)
        x = self.norm4(x)
        x = self.pool(x)

        x = self.con5(x)
        x = self.relu(x)
        x = self.norm5(x)
        x = self.pool(x)

        #print(x.shape)
        x = x.view(-1, dep5 * 5 * 5)
```

```
        x = self.drop(x)
        x = self.fullconn(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()
```

1.1.9 (IMPLEMENTATION) Model Architecture

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

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

Answer:

- I adapted code from an example basic model for image classification from: https://heartbeat.fritz.ai/basics-of-image-classification-with-pytorch-2f8973c51864
- I used relu activations for the convolutional layers as i know they are the preferred activation function according the the free deep learning course at udacity. I also of course used convolution layers as they are essential in image based deep learning.
- I added in batch normalization at each layer as i understand it can help improve the performance of models. Reference: https://medium.com/deeper-learning/glossary-of-deep-learning-batch-normalisation-8266dcd2fa82
- The pooling layer was used to reduce the size of the parameters in my model. The model has 5 convolutional layers so I assumed it could benefit from the max pooling. It also was included in the basic model I was following.
- To land at the number of covnets I tried 3-7 covnet layers and 5 performed better than 3 or 4, while 6 and 7 covnet layers had too much pooling. I tried to space out the pooling functions, but the net result was disappointing.
- I used a linear output for the classifcation as it was used in the basic example
- I used the shape function to determine the size of the final model to ensure I had the appropriate final dimensions.
- I used out_features = 133 as that is the number of dog breeds I am training for.
- I tried many different starting convolutions out channels, but I found that 16 was appropriate as it is a 4 x 4 output and lead to a reasonable final output channel size over the 5 layers.
- I included dropout to ensure the model was not overfitting the training set and would give generalizable results. I dialed up the dropout rate a little as I found some overfitting without it.
- My process involved lots of trial and error to land on a model that improved enough to beat the threshold of 10% accuracy on the test set.

1.1.10 (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 [144]: import torch.optim as optim

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

          ### TODO: select optimizer
          optimizer_scratch = torch.optim.SGD(model_scratch.parameters(), lr=0.1, momentum=0.9)
          scheduler_scratch = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer_scratch, 'min
```

1.1.11 (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 [7]: # the following import is required for training to be robust to truncated images
        from PIL import ImageFile
        ImageFile.LOAD_TRUNCATED_IMAGES = True
        import time
        def train(n_epochs, loaders, model, optimizer, scheduler, criterion, use_cuda, save_path
            """returns trained model"""
            # initialize tracker for minimum validation loss
            valid_loss_min = np.Inf
            print("Cuda is enabled: {}".format(use_cuda))
            for epoch in range(1, n_epochs+1):
                start = time.time()

                # 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']):

                    if use_cuda: data, target = data.cuda(), target.cuda()
                    ## find the loss and update the model parameters accordingly
                    # reference: https://stackoverflow.com/questions/48001598/why-do-we-need-to-
                    optimizer.zero_grad()
                    loss = criterion(model(data), target)
                    loss.backward()
```

```
            optimizer.step()

            ## record the average training loss, using something like

            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']):

            if use_cuda:  data, target = data.cuda(), target.cuda()
            valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (criterion(model(data), t

        # Return training update message
        scheduler.step(valid_loss)
        end = time.time()

        print('Epoch: {} of {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}. Epoch
            epoch,
            n_epochs,
            train_loss,
            valid_loss,
            (end - start)/60,
            (end - start)/60*(n_epochs-epoch)
            ))

        if valid_loss_min > valid_loss:
            print('Saving improved model: Validation loss decreased to {} from {}. Reduc
            valid_loss_min = valid_loss
            torch.save(model.state_dict(), save_path)
        else: print('Validation Loss not impoved this epoch. Model update not saved.')

    # return trained model
    return model
```

In [145]:
```
# train the model
model_scratch = train(100, loaders_scratch, model_scratch, optimizer_scratch, schedule
                      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'))
```

```
Cuda is enabled: True
Epoch: 1 of 100         Training Loss: 5.403888         Validation Loss: 4.851044. Epoch length:
Saving improved model: Validation loss decreased to 4.851044178009033 from inf. Reduced: inf
```

