dog_app

November 18, 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 `/dogImages`.

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

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

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

Reviewer comments 2: Please include the HTML report, you can get this by opening File > Download as... > HTML in the Jupyter notebook menu. PS. It's sufficient to only include the Jupyter notebook, HTML report file and the tested images from step 6.

Dear reviewer, I have published the jupyter notebook and HTML file at the git hub dog breed repository. Regarding the images used in step 6. I downloaded them and stored them in the repository img subdirectory. Git hub rejects publishing the original images sizes exceed the file size limit.

In [1]:
```
import numpy as np
import pandas as pd
import cv2                                  # image processing
import torch
import torch.nn as nn
import torch.optim as optim                 # optimizer and loss function
import torch.nn.functional as F             # activation functions
import torchvision.models as models         # to load vgg16 model
import torchvision.transforms as transforms
import os                                   # system calls
import matplotlib.pyplot as plt             # plotting
import requests                             # for downloading data from a given URL
import ast
from tqdm import tqdm                       # show the progress of a loop, i.e tqdm
from glob import glob                       # image I/O
from PIL import Image                       # image manipulation
from PIL import ImageFile
from torchvision import datasets

import warnings
warnings.filterwarnings('ignore')


ImageFile.LOAD_TRUNCATED_IMAGES = True
%matplotlib inline


# evaluate if cuda is available
cuda_available = torch.cuda.is_available()
```

In [2]:
```
# load filenames for human and dog images
human_files = np.array(glob("lfw/*/*"))
dog_files = np.array(glob("dogImages/*/*/*"))

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

```
There are 13233 total human images.
There are 8351 total dog images.
```

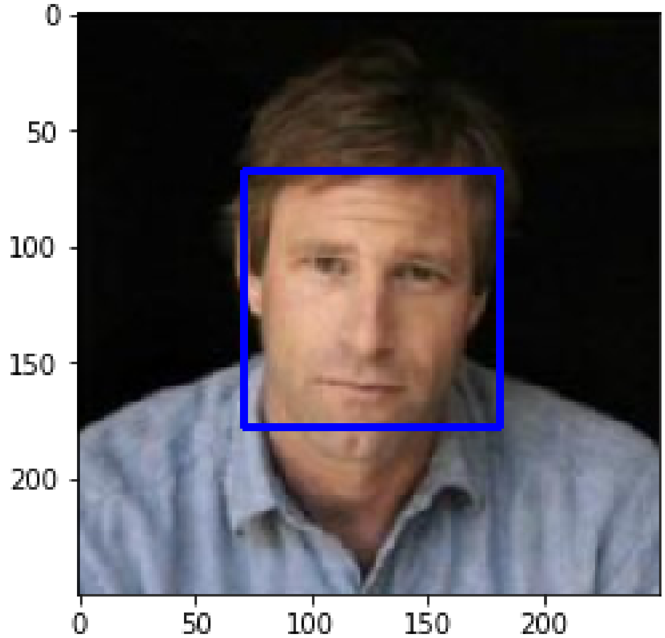
Step 1: Detect Humans

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 [37]:
```
# 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.

1.1.1 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 [28]: # returns "True" if face is detected in image stored at img_path
         def Haar_detector(img_path):
             # Load the model
             face_cascade = cv2.CascadeClassifier('haarcascades/haarcascade_frontalface_alt.xml

             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`.

```
In [73]: # NOTE: Helper function
         def detector_frequency(imgset, heuristic=None):
             bset = np.zeros((len(imgset)), dtype=bool)
             for i, fp in enumerate(imgset):
                 bset[i] = heuristic(fp)
             return bset
```

Answer:

Reviewer comments 1: What percentage of the first 100 images in `human_files` have a detected human face?

96%

Reviewer comments 1 : What percentage of the first 100 images in `dog_files` have a detected human face?

18%.

Short summary:

100 images corresponding to the people and dogs were tested for human face detection. Results show, that 96% and 18% of people and dog images contained a human face. 4% of human faces were not detected and 18% of dogs were incorrectly classified as having a human face. Results migh be explained since Haar classifier use change in contrast values between adjacent rectangular groups of pixels to detect features, such as eyes, mouth, and the edge of the face. However, many of these features are also present in dog and other classes of animal faces.

```
In [7]: human_files_short = human_files[:100]
        dog_files_short = dog_files[:100]
```

```
In [75]: #-#-# 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.
         hset = pd.DataFrame(detector_frequency(human_files_short, heuristic=Haar_detector))
         dset = pd.DataFrame(detector_frequency(dog_files_short, heuristic=Haar_detector))
```

```
In [76]: print('Human : ', hset[hset[0]==True].shape[0]/hset.shape[0])
         print('Dogs  : ', dset[dset[0]==True].shape[0]/dset.shape[0])
```

```
Human :  0.96
Dogs  :  0.18
```

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

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

1.1.3 Obtain Pre-trained VGG-16 Model

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

```
In [40]: # define VGG16 model
         vgg16 = models.vgg16(pretrained=True)

         # check if CUDA is available
         if cuda_available:
             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 [3]: # NOTE: Helper function
        def vgg16_labels():
            LABELS_MAP_URL = "https://gist.githubusercontent.com/yrevar/942d3a0ac09ec9e5eb3a/ra
            labels = ast.literal_eval(requests.get(LABELS_MAP_URL).text)
            return labels

In [3]: 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

        # load the model

        # define VGG16 model
        vgg16 = models.vgg16(pretrained=True)
        # check if CUDA is available
        if cuda_available:
            vgg16 = vgg16.cuda()


        img = Image.open(img_path)   # load the image from persistance

        # define image transformations
        transformations = transforms.Compose([
            transforms.Resize(size=224),
            transforms.CenterCrop((224,224)),
            transforms.ToTensor(),
            transforms.Normalize( mean=[0.485, 0.456, 0.406],
                                  std=[0.229, 0.224, 0.225])])

        # apply the image transformations
        img = transformations(img).unsqueeze_(0)

        # map to gpu
        if cuda_available:
            img = img.cuda()


        # predict the class for the image
        output = vgg16(img)

        # convert output probabilities to predicted class
        _, pred = torch.max(output, 1)

        pred = np.squeeze(pred.numpy()) if not cuda_available else np.squeeze(pred.cpu().nu
        return int(pred)
```

```
In [8]: # Test code
```

```
        labels = vgg16_labels()
        cid = VGG16_predict(dog_files_short[10])
        print(labels[cid])
```

```
Afghan hound, Afghan
```

```
In [48]: cid = VGG16_predict(human_files[3])
         print(labels[cid])
```

```
neck brace
```

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 [9]: ### returns "True" if a dog is detected in the image stored at img_path
        def VGG16_dog_detector(img_path):
            ## TODO: Complete the function.
            cid = VGG16_predict(img_path)
            if(cid >=151 and cid<=268):
                return True
            return False
```

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

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

Answer:

Reviewer comments 1: What percentage of the images in human_files_short have a detected dog?

None.

Reviewer comments 1: What percentage of the images in dog_files_short have a detected dog? 95%.

