dog_app

February 16, 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 [2]: 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.
```

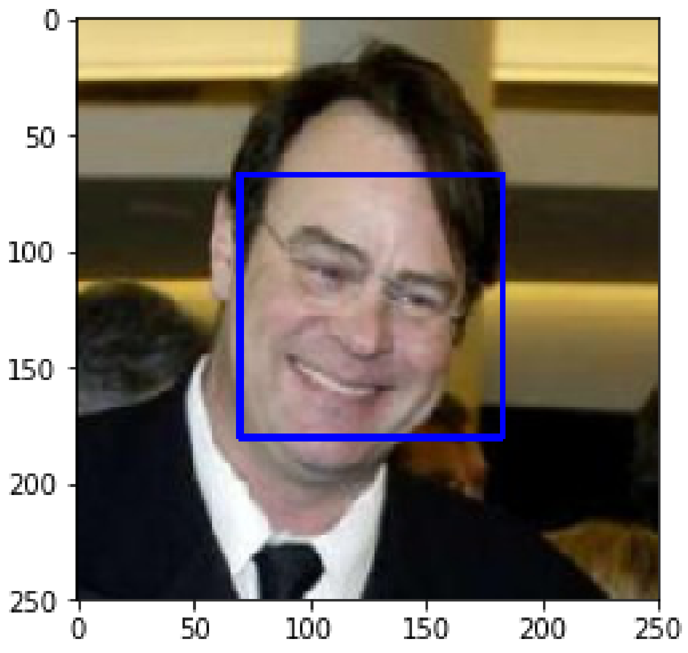
## Step 1: Detect Humans

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),(255,0,0),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 [4]: # returns "True" if face is detected in image stored at img_path
        def face_detector(img_path):
```

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

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

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

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

Answer: Percentage of human faces in human files is 98%. Percentage of dog faces in dog files is 17%

```
In [5]: 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

        human_face_in_human = [face_detector(img) for img in tqdm(human_files_short)]
        human_face_in_dog = [face_detector(img) for img in tqdm(dog_files_short)]

        print("percentage of detected human faces in human files is {}%".format(sum(human_face_i
        print("percentage of detected human faces in dog files is {}%".format(sum(human_face_in_
```

```
100%|| 100/100 [00:03<00:00, 27.27it/s]
100%|| 100/100 [00:39<00:00,  2.55it/s]

percentage of detected human faces in human files is 98%
percentage of detected human faces in dog files is 17%
```

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 anotherface detection algorithm.
        ### Feel free to use as many code cells as needed.
```

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

1.1.3 Obtain Pre-trained VGG-16 Model

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

```
In [6]: import torch
        import torchvision.models as models

        # define VGG16 model
        VGG16 = models.vgg16(pretrained=True)

        # check if CUDA is available
        use_cuda = torch.cuda.is_available()

        # move model to GPU if CUDA is available
        if use_cuda:
            VGG16 = VGG16.cuda()
```

Given an image, this pre-trained VGG-16 model returns a prediction (derived from the 1000 possible categories in ImageNet) for the object that is contained in the image.

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

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

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

```
In [7]: from PIL import Image
        import torchvision.transforms as transforms
        from torch.autograd import Variable

        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.open(img_path)
        data_transforms = transforms.Compose([transforms.Resize(size=(244,244)),
                                             transforms.ToTensor(),
                                             transforms.Normalize([0.485, 0.456, 0.406],
                                                                  [0.229, 0.224, 0.225])])
        img_preprocess = data_transforms(img)
        img_preprocess.unsqueeze_(0)

        image = img_preprocess.cuda()
        net = VGG16(image)

        return torch.max(net,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 [8]: ### returns "True" if a dog is detected in the image stored at img_path
        def dog_detector(img_path):
            ## TODO: Complete the function.
            prediction = VGG16_predict(img_path)
            return ((prediction <= 268) & (prediction >= 151)) # true/false
```

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

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

Answer: Percentage of detected dog face in human files is 1% Percentage of detected dog face in dog files is 100%

```
In [9]: ### TODO: Test the performance of the dog_detector function
        ### on the images in human_files_short and dog_files_short.
        from tqdm import tqdm

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

        dog_face_in_human_files = [dog_detector(img) for img in tqdm(human_files_short)]
        dog_face_in_dog_files = [dog_detector(img) for img in tqdm(dog_files_short)]

        print("Percentage of detected dog face in human files is {}%".format(np.sum(dog_face_in_
        print("Percentage of detected dog face in dog files is {}%".format(np.sum(dog_face_in_do

100%|| 100/100 [00:04<00:00, 24.52it/s]
100%|| 100/100 [00:05<00:00, 18.64it/s]

Percentage of detected dog face in human files is 1%
Percentage of detected dog face in dog files is 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 [ ]: ### (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 [10]: import os
         from torchvision import transforms,datasets

