dog_app1

October 27, 2019

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 `/dog_images`.

- Download the human dataset. Unzip the folder and place it in the home directory, at location `/lfw`.

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

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

```
In [31]: 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

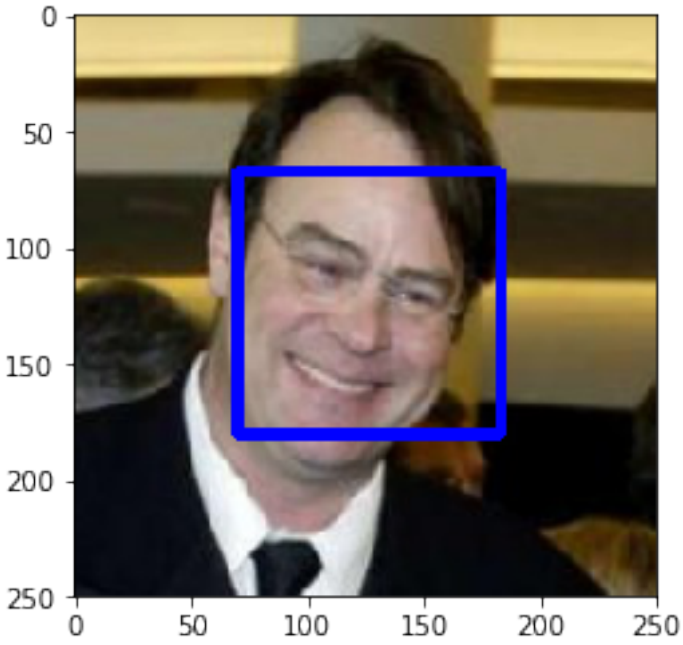
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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 [3]: 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),(255,1,1),4)

        # 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 [4]: # returns "True" if face is detected in image stored at img_path
        def human_face_detector(img_path):
            img = cv2.imread(img_path)
```

```
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    faces = face_cascade.detectMultiScale(gray)
    return len(faces) > 0
```

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

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

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

Answer: (You can print out your results and/or write your percentages in this cell) Printed the results below. Faces detected in 98.00% of the sample human dataset. Faces detected in 17.00% of the sample dog dataset.

In [6]:
```
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.

faces = np.vectorize(human_face_detector)
human_faces = faces(human_files_short)
dog_faces = faces(dog_files_short)

print('The faces detected is {:.2f}% from the sample human dataset.'.format((sum(human_f
print('The faces detected is {:.2f}% from the sample dog dataset.'.format((sum(dog_faces
```

```
The faces detected is 98.00% from the sample human dataset.
The faces detected is 17.00% from the sample dog dataset.
```

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 [7]:
```
### (Optional)
### TODO: Test performance of anotherface detection algorithm.
### Feel free to use as many code cells as needed.
```

Step 2: Detect Dogs

In this section, we use a pre-trained model to detect dogs in images.

1.1.3 Obtain Pre-trained VGG-16 Model

The code cell below downloads the VGG-16 model, along with weights that have been trained on ImageNet, a very large, very popular dataset used for image classification and other vision tasks. ImageNet contains over 10 million URLs, each linking to an image containing an object from one of 1000 categories.

```
In [8]: 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 [9]: from PIL import Image
        import torchvision.transforms as transforms

        def image_loader_path(img_path):
            image = Image.open(img_path).convert('RGB')
            in_transform = transforms.Compose([
                                transforms.Resize(size=(244, 244)),
                                transforms.ToTensor()])
            image = in_transform(image)[:3,:,:].unsqueeze(0)
            return image

        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

    img = image_loader_path(img_path)
    if use_cuda:
        img = img.cuda()
    ret = VGG16(img)
    return torch.max(ret,1)[1].item() # 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 [10]: def dog_detector(img_path):
             prediction = VGG16_predict(img_path)

             #print(prediction)
             return ((prediction <= 268) & (prediction >= 151))
```

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: 0% dog images in human files 99% dog files in dog files

```
In [12]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.

         def dog_detector_test(files):
             detection_cnt = 0;
             total_cnt = len(files)
             for file in files:
```

```
            detection_cnt += dog_detector(file)
        return detection_cnt, total_cnt

    print("Detect a dog in human_files: {} / {}".format(dog_detector_test(human_files_short
    print("Detect a dog in dog_files: {} / {}".format(dog_detector_test(dog_files_short)[0]
```

```
Detect a dog in human_files: 0 / 100
Detect a dog in dog_files: 99 / 100
```

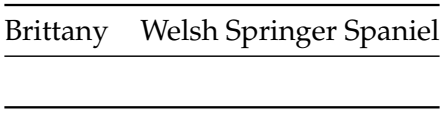
We suggest VGG-16 as a potential network to detect dog images in your algorithm, but you are free to explore other pre-trained networks (such as Inception-v3, ResNet-50, etc). Please use the code cell below to test other pre-trained PyTorch models. If you decide to pursue this *optional* task, report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

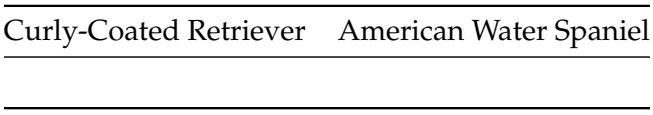
Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN that classifies dog breeds. You must create your CNN *from scratch* (so, you can't use transfer learning *yet*!), and you must attain a test accuracy of 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.



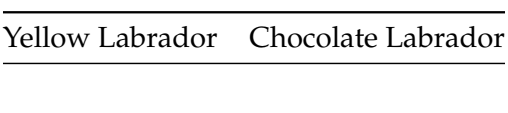
Brittany Welsh Springer Spaniel

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!

1.1.7 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

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

```
In [14]: import os
         from torchvision import datasets
         import torchvision.transforms as transforms
         import torch
         import numpy as np
         from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True

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

         batch_size = 10
         num_workers = 0

         data_directory = '/data/dog_images/'
         train_directory = os.path.join(data_directory, 'train/')
         valid_directory = os.path.join(data_directory, 'valid/')
         test_directory = os.path.join(data_directory, 'test/')

In [16]: standard_normalization = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                        std=[0.229, 0.224, 0.225])
         data_transforms = {'train': transforms.Compose([transforms.RandomResizedCrop(224),
                                                         transforms.RandomHorizontalFlip(),
                                                         transforms.ToTensor(),
                                                         standard_normalization]),
                            'val': transforms.Compose([transforms.Resize(256),
                                                       transforms.CenterCrop(224),
                                                       transforms.ToTensor(),
                                                       standard_normalization]),
                            'test': transforms.Compose([transforms.Resize(size=(224,224)),
                                                        transforms.ToTensor(),
                                                        standard_normalization])
                           }
         train_data = datasets.ImageFolder(train_directory, transform=data_transforms['train'])
```

```
valid_data = datasets.ImageFolder(valid_directory, transform=data_transforms['val'])
test_data = datasets.ImageFolder(test_directory, transform=data_transforms['test'])

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

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:

Applied torch transforms as followed Croped the Image at the center using ; transforms.CenterCrop(224) Resized the images usimg transforms;Resize(256) Transformed tensor image ;transforms.ToTensor() Normalized the input images to be of standard size using mean and std; standard_normalization = transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [17]: import torch.nn as nn
         import torch.nn.functional as F
         import numpy as np
         ImageFile.LOAD_TRUNCATED_IMAGES = True

         # define the CNN architecture
         num_classes = 133
         class Net(nn.Module):

             ### TODO: choose an architecture, and complete the class
             def __init__(self):
                 super(Net, self).__init__()
```

```
                ## Define layers of a CNN
                self.conv1 = nn.Conv2d(3, 32, 3, stride=2, padding=1)
                self.conv2 = nn.Conv2d(32, 64, 3, stride=2, padding=1)
                self.conv3 = nn.Conv2d(64, 128, 3, padding=1)
                self.pool = nn.MaxPool2d(2, 2)
                self.fc1 = nn.Linear(7*7*128, 500)
                self.fc2 = nn.Linear(500, num_classes)
                self.dropout = nn.Dropout(0.3)

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

                # flatten
                x = x.view(-1, 7*7*128)

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

                x = self.dropout(x)
                x = self.fc2(x)
                return x

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

        # instantiate the CNN
        model_scratch = Net()
        print(model_scratch)
        # move tensors to GPU if CUDA is available
        if use_cuda:
            model_scratch.cuda()
```

```
Net(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=6272, out_features=500, bias=True)
  (fc2): Linear(in_features=500, out_features=133, bias=True)
  (dropout): Dropout(p=0.3)
)
```

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

Answer: normalized imput images using Defined convolutional layer using nn.conv2d in init function as shown below.The convolutional layer was to perform feature extraction. It learns to find spatial features in input images. Then applied filters (convolutional kernels) Define the feed forward function of the model and flattended the images Used ReLU activation function; this function simply turns negative pixel values to 0 Used maxpooling layer for dimensionality reduction (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)