```
In [78]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.
         hset = pd.DataFrame(detector_frequency(human_files_short, heuristic=VGG16_dog_detecto
         dset = pd.DataFrame(detector_frequency(dog_files_short  , heuristic=VGG16_dog_detecto
```

```
In [79]: print('Human : ', hset[hset[0]==True].shape[0]/hset.shape[0])
         print('Dogs  : ', dset[dset[0]==True].shape[0]/dset.shape[0])
```

```
Human :  0.0
Dogs  :  0.95
```

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

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

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

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

Brittany	Welsh Springer Spaniel

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

Curly-Coated Retriever	American Water Spaniel

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

Yellow Labrador	Chocolate Labrador

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

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

1.1.7 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

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

Reviewer comments 2: Write three separate data loaders for the training, validation, and test datasets of dog images. These images should be pre-processed to be of the correct size. The preprocessing code is good, the only problem is that you also applied data augmentation on the validation and test set. The reason we don't want data augmentation for validation and testing is that we want the validation and test sets to represent real images as much as possible. If we augment the images they might not be realistic anymore. For the transformation for the validation and test set it's sufficient to only resize the image to 224x224 pixels.

Dear reviewer. I appreciate the acute observation. Unconsciously I reused the same data loader for validation and test phases. I have written the two new data loaders, namely: the validation and test loaders. These only apply image resize (224,224) and normalization. I ran new experiments using the transfer learning model that previously achieved test precision of 84%. Follow this Section **??** for new results.

In [36]:
```
### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes
data_transform = transforms.Compose([
        transforms.RandomHorizontalFlip(),
        transforms.RandomRotation(10),
        transforms.Resize(size=224),
        transforms.CenterCrop((224,224)),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406],
                             std=[0.229, 0.224, 0.225])])

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

data_transform_validate = transforms.Compose([
        transforms.Resize(size=224),
        transforms.CenterCrop((224,224)),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406],
                             std=[0.229, 0.224, 0.225])])

data_dir = 'dogImages/'
batch = 20
workers=0
```

```
trainset = datasets.ImageFolder(os.path.join(data_dir, 'train/'), transform=data_trans
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch, num_workers=worl

testset = datasets.ImageFolder(os.path.join(data_dir, 'test/'), transform=data_transfo
testloader = torch.utils.data.DataLoader(testset, batch_size=batch, num_workers=worke

validset = datasets.ImageFolder(os.path.join(data_dir, 'valid/'), transform=data_trans
validloader = torch.utils.data.DataLoader(validset, batch_size=batch, num_workers=worl

loaders_scratch = dict()
loaders_scratch['train'] = trainloader
loaders_scratch['test'] = testloader
loaders_scratch['valid'] = validloader
```

In [32]:

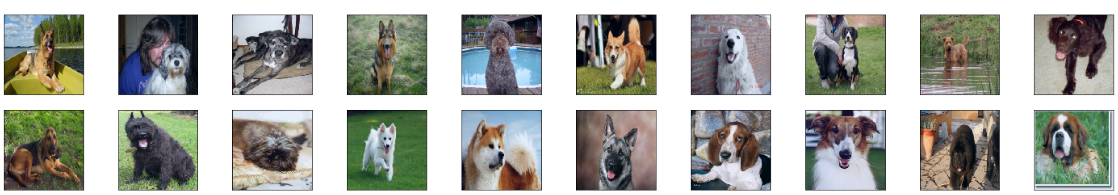
```
# Helper function:
def display_workload(loader):
    dataiter = iter(loader)
    imgset, lbset = dataiter.next()
    # imgset = imgset.numpy()

    fig = plt.figure(figsize=(25,4))
    for idx in np.arange(20):
        ax = fig.add_subplot(2, 20/2, idx+1, xticks=[], yticks=[])
        img = imgset[idx]
        img = (img-torch.min(img))/(torch.max(img)-torch.min(img)) #apply min-max nor
        img = np.transpose(img,(1, 2, 0))
        plt.imshow(img)
```

In [11]:

```
display_workload(loaders_scratch['train'])
```

In [33]:

```
display_workload(loaders_scratch['test'])
```



In [34]:
```
display_workload(loaders_scratch['valid'])
```

Question 3: Describe your chosen procedure for preprocessing the data. 1. How does your code resize the images (by cropping, stretching, etc)? What size did you pick for the input tensor, and why? 2. Did you decide to augment the dataset? If so, how (through translations, flips, rotations, etc)? If not, why not?

Answer:

Reviewer comments 1: How does your code resize the images (by cropping, stretching, etc)? What size did you pick for the input tensor, and why?

Images were randomly rotated, resized to 224 x 224 and used 3 channels, centered crop, and normalized. Images are relatively large, compared with MNIST images sizes. The reason for working with the proposed size, was to preseve, as many as possible, the original image properties and invariants. So that, during training these are generalized by the learning heuristic. Secondly, I also considered the amount of available resources, CPU and GPU memory buffer sizes, as a criterion for bounding the image size. I tested several image sizes.

Reviewer comments 1: Did you decide to augment the dataset? If so, how (through translations, flips, rotations, etc)? If not, why not?

There are numerous augmentation techniques, i.e: geometric transformations, color space augmentations, kernel filters, mixing images, random erasing, feature space augmentation, adversarial training, generative adversarial networks, among others. See (Connor Shorten and Taghi M. Khoshgoftaar, 2019). Augmentation techniques are often applied as a pre-processing phase, were produced images are added to the original dataset. Kernel filters are commonly applied to blur or sharpen images. Convolutional neural networks include kernel filters in its encoding phase. Thus it automatically includes data augmentation. I did not apply data augmentation as a pre-processing phase.

1.1.8 (IMPLEMENTATION) Model Architecture

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

In [42]:
```
# define the CNN architecture
class Net(nn.Module):
    ### TODO: choose an architecture, and complete the class
    def __init__(self):
        super(Net, self).__init__()

        #define the convolutional layer
        self.conv1 = nn.Conv2d(3,16,3,padding=1)
        self.conv2 = nn.Conv2d(16,32,3,padding=1)
        self.conv3 = nn.Conv2d(32,64,3,padding=1)
```

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

        ## Define layers of a CNN
        self.fc1 = nn.Linear(64*28*28,1000)
        self.fc2 = nn.Linear(1000,1000)
        self.fc3 = nn.Linear(1000,500)
        self.fc4 = nn.Linear(500,133)

        self.batch_norm2 = nn.BatchNorm1d(num_features=1000)
        self.batch_norm3 = nn.BatchNorm1d(num_features=500)
        self.dropout = nn.Dropout(0.25)

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

        x = x.view(x.size(0), -1)

        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = F.relu(self.fc2(x))
        x = self.dropout(x)
        x = self.batch_norm2(x)
        x = F.relu(self.fc3(x))
        x = self.batch_norm3(x)
        x = self.fc4(x)

        return x

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

# instantiate the CNN
model_scratch = Net()

# move tensors to GPU if CUDA is available
if cuda_available:
    model_scratch.cuda()
```

In [43]:
```
print(model_scratch)
```

```
Net(
  (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