         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 = 20
         num_workers = 2

         data_transforms = transforms.Compose([transforms.Resize(224),
                                               transforms.RandomCrop(224),
                                               transforms.RandomHorizontalFlip(),
                                               transforms.RandomVerticalFlip(),
                                               #transforms.RandomAffine(20, translate=None, sca
                                              # transforms.ColorJitter(brightness=0.4, contrast
                                               transforms.ToTensor(),
                                               transforms.Normalize([0.485, 0.456, 0.406],
                                                                    [0.229, 0.224, 0.225])])
```

```
test_transform = transforms.Compose(
            [transforms.Resize(224),
             transforms.RandomCrop(224),
             #transforms.Grayscale(3),
             transforms.ToTensor(),
             transforms.Normalize([0.485, 0.456, 0.406],
                                  [0.229, 0.224, 0.225])]
        )


data_dir = "/data/dog_images"
train_dir = os.path.join(data_dir, 'train/')
valid_dir = os.path.join(data_dir, 'valid/')
test_dir = os.path.join(data_dir, 'test/')

train_data = datasets.ImageFolder(train_dir, transform=data_transforms)
valid_data = datasets.ImageFolder(valid_dir, transform=data_transforms)
test_data = datasets.ImageFolder(test_dir, transform=test_transform)

train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle =
valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=batch_size, shuffle =
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size, num_workers

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: I have decided to resize the image to 224 as it is the appropriate image size for the CNN architecture. I augmented the dataset by random crop, random horizontal and vertical flip. I thought by changing the color saturations and do some affine transformations might increase the testing data accuracy, but it is not. So, I have limited the data augmentation to those above for the training and validation dataset, then for testing data set, I just augmented it with the random crop. I set number of workers to 2 to reduce the computing times.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [11]: import torch.nn as nn
         import torch.nn.functional as F
```

```
# define the CNN architecture
class Net(nn.Module):
    ### TODO: choose an architecture, and complete the class
    def __init__(self):
        super(Net, self).__init__()
        ## Define layers of a CNN
        self.conv1 = nn.Conv2d(3, 16, 3)
        self.conv2 = nn.Conv2d(16, 32, 3)
        self.conv3 = nn.Conv2d(32, 64, 3)
        self.conv4 = nn.Conv2d(64, 128, 3)
        self.conv5 = nn.Conv2d(128,256, 3)
        self.pool = nn.MaxPool2d(2, 2)

        self.conv_bn1 = nn.BatchNorm2d(224,3)
        self.conv_bn2 = nn.BatchNorm2d(16)
        self.conv_bn3 = nn.BatchNorm2d(32)
        self.conv_bn4 = nn.BatchNorm2d(64)
        self.conv_bn5 = nn.BatchNorm2d(128)
        self.conv_bn6 = nn.BatchNorm2d(256)

        self.fc1 = nn.Linear(5 * 5 * 256, 500)
        self.fc2 = nn.Linear(500, 133)
        self.dropout = nn.Dropout(0.3)

    def forward(self, x):
        ## Define forward behavior
        x = F.relu(self.conv1(x))
        x = self.pool(x)
        x = self.conv_bn2(x)
        x = F.relu(self.conv2(x))
        x = self.pool(x)
        x = self.conv_bn3(x)
        x = F.relu(self.conv3(x))
        x = self.pool(x)
        x = self.conv_bn4(x)
        x = F.relu(self.conv4(x))
        x = self.pool(x)
        x = self.conv_bn5(x)
        x = F.relu(self.conv5(x))
        x = self.pool(x)
        x = self.conv_bn6(x)
        # flatten image input
        x = x.view(-1, 5 * 5 * 256)
        # add dropout layer
        x = self.dropout(x)
        # add 1st hidden layer, with relu activation function
        x = F.relu(self.fc1(x))
        # add dropout layer
```

```
                x = self.dropout(x)
                # add last hidden layer
                x = self.fc2(x)

                return x

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

        # instantiate the CNN
        model_scratch = Net()

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

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

Answer: I use 5 convolutional layers with every layers performance are enhanced by using batch normalization. Dropout is used in the fully connected layer to prevent overfitting while training the model. Same goes to batch normalization. Batch normalization is a way to prevent overfitting for convolutional layers. I started with a small layer of 3 and ends with 256. Then, I just used two fully connected layers with a dropout probability of 0.3. I used relu as the activation functions for all the layers except for the last fully connected layer. The step of building a convolutional layer is, the layer, activating it, pooling it then batch normalized it.