Result

```
Net(
```

(conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1)) (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1)) (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False) (fc1): Linear(in_features=6272, out_features=500, bias=True) (fc2): Linear(in_features=500, out_features=133, bias=True) (dropout): Dropout(p=0.3))

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 [18]:
```
import torch.optim as optimization

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

### TODO: select optimizer
optimizer_scratch = optimization.SGD(model_scratch.parameters(), lr = 0.05)
```

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 [19]:
```
def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path, last_val
    """returns trained model"""
    # initialize tracker for minimum validation loss
    if last_validation_loss is not None:
        valid_loss_min = last_validation_loss
    else:
        valid_loss_min = np.Inf

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

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

    # initialize weights to zero
    optimizer.zero_grad()

    output = model(data)

    # calculate loss
    loss = criterion(output, target)

    # back prop
    loss.backward()

    # grad
    optimizer.step()

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

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


# validate the model

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


# print training/validation statistics
print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
```

```
                    epoch,
                    train_loss,
                    valid_loss
                    ))

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

            # return trained model
            return model


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

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

```
Epoch 1, Batch 1 loss: 4.895126
Epoch 1, Batch 101 loss: 4.887576
Epoch 1, Batch 201 loss: 4.881241
Epoch 1, Batch 301 loss: 4.880157
Epoch 1, Batch 401 loss: 4.874125
Epoch 1, Batch 501 loss: 4.865132
Epoch 1, Batch 601 loss: 4.852956
Epoch: 1         Training Loss: 4.841608         Validation Loss: 4.783294
Validation loss decreased (inf --> 4.783294).  Saving model ...
Epoch 2, Batch 1 loss: 4.824551
Epoch 2, Batch 101 loss: 4.744110
Epoch 2, Batch 201 loss: 4.724648
Epoch 2, Batch 301 loss: 4.709957
Epoch 2, Batch 401 loss: 4.699346
Epoch 2, Batch 501 loss: 4.685124
Epoch 2, Batch 601 loss: 4.677392
Epoch: 2         Training Loss: 4.674028         Validation Loss: 4.532271
Validation loss decreased (4.783294 --> 4.532271).  Saving model ...
Epoch 3, Batch 1 loss: 4.575162
Epoch 3, Batch 101 loss: 4.579615
Epoch 3, Batch 201 loss: 4.561616
Epoch 3, Batch 301 loss: 4.563613
Epoch 3, Batch 401 loss: 4.568072
Epoch 3, Batch 501 loss: 4.566312
```

```
Epoch 3, Batch 601 loss: 4.572583
Epoch: 3         Training Loss: 4.572161         Validation Loss: 4.419909
Validation loss decreased (4.532271 --> 4.419909).  Saving model ...
Epoch 4, Batch 1 loss: 4.338643
Epoch 4, Batch 101 loss: 4.547853
Epoch 4, Batch 201 loss: 4.521580
Epoch 4, Batch 301 loss: 4.526608
Epoch 4, Batch 401 loss: 4.517502
Epoch 4, Batch 501 loss: 4.517090
Epoch 4, Batch 601 loss: 4.514818
Epoch: 4         Training Loss: 4.516067         Validation Loss: 4.296553
Validation loss decreased (4.419909 --> 4.296553).  Saving model ...
Epoch 5, Batch 1 loss: 4.348146
Epoch 5, Batch 101 loss: 4.420317
Epoch 5, Batch 201 loss: 4.426586
Epoch 5, Batch 301 loss: 4.415782
Epoch 5, Batch 401 loss: 4.411962
Epoch 5, Batch 501 loss: 4.407943
Epoch 5, Batch 601 loss: 4.409576
Epoch: 5         Training Loss: 4.412901         Validation Loss: 4.166154
Validation loss decreased (4.296553 --> 4.166154).  Saving model ...
Epoch 6, Batch 1 loss: 3.907252
Epoch 6, Batch 101 loss: 4.379731
Epoch 6, Batch 201 loss: 4.375529
Epoch 6, Batch 301 loss: 4.355063
Epoch 6, Batch 401 loss: 4.349281
Epoch 6, Batch 501 loss: 4.348279
Epoch 6, Batch 601 loss: 4.337055
Epoch: 6         Training Loss: 4.339267         Validation Loss: 4.056233
Validation loss decreased (4.166154 --> 4.056233).  Saving model ...
Epoch 7, Batch 1 loss: 4.623010
Epoch 7, Batch 101 loss: 4.258986
Epoch 7, Batch 201 loss: 4.261263
Epoch 7, Batch 301 loss: 4.264733
Epoch 7, Batch 401 loss: 4.271855
Epoch 7, Batch 501 loss: 4.270550
Epoch 7, Batch 601 loss: 4.268413
Epoch: 7         Training Loss: 4.265335         Validation Loss: 4.051161
Validation loss decreased (4.056233 --> 4.051161).  Saving model ...
Epoch 8, Batch 1 loss: 3.855294
Epoch 8, Batch 101 loss: 4.211717
Epoch 8, Batch 201 loss: 4.194986
Epoch 8, Batch 301 loss: 4.199516
Epoch 8, Batch 401 loss: 4.199002
Epoch 8, Batch 501 loss: 4.199041
Epoch 8, Batch 601 loss: 4.193763
Epoch: 8         Training Loss: 4.191248         Validation Loss: 4.029790
Validation loss decreased (4.051161 --> 4.029790).  Saving model ...
```

```
Epoch 9, Batch 1 loss: 4.359785
Epoch 9, Batch 101 loss: 4.134468
Epoch 9, Batch 201 loss: 4.131169
Epoch 9, Batch 301 loss: 4.127033
Epoch 9, Batch 401 loss: 4.138646
Epoch 9, Batch 501 loss: 4.141179
Epoch 9, Batch 601 loss: 4.140949
Epoch: 9         Training Loss: 4.136400         Validation Loss: 3.997084
Validation loss decreased (4.029790 --> 3.997084).  Saving model ...
Epoch 10, Batch 1 loss: 4.050555
Epoch 10, Batch 101 loss: 4.086525
Epoch 10, Batch 201 loss: 4.106220
Epoch 10, Batch 301 loss: 4.093245
Epoch 10, Batch 401 loss: 4.103481
Epoch 10, Batch 501 loss: 4.098190
Epoch 10, Batch 601 loss: 4.095615
Epoch: 10         Training Loss: 4.092600         Validation Loss: 3.878079
Validation loss decreased (3.997084 --> 3.878079).  Saving model ...
Epoch 11, Batch 1 loss: 4.444954
Epoch 11, Batch 101 loss: 4.050182
Epoch 11, Batch 201 loss: 4.030354
Epoch 11, Batch 301 loss: 4.030535
Epoch 11, Batch 401 loss: 4.033630
Epoch 11, Batch 501 loss: 4.030192
Epoch 11, Batch 601 loss: 4.036099
Epoch: 11         Training Loss: 4.040797         Validation Loss: 3.758311
Validation loss decreased (3.878079 --> 3.758311).  Saving model ...
Epoch 12, Batch 1 loss: 3.775643
Epoch 12, Batch 101 loss: 3.976257
Epoch 12, Batch 201 loss: 3.974221
Epoch 12, Batch 301 loss: 3.984025
Epoch 12, Batch 401 loss: 3.970132
Epoch 12, Batch 501 loss: 3.971983
Epoch 12, Batch 601 loss: 3.971455
Epoch: 12         Training Loss: 3.969862         Validation Loss: 3.774840
Epoch 13, Batch 1 loss: 4.446290
Epoch 13, Batch 101 loss: 3.965396
Epoch 13, Batch 201 loss: 3.933020
Epoch 13, Batch 301 loss: 3.922987
Epoch 13, Batch 401 loss: 3.910756
Epoch 13, Batch 501 loss: 3.920129
Epoch 13, Batch 601 loss: 3.930464
Epoch: 13         Training Loss: 3.932479         Validation Loss: 3.721658
Validation loss decreased (3.758311 --> 3.721658).  Saving model ...
Epoch 14, Batch 1 loss: 3.972277
Epoch 14, Batch 101 loss: 3.864096
Epoch 14, Batch 201 loss: 3.875204
Epoch 14, Batch 301 loss: 3.872365
```

