

# dog\_app

April 1, 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 [1]: import numpy as np
       from glob import glob

       # load filenames for human and dog images
       human_files = np.array(glob("/data/lfw/*/*"))
       dog_files = np.array(glob("/data/dog_images/*/*/*"))

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

There are 13233 total human images.

There are 8351 total dog images.

### ## Step 1: Detect Humans

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

OpenCV provides many pre-trained face detectors, stored as XML files on [github](#). We have downloaded one of these detectors and stored it in the `haarcascades` directory. In the next code cell, we demonstrate how to use this detector to find human faces in a sample image.

```
In [2]: import cv2
       import matplotlib.pyplot as plt
       %matplotlib inline

       # extract pre-trained face detector
       face_cascade = cv2.CascadeClassifier('haarcascades/haarcascade_frontalface_alt.xml')

       # load color (BGR) image
       img = cv2.imread(human_files[0])
       # convert BGR image to grayscale
       gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

       # find faces in image
       faces = face_cascade.detectMultiScale(gray)

       # print number of faces detected in the image
       print('Number of faces detected:', len(faces))

       # get bounding box for each detected face
       for (x,y,w,h) in faces:
           # add bounding box to color image
           cv2.rectangle(img,(x,y),(x+w,y+h),(255,0,0),2)
```

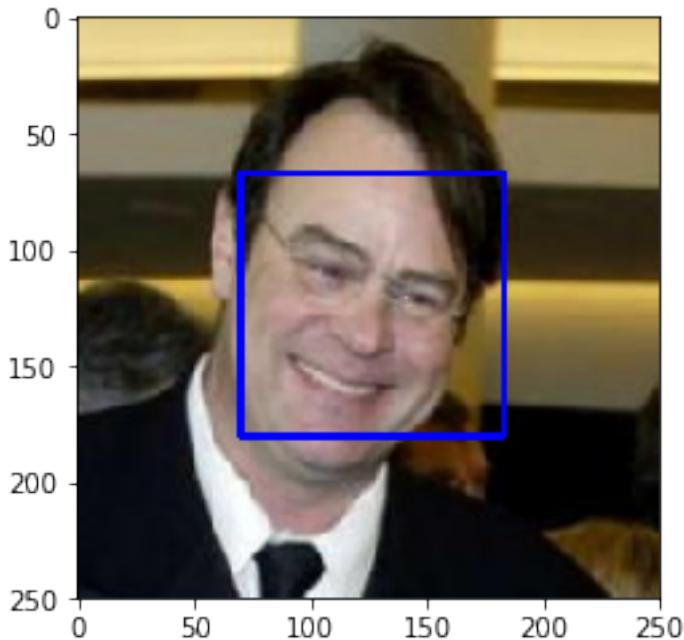
```

# convert BGR image to RGB for plotting
cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

# display the image, along with bounding box
plt.imshow(cv_rgb)
plt.show()

```

Number of faces detected: 1



Before using any of the face detectors, it is standard procedure to convert the images to grayscale. The `detectMultiScale` function executes the classifier stored in `face_cascade` and takes the grayscale image as a parameter.

In the above code, `faces` is a numpy array of detected faces, where each row corresponds to a detected face. Each detected face is a 1D array with four entries that specifies the bounding box of the detected face. The first two entries in the array (extracted in the above code as `x` and `y`) specify the horizontal and vertical positions of the top left corner of the bounding box. The last two entries in the array (extracted here as `w` and `h`) specify the width and height of the box.