```
  (conv3): Conv2d(32, 64, 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=50176, out_features=1000, bias=True)
  (fc2): Linear(in_features=1000, out_features=1000, bias=True)
  (fc3): Linear(in_features=1000, out_features=500, bias=True)
  (fc4): Linear(in_features=500, out_features=133, bias=True)
  (batch_norm2): BatchNorm1d(1000, eps=1e-05, momentum=0.1, affine=True, track_running_stats=T
  (batch_norm3): BatchNorm1d(500, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tr
  (dropout): Dropout(p=0.25, inplace=False)
)
```

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

Answer:

Reviewer comments 1: Outline the steps you took to get to your final CNN architecture and your reasoning at each step.

The CNN consists of three convolutional layers whose aim is to augment the data set by both reducing image noise and increasing the acutence of images. After each convolutional phase, negative values are removed via the relu activation function. Since the size and number of images resulting after a convolutional phase may be large, pooling is applied in order to reduce the amount of information and size of the image. Pooling may introduce information loss, but may be reduced by data augmentation. Finally, dropout is introduced in order to reduce overfitting. The depth, number of layers, in the reduction phase is bounded by the amount of information loss tolerable for the problem instance. In the proposed solution, images were shrunk significantly until there size reached 28x28.

The neural network layer, consists of four layers. Two dropout layers were added in order to reduce overfitting. Batch normalization is added in order to accelerate training and reduce covariance shift. Two phase of normalization was applied.

The network was tested only a few times, but results were not promising. A test accuracy of 13% was achieved. The proposed feature extraction phase depth, which compromise the convolutional layers, is shallow when compared to VGG16 feature extraction phase. Perhaps, the performance of the proposed network might be improved by improving such phase.

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

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

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 [45]:
```
# the following import is required for training to be robust to truncated images
ImageFile.LOAD_TRUNCATED_IMAGES = True

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

    # record train ans validation lossess
    loss_batch = []
    loss_valid = []

    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_

            # clear the gradients of all optimized variables
            optimizer.zero_grad()
            # forward pass: compute predicted outputs by passing inputs to the model
            output = model(data)
            # calculate the batch loss
            loss = criterion(output, target)
            # backward pass: compute gradient of the loss with respect to model param
            loss.backward()
            # perform a single optimization step (parameter update)
            optimizer.step()
            # update training loss
            train_loss += loss.item()*data.size(0)

        ######################
        # validate the model #
        ######################
        model.eval()
        for batch_idx, (data, target) in enumerate(loaders['valid']):
            # move to GPU
```

```python
                if use_cuda:
                    data, target = data.cuda(), target.cuda()
                ## update the average validation loss

                # forward pass: compute predicted outputs by passing inputs to the model
                output = model(data)
                # calculate the batch loss
                loss = criterion(output, target)
                # update average validation loss
                valid_loss += loss.item()*data.size(0)


            # calculate average losses
            train_loss = train_loss/len(loaders['train'].sampler)
            valid_loss = valid_loss/len(loaders['valid'].sampler)
            #record average losssess
            loss_batch.append(train_loss)
            loss_valid.append(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
            # save model if validation loss has decreased
            if valid_loss <= valid_loss_min:
                print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.f
                valid_loss_min,
                valid_loss))
                torch.save(model.state_dict(),save_path)
                valid_loss_min = valid_loss
        return loss_batch, valid_loss, model


    # train the model
    loss_batch, valid_loss, model_scratch = train(100, loaders_scratch, model_scratch, opt
                          criterion_scratch, cuda_available, 'model_scratch2.pt')

    # load the model that got the best validation accuracy
    model_scratch.load_state_dict(torch.load('model_scratch2.pt'))
```

```
Epoch: 1         Training Loss: 4.983277         Validation Loss: 126.376283
Validation loss decreased (inf --> 126.376283).  Saving model ...
Epoch: 2         Training Loss: 4.807230         Validation Loss: 4.897680
Validation loss decreased (126.376283 --> 4.897680).  Saving model ...
```

```
Epoch: 3        Training Loss: 4.669751        Validation Loss: 4.891207
Validation loss decreased (4.897680 --> 4.891207).  Saving model ...
Epoch: 4        Training Loss: 4.511940        Validation Loss: 4.522735
Validation loss decreased (4.891207 --> 4.522735).  Saving model ...
Epoch: 5        Training Loss: 4.395568        Validation Loss: 4.641390
Epoch: 6        Training Loss: 4.297917        Validation Loss: 4.675839
Epoch: 7        Training Loss: 4.209301        Validation Loss: 4.440095
Validation loss decreased (4.522735 --> 4.440095).  Saving model ...
Epoch: 8        Training Loss: 4.105937        Validation Loss: 4.399250
Validation loss decreased (4.440095 --> 4.399250).  Saving model ...
Epoch: 9        Training Loss: 4.016523        Validation Loss: 4.176144
Validation loss decreased (4.399250 --> 4.176144).  Saving model ...
Epoch: 10        Training Loss: 3.911892        Validation Loss: 4.216344
Epoch: 11        Training Loss: 3.837239        Validation Loss: 4.367939
Epoch: 12        Training Loss: 3.736571        Validation Loss: 4.039827
Validation loss decreased (4.176144 --> 4.039827).  Saving model ...
Epoch: 13        Training Loss: 3.646937        Validation Loss: 3.958032
Validation loss decreased (4.039827 --> 3.958032).  Saving model ...
Epoch: 14        Training Loss: 3.554220        Validation Loss: 4.116770
Epoch: 15        Training Loss: 3.478735        Validation Loss: 3.948525
Validation loss decreased (3.958032 --> 3.948525).  Saving model ...
Epoch: 16        Training Loss: 3.363377        Validation Loss: 3.884706
Validation loss decreased (3.948525 --> 3.884706).  Saving model ...
Epoch: 17        Training Loss: 3.274325        Validation Loss: 3.838474
Validation loss decreased (3.884706 --> 3.838474).  Saving model ...
Epoch: 18        Training Loss: 3.169669        Validation Loss: 3.942212
Epoch: 19        Training Loss: 3.074814        Validation Loss: 4.224119
Epoch: 20        Training Loss: 2.954479        Validation Loss: 3.864537
Epoch: 21        Training Loss: 2.834234        Validation Loss: 3.882875
Epoch: 22        Training Loss: 2.738972        Validation Loss: 3.845746
Epoch: 23        Training Loss: 2.632764        Validation Loss: 3.886914
Epoch: 24        Training Loss: 2.496515        Validation Loss: 3.909535
Epoch: 25        Training Loss: 2.390710        Validation Loss: 3.821160
Validation loss decreased (3.838474 --> 3.821160).  Saving model ...
Epoch: 26        Training Loss: 2.253152        Validation Loss: 3.874713
Epoch: 27        Training Loss: 2.142426        Validation Loss: 3.911588
Epoch: 28        Training Loss: 2.034972        Validation Loss: 3.964688
Epoch: 29        Training Loss: 1.899691        Validation Loss: 3.840306
Epoch: 30        Training Loss: 1.748688        Validation Loss: 3.940062
Epoch: 31        Training Loss: 1.647949        Validation Loss: 3.963384
Epoch: 32        Training Loss: 1.519266        Validation Loss: 4.065378
Epoch: 33        Training Loss: 1.422767        Validation Loss: 4.038896
Epoch: 34        Training Loss: 1.363646        Validation Loss: 4.109205
Epoch: 35        Training Loss: 1.255184        Validation Loss: 4.021675
Epoch: 36        Training Loss: 1.122894        Validation Loss: 4.005271
Epoch: 37        Training Loss: 1.043092        Validation Loss: 4.046155
Epoch: 38        Training Loss: 0.984172        Validation Loss: 4.080964
Epoch: 39        Training Loss: 0.863840        Validation Loss: 4.107561
```