```
Epoch 14, Batch 401 loss: 3.871518
Epoch 14, Batch 501 loss: 3.863610
Epoch 14, Batch 601 loss: 3.865977
Epoch: 14         Training Loss: 3.872727         Validation Loss: 3.666088
Validation loss decreased (3.721658 --> 3.666088).  Saving model ...
Epoch 15, Batch 1 loss: 3.683909
Epoch 15, Batch 101 loss: 3.814241
Epoch 15, Batch 201 loss: 3.792981
Epoch 15, Batch 301 loss: 3.845436
Epoch 15, Batch 401 loss: 3.846017
Epoch 15, Batch 501 loss: 3.846382
Epoch 15, Batch 601 loss: 3.843997
Epoch: 15         Training Loss: 3.838495         Validation Loss: 3.597543
Validation loss decreased (3.666088 --> 3.597543).  Saving model ...
Epoch 16, Batch 1 loss: 2.968808
Epoch 16, Batch 101 loss: 3.822895
Epoch 16, Batch 201 loss: 3.818515
Epoch 16, Batch 301 loss: 3.803381
Epoch 16, Batch 401 loss: 3.818937
Epoch 16, Batch 501 loss: 3.804001
Epoch 16, Batch 601 loss: 3.802612
Epoch: 16         Training Loss: 3.810800         Validation Loss: 3.627590
Epoch 17, Batch 1 loss: 2.730054
Epoch 17, Batch 101 loss: 3.801003
Epoch 17, Batch 201 loss: 3.798165
Epoch 17, Batch 301 loss: 3.797759
Epoch 17, Batch 401 loss: 3.799393
Epoch 17, Batch 501 loss: 3.802302
Epoch 17, Batch 601 loss: 3.797270
Epoch: 17         Training Loss: 3.788012         Validation Loss: 3.552660
Validation loss decreased (3.597543 --> 3.552660).  Saving model ...
Epoch 18, Batch 1 loss: 3.316749
Epoch 18, Batch 101 loss: 3.704961
Epoch 18, Batch 201 loss: 3.730834
Epoch 18, Batch 301 loss: 3.728914
Epoch 18, Batch 401 loss: 3.724653
Epoch 18, Batch 501 loss: 3.735294
Epoch 18, Batch 601 loss: 3.734078
Epoch: 18         Training Loss: 3.736353         Validation Loss: 3.713028
Epoch 19, Batch 1 loss: 3.944765
Epoch 19, Batch 101 loss: 3.693385
Epoch 19, Batch 201 loss: 3.643083
Epoch 19, Batch 301 loss: 3.673571
Epoch 19, Batch 401 loss: 3.699407
Epoch 19, Batch 501 loss: 3.717690
Epoch 19, Batch 601 loss: 3.722142
Epoch: 19         Training Loss: 3.724594         Validation Loss: 3.728964
Epoch 20, Batch 1 loss: 4.334942
```

```
Epoch 20, Batch 101 loss: 3.660181
Epoch 20, Batch 201 loss: 3.658033
Epoch 20, Batch 301 loss: 3.671723
Epoch 20, Batch 401 loss: 3.666684
Epoch 20, Batch 501 loss: 3.667315
Epoch 20, Batch 601 loss: 3.671790
Epoch: 20         Training Loss: 3.676121         Validation Loss: 3.598419
Epoch 21, Batch 1 loss: 4.178956
Epoch 21, Batch 101 loss: 3.620459
Epoch 21, Batch 201 loss: 3.622288
Epoch 21, Batch 301 loss: 3.647300
Epoch 21, Batch 401 loss: 3.666295
Epoch 21, Batch 501 loss: 3.648618
Epoch 21, Batch 601 loss: 3.655122
Epoch: 21         Training Loss: 3.647373         Validation Loss: 3.739750
Epoch 22, Batch 1 loss: 3.429747
Epoch 22, Batch 101 loss: 3.533105
Epoch 22, Batch 201 loss: 3.585959
Epoch 22, Batch 301 loss: 3.603197
Epoch 22, Batch 401 loss: 3.611592
Epoch 22, Batch 501 loss: 3.616976
Epoch 22, Batch 601 loss: 3.628656
Epoch: 22         Training Loss: 3.626004         Validation Loss: 3.512274
Validation loss decreased (3.552660 --> 3.512274).  Saving model ...
Epoch 23, Batch 1 loss: 2.822127
Epoch 23, Batch 101 loss: 3.551262
Epoch 23, Batch 201 loss: 3.613368
Epoch 23, Batch 301 loss: 3.619877
Epoch 23, Batch 401 loss: 3.616857
Epoch 23, Batch 501 loss: 3.587629
Epoch 23, Batch 601 loss: 3.612062
Epoch: 23         Training Loss: 3.613665         Validation Loss: 3.360026
Validation loss decreased (3.512274 --> 3.360026).  Saving model ...
Epoch 24, Batch 1 loss: 3.362460
Epoch 24, Batch 101 loss: 3.595397
Epoch 24, Batch 201 loss: 3.561870
Epoch 24, Batch 301 loss: 3.572133
Epoch 24, Batch 401 loss: 3.565087
Epoch 24, Batch 501 loss: 3.576833
Epoch 24, Batch 601 loss: 3.578206
Epoch: 24         Training Loss: 3.579989         Validation Loss: 3.471230
Epoch 25, Batch 1 loss: 3.230272
Epoch 25, Batch 101 loss: 3.527457
Epoch 25, Batch 201 loss: 3.535059
Epoch 25, Batch 301 loss: 3.529006
Epoch 25, Batch 401 loss: 3.550554
Epoch 25, Batch 501 loss: 3.549018
Epoch 25, Batch 601 loss: 3.558392
```

```
Epoch: 25         Training Loss: 3.561114         Validation Loss: 3.491200
Epoch 26, Batch 1 loss: 3.890659
Epoch 26, Batch 101 loss: 3.613238
Epoch 26, Batch 201 loss: 3.545858
Epoch 26, Batch 301 loss: 3.551778
Epoch 26, Batch 401 loss: 3.539084
Epoch 26, Batch 501 loss: 3.553264
Epoch 26, Batch 601 loss: 3.556667
Epoch: 26         Training Loss: 3.551104         Validation Loss: 3.428080
Epoch 27, Batch 1 loss: 4.045125
Epoch 27, Batch 101 loss: 3.427742
Epoch 27, Batch 201 loss: 3.478087
Epoch 27, Batch 301 loss: 3.486518
Epoch 27, Batch 401 loss: 3.508609
Epoch 27, Batch 501 loss: 3.501081
Epoch 27, Batch 601 loss: 3.509737
Epoch: 27         Training Loss: 3.514867         Validation Loss: 3.404427
Epoch 28, Batch 1 loss: 3.290401
Epoch 28, Batch 101 loss: 3.475600
Epoch 28, Batch 201 loss: 3.504753
Epoch 28, Batch 301 loss: 3.528502
Epoch 28, Batch 401 loss: 3.518493
Epoch 28, Batch 501 loss: 3.510220
Epoch 28, Batch 601 loss: 3.536776
Epoch: 28         Training Loss: 3.542828         Validation Loss: 3.356809
Validation loss decreased (3.360026 --> 3.356809).  Saving model ...
Epoch 29, Batch 1 loss: 2.416568
Epoch 29, Batch 101 loss: 3.462648
Epoch 29, Batch 201 loss: 3.551656
Epoch 29, Batch 301 loss: 3.560596
Epoch 29, Batch 401 loss: 3.551981
Epoch 29, Batch 501 loss: 3.547531
Epoch 29, Batch 601 loss: 3.539278
Epoch: 29         Training Loss: 3.523152         Validation Loss: 3.405803
Epoch 30, Batch 1 loss: 3.517871
Epoch 30, Batch 101 loss: 3.396412
Epoch 30, Batch 201 loss: 3.438354
Epoch 30, Batch 301 loss: 3.396850
Epoch 30, Batch 401 loss: 3.401706
Epoch 30, Batch 501 loss: 3.448124
Epoch 30, Batch 601 loss: 3.457913
Epoch: 30         Training Loss: 3.449814         Validation Loss: 3.392078
```

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 [20]: 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: 3.463461


Test Accuracy: 18% (158/836)
```

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 [22]: ## TODO: Specify data loaders
         loaders_transfer = loaders_scratch.copy()
```

1.1.13 (IMPLEMENTATION) Model Architecture

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

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

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

         for param in model_transfer.parameters():
             param.requires_grad = False
         model_transfer.fc = nn.Linear(2048, 133, bias=True)
         fc_parameters = model_transfer.fc.parameters()
         for param in fc_parameters:
             param.requires_grad = True
         print (model_transfer)
         if use_cuda:
             model_transfer = model_transfer.cuda()
```

```
Downloading: "https://download.pytorch.org/models/resnet50-19c8e357.pth" to /root/.torch/models/
100%|| 102502400/102502400 [00:08<00:00, 11874143.59it/s]


ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (downsample): Sequential(
        (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
```

```
  )
  (1): Bottleneck(
    (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
  (2): Bottleneck(
    (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
)
(layer2): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
    (downsample): Sequential(
      (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
  (2): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

```
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
  (3): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
)
(layer3): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
    (downsample): Sequential(
      (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
  (2): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
  (3): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
```

```
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
  (4): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
  (5): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
)
(layer4): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
    (downsample): Sequential(
      (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): Bottleneck(
    (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
```

```
    )
    (2): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
  )
  (avgpool): AvgPool2d(kernel_size=7, stride=1, padding=0)
  (fc): Linear(in_features=2048, out_features=133, bias=True)
)
```

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

Answer: Used ResNet model for classification as i thought it would be great in image classification ,which is a problem like this one I am solving.I thought it would p@erform great with this dataset.

Specified the model with random weights as shown ResNet((conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)

Provided pre-trained models, using the PyTorch torch.utils.model and used ReLu Activation function

(relu): ReLU(inplace)

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 [24]: criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = optimization.SGD(model_transfer.fc.parameters(), lr=0.001)
```