```
Epoch: 2 of 100         Training Loss: 4.772555         Validation Loss: 4.749016. Epoch length:
Saving improved model: Validation loss decreased to 4.749016284942627 from 4.851044178009033. Re
Epoch: 3 of 100         Training Loss: 4.691324         Validation Loss: 4.684814. Epoch length:
Saving improved model: Validation loss decreased to 4.684813976287842 from 4.749016284942627. Re
Epoch: 4 of 100         Training Loss: 4.625930         Validation Loss: 4.742805. Epoch length:
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 5 of 100         Training Loss: 4.598840         Validation Loss: 4.669291. Epoch length:
Saving improved model: Validation loss decreased to 4.669291019439697 from 4.684813976287842. Re
Epoch: 6 of 100         Training Loss: 4.591786         Validation Loss: 4.569451. Epoch length:
Saving improved model: Validation loss decreased to 4.569450855255127 from 4.669291019439697. Re
Epoch: 7 of 100         Training Loss: 4.520760         Validation Loss: 4.739399. Epoch length:
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 8 of 100         Training Loss: 4.521183         Validation Loss: 4.647558. Epoch length:
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 9 of 100         Training Loss: 4.471560         Validation Loss: 4.623035. Epoch length:
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 10 of 100         Training Loss: 4.475198         Validation Loss: 4.506001. Epoch length
Saving improved model: Validation loss decreased to 4.506000518798828 from 4.569450855255127. Re
Epoch: 11 of 100         Training Loss: 4.491115         Validation Loss: 4.518075. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 12 of 100         Training Loss: 4.435848         Validation Loss: 4.673770. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 13 of 100         Training Loss: 4.422563         Validation Loss: 4.594373. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 14 of 100         Training Loss: 4.363161         Validation Loss: 4.567577. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 15 of 100         Training Loss: 4.330350         Validation Loss: 4.466792. Epoch length
Saving improved model: Validation loss decreased to 4.466792106628418 from 4.506000518798828. Re
Epoch: 16 of 100         Training Loss: 4.352002         Validation Loss: 4.572763. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 17 of 100         Training Loss: 4.317725         Validation Loss: 4.678296. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 18 of 100         Training Loss: 4.284043         Validation Loss: 4.513853. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 19 of 100         Training Loss: 4.242736         Validation Loss: 4.412345. Epoch length
Saving improved model: Validation loss decreased to 4.4123454093933105 from 4.466792106628418. R
Epoch: 20 of 100         Training Loss: 4.239399         Validation Loss: 4.439794. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 21 of 100         Training Loss: 4.276450         Validation Loss: 4.479249. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 22 of 100         Training Loss: 4.242011         Validation Loss: 4.590837. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 23 of 100         Training Loss: 4.202572         Validation Loss: 4.485117. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 24 of 100         Training Loss: 4.152402         Validation Loss: 4.528109. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 25 of 100         Training Loss: 4.105244         Validation Loss: 4.382455. Epoch length
Saving improved model: Validation loss decreased to 4.382454872131348 from 4.4123454093933105. R
```

```
Epoch: 26 of 100         Training Loss: 4.079582         Validation Loss: 4.631456. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 27 of 100         Training Loss: 4.073783         Validation Loss: 4.307298. Epoch length
Saving improved model: Validation loss decreased to 4.307297706604004 from 4.382454872131348. Re
Epoch: 28 of 100         Training Loss: 4.048481         Validation Loss: 4.425885. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 29 of 100         Training Loss: 4.059120         Validation Loss: 4.309983. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 30 of 100         Training Loss: 4.055812         Validation Loss: 4.517257. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 31 of 100         Training Loss: 4.029723         Validation Loss: 4.329742. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 32 of 100         Training Loss: 3.996455         Validation Loss: 4.259314. Epoch length
Saving improved model: Validation loss decreased to 4.259314060211182 from 4.307297706604004. Re
Epoch: 33 of 100         Training Loss: 4.002159         Validation Loss: 4.442115. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 34 of 100         Training Loss: 3.945166         Validation Loss: 4.276726. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 35 of 100         Training Loss: 3.925663         Validation Loss: 4.159507. Epoch length
Saving improved model: Validation loss decreased to 4.159506797790527 from 4.259314060211182. Re
Epoch: 36 of 100         Training Loss: 3.927055         Validation Loss: 4.286820. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 37 of 100         Training Loss: 3.931464         Validation Loss: 4.364314. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 38 of 100         Training Loss: 3.914646         Validation Loss: 4.258191. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 39 of 100         Training Loss: 3.911138         Validation Loss: 4.266344. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 40 of 100         Training Loss: 3.850371         Validation Loss: 4.295825. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 41 of 100         Training Loss: 3.817836         Validation Loss: 4.238762. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 42 of 100         Training Loss: 3.865620         Validation Loss: 4.219911. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 43 of 100         Training Loss: 3.827065         Validation Loss: 4.342802. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 44 of 100         Training Loss: 3.881278         Validation Loss: 4.396882. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 45 of 100         Training Loss: 3.834246         Validation Loss: 4.441874. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 46 of 100         Training Loss: 3.795902         Validation Loss: 4.623216. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 47 of 100         Training Loss: 3.270597         Validation Loss: 3.638112. Epoch length
Saving improved model: Validation loss decreased to 3.6381118297576904 from 4.159506797790527. R
Epoch: 48 of 100         Training Loss: 3.054835         Validation Loss: 3.614161. Epoch length
Saving improved model: Validation loss decreased to 3.6141607761383057 from 3.6381118297576904.
Epoch: 49 of 100         Training Loss: 3.047942         Validation Loss: 3.528508. Epoch length
Saving improved model: Validation loss decreased to 3.528508186340332 from 3.6141607761383057. R
```