```
Epoch: 40          Training Loss: 0.809669          Validation Loss: 4.238978
Epoch: 41          Training Loss: 0.750227          Validation Loss: 4.213589
Epoch: 42          Training Loss: 0.676087          Validation Loss: 4.331332
Epoch: 43          Training Loss: 0.625549          Validation Loss: 4.256974
Epoch: 44          Training Loss: 0.573893          Validation Loss: 4.268788
Epoch: 45          Training Loss: 0.554618          Validation Loss: 4.332600
Epoch: 46          Training Loss: 0.515905          Validation Loss: 4.343984
Epoch: 47          Training Loss: 0.476054          Validation Loss: 4.472133
Epoch: 48          Training Loss: 0.450636          Validation Loss: 4.334009
Epoch: 49          Training Loss: 0.407509          Validation Loss: 4.401799
Epoch: 50          Training Loss: 0.392475          Validation Loss: 4.512152
Epoch: 51          Training Loss: 0.369875          Validation Loss: 4.464147
Epoch: 52          Training Loss: 0.351269          Validation Loss: 4.500285
Epoch: 53          Training Loss: 0.315536          Validation Loss: 4.584879
Epoch: 54          Training Loss: 0.319270          Validation Loss: 4.595180
Epoch: 55          Training Loss: 0.306266          Validation Loss: 4.578832
Epoch: 56          Training Loss: 0.282546          Validation Loss: 4.659887
Epoch: 57          Training Loss: 0.267772          Validation Loss: 4.656134
Epoch: 58          Training Loss: 0.232334          Validation Loss: 4.637737
Epoch: 59          Training Loss: 0.262685          Validation Loss: 4.702690
Epoch: 60          Training Loss: 0.223369          Validation Loss: 4.870451
Epoch: 61          Training Loss: 0.225377          Validation Loss: 4.781228
Epoch: 62          Training Loss: 0.215229          Validation Loss: 4.855132
Epoch: 63          Training Loss: 0.202986          Validation Loss: 4.796535
Epoch: 64          Training Loss: 0.207825          Validation Loss: 4.838460
Epoch: 65          Training Loss: 0.177890          Validation Loss: 4.919020
Epoch: 66          Training Loss: 0.180397          Validation Loss: 4.842749
Epoch: 67          Training Loss: 0.173632          Validation Loss: 4.844027
Epoch: 68          Training Loss: 0.178193          Validation Loss: 4.830736
Epoch: 69          Training Loss: 0.159565          Validation Loss: 4.857112
Epoch: 70          Training Loss: 0.160898          Validation Loss: 4.801183
Epoch: 71          Training Loss: 0.169295          Validation Loss: 5.008532
Epoch: 72          Training Loss: 0.150117          Validation Loss: 4.862096
Epoch: 73          Training Loss: 0.139683          Validation Loss: 4.878680
Epoch: 74          Training Loss: 0.152478          Validation Loss: 4.808571
Epoch: 75          Training Loss: 0.134955          Validation Loss: 4.871299
Epoch: 76          Training Loss: 0.125851          Validation Loss: 4.848272
Epoch: 77          Training Loss: 0.126269          Validation Loss: 4.970186
Epoch: 78          Training Loss: 0.128774          Validation Loss: 4.977873
Epoch: 79          Training Loss: 0.119772          Validation Loss: 4.924303
Epoch: 80          Training Loss: 0.107659          Validation Loss: 5.115578
Epoch: 81          Training Loss: 0.109833          Validation Loss: 5.072667
Epoch: 82          Training Loss: 0.109573          Validation Loss: 4.979229
Epoch: 83          Training Loss: 0.107458          Validation Loss: 5.177440
Epoch: 84          Training Loss: 0.106956          Validation Loss: 5.068801
Epoch: 85          Training Loss: 0.102681          Validation Loss: 4.993513
Epoch: 86          Training Loss: 0.106469          Validation Loss: 5.100525
Epoch: 87          Training Loss: 0.093274          Validation Loss: 4.992168
```

```
Epoch: 88         Training Loss: 0.094210         Validation Loss: 5.056087
Epoch: 89         Training Loss: 0.100516         Validation Loss: 5.055184
Epoch: 90         Training Loss: 0.103470         Validation Loss: 5.180474
Epoch: 91         Training Loss: 0.096631         Validation Loss: 5.026805
Epoch: 92         Training Loss: 0.082145         Validation Loss: 5.080624
Epoch: 93         Training Loss: 0.087390         Validation Loss: 5.103360
Epoch: 94         Training Loss: 0.084554         Validation Loss: 5.028766
Epoch: 95         Training Loss: 0.074126         Validation Loss: 5.183977
Epoch: 96         Training Loss: 0.088981         Validation Loss: 5.148665
Epoch: 97         Training Loss: 0.086385         Validation Loss: 5.187917
Epoch: 98         Training Loss: 0.076803         Validation Loss: 5.196313
Epoch: 99         Training Loss: 0.081358         Validation Loss: 5.070851
Epoch: 100         Training Loss: 0.085773         Validation Loss: 5.088829
```

```
Out[45]: <All keys matched successfully>
```

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 [37]: 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 [13]: # call test function
         test(loaders_scratch, model_scratch, criterion_scratch, cuda_available)
```

```
Test Loss: 3.817867


Test Accuracy: 13% (116/836)
```

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 [46]: ## TODO: Specify data loaders
         loaders_transfer = dict()
         loaders_transfer['train'] = trainloader
         loaders_transfer['test'] = testloader
         loaders_transfer['valid'] = validloader
```

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 [25]: class classifier(nn.Module):
             def __init__(self, c_0_inputs):
                 super().__init__()
                 self.fc1 = nn.Linear(c_0_inputs, 1000)
                 self.fc2 = nn.Linear(1000, 1000)
                 self.fc3 = nn.Linear(1000, 500)
                 self.fc4 = nn.Linear(500,133)

                 self.batch_norm2 = nn.BatchNorm1d(num_features=1000)
                 self.batch_norm3 = nn.BatchNorm1d(num_features=500)
                 self.dropout = nn.Dropout(0.25)
```

```
    def forward(self, x):
        # make sure input tensor is flattened
        x = x.view(x.size(0), -1)

        x = self.fc1(x)
        x = F.relu(x)
        x = self.dropout(x)
        x = self.fc2(x)
        x = F.relu(x)
        x = self.dropout(x)
        x = self.batch_norm2(x)
        x = self.fc3(x)
        x = F.relu(x)
        x = self.batch_norm3(x)
        x = self.fc4(x)

        return x
```