1.1.15 (IMPLEMENTATION) Train and Validate the Model

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

```
In [25]: # train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer, criterion_trans
         def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
             """returns trained model"""
             # initialize tracker for minimum validation loss
             valid_loss_min = np.Inf

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

```
valid_loss = 0.0


# train the model
model.train()
for batch_idx, (data, target) in enumerate(loaders['train']):
    # move to GPU
    if use_cuda:
        data, target = data.cuda(), target.cuda()
    # initialize weights to zero
    optimizer.zero_grad()

    output = model(data)

    # calculate loss
    loss = criterion(output, target)

    # back prop
    loss.backward()

    # grad
    optimizer.step()

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

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


# validate the model

model.eval()
for batch_idx, (data, target) in enumerate(loaders['valid']):
    # move to GPU
    if use_cuda:
        data, target = data.cuda(), target.cuda()
    ## update the average validation loss
    output = model(data)
    loss = criterion(output, target)
    valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data - valid_loss)
# print training/validation statistics
print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
    epoch,
    train_loss,
    valid_loss
    ))
```

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

            # return trained model
            return model
        train(30, loaders_transfer, model_transfer, optimizer_transfer, criterion_transfer, use
```

```
Epoch 1, Batch 1 loss: 5.049479
Epoch 1, Batch 101 loss: 4.928988
Epoch 1, Batch 201 loss: 4.893566
Epoch 1, Batch 301 loss: 4.862844
Epoch 1, Batch 401 loss: 4.832908
Epoch 1, Batch 501 loss: 4.808810
Epoch 1, Batch 601 loss: 4.786127
Epoch: 1         Training Loss: 4.773101         Validation Loss: 4.468023
Validation loss decreased (inf --> 4.468023).  Saving model ...
Epoch 2, Batch 1 loss: 4.688281
Epoch 2, Batch 101 loss: 4.577074
Epoch 2, Batch 201 loss: 4.548672
Epoch 2, Batch 301 loss: 4.523154
Epoch 2, Batch 401 loss: 4.504277
Epoch 2, Batch 501 loss: 4.488003
Epoch 2, Batch 601 loss: 4.468063
Epoch: 2         Training Loss: 4.456735         Validation Loss: 4.060163
Validation loss decreased (4.468023 --> 4.060163).  Saving model ...
Epoch 3, Batch 1 loss: 4.072095
Epoch 3, Batch 101 loss: 4.294301
Epoch 3, Batch 201 loss: 4.273100
Epoch 3, Batch 301 loss: 4.245363
Epoch 3, Batch 401 loss: 4.223639
Epoch 3, Batch 501 loss: 4.196660
Epoch 3, Batch 601 loss: 4.179577
Epoch: 3         Training Loss: 4.165814         Validation Loss: 3.686569
Validation loss decreased (4.060163 --> 3.686569).  Saving model ...
Epoch 4, Batch 1 loss: 4.212717
Epoch 4, Batch 101 loss: 3.960767
Epoch 4, Batch 201 loss: 3.979258
Epoch 4, Batch 301 loss: 3.959760
Epoch 4, Batch 401 loss: 3.943068
Epoch 4, Batch 501 loss: 3.924000
Epoch 4, Batch 601 loss: 3.909581
Epoch: 4         Training Loss: 3.902376         Validation Loss: 3.343493
Validation loss decreased (3.686569 --> 3.343493).  Saving model ...
```

```
Epoch 5, Batch 1 loss: 3.779042
Epoch 5, Batch 101 loss: 3.723246
Epoch 5, Batch 201 loss: 3.717272
Epoch 5, Batch 301 loss: 3.717098
Epoch 5, Batch 401 loss: 3.693968
Epoch 5, Batch 501 loss: 3.685405
Epoch 5, Batch 601 loss: 3.677633
Epoch: 5         Training Loss: 3.664043         Validation Loss: 2.994759
Validation loss decreased (3.343493 --> 2.994759).  Saving model ...
Epoch 6, Batch 1 loss: 3.632957
Epoch 6, Batch 101 loss: 3.514032
Epoch 6, Batch 201 loss: 3.510237
Epoch 6, Batch 301 loss: 3.485984
Epoch 6, Batch 401 loss: 3.468683
Epoch 6, Batch 501 loss: 3.456229
Epoch 6, Batch 601 loss: 3.445699
Epoch: 6         Training Loss: 3.440652         Validation Loss: 2.778383
Validation loss decreased (2.994759 --> 2.778383).  Saving model ...
Epoch 7, Batch 1 loss: 3.700066
Epoch 7, Batch 101 loss: 3.358840
Epoch 7, Batch 201 loss: 3.328170
Epoch 7, Batch 301 loss: 3.304173
Epoch 7, Batch 401 loss: 3.286624
Epoch 7, Batch 501 loss: 3.275224
Epoch 7, Batch 601 loss: 3.254165
Epoch: 7         Training Loss: 3.246510         Validation Loss: 2.529352
Validation loss decreased (2.778383 --> 2.529352).  Saving model ...
Epoch 8, Batch 1 loss: 3.337461
Epoch 8, Batch 101 loss: 3.153722
Epoch 8, Batch 201 loss: 3.131077
Epoch 8, Batch 301 loss: 3.115461
Epoch 8, Batch 401 loss: 3.101219
Epoch 8, Batch 501 loss: 3.092914
Epoch 8, Batch 601 loss: 3.082258
Epoch: 8         Training Loss: 3.071780         Validation Loss: 2.309236
Validation loss decreased (2.529352 --> 2.309236).  Saving model ...
Epoch 9, Batch 1 loss: 2.939083
Epoch 9, Batch 101 loss: 2.982409
Epoch 9, Batch 201 loss: 2.957384
Epoch 9, Batch 301 loss: 2.943712
Epoch 9, Batch 401 loss: 2.928313
Epoch 9, Batch 501 loss: 2.923086
Epoch 9, Batch 601 loss: 2.912857
Epoch: 9         Training Loss: 2.902574         Validation Loss: 2.109102
Validation loss decreased (2.309236 --> 2.109102).  Saving model ...
Epoch 10, Batch 1 loss: 2.486369
Epoch 10, Batch 101 loss: 2.760569
Epoch 10, Batch 201 loss: 2.755246
```

```
Epoch 10, Batch 301 loss: 2.755124
Epoch 10, Batch 401 loss: 2.770142
Epoch 10, Batch 501 loss: 2.754657
Epoch 10, Batch 601 loss: 2.748756
Epoch: 10         Training Loss: 2.742423         Validation Loss: 1.955587
Validation loss decreased (2.109102 --> 1.955587).  Saving model ...
Epoch 11, Batch 1 loss: 2.417129
Epoch 11, Batch 101 loss: 2.628525
Epoch 11, Batch 201 loss: 2.639730
Epoch 11, Batch 301 loss: 2.659477
Epoch 11, Batch 401 loss: 2.656804
Epoch 11, Batch 501 loss: 2.662787
Epoch 11, Batch 601 loss: 2.656043
Epoch: 11         Training Loss: 2.649472         Validation Loss: 1.832452
Validation loss decreased (1.955587 --> 1.832452).  Saving model ...
Epoch 12, Batch 1 loss: 2.653443
Epoch 12, Batch 101 loss: 2.565390
Epoch 12, Batch 201 loss: 2.567378
Epoch 12, Batch 301 loss: 2.550606
Epoch 12, Batch 401 loss: 2.543113
Epoch 12, Batch 501 loss: 2.537696
Epoch 12, Batch 601 loss: 2.528792
Epoch: 12         Training Loss: 2.519427         Validation Loss: 1.717478
Validation loss decreased (1.832452 --> 1.717478).  Saving model ...
Epoch 13, Batch 1 loss: 2.480563
Epoch 13, Batch 101 loss: 2.424089
Epoch 13, Batch 201 loss: 2.421093
Epoch 13, Batch 301 loss: 2.433203
Epoch 13, Batch 401 loss: 2.429163
Epoch 13, Batch 501 loss: 2.428884
Epoch 13, Batch 601 loss: 2.413926
Epoch: 13         Training Loss: 2.408528         Validation Loss: 1.589405
Validation loss decreased (1.717478 --> 1.589405).  Saving model ...
Epoch 14, Batch 1 loss: 1.999356
Epoch 14, Batch 101 loss: 2.366043
Epoch 14, Batch 201 loss: 2.361404
Epoch 14, Batch 301 loss: 2.352121
Epoch 14, Batch 401 loss: 2.336597
Epoch 14, Batch 501 loss: 2.326105
Epoch 14, Batch 601 loss: 2.324795
Epoch: 14         Training Loss: 2.316738         Validation Loss: 1.458263
Validation loss decreased (1.589405 --> 1.458263).  Saving model ...
Epoch 15, Batch 1 loss: 1.882346
Epoch 15, Batch 101 loss: 2.257885
Epoch 15, Batch 201 loss: 2.254256
Epoch 15, Batch 301 loss: 2.265151
Epoch 15, Batch 401 loss: 2.266720
Epoch 15, Batch 501 loss: 2.259514
```

```
Epoch 15, Batch 601 loss: 2.257015
Epoch: 15         Training Loss: 2.251977         Validation Loss: 1.384440
Validation loss decreased (1.458263 --> 1.384440).  Saving model ...
Epoch 16, Batch 1 loss: 1.979009
Epoch 16, Batch 101 loss: 2.210666
Epoch 16, Batch 201 loss: 2.175543
Epoch 16, Batch 301 loss: 2.191303
Epoch 16, Batch 401 loss: 2.179758
Epoch 16, Batch 501 loss: 2.165377
Epoch 16, Batch 601 loss: 2.168794
Epoch: 16         Training Loss: 2.168741         Validation Loss: 1.332244
Validation loss decreased (1.384440 --> 1.332244).  Saving model ...
Epoch 17, Batch 1 loss: 2.643209