### 1.1.1 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 [3]: # returns "True" if face is detected in image stored at img_path
def face_detector(img_path):

```

```

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

```

### 1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

**Question 1:** Use the code cell below to test the performance of the `face_detector` function.

- What percentage of the first 100 images in `human_files` have a detected human face?
- What percentage of the first 100 images in `dog_files` have a detected human face?

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

**Answer:** (You can print out your results and/or write your percentages in this cell)

In [4]: `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.
humans = 0
for human_face in human_files_short:
    if face_detector(human_face):
        humans += 1
percentage = (humans/len(human_files_short)) * 100
print ('Percentage of human face in human_files:{}%'.format(percentage))

dogs = 0
for dog_face in dog_files_short:
    if face_detector(dog_face):
        dogs += 1
percentage = (dogs/len(dog_files_short)) * 100
print ('Percentage of human face in dog_files:{}%'.format(percentage))

Percentage of human face in human_files:98.0%
Percentage of human face in dog_files:17.0%

```

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 [5]: ### (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 [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()
```

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg16-397923af.pth
100%|| 553433881/553433881 [00:31<00:00, 17849931.50it/s]
```

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

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

In the next code cell, you will write a function that accepts a path to an image (such as 'dogImages/train/001.Affenpinscher/Affenpinscher\_00001.jpg') as input and returns the index corresponding to the ImageNet class that is predicted by the pre-trained VGG-16 model. The output should always be an integer between 0 and 999, inclusive.

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

```
In [9]: from PIL import Image
import torchvision.transforms as transforms
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
    image = Image.open(img_path)

    # Define transformations for the image, should (note that imagenet models are trained
    transformation = transforms.Compose([
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485,0.456,0.406],std=[0.229,0.224,0.225])
    ])

    # Preprocess the image
    image_tensor = transformation(image).float()

    # Add an extra batch dimension since pytorch treats all images as batches
    image_tensor = image_tensor.unsqueeze_(0)
    image = Variable(image_tensor)
    # Turn the input into a Variable
    if use_cuda:
        image = Variable(image_tensor.cuda())

    VGG16.eval()
    # Predict the class of the image
    output = VGG16(image)
    output = output.cpu()
    index = output.data.numpy().argmax()

    return index

```

In [10]: #predict dog using ImageNet class  
VGG16\_predict(dog\_files\_short[0])

Out[10]: 225

### 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 [11]: ### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    ## TODO: Complete the function.

    idx = VGG16_predict(img_path)

    return idx >= 151 and idx <=268

print(dog_detector(dog_files[0]))
print(dog_detector(human_files_short[0]))
```

```
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:**

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

humans = 0
for human_face in human_files_short:
    if dog_detector(human_face):
        humans += 1
percentage = (humans/len (human_files_short)) * 100
print ('Percentage of dog face in human_files:{}%'.format (percentage))

dogs = 0
for dog_face in dog_files_short:
    if dog_detector(dog_face):
        dogs += 1
percentage = (dogs/len (dog_files_short)) * 100
print ('Percentage of dog face in dog_files:{}%'.format (percentage))
```

```
Percentage of dog face in human_files:1.0%
Percentage of dog face in dog_files:91.0%
```

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

```
In [13]: ### (Optional)
### TODO: Report the performance of another pre-trained network.
### Feel free to use as many code cells as needed.
```

---

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

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

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

---

Brittany	Welsh Springer Spaniel
----------	------------------------

---

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

---

Curly-Coated Retriever	American Water Spaniel
------------------------	------------------------

---

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

---

Yellow Labrador	Chocolate Labrador
-----------------	--------------------

---

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

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

### 1.1.7 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

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

```
In [14]: import os
        from torchvision import datasets
        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
        data_dir = '/data/dog_images'
        train_dir = data_dir + '/train'
        valid_dir = data_dir + '/valid'
        test_dir = data_dir + '/test'

        # number of subprocesses to use for data loading
        num_workers = 0
        # how many samples per batch to load
        batch_size = 20

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

        preprocess = transforms.Compose([
            transforms.RandomResizedCrop(224),
            transforms.RandomHorizontalFlip(),
            transforms.RandomRotation(10),
            transforms.ToTensor(),
            normalize
        ])

        transform = transforms.Compose([
            transforms.Resize((224,224)),
            transforms.ToTensor(),
            normalize
        ])

        train_data = datasets.ImageFolder(train_dir, transform = preprocess)

        valid_data = datasets.ImageFolder(valid_dir, transform = transform)

        test_data = datasets.ImageFolder(test_dir, transform = transform)

        train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,shuffle=True)
```

```

        num_workers=num_workers)
valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=batch_size,shuffle=True,
                                           num_workers=num_workers)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size,shuffle=True,
                                         num_workers=num_workers)