```
Epoch: 50 of 100         Training Loss: 3.034617         Validation Loss: 3.520871. Epoch length
Saving improved model: Validation loss decreased to 3.520871162414551 from 3.528508186340332. Re
Epoch: 51 of 100         Training Loss: 2.955770         Validation Loss: 3.552707. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 52 of 100         Training Loss: 2.934195         Validation Loss: 3.505131. Epoch length
Saving improved model: Validation loss decreased to 3.505131483078003 from 3.520871162414551. Re
Epoch: 53 of 100         Training Loss: 2.956889         Validation Loss: 3.520300. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 54 of 100         Training Loss: 2.932089         Validation Loss: 3.539931. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 55 of 100         Training Loss: 2.897115         Validation Loss: 3.547032. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 56 of 100         Training Loss: 2.905297         Validation Loss: 3.636431. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 57 of 100         Training Loss: 2.890358         Validation Loss: 3.464749. Epoch length
Saving improved model: Validation loss decreased to 3.4647488594055176 from 3.505131483078003. R
Epoch: 58 of 100         Training Loss: 2.886464         Validation Loss: 3.514436. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 59 of 100         Training Loss: 2.903892         Validation Loss: 3.472959. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 60 of 100         Training Loss: 2.875675         Validation Loss: 3.475946. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 61 of 100         Training Loss: 2.828197         Validation Loss: 3.502036. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 62 of 100         Training Loss: 2.815260         Validation Loss: 3.382159. Epoch length
Saving improved model: Validation loss decreased to 3.382159471511841 from 3.4647488594055176. R
Epoch: 63 of 100         Training Loss: 2.833219         Validation Loss: 3.429278. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 64 of 100         Training Loss: 2.817970         Validation Loss: 3.396978. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 65 of 100         Training Loss: 2.803035         Validation Loss: 3.492963. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 66 of 100         Training Loss: 2.787789         Validation Loss: 3.454276. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 67 of 100         Training Loss: 2.805902         Validation Loss: 3.412423. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 68 of 100         Training Loss: 2.792814         Validation Loss: 3.500940. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 69 of 100         Training Loss: 2.782984         Validation Loss: 3.531445. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 70 of 100         Training Loss: 2.764810         Validation Loss: 3.412306. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 71 of 100         Training Loss: 2.757029         Validation Loss: 3.338508. Epoch length
Saving improved model: Validation loss decreased to 3.338507890701294 from 3.382159471511841. Re
Epoch: 72 of 100         Training Loss: 2.724384         Validation Loss: 3.450961. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 73 of 100         Training Loss: 2.743159         Validation Loss: 3.360616. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
```