Epoch 17, Batch 101 loss: 2.198898
Epoch 17, Batch 201 loss: 2.154895
Epoch 17, Batch 301 loss: 2.142068
Epoch 17, Batch 401 loss: 2.134517
Epoch 17, Batch 501 loss: 2.120219
Epoch 17, Batch 601 loss: 2.110243
Epoch: 17         Training Loss: 2.112270         Validation Loss: 1.268570
Validation loss decreased (1.332244 --> 1.268570).  Saving model ...
Epoch 18, Batch 1 loss: 1.992575
Epoch 18, Batch 101 loss: 2.036540
Epoch 18, Batch 201 loss: 2.030206
Epoch 18, Batch 301 loss: 2.036948
Epoch 18, Batch 401 loss: 2.051291
Epoch 18, Batch 501 loss: 2.055663
Epoch 18, Batch 601 loss: 2.036342
Epoch: 18         Training Loss: 2.034009         Validation Loss: 1.209780
Validation loss decreased (1.268570 --> 1.209780).  Saving model ...
Epoch 19, Batch 1 loss: 1.713289
Epoch 19, Batch 101 loss: 2.010318
Epoch 19, Batch 201 loss: 2.006399
Epoch 19, Batch 301 loss: 2.005540
Epoch 19, Batch 401 loss: 1.998441
Epoch 19, Batch 501 loss: 1.993348
Epoch 19, Batch 601 loss: 1.992202
Epoch: 19         Training Loss: 1.987881         Validation Loss: 1.160963
Validation loss decreased (1.209780 --> 1.160963).  Saving model ...
Epoch 20, Batch 1 loss: 1.806684
Epoch 20, Batch 101 loss: 1.947480
Epoch 20, Batch 201 loss: 1.930054
Epoch 20, Batch 301 loss: 1.929094
Epoch 20, Batch 401 loss: 1.907441
Epoch 20, Batch 501 loss: 1.905106
Epoch 20, Batch 601 loss: 1.908444
Epoch: 20         Training Loss: 1.907725         Validation Loss: 1.112334
Validation loss decreased (1.160963 --> 1.112334).  Saving model ...
```

```
Epoch 21, Batch 1 loss: 1.593995
Epoch 21, Batch 101 loss: 1.932754
Epoch 21, Batch 201 loss: 1.898248
Epoch 21, Batch 301 loss: 1.891574
Epoch 21, Batch 401 loss: 1.880328
Epoch 21, Batch 501 loss: 1.868941
Epoch 21, Batch 601 loss: 1.864192
Epoch: 21         Training Loss: 1.866123         Validation Loss: 1.042084
Validation loss decreased (1.112334 --> 1.042084).  Saving model ...
Epoch 22, Batch 1 loss: 1.977712
Epoch 22, Batch 101 loss: 1.809538
Epoch 22, Batch 201 loss: 1.797034
Epoch 22, Batch 301 loss: 1.834239
Epoch 22, Batch 401 loss: 1.852501
Epoch 22, Batch 501 loss: 1.842766
Epoch 22, Batch 601 loss: 1.842169
Epoch: 22         Training Loss: 1.839188         Validation Loss: 1.029432
Validation loss decreased (1.042084 --> 1.029432).  Saving model ...
Epoch 23, Batch 1 loss: 2.631504
Epoch 23, Batch 101 loss: 1.785639
Epoch 23, Batch 201 loss: 1.781698
Epoch 23, Batch 301 loss: 1.778738
Epoch 23, Batch 401 loss: 1.781332
Epoch 23, Batch 501 loss: 1.792007
Epoch 23, Batch 601 loss: 1.793207
Epoch: 23         Training Loss: 1.791036         Validation Loss: 1.007512
Validation loss decreased (1.029432 --> 1.007512).  Saving model ...
Epoch 24, Batch 1 loss: 1.871730
Epoch 24, Batch 101 loss: 1.780238
Epoch 24, Batch 201 loss: 1.774065
Epoch 24, Batch 301 loss: 1.764668
Epoch 24, Batch 401 loss: 1.766666
Epoch 24, Batch 501 loss: 1.768214
Epoch 24, Batch 601 loss: 1.757302
Epoch: 24         Training Loss: 1.753909         Validation Loss: 0.940482
Validation loss decreased (1.007512 --> 0.940482).  Saving model ...
Epoch 25, Batch 1 loss: 2.068026
Epoch 25, Batch 101 loss: 1.684406
Epoch 25, Batch 201 loss: 1.700142
Epoch 25, Batch 301 loss: 1.719033
Epoch 25, Batch 401 loss: 1.710144
Epoch 25, Batch 501 loss: 1.718319
Epoch 25, Batch 601 loss: 1.715739
Epoch: 25         Training Loss: 1.712883         Validation Loss: 0.930425
Validation loss decreased (0.940482 --> 0.930425).  Saving model ...
Epoch 26, Batch 1 loss: 2.333452
Epoch 26, Batch 101 loss: 1.696344
Epoch 26, Batch 201 loss: 1.684949
```

```
Epoch 26, Batch 301 loss: 1.684924
Epoch 26, Batch 401 loss: 1.671818
Epoch 26, Batch 501 loss: 1.669049
Epoch 26, Batch 601 loss: 1.672634
Epoch: 26         Training Loss: 1.677956         Validation Loss: 0.909551
Validation loss decreased (0.930425 --> 0.909551).  Saving model ...
Epoch 27, Batch 1 loss: 1.695498
Epoch 27, Batch 101 loss: 1.659133
Epoch 27, Batch 201 loss: 1.671737
Epoch 27, Batch 301 loss: 1.662642
Epoch 27, Batch 401 loss: 1.662394
Epoch 27, Batch 501 loss: 1.649155
Epoch 27, Batch 601 loss: 1.640779
Epoch: 27         Training Loss: 1.643564         Validation Loss: 0.866863
Validation loss decreased (0.909551 --> 0.866863).  Saving model ...
Epoch 28, Batch 1 loss: 1.280798
Epoch 28, Batch 101 loss: 1.605443
Epoch 28, Batch 201 loss: 1.630353
Epoch 28, Batch 301 loss: 1.647823
Epoch 28, Batch 401 loss: 1.650718
Epoch 28, Batch 501 loss: 1.641141
Epoch 28, Batch 601 loss: 1.644341
Epoch: 28         Training Loss: 1.629603         Validation Loss: 0.817755
Validation loss decreased (0.866863 --> 0.817755).  Saving model ...
Epoch 29, Batch 1 loss: 1.399801
Epoch 29, Batch 101 loss: 1.570876
Epoch 29, Batch 201 loss: 1.601737
Epoch 29, Batch 301 loss: 1.607360
Epoch 29, Batch 401 loss: 1.600789
Epoch 29, Batch 501 loss: 1.612257
Epoch 29, Batch 601 loss: 1.601156
Epoch: 29         Training Loss: 1.600973         Validation Loss: 0.817622
Validation loss decreased (0.817755 --> 0.817622).  Saving model ...
Epoch 30, Batch 1 loss: 1.715226
Epoch 30, Batch 101 loss: 1.544761
Epoch 30, Batch 201 loss: 1.574159
Epoch 30, Batch 301 loss: 1.575769
Epoch 30, Batch 401 loss: 1.571048
Epoch 30, Batch 501 loss: 1.569348
Epoch 30, Batch 601 loss: 1.574208
Epoch: 30         Training Loss: 1.570685         Validation Loss: 0.820115
```

```
Out[25]: ResNet(
           (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
           (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
           (relu): ReLU(inplace)
           (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
```

```
(layer1): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
    (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (relu): ReLU(inplace)
    (downsample): Sequential(
      (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    )
  )
  (1): Bottleneck(
    (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
    (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (relu): ReLU(inplace)
  )
  (2): Bottleneck(
    (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
    (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (relu): ReLU(inplace)
  )
)
(layer2): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (relu): ReLU(inplace)
    (downsample): Sequential(
      (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    )
  )
  (1): Bottleneck(
```

```
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (relu): ReLU(inplace)
  )
  (2): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (relu): ReLU(inplace)
  )
  (3): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (relu): ReLU(inplace)
  )
)
(layer3): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
    (relu): ReLU(inplace)
    (downsample): Sequential(
      (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
    )
  )
  (1): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
```

```
    (relu): ReLU(inplace)
  )
  (2): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
    (relu): ReLU(inplace)
  )
  (3): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
    (relu): ReLU(inplace)
  )
  (4): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
    (relu): ReLU(inplace)
  )
  (5): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
    (relu): ReLU(inplace)
  )
)
(layer4): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stat
    (relu): ReLU(inplace)
```

```
      (downsample): Sequential(
        (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      (relu): ReLU(inplace)
    )
    (2): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      (relu): ReLU(inplace)
    )
  )
  (avgpool): AvgPool2d(kernel_size=7, stride=1, padding=0)
  (fc): Linear(in_features=2048, out_features=133, bias=True)
)
```

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

```
Test Loss: 0.893086


Test Accuracy: 79% (668/836)
```

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.