In [26]:
```
## TODO: Specify model architecture
vgg16 = models.vgg16(pretrained=True)

# Freeze training for all "features" layers
for param in vgg16.features.parameters():
    param.requires_grad = False

# I will replace all the classification layers.

c_0_inputs  = vgg16.classifier[0].in_features
vgg16.classifier = classifier(c_0_inputs)

print(vgg16.classifier)

if cuda_available:
    model_transfer = vgg16.cuda()
```

```
classifier(
  (fc1): Linear(in_features=25088, out_features=1000, bias=True)
  (fc2): Linear(in_features=1000, out_features=1000, bias=True)
  (fc3): Linear(in_features=1000, out_features=500, bias=True)
  (fc4): Linear(in_features=500, out_features=133, bias=True)
  (batch_norm2): BatchNorm1d(1000, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tr
  (batch_norm3): BatchNorm1d(500, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
  (dropout): Dropout(p=0.25, inplace=False)
)
```

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.

Reviewer comments 1: 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.

In this exercise, I kept vgg16 feature extraction phase and replaced the learning phase entirely by replacing it with the learning phase of the cnn that I created earlier. I did this, in order to test my hypothesis, that the feature extraction phase in the custom made cnn was not properly design. Features in the feature extraction phase were kept and in the learning phase were retrained. Results were very promising, test accuracy of 84% was achieved.

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 [39]:
```
criterion_transfer = nn.CrossEntropyLoss()
optimizer_transfer = optim.SGD(vgg16.classifier.parameters(), lr=0.001)
```