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 [12]: from torch import optim

         criterion_scratch = nn.CrossEntropyLoss()
         learning_rate = 0.0001
         reduction = 1/(10**(1./10))
         optimizer_scratch = optim.Adam(model_scratch.parameters(), lr=learning_rate)
         scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer_scratch, factor=reduct
```

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

             for epoch in range(1, n_epochs+1):
```

```
# initialize variables to monitor training and validation loss
train_loss = 0.0
valid_loss = 0.0

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

```
            #reduce learning rate
            scheduler.step(valid_loss)
            for param_group in optimizer.param_groups:
                  print('learning rate: ' + str(param_group['lr']))

            # print training/validation statistics
            print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
                epoch,
                train_loss,
                valid_loss
                ))

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

```
In [44]: # 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'))
```

```
learning rate: 0.0001
Epoch: 1         Training Loss: 2.294186         Validation Loss: 3.089329
Validation loss decreased (inf --> 3.089329).  Saving model ...
learning rate: 0.0001
Epoch: 2         Training Loss: 2.276147         Validation Loss: 3.059518
Validation loss decreased (3.089329 --> 3.059518).  Saving model ...
learning rate: 0.0001
Epoch: 3         Training Loss: 2.185283         Validation Loss: 3.135017
learning rate: 0.0001
Epoch: 4         Training Loss: 2.129291         Validation Loss: 3.053269
Validation loss decreased (3.059518 --> 3.053269).  Saving model ...
learning rate: 0.0001
Epoch: 5         Training Loss: 2.078873         Validation Loss: 3.065223
learning rate: 0.0001
Epoch: 6         Training Loss: 2.017567         Validation Loss: 3.022861
Validation loss decreased (3.053269 --> 3.022861).  Saving model ...
learning rate: 0.0001
Epoch: 7         Training Loss: 1.989338         Validation Loss: 3.021452
Validation loss decreased (3.022861 --> 3.021452).  Saving model ...
```

```
learning rate: 0.0001
Epoch: 8        Training Loss: 1.911439         Validation Loss: 2.999404
Validation loss decreased (3.021452 --> 2.999404).  Saving model ...
learning rate: 0.0001
Epoch: 9        Training Loss: 1.887812         Validation Loss: 3.049808
learning rate: 0.0001
Epoch: 10       Training Loss: 1.809728         Validation Loss: 2.991976
Validation loss decreased (2.999404 --> 2.991976).  Saving model ...
learning rate: 0.0001
Epoch: 11       Training Loss: 1.773462         Validation Loss: 3.013681
learning rate: 0.0001
Epoch: 12       Training Loss: 1.718708         Validation Loss: 2.955194
Validation loss decreased (2.991976 --> 2.955194).  Saving model ...
learning rate: 0.0001
Epoch: 13       Training Loss: 1.676528         Validation Loss: 2.943834
Validation loss decreased (2.955194 --> 2.943834).  Saving model ...
learning rate: 0.0001
Epoch: 14       Training Loss: 1.640356         Validation Loss: 2.908609
Validation loss decreased (2.943834 --> 2.908609).  Saving model ...
learning rate: 0.0001
Epoch: 15       Training Loss: 1.592916         Validation Loss: 2.911945
learning rate: 7.943282347242815e-05
Epoch: 16       Training Loss: 1.538373         Validation Loss: 2.913129
learning rate: 7.943282347242815e-05
Epoch: 17       Training Loss: 1.500865         Validation Loss: 2.912822
learning rate: 7.943282347242815e-05
Epoch: 18       Training Loss: 1.429383         Validation Loss: 2.981903
learning rate: 6.309573444801932e-05
Epoch: 19       Training Loss: 1.399902         Validation Loss: 2.970947
learning rate: 6.309573444801932e-05
Epoch: 20       Training Loss: 1.356544         Validation Loss: 2.925435
learning rate: 6.309573444801932e-05
Epoch: 21       Training Loss: 1.342978         Validation Loss: 2.975042
learning rate: 5.0118723362727224e-05
Epoch: 22       Training Loss: 1.305417         Validation Loss: 2.911434
learning rate: 5.0118723362727224e-05
Epoch: 23       Training Loss: 1.272479         Validation Loss: 2.940586
learning rate: 5.0118723362727224e-05
Epoch: 24       Training Loss: 1.234848         Validation Loss: 2.939387
learning rate: 3.981071705534972e-05
Epoch: 25       Training Loss: 1.208341         Validation Loss: 2.910188
learning rate: 3.981071705534972e-05
Epoch: 26       Training Loss: 1.151844         Validation Loss: 2.984124
learning rate: 3.981071705534972e-05
Epoch: 27       Training Loss: 1.134362         Validation Loss: 2.917829
learning rate: 3.162277660168379e-05
Epoch: 28       Training Loss: 1.141601         Validation Loss: 2.910304
learning rate: 3.162277660168379e-05
```

```
Epoch: 29         Training Loss: 1.095747        Validation Loss: 2.883897
Validation loss decreased (2.908609 --> 2.883897).  Saving model ...
learning rate: 3.162277660168379e-05
Epoch: 30         Training Loss: 1.082809        Validation Loss: 2.900528
```