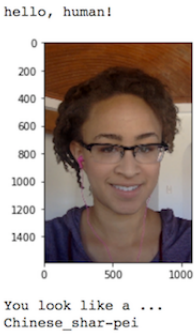
```
In [37]: ### TODO: Write a function that takes a path to an image as input
         ### and returns the dog breed that is predicted by the model.
```

```
# list of class names by index, i.e. a name can be accessed like class_names[0]
from PIL import Image
import torchvision.transforms as transforms

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

loaders_transfer['train'].dataset.classes[:10]
class_names[:10]
def loadinput_image(img_path):
    image = Image.open(img_path).convert('RGB')
    prediction_transform = transforms.Compose([transforms.Resize(size=(224, 224)),
                                    transforms.ToTensor(),
                                    standard_normalization])

    # discard the transparent, alpha channel (that's the :3) and add the batch dimensio
    image = prediction_transform(image)[:3,:,:].unsqueeze(0)
    return image

def predictbreed_transfer_learning(model, class_names, img_path):
    # load the image and return the predicted breed
    img = load_input_image(img_path)
    model = model.cpu()
    model.eval()
    idx = torch.argmax(model(img))
    return class_names[idx]

for img_file in os.listdir('/data/dog_images/test/004.Akita/'):
    img_path = os.path.join('/data/dog_images/test/004.Akita/', img_file)
    predition = predict_breed_transfer(model_transfer, class_names, img_path)
    print("image_file_name: {0}, \t predition breed: {1}".format(img_path, predition))
```

```
image_file_name: /data/dog_images/test/004.Akita/Akita_00296.jpg,          predition breed: Akit
image_file_name: /data/dog_images/test/004.Akita/Akita_00258.jpg,          predition breed: Akit
image_file_name: /data/dog_images/test/004.Akita/Akita_00263.jpg,          predition breed: Akit
image_file_name: /data/dog_images/test/004.Akita/Akita_00244.jpg,          predition breed: Alas
image_file_name: /data/dog_images/test/004.Akita/Akita_00276.jpg,          predition breed: Akit
image_file_name: /data/dog_images/test/004.Akita/Akita_00262.jpg,          predition breed: Akit
image_file_name: /data/dog_images/test/004.Akita/Akita_00270.jpg,          predition breed: Norw
image_file_name: /data/dog_images/test/004.Akita/Akita_00282.jpg,          predition breed: Akit
```

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



Sample Human Output

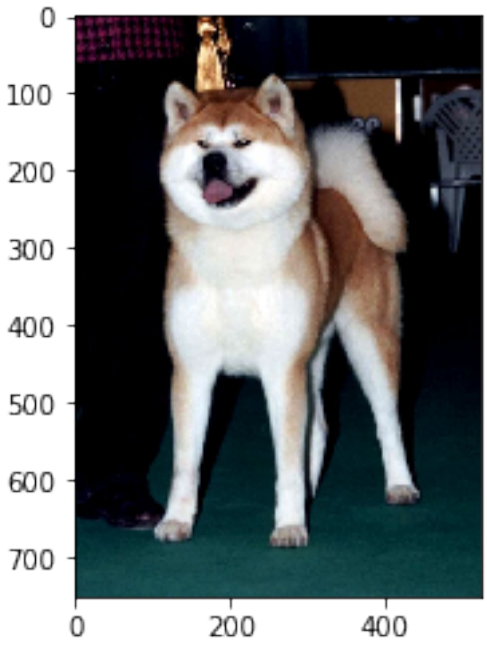
breed. - if a **human** is detected in the image, return the resembling dog breed. - if **neither** is detected in the image, provide output that indicates an error.

You are welcome to write your own functions for detecting humans and dogs in images, but feel free to use the `face_detector` and `human_detector` functions developed above. You are **required** to use your CNN from Step 4 to predict dog breed.

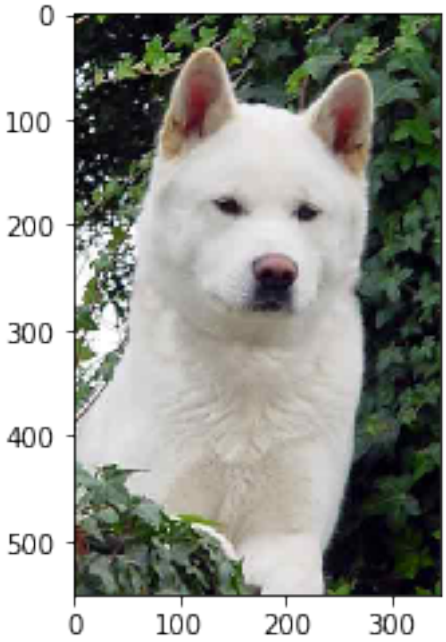
Some sample output for our algorithm is provided below, but feel free to design your own user experience!

1.1.18 (IMPLEMENTATION) Write your Algorithm

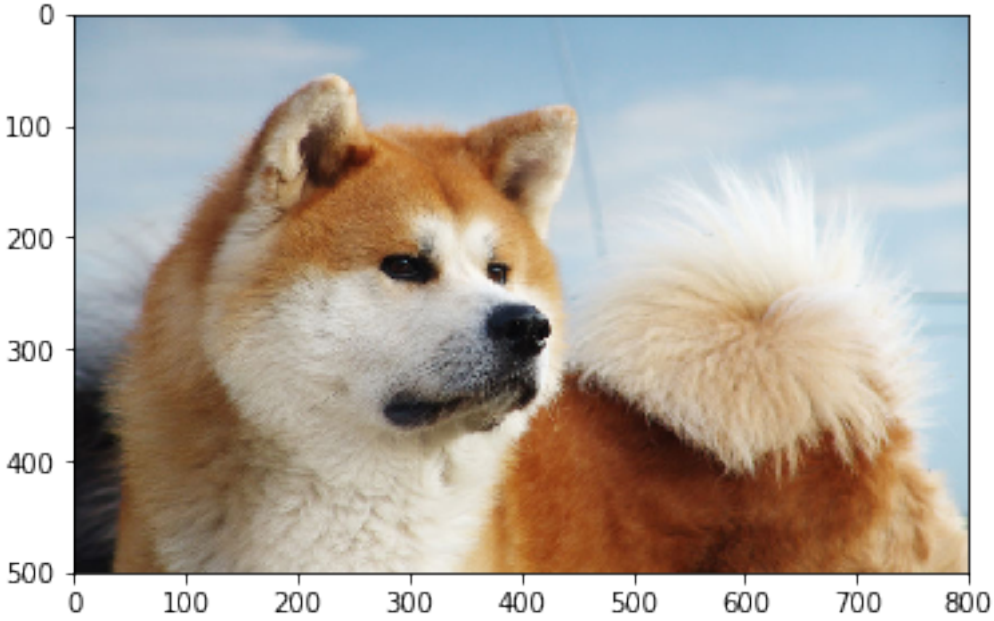
In [43]:
```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.

def my_alg(img_path):
    ## handle cases for a human face, dog, and neither
    img = Image.open(img_path)
    plt.imshow(img)
    plt.show()

    if dog_detector(img_path) is True:
        prediction = predict_breed_transfer(model_transfer, class_names, img_path)
        print("Dog Detected;\nIt is {0}".format(prediction))

    elif human_face_detector(img_path) > 0:
        prediction = predict_breed_transfer(model_transfer, class_names, img_path)
        print("Hello Human!\nYou look like a {0}".format(prediction))
    else:
        print("Error! Can't detect anything..")
for img_file in os.listdir('/data/dog_images/test/004.Akita/'):
    img_path = os.path.join('/data/dog_images/test/004.Akita/', img_file)
    my_alg(img_path)
```