```
Epoch: 74 of 100         Training Loss: 2.737907         Validation Loss: 3.336989. Epoch length
Saving improved model: Validation loss decreased to 3.336989164352417 from 3.338507890701294. Re
Epoch: 75 of 100         Training Loss: 2.742806         Validation Loss: 3.483648. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 76 of 100         Training Loss: 2.689913         Validation Loss: 3.423991. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 77 of 100         Training Loss: 2.732053         Validation Loss: 3.398944. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 78 of 100         Training Loss: 2.734626         Validation Loss: 3.425650. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 79 of 100         Training Loss: 2.669559         Validation Loss: 3.376423. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 80 of 100         Training Loss: 2.706006         Validation Loss: 3.423292. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 81 of 100         Training Loss: 2.695558         Validation Loss: 3.329628. Epoch length
Saving improved model: Validation loss decreased to 3.329627513885498 from 3.336989164352417. Re
Epoch: 82 of 100         Training Loss: 2.666815         Validation Loss: 3.424276. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 83 of 100         Training Loss: 2.701338         Validation Loss: 3.288738. Epoch length
Saving improved model: Validation loss decreased to 3.2887380123138428 from 3.329627513885498. R
Epoch: 84 of 100         Training Loss: 2.675562         Validation Loss: 3.354362. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 85 of 100         Training Loss: 2.612161         Validation Loss: 3.496212. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 86 of 100         Training Loss: 2.654893         Validation Loss: 3.422738. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 87 of 100         Training Loss: 2.658676         Validation Loss: 3.394997. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 88 of 100         Training Loss: 2.664322         Validation Loss: 3.384317. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 89 of 100         Training Loss: 2.586216         Validation Loss: 3.423728. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 90 of 100         Training Loss: 2.635203         Validation Loss: 3.359294. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 91 of 100         Training Loss: 2.633909         Validation Loss: 3.393426. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 92 of 100         Training Loss: 2.598034         Validation Loss: 3.344603. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 93 of 100         Training Loss: 2.586331         Validation Loss: 3.450483. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 94 of 100         Training Loss: 2.633556         Validation Loss: 3.353187. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 95 of 100         Training Loss: 2.536518         Validation Loss: 3.345610. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 96 of 100         Training Loss: 2.553946         Validation Loss: 3.309890. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 97 of 100         Training Loss: 2.498982         Validation Loss: 3.286573. Epoch length
Saving improved model: Validation loss decreased to 3.2865726947784424 from 3.2887380123138428.
```

```
Epoch: 98 of 100         Training Loss: 2.508299         Validation Loss: 3.383020. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 99 of 100         Training Loss: 2.537921         Validation Loss: 3.273152. Epoch length
Saving improved model: Validation loss decreased to 3.2731521129608154 from 3.2865726947784424.
Epoch: 100 of 100         Training Loss: 2.483848         Validation Loss: 3.260370. Epoch lengt
Saving improved model: Validation loss decreased to 3.2603702545166016 from 3.2731521129608154.
```

1.1.12 (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 [ ]: #model_scratch.load_state_dict(torch.load('model_scratch.pt'))
```

```
In [8]: 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 [146]: # call test function
          test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.308095
```

```
Test Accuracy: 25% (212/836)
```

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.

1.1.13 (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 [10]:
```
## 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
transform_pipeline_train = transforms.Compose([transforms.RandomResizedCrop(224),
                                               transforms.RandomHorizontalFlip(),
                                               transforms.RandomRotation(10),
                                               transforms.ToTensor(),
                                               transforms.Normalize(mean=[0.485, 0.456, 0.406
                                              ])

transform_pipeline_test = transforms.Compose([transforms.RandomResizedCrop(224),
                                               transforms.ToTensor(),
                                               transforms.Normalize(mean=[0.485, 0.456, 0.406
                                              ])

# https://pytorch.org/docs/stable/torchvision/datasets.html#imagefolder
train_df = datasets.ImageFolder('dogImages/train', transform=transform_pipeline_train)
val_df = datasets.ImageFolder('dogImages/valid', transform=transform_pipeline_test)
test_df = datasets.ImageFolder('dogImages/test', transform=transform_pipeline_test)

bsize = 64

train_dl = torch.utils.data.DataLoader(dataset = train_df, batch_size= bsize)
val_dl = torch.utils.data.DataLoader(dataset = val_df, batch_size= 15)
```

```
test_dl = torch.utils.data.DataLoader(dataset = test_df, batch_size= 15)

loaders_transfer = {'train': train_dl, 'valid': val_dl, 'test': test_dl}
```

1.1.14 (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 [14]: import torchvision.models as models
         import torch.nn as nn

         ## TODO: Specify model architecture
         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
         numofbreeds = 133
         # Freeze All Layers
         for parameters in model_transfer.parameters(): parameters.requires_grad = False
         # Replace last layer of classifier to match number of breeds
         # last layer is unfrozen by default
         model_transfer.classifier[6] = nn.Linear(model_transfer.classifier[6].in_features, numo

         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 decided to use the vgg16 model as we used above for the dog breed detector. I decided to use this for transfer learning as it has already shown promise as a dog detector. I assumed it would work well fine tuned to the dogs specifically. This is why I think the architecture would work for this classification problem. - I froze all the layers in the model, but replaced the final linear layer of the classifier with a new linear layer that included the same number of input features, but would output to the number of breeds.