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 [14]: 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 [45]: # call test function
         test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 2.815560


Test Accuracy: 31% (262/836)
```

## 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 [15]: ## TODO: Specify data loaders
         import os
         from torchvision import transforms,datasets


         data_dir = "/data/dog_images"
         train_dir = os.path.join(data_dir, 'train/')
         valid_dir = os.path.join(data_dir, 'valid/')
         test_dir = os.path.join(data_dir, 'test/')

         train_data = datasets.ImageFolder(train_dir, transform=data_transforms)
         valid_data = datasets.ImageFolder(valid_dir, transform=data_transforms)
         test_data = datasets.ImageFolder(test_dir, transform=test_transform)

         train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle =
         valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=batch_size, shuffle =
         test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size, num_workers

         loaders_transfer = {
             'train': train_loader,
             'valid': valid_loader,
             'test': test_loader
         }
```

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

         ## TODO: Specify model architecture

         model_transfer = models.densenet161(pretrained=True)
```

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

for param in model.parameters():
    param.requires_grad = False

from collections import OrderedDict
fc = nn.Sequential(OrderedDict([
                          ('fc1', nn.Linear(2208, 1104)),
                          ('relu', nn.ReLU()),
                          ('dropout',nn.Dropout(0.5)),
                          ('fc2', nn.Linear(1104, 133)),
                          ('output', nn.LogSoftmax(dim=1))
                          ]))

model_transfer.classifier = fc
```

```
/opt/conda/lib/python3.6/site-packages/torchvision-0.2.1-py3.6.egg/torchvision/models/densenet.p


DenseNet(
  (features): Sequential(
    (conv0): Conv2d(3, 96, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
    (norm0): BatchNorm2d(96, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu0): ReLU(inplace)
    (pool0): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
    (denseblock1): _DenseBlock(
      (denselayer1): _DenseLayer(
        (norm1): BatchNorm2d(96, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu1): ReLU(inplace)
        (conv1): Conv2d(96, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        (relu2): ReLU(inplace)
        (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      )
      (denselayer2): _DenseLayer(
        (norm1): BatchNorm2d(144, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        (relu1): ReLU(inplace)
        (conv1): Conv2d(144, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        (relu2): ReLU(inplace)
        (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      )
      (denselayer3): _DenseLayer(
        (norm1): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        (relu1): ReLU(inplace)
        (conv1): Conv2d(192, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer4): _DenseLayer(
      (norm1): BatchNorm2d(240, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(240, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer5): _DenseLayer(
      (norm1): BatchNorm2d(288, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(288, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer6): _DenseLayer(
      (norm1): BatchNorm2d(336, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(336, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
  )
  (transition1): _Transition(
    (norm): BatchNorm2d(384, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
    (conv): Conv2d(384, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (pool): AvgPool2d(kernel_size=2, stride=2, padding=0)
  )
  (denseblock2): _DenseBlock(
    (denselayer1): _DenseLayer(
      (norm1): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(192, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer2): _DenseLayer(
      (norm1): BatchNorm2d(240, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(240, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer3): _DenseLayer(
      (norm1): BatchNorm2d(288, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(288, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer4): _DenseLayer(
      (norm1): BatchNorm2d(336, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(336, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer5): _DenseLayer(
      (norm1): BatchNorm2d(384, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(384, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer6): _DenseLayer(
      (norm1): BatchNorm2d(432, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(432, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer7): _DenseLayer(
      (norm1): BatchNorm2d(480, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(480, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer8): _DenseLayer(
      (norm1): BatchNorm2d(528, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(528, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer9): _DenseLayer(
      (norm1): BatchNorm2d(576, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(576, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer10): _DenseLayer(
      (norm1): BatchNorm2d(624, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(624, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer11): _DenseLayer(
      (norm1): BatchNorm2d(672, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(672, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer12): _DenseLayer(
      (norm1): BatchNorm2d(720, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(720, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
  )
  (transition2): _Transition(
    (norm): BatchNorm2d(768, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
    (conv): Conv2d(768, 384, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (pool): AvgPool2d(kernel_size=2, stride=2, padding=0)
  )
  (denseblock3): _DenseBlock(
    (denselayer1): _DenseLayer(
      (norm1): BatchNorm2d(384, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(384, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer2): _DenseLayer(
      (norm1): BatchNorm2d(432, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(432, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer3): _DenseLayer(
      (norm1): BatchNorm2d(480, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(480, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer4): _DenseLayer(
      (norm1): BatchNorm2d(528, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(528, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer5): _DenseLayer(
      (norm1): BatchNorm2d(576, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(576, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer6): _DenseLayer(
      (norm1): BatchNorm2d(624, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(624, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer7): _DenseLayer(
      (norm1): BatchNorm2d(672, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(672, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer8): _DenseLayer(
      (norm1): BatchNorm2d(720, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(720, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer9): _DenseLayer(
      (norm1): BatchNorm2d(768, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(768, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer10): _DenseLayer(