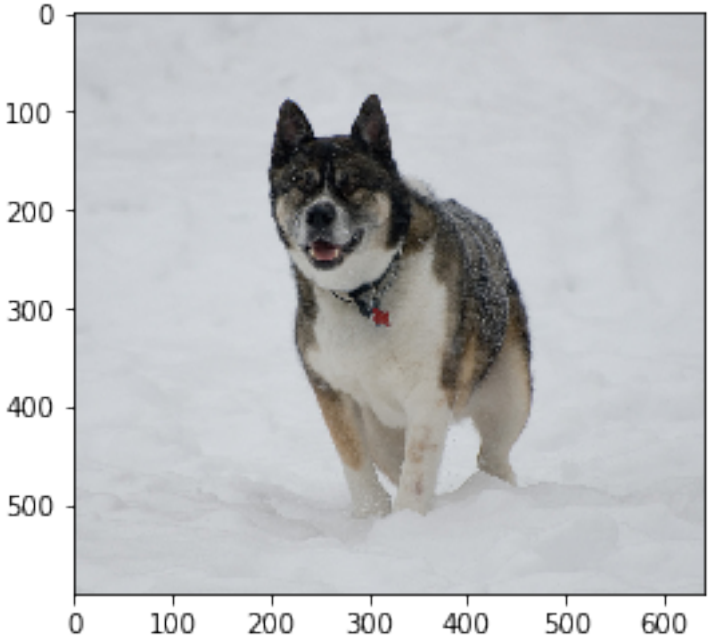

```
Dog Detected;
It is Akita
```



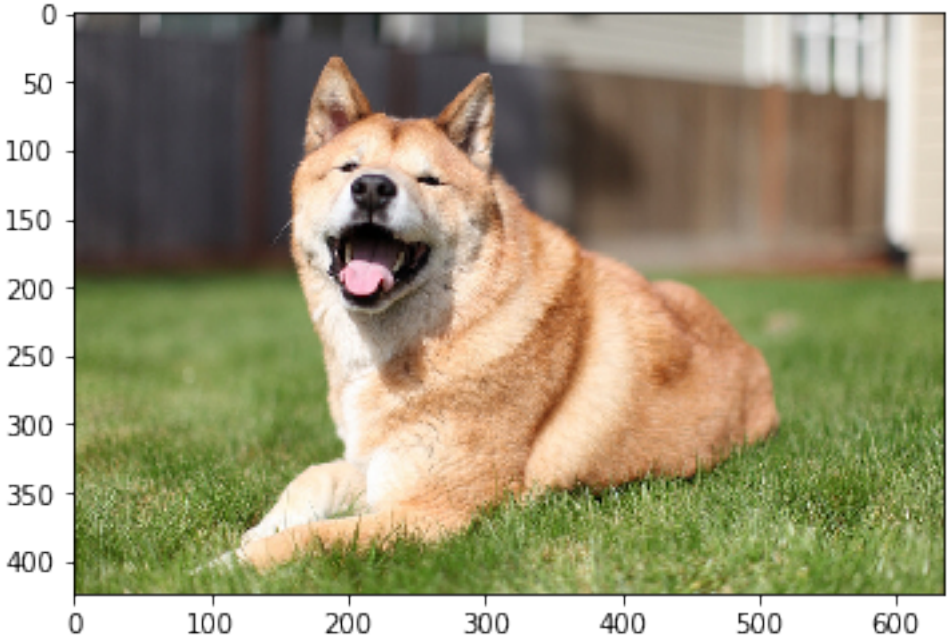
```
Dog Detected;
It is Akita
```



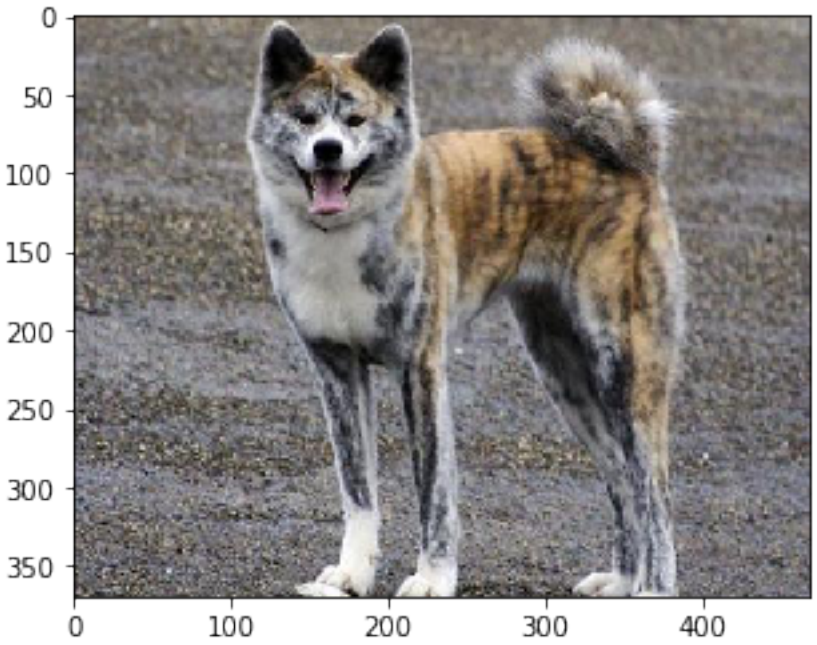

```
Dog Detected;
It is Akita
```



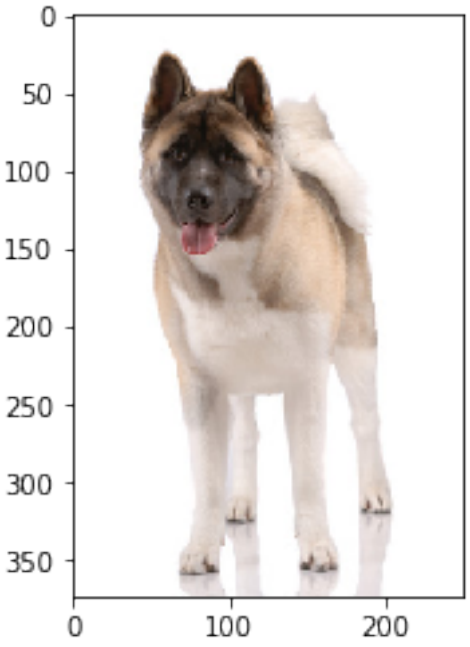
```
Error! Can't detect anything..
```



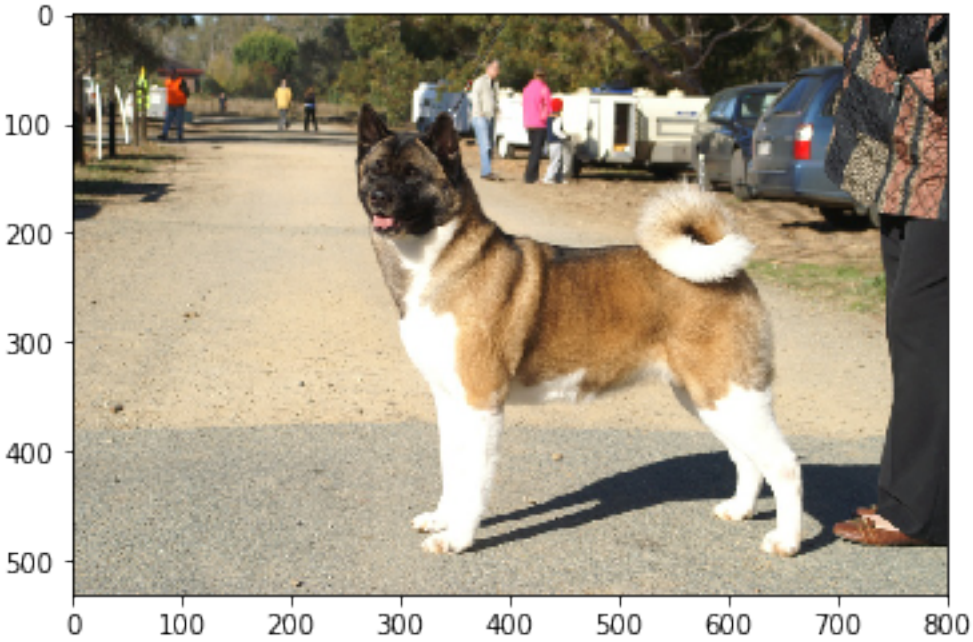

```
Dog Detected;
It is Akita
```




```
Error! Can't detect anything..
```



```
Dog Detected;
It is Norwegian elkhound
```




```
Dog Detected;
It is Akita
```

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!