1.1.15 (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 [15]: 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 = torch.optim.SGD(filter(lambda p: p.requires_grad,model_transfer.cl
         scheduler_transfer = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer_transfer, 'mi
```

1.1.16 (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 [17]:
```
# train the model

        model_transfer = train(40, loaders_transfer, model_transfer, optimizer_transfer, schedu
                               criterion_transfer, use_cuda, 'model_transfer.pt')

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

```
Cuda is enabled: True
Epoch: 1 of 40         Training Loss: 136.004471         Validation Loss: 73.496819. Epoch lengt
Saving improved model: Validation loss decreased to 73.49681854248047 from inf. Reduced: inf
Epoch: 2 of 40         Training Loss: 136.289566         Validation Loss: 68.631775. Epoch lengt
Saving improved model: Validation loss decreased to 68.63177490234375 from 73.49681854248047. Re
Epoch: 3 of 40         Training Loss: 125.786407         Validation Loss: 59.717144. Epoch lengt
Saving improved model: Validation loss decreased to 59.71714401245117 from 68.63177490234375. Re
Epoch: 4 of 40         Training Loss: 112.551094         Validation Loss: 59.596256. Epoch lengt
Saving improved model: Validation loss decreased to 59.596256256103516 from 59.71714401245117. R
Epoch: 5 of 40         Training Loss: 104.894920         Validation Loss: 48.786407. Epoch lengt
Saving improved model: Validation loss decreased to 48.786407470703125 from 59.596256256103516.
Epoch: 6 of 40         Training Loss: 98.637985         Validation Loss: 51.042755. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 7 of 40         Training Loss: 102.808235         Validation Loss: 50.899044. Epoch lengt
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 8 of 40         Training Loss: 95.586586         Validation Loss: 46.837708. Epoch length
Saving improved model: Validation loss decreased to 46.83770751953125 from 48.786407470703125. R
Epoch: 9 of 40         Training Loss: 86.417084         Validation Loss: 48.987713. Epoch length
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 10 of 40         Training Loss: 86.785690         Validation Loss: 46.825764. Epoch lengt
Saving improved model: Validation loss decreased to 46.82576370239258 from 46.83770751953125. Re
Epoch: 11 of 40         Training Loss: 82.650826         Validation Loss: 41.881889. Epoch lengt
Saving improved model: Validation loss decreased to 41.88188934326172 from 46.82576370239258. Re
Epoch: 12 of 40         Training Loss: 78.831635         Validation Loss: 39.011021. Epoch lengt
Saving improved model: Validation loss decreased to 39.01102066040039 from 41.88188934326172. Re
Epoch: 13 of 40         Training Loss: 78.180199         Validation Loss: 43.985962. Epoch lengt
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 14 of 40         Training Loss: 79.401825         Validation Loss: 41.724716. Epoch lengt
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 15 of 40         Training Loss: 78.128296         Validation Loss: 41.561474. Epoch lengt
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 16 of 40         Training Loss: 79.009056         Validation Loss: 44.143257. Epoch lengt
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 17 of 40         Training Loss: 76.536736         Validation Loss: 41.524982. Epoch lengt
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 18 of 40         Training Loss: 73.980362         Validation Loss: 42.210766. Epoch lengt
```

```
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 19 of 40         Training Loss: 76.542618         Validation Loss: 41.999866. Epoch lengt
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 20 of 40         Training Loss: 71.908218         Validation Loss: 41.625656. Epoch lengt
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 21 of 40         Training Loss: 71.665771         Validation Loss: 40.235485. Epoch lengt
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 22 of 40         Training Loss: 74.574883         Validation Loss: 46.941109. Epoch lengt
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 23 of 40         Training Loss: 75.377640         Validation Loss: 40.130211. Epoch lengt
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 24 of 40         Training Loss: 46.572563         Validation Loss: 35.622299. Epoch lengt
Saving improved model: Validation loss decreased to 35.62229919433594 from 39.01102066040039. Re
Epoch: 25 of 40         Training Loss: 37.973667         Validation Loss: 31.101675. Epoch lengt
Saving improved model: Validation loss decreased to 31.101675033569336 from 35.62229919433594. R
Epoch: 26 of 40         Training Loss: 32.967941         Validation Loss: 28.488159. Epoch lengt
Saving improved model: Validation loss decreased to 28.4881591796875 from 31.101675033569336. Re
Epoch: 27 of 40         Training Loss: 30.062334         Validation Loss: 26.917006. Epoch lengt
Saving improved model: Validation loss decreased to 26.91700553894043 from 28.4881591796875. Red
Epoch: 28 of 40         Training Loss: 29.597939         Validation Loss: 28.128750. Epoch lengt
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 29 of 40         Training Loss: 29.826130         Validation Loss: 24.029089. Epoch lengt
Saving improved model: Validation loss decreased to 24.029088973999023 from 26.91700553894043. R
Epoch: 30 of 40         Training Loss: 28.936689         Validation Loss: 26.919746. Epoch lengt
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 31 of 40         Training Loss: 29.193590         Validation Loss: 29.073254. Epoch lengt
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 32 of 40         Training Loss: 27.272236         Validation Loss: 27.716196. Epoch lengt
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 33 of 40         Training Loss: 28.200151         Validation Loss: 24.848421. Epoch lengt
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 34 of 40         Training Loss: 27.256527         Validation Loss: 26.744051. Epoch lengt
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 35 of 40         Training Loss: 27.600412         Validation Loss: 26.085072. Epoch lengt
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 36 of 40         Training Loss: 27.383461         Validation Loss: 25.838818. Epoch lengt
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 37 of 40         Training Loss: 26.965219         Validation Loss: 24.973656. Epoch lengt
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 38 of 40         Training Loss: 27.590139         Validation Loss: 28.185150. Epoch lengt
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 39 of 40         Training Loss: 27.220772         Validation Loss: 24.239990. Epoch lengt
Validation Loss not impoved this epoch. Model update not saved.
Epoch: 40 of 40         Training Loss: 27.045982         Validation Loss: 25.941187. Epoch lengt
Validation Loss not impoved this epoch. Model update not saved.
```