      (norm1): BatchNorm2d(816, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(816, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer11): _DenseLayer(
      (norm1): BatchNorm2d(864, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(864, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer12): _DenseLayer(
      (norm1): BatchNorm2d(912, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(912, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer13): _DenseLayer(
      (norm1): BatchNorm2d(960, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(960, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
    (relu2): ReLU(inplace)
    (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  )
  (denselayer14): _DenseLayer(
    (norm1): BatchNorm2d(1008, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
    (relu1): ReLU(inplace)
    (conv1): Conv2d(1008, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
    (relu2): ReLU(inplace)
    (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  )
  (denselayer15): _DenseLayer(
    (norm1): BatchNorm2d(1056, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
    (relu1): ReLU(inplace)
    (conv1): Conv2d(1056, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
    (relu2): ReLU(inplace)
    (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  )
  (denselayer16): _DenseLayer(
    (norm1): BatchNorm2d(1104, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
    (relu1): ReLU(inplace)
    (conv1): Conv2d(1104, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
    (relu2): ReLU(inplace)
    (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  )
  (denselayer17): _DenseLayer(
    (norm1): BatchNorm2d(1152, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
    (relu1): ReLU(inplace)
    (conv1): Conv2d(1152, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
    (relu2): ReLU(inplace)
    (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  )
  (denselayer18): _DenseLayer(
    (norm1): BatchNorm2d(1200, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
    (relu1): ReLU(inplace)
    (conv1): Conv2d(1200, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
    (relu2): ReLU(inplace)
    (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  )
  (denselayer19): _DenseLayer(
    (norm1): BatchNorm2d(1248, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
    (relu1): ReLU(inplace)
    (conv1): Conv2d(1248, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
```

```
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer20): _DenseLayer(
      (norm1): BatchNorm2d(1296, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
      (relu1): ReLU(inplace)
      (conv1): Conv2d(1296, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer21): _DenseLayer(
      (norm1): BatchNorm2d(1344, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
      (relu1): ReLU(inplace)
      (conv1): Conv2d(1344, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer22): _DenseLayer(
      (norm1): BatchNorm2d(1392, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
      (relu1): ReLU(inplace)
      (conv1): Conv2d(1392, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer23): _DenseLayer(
      (norm1): BatchNorm2d(1440, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
      (relu1): ReLU(inplace)
      (conv1): Conv2d(1440, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer24): _DenseLayer(
      (norm1): BatchNorm2d(1488, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
      (relu1): ReLU(inplace)
      (conv1): Conv2d(1488, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer25): _DenseLayer(
      (norm1): BatchNorm2d(1536, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
      (relu1): ReLU(inplace)
      (conv1): Conv2d(1536, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer26): _DenseLayer(
      (norm1): BatchNorm2d(1584, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
      (relu1): ReLU(inplace)
      (conv1): Conv2d(1584, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer27): _DenseLayer(
      (norm1): BatchNorm2d(1632, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
      (relu1): ReLU(inplace)
      (conv1): Conv2d(1632, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer28): _DenseLayer(
      (norm1): BatchNorm2d(1680, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
      (relu1): ReLU(inplace)
      (conv1): Conv2d(1680, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer29): _DenseLayer(
      (norm1): BatchNorm2d(1728, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
      (relu1): ReLU(inplace)
      (conv1): Conv2d(1728, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer30): _DenseLayer(
      (norm1): BatchNorm2d(1776, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
      (relu1): ReLU(inplace)
      (conv1): Conv2d(1776, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer31): _DenseLayer(
      (norm1): BatchNorm2d(1824, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
      (relu1): ReLU(inplace)
      (conv1): Conv2d(1824, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
```

```
        (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        (relu2): ReLU(inplace)
        (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      )
      (denselayer32): _DenseLayer(
        (norm1): BatchNorm2d(1872, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
        (relu1): ReLU(inplace)
        (conv1): Conv2d(1872, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        (relu2): ReLU(inplace)
        (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      )
      (denselayer33): _DenseLayer(
        (norm1): BatchNorm2d(1920, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
        (relu1): ReLU(inplace)
        (conv1): Conv2d(1920, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        (relu2): ReLU(inplace)
        (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      )
      (denselayer34): _DenseLayer(
        (norm1): BatchNorm2d(1968, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
        (relu1): ReLU(inplace)
        (conv1): Conv2d(1968, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        (relu2): ReLU(inplace)
        (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      )
      (denselayer35): _DenseLayer(
        (norm1): BatchNorm2d(2016, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
        (relu1): ReLU(inplace)
        (conv1): Conv2d(2016, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        (relu2): ReLU(inplace)
        (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      )
      (denselayer36): _DenseLayer(
        (norm1): BatchNorm2d(2064, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
        (relu1): ReLU(inplace)
        (conv1): Conv2d(2064, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        (relu2): ReLU(inplace)
        (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      )
    )
    (transition3): _Transition(
      (norm): BatchNorm2d(2112, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (conv): Conv2d(2112, 1056, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (pool): AvgPool2d(kernel_size=2, stride=2, padding=0)
    )
    (denseblock4): _DenseBlock(
      (denselayer1): _DenseLayer(
        (norm1): BatchNorm2d(1056, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
        (relu1): ReLU(inplace)
        (conv1): Conv2d(1056, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        (relu2): ReLU(inplace)
        (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      )