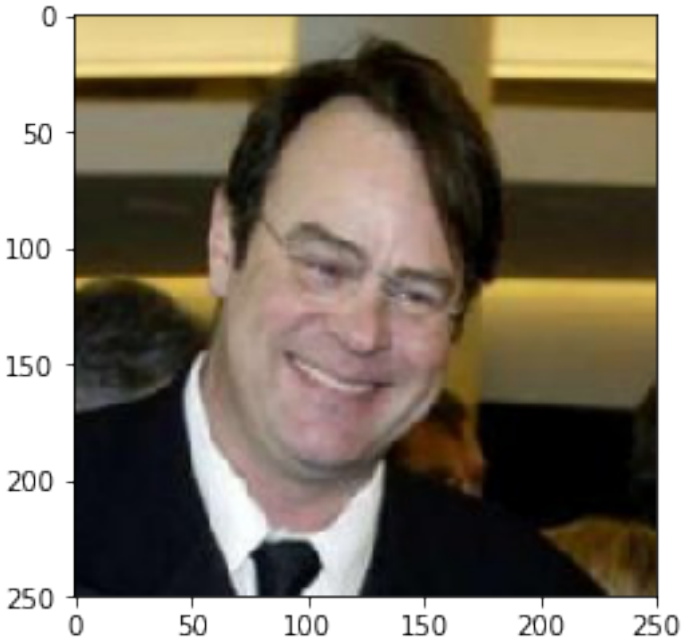
Test your algorithm at least six images on your computer. Feel free to use any images you like. Use at least two human and two dog images.

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

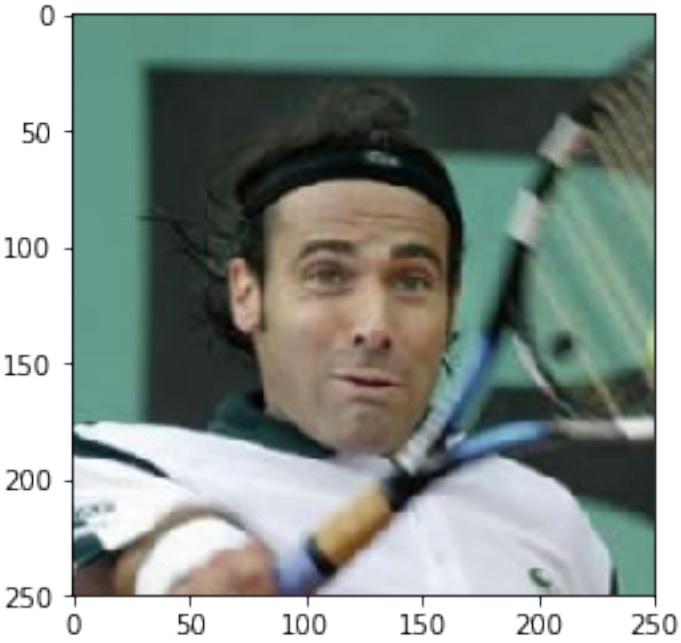
Answer: The output is better Tuning parameters such as learning rate, weight, batch_size would make it perform better Training on mora images might also improve the performance

```
In [45]: ## TODO: Execute your algorithm from Step 6 on
         ## at least 6 images on your computer.
```

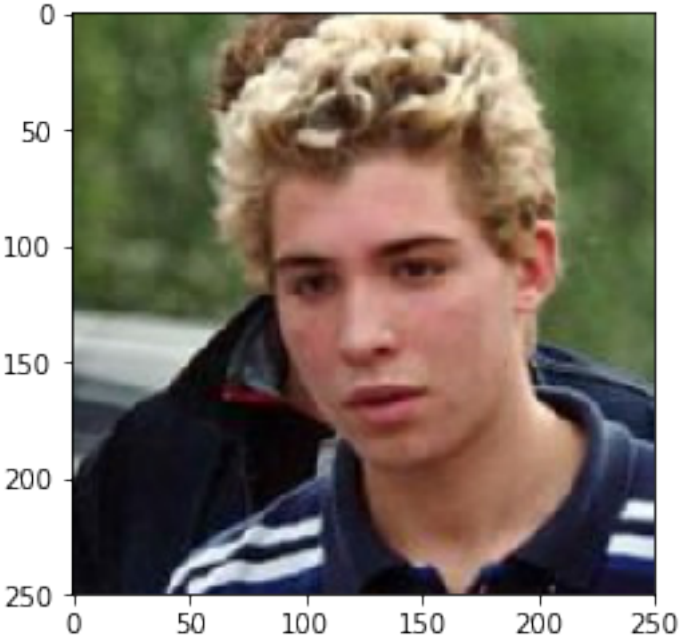
```
## Feel free to use as many code cells as needed.
my_human_files = ['./data/ifw/AJ_Cook/AJ_Cook_0001.jpg', './data/ifw/Adam_Ant/Adam_Ant_
my_dog_files = ['./data/dog_images/test/016.Beagle/Beeagle_01130.ipg', './data/dog_imag

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



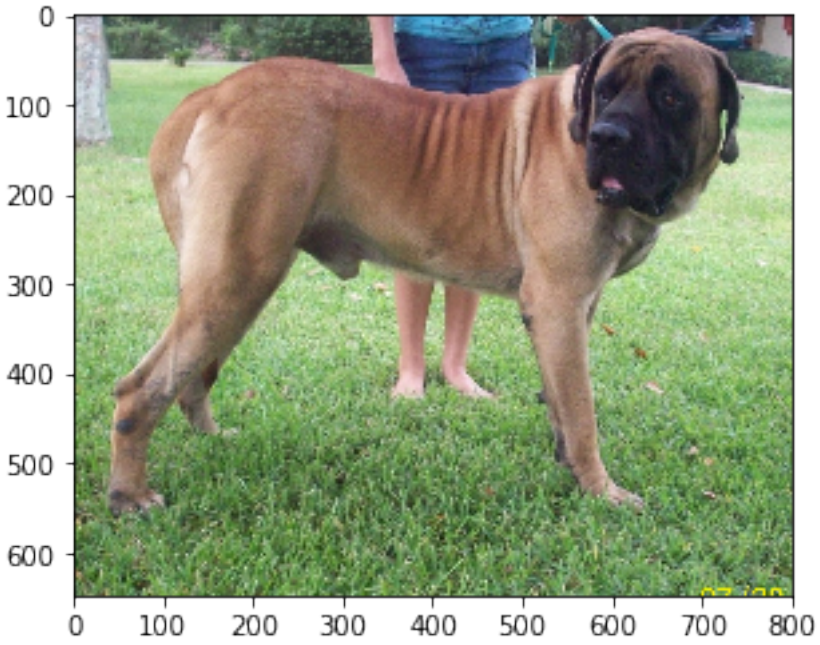

```
Hello Human!
You look like a Chihuahua
```



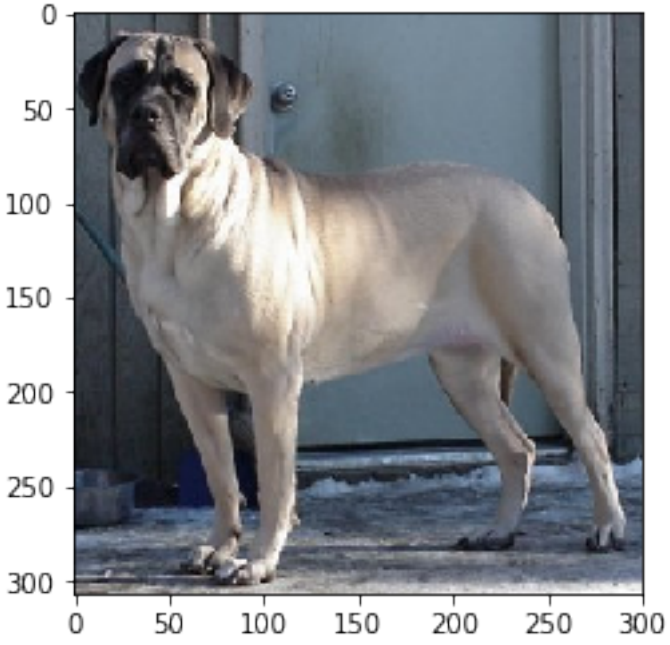

```
Hello Human!
You look like a Bull terrier
```



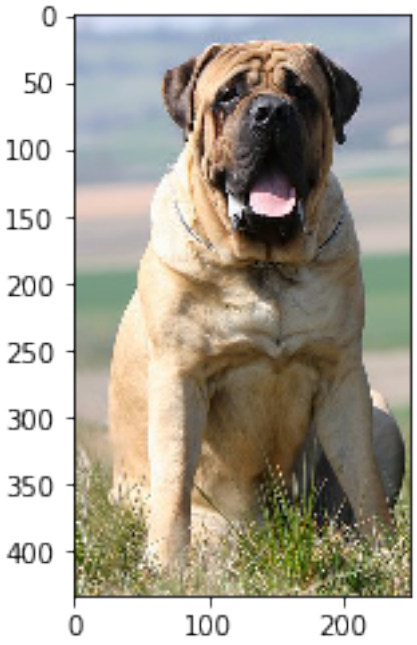
```
Hello Human!
You look like a American water spaniel
```




```
Dog Detected;
It is Bullmastiff
```



```
Dog Detected;
It is Bullmastiff
```




```
Dog Detected;
It is Bullmastiff
```