1.1.15 (IMPLEMENTATION) Train and Validate the Model

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

In [25]:
```
# train the transfer model
loss_batch, valid_loss, model_transfer = train(100, loaders_transfer, model_transfer,
                     criterion_transfer, cuda_available, 'model_scratch3.pt')


# load the model that got the best validation accuracy (uncomment the line below)
model_transfer.load_state_dict(torch.load('model_scratch3.pt'))
```

```
Epoch: 1         Training Loss: 4.597066         Validation Loss: 4.071239
Validation loss decreased (inf --> 4.071239).  Saving model ...
Epoch: 2         Training Loss: 3.832786         Validation Loss: 3.294235
Validation loss decreased (4.071239 --> 3.294235).  Saving model ...
Epoch: 3         Training Loss: 3.225330         Validation Loss: 2.744397
Validation loss decreased (3.294235 --> 2.744397).  Saving model ...
Epoch: 4         Training Loss: 2.743811         Validation Loss: 2.333085
Validation loss decreased (2.744397 --> 2.333085).  Saving model ...
Epoch: 5         Training Loss: 2.377515         Validation Loss: 2.024208
Validation loss decreased (2.333085 --> 2.024208).  Saving model ...
Epoch: 6         Training Loss: 2.087905         Validation Loss: 1.792346
Validation loss decreased (2.024208 --> 1.792346).  Saving model ...
Epoch: 7         Training Loss: 1.852640         Validation Loss: 1.595844
Validation loss decreased (1.792346 --> 1.595844).  Saving model ...
Epoch: 8         Training Loss: 1.657355         Validation Loss: 1.470441
Validation loss decreased (1.595844 --> 1.470441).  Saving model ...
Epoch: 9         Training Loss: 1.489828         Validation Loss: 1.308649
Validation loss decreased (1.470441 --> 1.308649).  Saving model ...
Epoch: 10         Training Loss: 1.368397         Validation Loss: 1.211461
```

```
Validation loss decreased (1.308649 --> 1.211461).  Saving model ...
Epoch: 11       Training Loss: 1.240888       Validation Loss: 1.113330
Validation loss decreased (1.211461 --> 1.113330).  Saving model ...
Epoch: 12       Training Loss: 1.128628       Validation Loss: 1.017311
Validation loss decreased (1.113330 --> 1.017311).  Saving model ...
Epoch: 13       Training Loss: 1.043189       Validation Loss: 0.965731
Validation loss decreased (1.017311 --> 0.965731).  Saving model ...
Epoch: 14       Training Loss: 0.956369       Validation Loss: 0.918134
Validation loss decreased (0.965731 --> 0.918134).  Saving model ...
Epoch: 15       Training Loss: 0.885994       Validation Loss: 0.872082
Validation loss decreased (0.918134 --> 0.872082).  Saving model ...
Epoch: 16       Training Loss: 0.817838       Validation Loss: 0.828464
Validation loss decreased (0.872082 --> 0.828464).  Saving model ...
Epoch: 17       Training Loss: 0.768766       Validation Loss: 0.795059
Validation loss decreased (0.828464 --> 0.795059).  Saving model ...
Epoch: 18       Training Loss: 0.702580       Validation Loss: 0.771254
Validation loss decreased (0.795059 --> 0.771254).  Saving model ...
Epoch: 19       Training Loss: 0.663819       Validation Loss: 0.726378
Validation loss decreased (0.771254 --> 0.726378).  Saving model ...
Epoch: 20       Training Loss: 0.616127       Validation Loss: 0.679994
Validation loss decreased (0.726378 --> 0.679994).  Saving model ...
Epoch: 21       Training Loss: 0.565106       Validation Loss: 0.686237
Epoch: 22       Training Loss: 0.533259       Validation Loss: 0.677633
Validation loss decreased (0.679994 --> 0.677633).  Saving model ...
Epoch: 23       Training Loss: 0.499670       Validation Loss: 0.669004
Validation loss decreased (0.677633 --> 0.669004).  Saving model ...
Epoch: 24       Training Loss: 0.472985       Validation Loss: 0.642696
Validation loss decreased (0.669004 --> 0.642696).  Saving model ...
Epoch: 25       Training Loss: 0.442701       Validation Loss: 0.632628
Validation loss decreased (0.642696 --> 0.632628).  Saving model ...
Epoch: 26       Training Loss: 0.418468       Validation Loss: 0.604674
Validation loss decreased (0.632628 --> 0.604674).  Saving model ...
Epoch: 27       Training Loss: 0.389862       Validation Loss: 0.598525
Validation loss decreased (0.604674 --> 0.598525).  Saving model ...
Epoch: 28       Training Loss: 0.362864       Validation Loss: 0.582931
Validation loss decreased (0.598525 --> 0.582931).  Saving model ...
Epoch: 29       Training Loss: 0.341189       Validation Loss: 0.589163
Epoch: 30       Training Loss: 0.330714       Validation Loss: 0.564195
Validation loss decreased (0.582931 --> 0.564195).  Saving model ...
Epoch: 31       Training Loss: 0.310584       Validation Loss: 0.572365
Epoch: 32       Training Loss: 0.292987       Validation Loss: 0.559096
Validation loss decreased (0.564195 --> 0.559096).  Saving model ...
Epoch: 33       Training Loss: 0.274128       Validation Loss: 0.556561
Validation loss decreased (0.559096 --> 0.556561).  Saving model ...
Epoch: 34       Training Loss: 0.263129       Validation Loss: 0.539932
Validation loss decreased (0.556561 --> 0.539932).  Saving model ...
Epoch: 35       Training Loss: 0.246184       Validation Loss: 0.541683
Epoch: 36       Training Loss: 0.233690       Validation Loss: 0.531535
```

```
Validation loss decreased (0.539932 --> 0.531535).  Saving model ...
Epoch: 37         Training Loss: 0.224427         Validation Loss: 0.531884
Epoch: 38         Training Loss: 0.213584         Validation Loss: 0.533269
Epoch: 39         Training Loss: 0.205544         Validation Loss: 0.526437
Validation loss decreased (0.531535 --> 0.526437).  Saving model ...
Epoch: 40         Training Loss: 0.189440         Validation Loss: 0.515294
Validation loss decreased (0.526437 --> 0.515294).  Saving model ...
Epoch: 41         Training Loss: 0.180155         Validation Loss: 0.523311
Epoch: 42         Training Loss: 0.176149         Validation Loss: 0.510906
Validation loss decreased (0.515294 --> 0.510906).  Saving model ...
Epoch: 43         Training Loss: 0.165825         Validation Loss: 0.512147
Epoch: 44         Training Loss: 0.161219         Validation Loss: 0.508459
Validation loss decreased (0.510906 --> 0.508459).  Saving model ...
Epoch: 45         Training Loss: 0.152159         Validation Loss: 0.511513
Epoch: 46         Training Loss: 0.145803         Validation Loss: 0.512393
Epoch: 47         Training Loss: 0.140234         Validation Loss: 0.501484
Validation loss decreased (0.508459 --> 0.501484).  Saving model ...
Epoch: 48         Training Loss: 0.130443         Validation Loss: 0.495359
Validation loss decreased (0.501484 --> 0.495359).  Saving model ...
Epoch: 49         Training Loss: 0.129843         Validation Loss: 0.492279
Validation loss decreased (0.495359 --> 0.492279).  Saving model ...
Epoch: 50         Training Loss: 0.126662         Validation Loss: 0.486382
Validation loss decreased (0.492279 --> 0.486382).  Saving model ...
Epoch: 51         Training Loss: 0.121543         Validation Loss: 0.487514
Epoch: 52         Training Loss: 0.115404         Validation Loss: 0.473096
Validation loss decreased (0.486382 --> 0.473096).  Saving model ...
Epoch: 53         Training Loss: 0.109280         Validation Loss: 0.480492
Epoch: 54         Training Loss: 0.108048         Validation Loss: 0.495054
Epoch: 55         Training Loss: 0.100715         Validation Loss: 0.488889
Epoch: 56         Training Loss: 0.099200         Validation Loss: 0.493808
Epoch: 57         Training Loss: 0.096568         Validation Loss: 0.484988
Epoch: 58         Training Loss: 0.091654         Validation Loss: 0.491377
Epoch: 59         Training Loss: 0.090946         Validation Loss: 0.502080
Epoch: 60         Training Loss: 0.087332         Validation Loss: 0.498340
Epoch: 61         Training Loss: 0.086527         Validation Loss: 0.494710
Epoch: 62         Training Loss: 0.081584         Validation Loss: 0.482120
Epoch: 63         Training Loss: 0.076770         Validation Loss: 0.489178
Epoch: 64         Training Loss: 0.075926         Validation Loss: 0.474426
Epoch: 65         Training Loss: 0.075897         Validation Loss: 0.495338
Epoch: 66         Training Loss: 0.071602         Validation Loss: 0.502745
Epoch: 67         Training Loss: 0.072047         Validation Loss: 0.480503
Epoch: 68         Training Loss: 0.068900         Validation Loss: 0.489096
Epoch: 69         Training Loss: 0.066317         Validation Loss: 0.480594
Epoch: 70         Training Loss: 0.066531         Validation Loss: 0.486492
Epoch: 71         Training Loss: 0.062390         Validation Loss: 0.476981
Epoch: 72         Training Loss: 0.059585         Validation Loss: 0.484700
Epoch: 73         Training Loss: 0.061383         Validation Loss: 0.472365
Validation loss decreased (0.473096 --> 0.472365).  Saving model ...
```

```
Epoch: 74         Training Loss: 0.058774         Validation Loss: 0.493007
Epoch: 75         Training Loss: 0.059019         Validation Loss: 0.482265
Epoch: 76         Training Loss: 0.056204         Validation Loss: 0.487423
Epoch: 77         Training Loss: 0.057195         Validation Loss: 0.479320
Epoch: 78         Training Loss: 0.054174         Validation Loss: 0.490018
Epoch: 79         Training Loss: 0.052546         Validation Loss: 0.481440
Epoch: 80         Training Loss: 0.049684         Validation Loss: 0.463954
Validation loss decreased (0.472365 --> 0.463954).  Saving model ...
Epoch: 81         Training Loss: 0.051010         Validation Loss: 0.483553
Epoch: 82         Training Loss: 0.048822         Validation Loss: 0.460831
Validation loss decreased (0.463954 --> 0.460831).  Saving model ...
Epoch: 83         Training Loss: 0.048803         Validation Loss: 0.477836
Epoch: 84         Training Loss: 0.046561         Validation Loss: 0.487093
Epoch: 85         Training Loss: 0.045667         Validation Loss: 0.476361
Epoch: 86         Training Loss: 0.046498         Validation Loss: 0.478918
Epoch: 87         Training Loss: 0.044026         Validation Loss: 0.470675
Epoch: 88         Training Loss: 0.045211         Validation Loss: 0.464756
Epoch: 89         Training Loss: 0.044179         Validation Loss: 0.475361
Epoch: 90         Training Loss: 0.042325         Validation Loss: 0.483828
Epoch: 91         Training Loss: 0.042452         Validation Loss: 0.463791
Epoch: 92         Training Loss: 0.040530         Validation Loss: 0.461863
Epoch: 93         Training Loss: 0.041583         Validation Loss: 0.473152
Epoch: 94         Training Loss: 0.039412         Validation Loss: 0.461955
Epoch: 95         Training Loss: 0.039283         Validation Loss: 0.476914
Epoch: 96         Training Loss: 0.038170         Validation Loss: 0.462210
Epoch: 97         Training Loss: 0.038758         Validation Loss: 0.484857
Epoch: 98         Training Loss: 0.037592         Validation Loss: 0.476753
Epoch: 99         Training Loss: 0.037244         Validation Loss: 0.469756
Epoch: 100          Training Loss: 0.035534         Validation Loss: 0.473606
```

```
Out[25]: <All keys matched successfully>
```

(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 [21]: model_transfer.load_state_dict(torch.load('model_scratch3.pt'))
```

```
Out[21]: <All keys matched successfully>
```

```
In [13]: test(loaders_transfer, model_transfer, criterion_transfer, cuda_available)
```

```
Test Loss: 0.501152


Test Accuracy: 84% (706/836)
```

My comments 2: After modifying the loaders, I directly applied the transfer-learned model (model_scratch3.pt), which previously achieved a precision of 84% to the new test set data loader. The new achieved accuracy is of 49%. The code for this the new test is show in the following cell.