      (denselayer2): _DenseLayer(
        (norm1): BatchNorm2d(1104, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
        (relu1): ReLU(inplace)
        (conv1): Conv2d(1104, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        (relu2): ReLU(inplace)
        (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      )
      (denselayer3): _DenseLayer(
        (norm1): BatchNorm2d(1152, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
        (relu1): ReLU(inplace)
        (conv1): Conv2d(1152, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        (relu2): ReLU(inplace)
        (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      )
      (denselayer4): _DenseLayer(
        (norm1): BatchNorm2d(1200, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
        (relu1): ReLU(inplace)
        (conv1): Conv2d(1200, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        (relu2): ReLU(inplace)
        (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      )
      (denselayer5): _DenseLayer(
        (norm1): BatchNorm2d(1248, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
        (relu1): ReLU(inplace)
        (conv1): Conv2d(1248, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        (relu2): ReLU(inplace)
        (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      )
      (denselayer6): _DenseLayer(
        (norm1): BatchNorm2d(1296, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
        (relu1): ReLU(inplace)
        (conv1): Conv2d(1296, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
```

```
          (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
          (relu2): ReLU(inplace)
          (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        )
        (denselayer7): _DenseLayer(
          (norm1): BatchNorm2d(1344, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
          (relu1): ReLU(inplace)
          (conv1): Conv2d(1344, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
          (relu2): ReLU(inplace)
          (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        )
        (denselayer8): _DenseLayer(
          (norm1): BatchNorm2d(1392, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
          (relu1): ReLU(inplace)
          (conv1): Conv2d(1392, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
          (relu2): ReLU(inplace)
          (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        )
        (denselayer9): _DenseLayer(
          (norm1): BatchNorm2d(1440, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
          (relu1): ReLU(inplace)
          (conv1): Conv2d(1440, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
          (relu2): ReLU(inplace)
          (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        )
        (denselayer10): _DenseLayer(
          (norm1): BatchNorm2d(1488, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
          (relu1): ReLU(inplace)
          (conv1): Conv2d(1488, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
          (relu2): ReLU(inplace)
          (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        )
        (denselayer11): _DenseLayer(
          (norm1): BatchNorm2d(1536, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
          (relu1): ReLU(inplace)
          (conv1): Conv2d(1536, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
          (relu2): ReLU(inplace)
          (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        )
        (denselayer12): _DenseLayer(
          (norm1): BatchNorm2d(1584, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
          (relu1): ReLU(inplace)
          (conv1): Conv2d(1584, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
```

```
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer13): _DenseLayer(
      (norm1): BatchNorm2d(1632, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
      (relu1): ReLU(inplace)
      (conv1): Conv2d(1632, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer14): _DenseLayer(
      (norm1): BatchNorm2d(1680, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
      (relu1): ReLU(inplace)
      (conv1): Conv2d(1680, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer15): _DenseLayer(
      (norm1): BatchNorm2d(1728, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
      (relu1): ReLU(inplace)
      (conv1): Conv2d(1728, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer16): _DenseLayer(
      (norm1): BatchNorm2d(1776, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
      (relu1): ReLU(inplace)
      (conv1): Conv2d(1776, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer17): _DenseLayer(
      (norm1): BatchNorm2d(1824, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
      (relu1): ReLU(inplace)
      (conv1): Conv2d(1824, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer18): _DenseLayer(
      (norm1): BatchNorm2d(1872, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
      (relu1): ReLU(inplace)
      (conv1): Conv2d(1872, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
```

```
  (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
  (relu2): ReLU(inplace)
  (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
)
(denselayer19): _DenseLayer(
  (norm1): BatchNorm2d(1920, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
  (relu1): ReLU(inplace)
  (conv1): Conv2d(1920, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
  (relu2): ReLU(inplace)
  (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
)
(denselayer20): _DenseLayer(
  (norm1): BatchNorm2d(1968, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
  (relu1): ReLU(inplace)
  (conv1): Conv2d(1968, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
  (relu2): ReLU(inplace)
  (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
)
(denselayer21): _DenseLayer(
  (norm1): BatchNorm2d(2016, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
  (relu1): ReLU(inplace)
  (conv1): Conv2d(2016, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
  (relu2): ReLU(inplace)
  (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
)
(denselayer22): _DenseLayer(
  (norm1): BatchNorm2d(2064, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
  (relu1): ReLU(inplace)
  (conv1): Conv2d(2064, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
  (relu2): ReLU(inplace)
  (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
)
(denselayer23): _DenseLayer(
  (norm1): BatchNorm2d(2112, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
  (relu1): ReLU(inplace)
  (conv1): Conv2d(2112, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
  (relu2): ReLU(inplace)
  (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
)
(denselayer24): _DenseLayer(
  (norm1): BatchNorm2d(2160, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
  (relu1): ReLU(inplace)
  (conv1): Conv2d(2160, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
```

```
          (norm2): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
          (relu2): ReLU(inplace)
          (conv2): Conv2d(192, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        )
      )
      (norm5): BatchNorm2d(2208, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (classifier): Linear(in_features=2208, out_features=1000, bias=True)
  )
```

```
        ---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call last)

        <ipython-input-16-fc1635c5e5e7> in <module>()
          9 print(model_transfer)
         10
    ---> 11 for param in model.parameters():
         12     param.requires_grad = False
         13


        NameError: name 'model' is not defined
```