1.1.17 (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%.

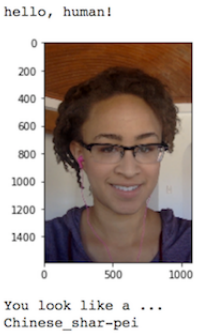
In [18]:
```
test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 29.636856


Test Accuracy: 74% (621/836)
```

1.1.18 (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 [19]:
```
### 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]
data_transfer = loaders_transfer.copy()
class_names = [item[4:].replace("_", " ") for item in data_transfer['train'].dataset.cl

def predict_breed_transfer(img_path, model_transfer, class_names):
    # load the image and return the predicted breed
    img = Image.open(img_path)

    # Use validation image transformations
    transform_pipeline = transforms.Compose([transforms.RandomResizedCrop(224),
                                             transforms.ToTensor(),
                                             transforms.Normalize(mean=[0.485, 0.456, 0.406
                                            ])

    # Pass image through pipeline
    img = transform_pipeline(img).unsqueeze(0)
    # ref: https://discuss.pytorch.org/t/preprocess-images-on-gpu/5096
    if use_cuda: img = img.cuda()

    img = model_transfer(img)

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

    index = torch.argmax(img)
    breed_name = class_names[index]

    return  breed_name
```



Sample Human Output

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!