```
In [38]: model_transfer.load_state_dict(torch.load('model_scratch3.pt'))
         test(loaders_transfer, model_transfer, criterion_transfer, cuda_available)
```

```
Test Loss: 2.046817


Test Accuracy: 49% (411/836)
```

My comments 2: Results degradate by a factor of 1.7!. It seems that image pre-processing phases corresponding to random horizontal flip, rotation, and center cropping significantly improved test accuracy. As described earlier, model scratch 3 was not retrained using a validation loader that omitted the previously mention transformations. To analyze if the model quality improves, it is retrained and retested. See code in the following cells.

```
In [47]: # train the transfer model
         loss_batch, valid_loss, model_transfer = train(100, loaders_transfer, model_transfer,
                                 criterion_transfer, cuda_available, 'model_scratch4.pt')
```

```
Epoch: 1         Training Loss: 0.047734         Validation Loss: 0.469103
Validation loss decreased (inf --> 0.469103).  Saving model ...
Epoch: 2         Training Loss: 0.048244         Validation Loss: 0.463061
Validation loss decreased (0.469103 --> 0.463061).  Saving model ...
Epoch: 3         Training Loss: 0.047332         Validation Loss: 0.467075
Epoch: 4         Training Loss: 0.043777         Validation Loss: 0.466281
Epoch: 5         Training Loss: 0.044889         Validation Loss: 0.468572
Epoch: 6         Training Loss: 0.045816         Validation Loss: 0.466513
Epoch: 7         Training Loss: 0.042952         Validation Loss: 0.474006
Epoch: 8         Training Loss: 0.042780         Validation Loss: 0.473524
Epoch: 9         Training Loss: 0.041193         Validation Loss: 0.463608
Epoch: 10         Training Loss: 0.040998         Validation Loss: 0.462575
Validation loss decreased (0.463061 --> 0.462575).  Saving model ...
Epoch: 11         Training Loss: 0.041538         Validation Loss: 0.471334
Epoch: 12         Training Loss: 0.039427         Validation Loss: 0.473825
Epoch: 13         Training Loss: 0.040018         Validation Loss: 0.468763
Epoch: 14         Training Loss: 0.039896         Validation Loss: 0.477060
Epoch: 15         Training Loss: 0.037594         Validation Loss: 0.470795
Epoch: 16         Training Loss: 0.036319         Validation Loss: 0.463526
Epoch: 17         Training Loss: 0.035309         Validation Loss: 0.461256
Validation loss decreased (0.462575 --> 0.461256).  Saving model ...
Epoch: 18         Training Loss: 0.034659         Validation Loss: 0.466602
Epoch: 19         Training Loss: 0.035590         Validation Loss: 0.464091
Epoch: 20         Training Loss: 0.033860         Validation Loss: 0.467963
Epoch: 21         Training Loss: 0.035094         Validation Loss: 0.465130
Epoch: 22         Training Loss: 0.032585         Validation Loss: 0.470218
```

```
Epoch: 23         Training Loss: 0.033375         Validation Loss: 0.464225
Epoch: 24         Training Loss: 0.032623         Validation Loss: 0.460103
Validation loss decreased (0.461256 --> 0.460103).  Saving model ...
Epoch: 25         Training Loss: 0.031729         Validation Loss: 0.460704
Epoch: 26         Training Loss: 0.032139         Validation Loss: 0.466804
Epoch: 27         Training Loss: 0.030654         Validation Loss: 0.461972
Epoch: 28         Training Loss: 0.032488         Validation Loss: 0.461988
Epoch: 29         Training Loss: 0.030130         Validation Loss: 0.467273
Epoch: 30         Training Loss: 0.031967         Validation Loss: 0.461604
Epoch: 31         Training Loss: 0.029690         Validation Loss: 0.464949
Epoch: 32         Training Loss: 0.029583         Validation Loss: 0.465034
Epoch: 33         Training Loss: 0.029677         Validation Loss: 0.471237
Epoch: 34         Training Loss: 0.029178         Validation Loss: 0.470556
Epoch: 35         Training Loss: 0.029086         Validation Loss: 0.473892
Epoch: 36         Training Loss: 0.028025         Validation Loss: 0.470112
Epoch: 37         Training Loss: 0.028330         Validation Loss: 0.471677
Epoch: 38         Training Loss: 0.026484         Validation Loss: 0.468820
Epoch: 39         Training Loss: 0.026376         Validation Loss: 0.468450
Epoch: 40         Training Loss: 0.027421         Validation Loss: 0.473695
Epoch: 41         Training Loss: 0.025220         Validation Loss: 0.473905
Epoch: 42         Training Loss: 0.026333         Validation Loss: 0.468383
Epoch: 43         Training Loss: 0.027277         Validation Loss: 0.473397
Epoch: 44         Training Loss: 0.026784         Validation Loss: 0.470728
Epoch: 45         Training Loss: 0.023879         Validation Loss: 0.466043
Epoch: 46         Training Loss: 0.025111         Validation Loss: 0.472101
Epoch: 47         Training Loss: 0.025083         Validation Loss: 0.472508
Epoch: 48         Training Loss: 0.024165         Validation Loss: 0.465119
Epoch: 49         Training Loss: 0.024136         Validation Loss: 0.473288
Epoch: 50         Training Loss: 0.024657         Validation Loss: 0.474809
Epoch: 51         Training Loss: 0.023883         Validation Loss: 0.472117
Epoch: 52         Training Loss: 0.023963         Validation Loss: 0.476374
Epoch: 53         Training Loss: 0.023631         Validation Loss: 0.471976
Epoch: 54         Training Loss: 0.023861         Validation Loss: 0.474686
Epoch: 55         Training Loss: 0.022959         Validation Loss: 0.473814
Epoch: 56         Training Loss: 0.022279         Validation Loss: 0.478200
Epoch: 57         Training Loss: 0.021797         Validation Loss: 0.468825
Epoch: 58         Training Loss: 0.022152         Validation Loss: 0.478918
Epoch: 59         Training Loss: 0.022037         Validation Loss: 0.470872
Epoch: 60         Training Loss: 0.021203         Validation Loss: 0.473166
Epoch: 61         Training Loss: 0.021411         Validation Loss: 0.475494
Epoch: 62         Training Loss: 0.020483         Validation Loss: 0.470029
Epoch: 63         Training Loss: 0.021653         Validation Loss: 0.478640
Epoch: 64         Training Loss: 0.021306         Validation Loss: 0.474579
Epoch: 65         Training Loss: 0.021165         Validation Loss: 0.472056
Epoch: 66         Training Loss: 0.020645         Validation Loss: 0.467473
Epoch: 67         Training Loss: 0.020256         Validation Loss: 0.471857
Epoch: 68         Training Loss: 0.020742         Validation Loss: 0.468695
Epoch: 69         Training Loss: 0.020832         Validation Loss: 0.473682
```