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

Answer: I have used the exact same transfer learning process when submitting for the flower classification project for last pytorch challenge. While experimenting on those flowers with 102 classes, I found that the last fully connected layer for their classification should be minimize to 2 and the dropout should not be more than 0.5. The dropout is used to discard unimportant nodes, so the higher the dropout, the higher the probability of important nodes are being discarded. So, I would like to test the same model for this type of data and see the results. I have also used the same criterion, optimizer, and data transfroms with my own model and would like to see how the results goes.

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 [17]: from torch import optim

         criterion_transfer = nn.CrossEntropyLoss()
         learning_rate = 0.0001
         reduction = 1/(10**(1./10))
```

```
        optimizer_transfer = optim.Adam(model_transfer.parameters(), lr=learning_rate)
        scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer_transfer, factor=reduc
```

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 [18]: # train the model
         model_transfer = train(20, loaders_transfer, model_transfer, optimizer_transfer,
                                criterion_transfer, use_cuda, 'model_transfer.pt')

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

```
learning rate: 0.0001
Epoch: 1         Training Loss: 3.240320         Validation Loss: 1.252229
Validation loss decreased (inf --> 1.252229).  Saving model ...
learning rate: 0.0001
Epoch: 2         Training Loss: 0.932457         Validation Loss: 0.853267
Validation loss decreased (1.252229 --> 0.853267).  Saving model ...
learning rate: 0.0001
Epoch: 3         Training Loss: 0.540245         Validation Loss: 0.691326
Validation loss decreased (0.853267 --> 0.691326).  Saving model ...
learning rate: 0.0001
Epoch: 4         Training Loss: 0.364930         Validation Loss: 0.669626
Validation loss decreased (0.691326 --> 0.669626).  Saving model ...
learning rate: 0.0001
Epoch: 5         Training Loss: 0.262106         Validation Loss: 0.632950
Validation loss decreased (0.669626 --> 0.632950).  Saving model ...
learning rate: 0.0001
Epoch: 6         Training Loss: 0.205965         Validation Loss: 0.677519
learning rate: 0.0001
Epoch: 7         Training Loss: 0.167186         Validation Loss: 0.632634
Validation loss decreased (0.632950 --> 0.632634).  Saving model ...
learning rate: 0.0001
Epoch: 8         Training Loss: 0.159708         Validation Loss: 0.659448
learning rate: 7.943282347242815e-05
Epoch: 9         Training Loss: 0.122318         Validation Loss: 0.814138
learning rate: 7.943282347242815e-05
Epoch: 10         Training Loss: 0.086469         Validation Loss: 0.563190
Validation loss decreased (0.632634 --> 0.563190).  Saving model ...
learning rate: 7.943282347242815e-05
Epoch: 11         Training Loss: 0.059396         Validation Loss: 0.666236
learning rate: 6.309573444801932e-05
Epoch: 12         Training Loss: 0.057601         Validation Loss: 0.639387
learning rate: 6.309573444801932e-05
Epoch: 13         Training Loss: 0.046728         Validation Loss: 0.590049
```