1.1.19 (IMPLEMENTATION) Write your Algorithm

```
In [20]: ### TODO: Write your algorithm.
         ### Feel free to use as many code cells as needed.

         def run_app(img_path, model_transfer, class_names):
             ## handle cases for a human face, dog, and neither
             human_found = face_detector(img_path)
             dog_found = dog_detector(img_path)

             if human_found > 0:
                 print('Hello Human! Your doggy-doppelganger is a: ' + predict_breed_transfer(im
             if dog_found > 0:
                 print('Good Doggy! You look like a: ' + predict_breed_transfer(img_path, model_
             if (human_found == 0) & (dog_found == 0):
                 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?

1.1.20 (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.

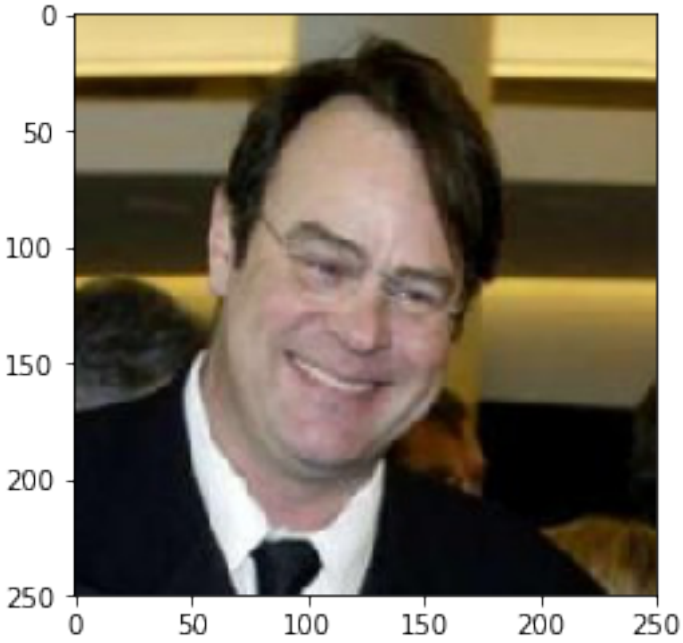
Answer: (Three possible points for improvement)

My model seemed to work better than I expected.

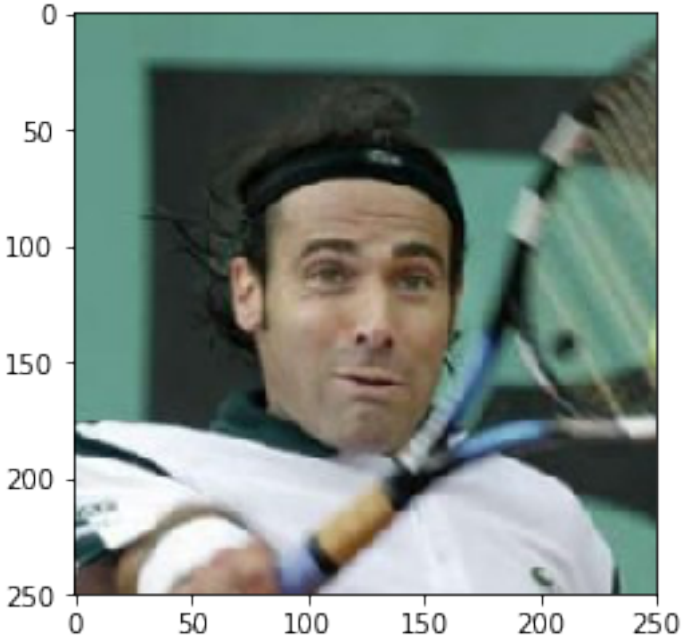
I believe I could improve the model generally with the following: 1. More training and validation samples of dogs (i.e. more data), or perhaps use data augmentation techniques to get even more training samples 2. Train the model longer on more epochs - perhaps using more expensive GPU instances to speed it up 3. Perhaps I may be able to unfreeze some of the latter covnets in order to find more features specific to this classification issue 4. Explore other more robust models included here: https://pytorch.org/docs/stable/torchvision/models.html

```
In [21]: ## 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])):

             print('Running dog breed app for file: {}'.format(file))
             run_app(file, model_transfer, class_names)

             plt.imshow(Image.open(file))
             plt.show()
```

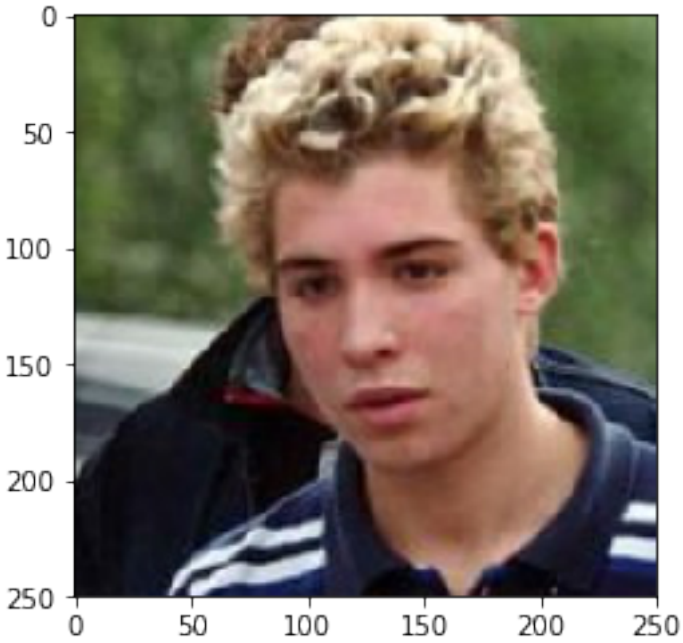
```
Running dog breed app for file: /data/lfw/Dan_Ackroyd/Dan_Ackroyd_0001.jpg
Hello Human! Your doggy-doppelganger is a: Dachshund
```



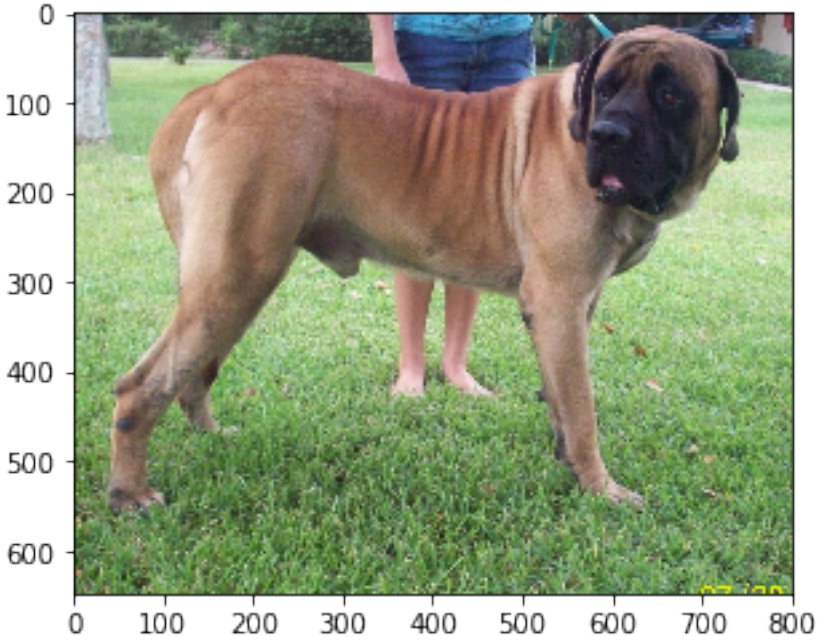