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

In [15]: print("Size of training set is: "+str(len(train_data)))
print("Size of validation set is: "+str(len(valid_data)))
print("Size of testing set is: "+str(len(test_data)))

Size of training set is: 6680
Size of validation set is: 835
Size of testing set is: 836

In [16]: class_names = train_data.classes
num_classes = len(class_names)

print("Number of classes:", num_classes)
print("\nClass names: \n\n", class_names)

Number of classes: 133

Class names:

['001.Affenpinscher', '002.Afghan_hound', '003.Airedale_terrier', '004.Akita', '005.Alaskan_mal

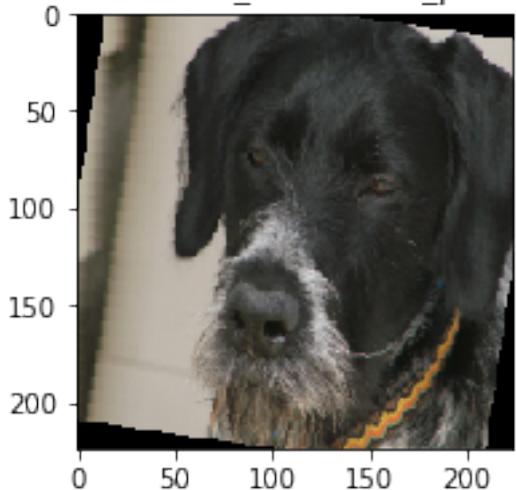
In [17]: # import torchvision
          # Get a batch of training data
inputs, classes = next( iter(loaders_scratch['train']) )

for image, label in zip(inputs, classes):
    image = image.to("cpu").clone().detach()
    image = image.numpy().squeeze()
    image = image.transpose(1,2,0)
    # normalize image
    image = image * np.array((0.229, 0.224, 0.225)) + np.array((0.485, 0.456, 0.406))
    image = image.clip(0, 1)