```
learning rate: 6.309573444801932e-05
Epoch: 14         Training Loss: 0.035784         Validation Loss: 0.587298
learning rate: 5.0118723362727224e-05
Epoch: 15         Training Loss: 0.045459         Validation Loss: 0.700204
learning rate: 5.0118723362727224e-05
Epoch: 16         Training Loss: 0.044054         Validation Loss: 0.549058
Validation loss decreased (0.563190 --> 0.549058).  Saving model ...
learning rate: 5.0118723362727224e-05
Epoch: 17         Training Loss: 0.026989         Validation Loss: 0.590972
learning rate: 3.981071705534972e-05
Epoch: 18         Training Loss: 0.025702         Validation Loss: 0.559347
learning rate: 3.981071705534972e-05
Epoch: 19         Training Loss: 0.021097         Validation Loss: 0.524101
Validation loss decreased (0.549058 --> 0.524101).  Saving model ...
learning rate: 3.981071705534972e-05
Epoch: 20         Training Loss: 0.021157         Validation Loss: 0.571843
```

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

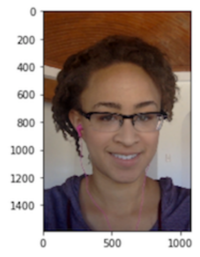
```
Test Loss: 0.613859


Test Accuracy: 85% (716/836)
```

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

         data_transfer = {'train': train_data,
                          'valid': valid_data,
                          'test': test_data}

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

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

```
hello, human!
```



```
You look like a ...
Chinese_shar-pei
```

Sample Human Output

```
    data_transforms = transforms.Compose([transforms.Resize(size=(244,244)),
                                          transforms.ToTensor(),
                                          transforms.Normalize([0.485, 0.456, 0.406],
                                                               [0.229, 0.224, 0.225])])
    img_preprocess = data_transforms(img)
    img_preprocess.unsqueeze_(0)

    image = img_preprocess.cuda()
    output = model_transfer(image)

    _, prediction = torch.max(output.data,1)
    predict_breed = class_names[prediction-1]

    return predict_breed
```

## Step 5: Write your Algorithm

Write an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then, - if a **dog** is detected in the image, return the predicted breed. - if a **human** is detected in the image, return the resembling dog breed. - if **neither** is detected in the image, provide output that indicates an error.

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

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

1.1.18 (IMPLEMENTATION) Write your Algorithm

```
In [40]: ### TODO: Write your algorithm.
         ### Feel free to use as many code cells as needed.

         def run_app(img_path):
             ## handle cases for a human face, dog, and neither
             if face_detector(img_path) == True or dog_detector(img_path) == True:
```

```
            print ("Your predicted breed is")
            return predict_breed_transfer(img_path)
        else:
            return "no dogs or humans in the image"
```

## 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.

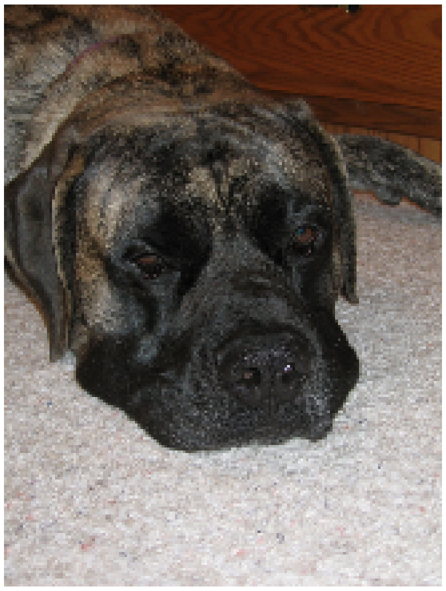
Question 6: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

Answer: Yes, it is better. Three improvements that we can do is, 1) increasing the training epoch as I just set it to 20. But even with 20, I can achiveve 85% accuracy. So, it is a good model I think. 2)Try different model like ResNet etc. 3) Try different optimizer and criterion. Modified the convolutional layers of the model might increase the accuracy.

```
In [44]: ## TODO: Execute your algorithm from Step 6 on
         ## at least 6 images on your computer.
         ## Feel free to use as many code cells as needed.


         ## suggested code, below
         fig_index = 0
         for file in np.hstack((human_files[5:8], dog_files[10:13])):
             predict_breed = run_app(file)
             fig_index += 1

             img=np.asarray(Image.open(file))

             fig = plt.figure(figsize=(16,4))
             ax = fig.add_subplot(1,2,1)
             ax.imshow(img)
             plt.axis('off')

             print('{}. '.format(predict_breed))
```

```
Your predicted breed is
American staffordshire terrier.
Your predicted breed is
Doberman pinscher.
Your predicted breed is
Doberman pinscher.
```

```
Your predicted breed is
Manchester terrier.
Your predicted breed is
Manchester terrier.
Your predicted breed is
Manchester terrier.
```



```
In [ ]:
```