```
Epoch: 70         Training Loss: 0.020289         Validation Loss: 0.468472
Epoch: 71         Training Loss: 0.019761         Validation Loss: 0.467030
Epoch: 72         Training Loss: 0.019826         Validation Loss: 0.471278
Epoch: 73         Training Loss: 0.019536         Validation Loss: 0.465755
Epoch: 74         Training Loss: 0.018913         Validation Loss: 0.472406
Epoch: 75         Training Loss: 0.018993         Validation Loss: 0.462947
Epoch: 76         Training Loss: 0.018453         Validation Loss: 0.470613
Epoch: 77         Training Loss: 0.018279         Validation Loss: 0.466213
Epoch: 78         Training Loss: 0.018502         Validation Loss: 0.471229
Epoch: 79         Training Loss: 0.018746         Validation Loss: 0.471243
Epoch: 80         Training Loss: 0.019141         Validation Loss: 0.467318
Epoch: 81         Training Loss: 0.018655         Validation Loss: 0.469253
Epoch: 82         Training Loss: 0.018614         Validation Loss: 0.472540
Epoch: 83         Training Loss: 0.017051         Validation Loss: 0.475628
Epoch: 84         Training Loss: 0.017579         Validation Loss: 0.472243
Epoch: 85         Training Loss: 0.019195         Validation Loss: 0.466429
Epoch: 86         Training Loss: 0.017705         Validation Loss: 0.470477
Epoch: 87         Training Loss: 0.017896         Validation Loss: 0.473380
Epoch: 88         Training Loss: 0.017020         Validation Loss: 0.480217
Epoch: 89         Training Loss: 0.017000         Validation Loss: 0.477352
Epoch: 90         Training Loss: 0.016935         Validation Loss: 0.477624
Epoch: 91         Training Loss: 0.017109         Validation Loss: 0.477153
Epoch: 92         Training Loss: 0.016254         Validation Loss: 0.472896
Epoch: 93         Training Loss: 0.017090         Validation Loss: 0.478364
Epoch: 94         Training Loss: 0.017183         Validation Loss: 0.480340
Epoch: 95         Training Loss: 0.018025         Validation Loss: 0.474726
Epoch: 96         Training Loss: 0.015349         Validation Loss: 0.473520
Epoch: 97         Training Loss: 0.016989         Validation Loss: 0.477230
Epoch: 98         Training Loss: 0.016524         Validation Loss: 0.474687
Epoch: 99         Training Loss: 0.016236         Validation Loss: 0.476214
Epoch: 100          Training Loss: 0.015955          Validation Loss: 0.467174
```

```
In [48]: model_transfer.load_state_dict(torch.load('model_scratch4.pt'))
         test(loaders_transfer, model_transfer, criterion_transfer, cuda_available)
```

```
Test Loss: 0.566806


Test Accuracy: 84% (708/836)
```

My comments 2: After retraining the model, test accuracy was restore to 84%. It actually, improved slightly from a ratio of 706/836 to 708/836.

1.1.16 (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 [213]:
```
# HELPER function. Gets the workload labels (unused, dont recomend this function)
def get_labels_from_loader(loader, name):
    items  = [items for (_, items) in enumerate(loader[name])]
    lists  = [item[1].tolist() for item in items]
    usval  = sorted(set(sum(lists,[]))) # unique sorted values indexed from 0 onward
    labels = [int(l) for l in usval]
    return labels

loader_class_labels = get_labels_from_loader(loaders_transfer, 'train')
```

In [22]:
```
def get_labels_from_filesystem(img_working_dir):
    dictionary = dict()
    files = os.listdir(img_working_dir)
    for file in files:
        key, val = file.split('.')
        dictionary[int(key)] = val
    return dictionary

class_labels = get_labels_from_filesystem(os.getcwd() + "\\dogImages\\test")
```

In [24]:
```
# HELPER function: given a file path, display the image
def display_img(img_path, name):
    params = {"text.color" : "blue"}
    img = Image.open(img_path)
    plt.imshow(img)
    plt.title(name, pad=10)
    plt.show()
    plt.rcParams.update(params)
```

In [324]:
```
def show_predictions(model, loader, train_on_gpu, labels):
    # obtain one batch of test images
    dataiter = iter(loader)
    images, _ = dataiter.next() # at a later time analyze labels
    images.numpy()
    imgs = images

    # move model inputs to cuda, if GPU available
    #if train_on_gpu:
   #     images = images.cuda()
    if cuda_available:
        images = images.cuda()
        model = model.cuda()


    # get sample outputs
    output = model(images)

    # convert output probabilities to predicted class
```

```
            _, preds_tensor = torch.max(output, 1)

            if train_on_gpu:
                preds = np.squeeze(preds_tensor.cpu().numpy())
            else:
                preds = np.squeeze(preds_tensor.numpy())

            # plot the images in the batch, along with predicted and true labels
            fig = plt.figure(figsize=(25, 4))
            for idx in np.arange(20):
                ax = fig.add_subplot(2, 20/2, idx+1, xticks=[], yticks=[])
                img = imgs[idx]
                img = (img-torch.min(img))/(torch.max(img)-torch.min(img)) #apply min-max no
                plt.imshow(np.transpose(img, (1, 2, 0)))
                k = preds[idx]+1
                ax.set_title("{}".format(labels[k]))
                #ax.set_title("{} ({})".format(classes[preds[idx]], classes[labels[idx]]),
                #           color=("green" if preds[idx]==labels[idx].item() else "red"))
```

In [20]:
```
### 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]
def  Dog_breed_detector(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
    '''
    # load and prepare the model
    model = models.vgg16(pretrained=True)

    # Freeze training for all "features" layers
    for param in model.features.parameters():
        param.requires_grad = False

    c_0_inputs  = model.classifier[0].in_features
    model.classifier = classifier(c_0_inputs)

    model.load_state_dict(torch.load('model_scratch3.pt'))
    if cuda_available:
        model = model.cuda()
    model.eval()
```

```
        img = Image.open(img_path)

        # define image transformations
        transformations = transforms.Compose([
            transforms.Resize(size=224),
            transforms.CenterCrop((224,224)),
            transforms.ToTensor(),
            transforms.Normalize(
                    mean=[0.485, 0.456, 0.406],
                    std=[0.229, 0.224, 0.225])])

        # apply the image transformations
        img = transformations(img).unsqueeze_(0)

        # map to gpu
        if cuda_available:
            img   = img.cuda()

        output = model(img)

        _, pred = torch.max(output, 1)

        pred = np.squeeze(pred.numpy()) if not cuda_available else np.squeeze(pred.cpu().n
        return int(pred) + 1
```

```
In [25]: # Test code for transfer_predict. I did not use the class_labels, but the inverse tab
         img_path  = dog_files[2000]
         cid       = Dog_breed_detector(img_path) # previously named transfer_predict
         labels    = get_labels_from_filesystem(os.getcwd()+ "\\dogImages\\test")
         name      = "Predicted: label " + str(cid) + ", human readable name: " + labels[cid] +
         display_img(img_path, name)
```