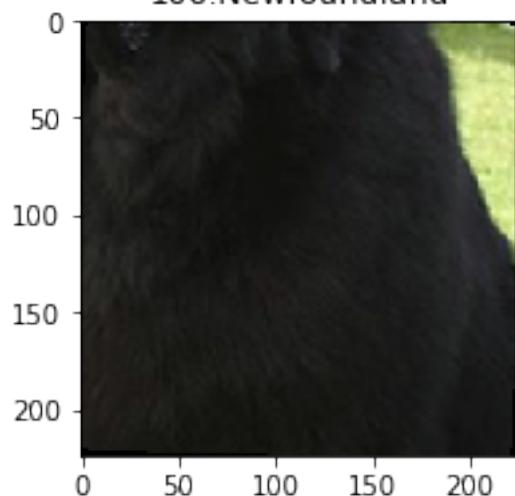
    fig = plt.figure(figsize=(12,3))
    plt.imshow(image)
    plt.title(class_names[label])

```

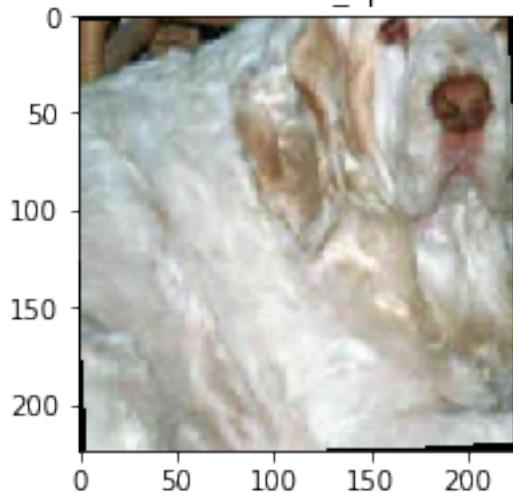
073.German\_wirehaired\_pointer



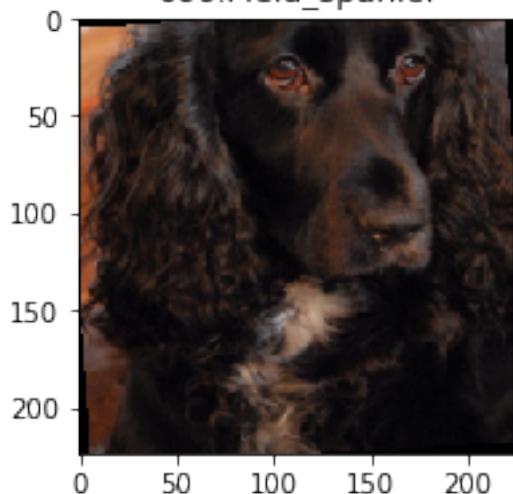
106.Newfoundland



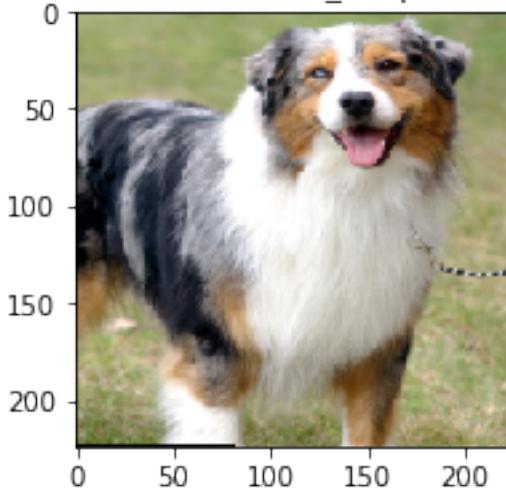
052.Clumber\_siel



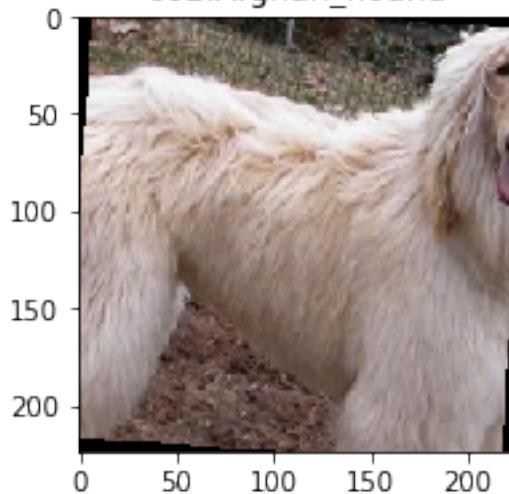
066.Field\_siel



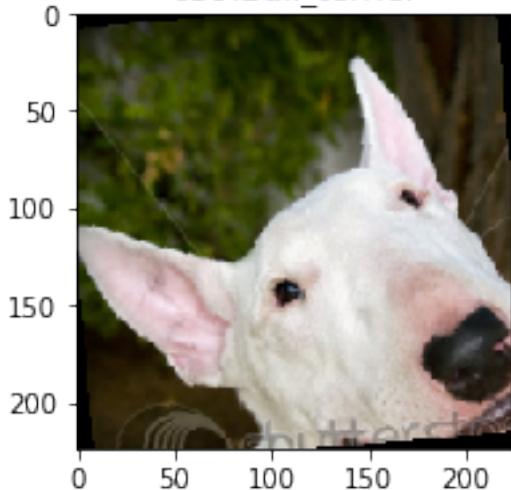
012.Australian\_shepherd



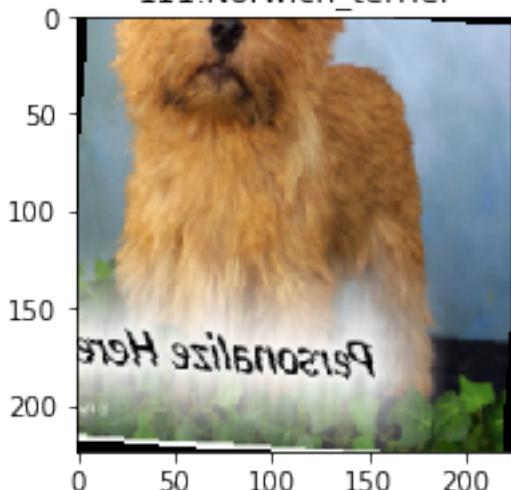
002.Afghan\_hound



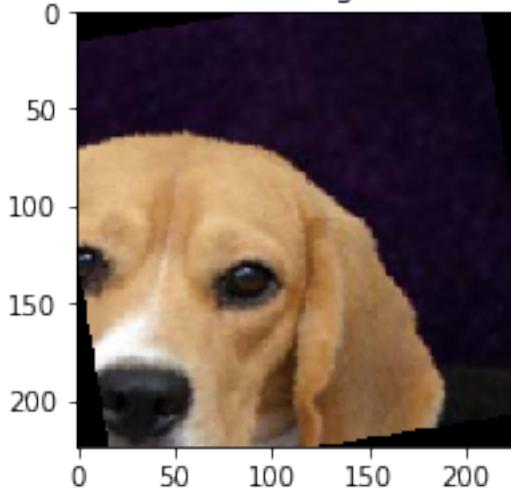
039.Bull\_terrier



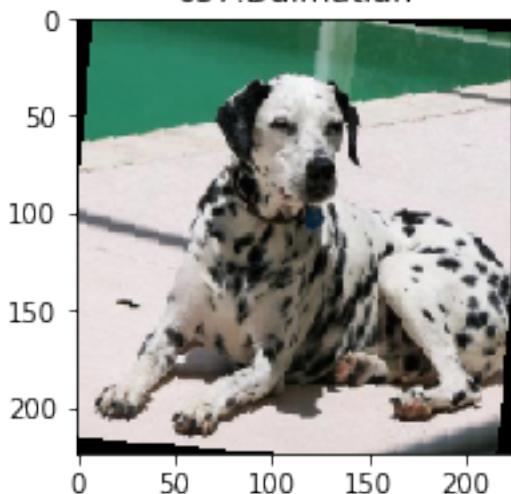
111.Norwich\_terrier