```
Running dog breed app for file: /data/lfw/Alex_Corretja/Alex_Corretja_0001.jpg
Hello Human! Your doggy-doppelganger is a: Dachshund
```



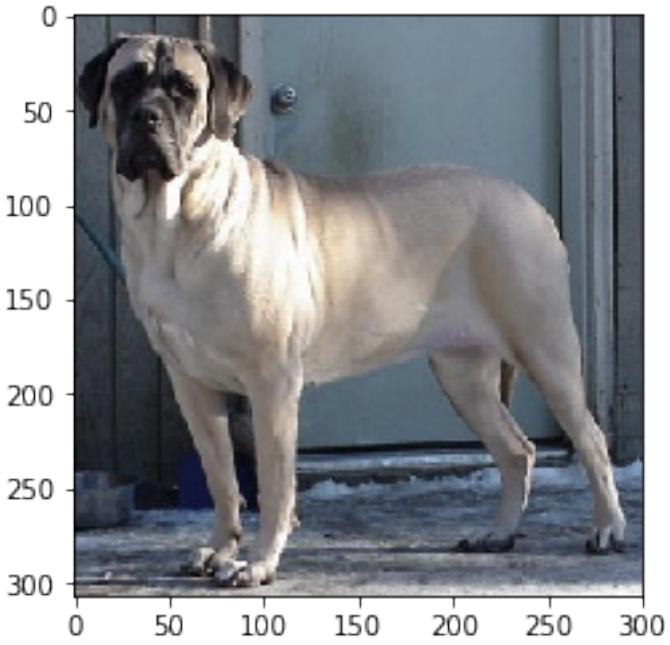
```
Running dog breed app for file: /data/lfw/Daniele_Bergamin/Daniele_Bergamin_0001.jpg
Hello Human! Your doggy-doppelganger is a: Silky terrier
```



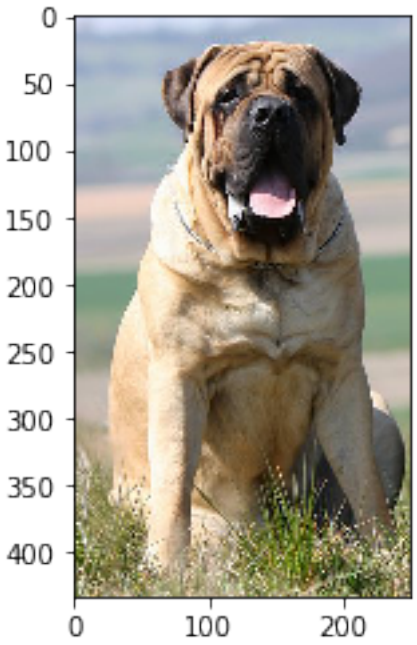

```
Running dog breed app for file: /data/dog_images/train/103.Mastiff/Mastiff_06833.jpg
Good Doggy! You look like a: Mastiff
```



```
Running dog breed app for file: /data/dog_images/train/103.Mastiff/Mastiff_06826.jpg
Good Doggy! You look like a: Mastiff
```




```
Running dog breed app for file: /data/dog_images/train/103.Mastiff/Mastiff_06871.jpg
Error: No Dog or Human found in picture. Is there either in picture?
```


0
50
100
150
200
250
300
350
400
0
100
200