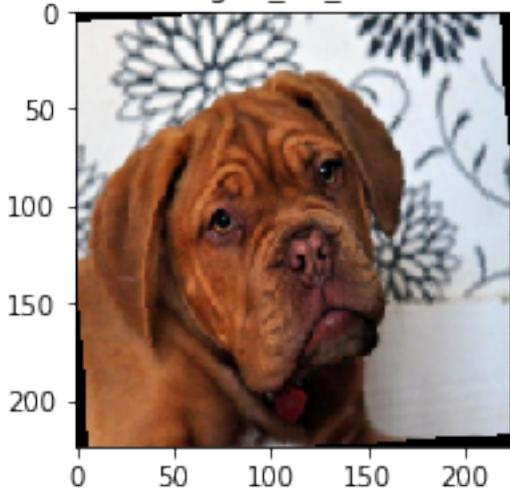
016.Beagle



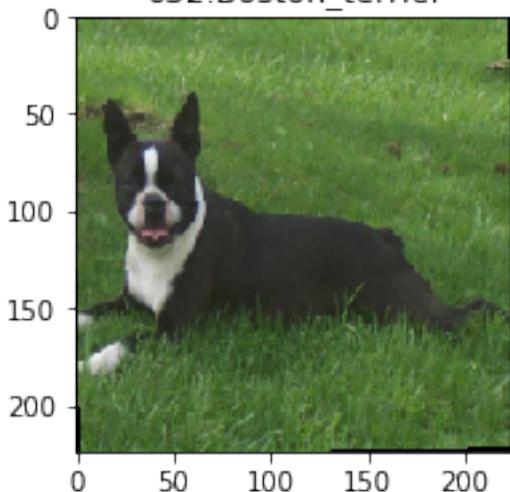
057.Dalmatian



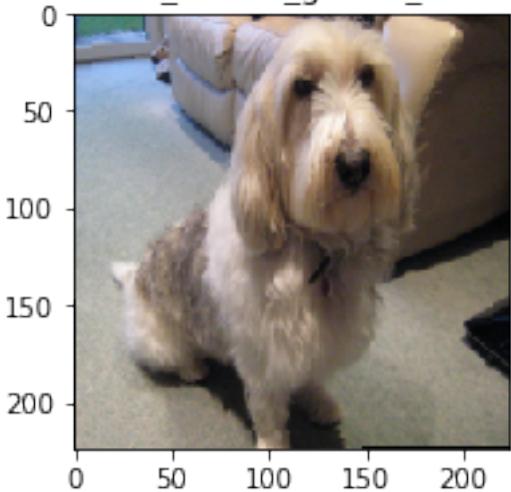
060.Dogue\_de\_bordeaux



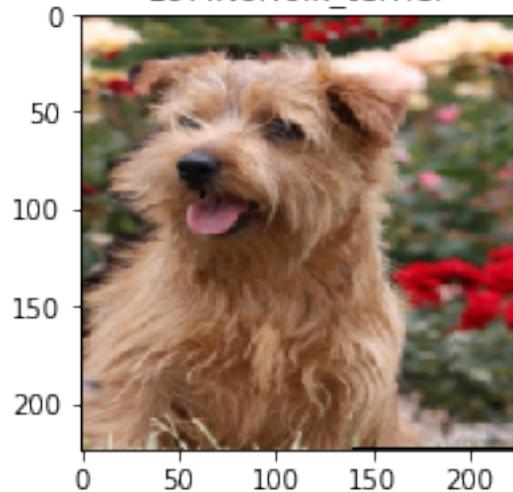
032.Boston\_terrier



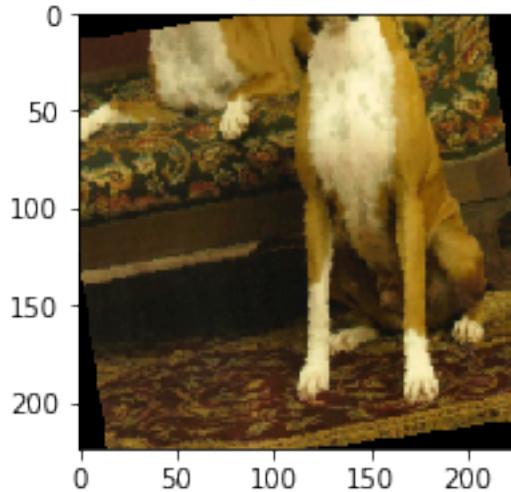
119.Petit\_basset\_griffon\_vendeen



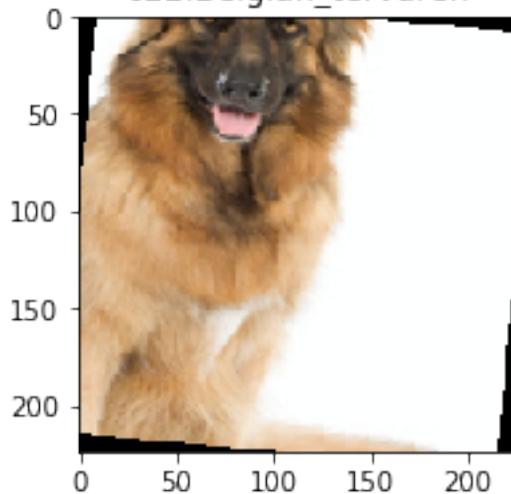
107.Norfolk\_terrier



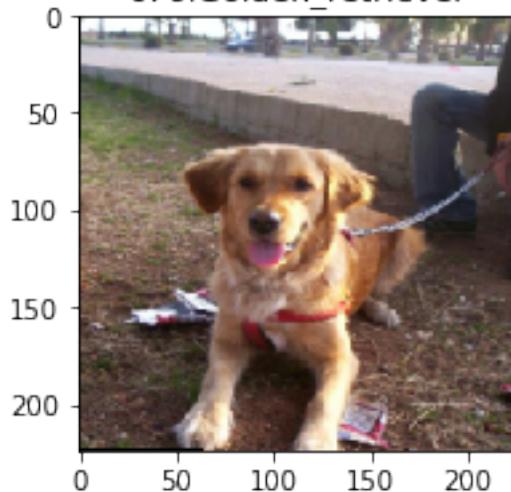
034.Boxer



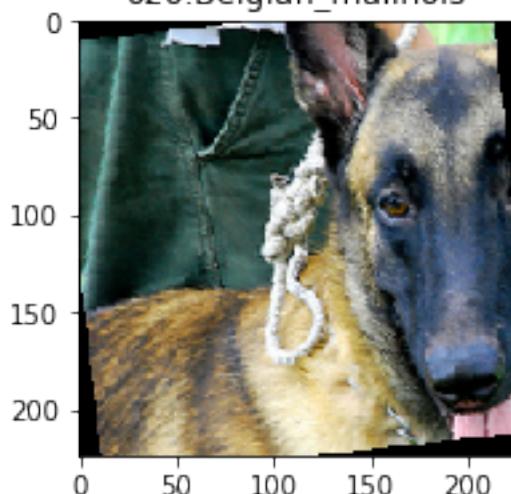
022.Belgian\_tervuren

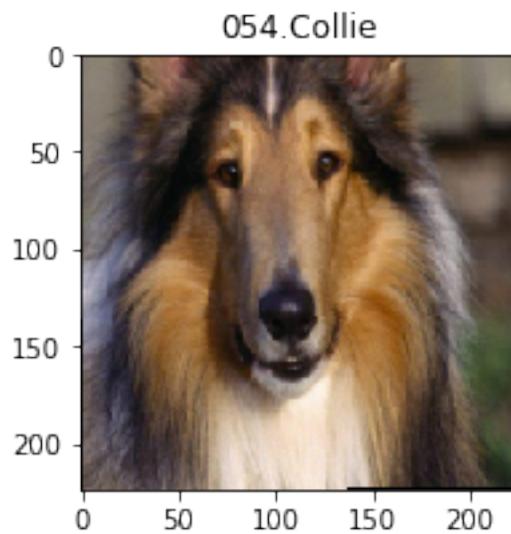
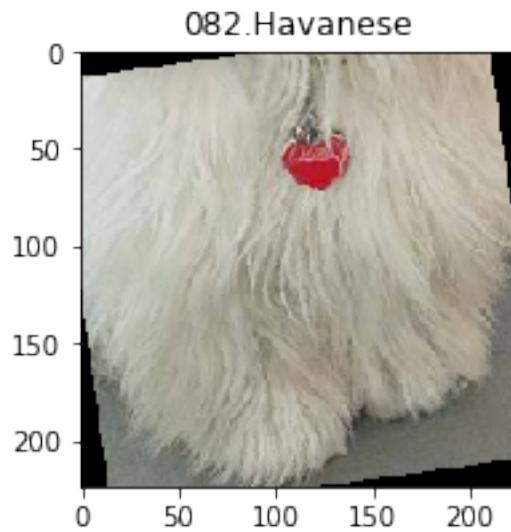


076.Golden\_retriever



020.Belgian\_malinois





**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: RandomResizedCrop, RandomHorizontalFlip and RandomRotation were used to augment and resize the train dataset. All images resized to 224\*224 pixels which were the setting for the input tensor. This size is deal because top1 and top5 errors are observed to be lesser for this setting.

### 1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [18]: 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__()

                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(256 * 5 * 5, 500)
                self.fc2 = nn.Linear(500, 133)

            def forward(self, x):
                ## Define forward behavior
                # add sequence of convolutional and max pooling layers
                x = self.pool(F.relu(self.conv1(x)))
                x = self.conv_bn2(x)

                x = self.pool(F.relu(self.conv2(x)))
                x = self.conv_bn3(x)

                x = self.pool(F.relu(self.conv3(x)))
                x = self.conv_bn4(x)

                x = self.pool(F.relu(self.conv4(x)))
                x = self.conv_bn5(x)

                x = self.pool(F.relu(self.conv5(x)))
                x = self.conv_bn6(x)
```

```

        # flatten image input
        x = x.view(-1, 256 * 5 * 5)
        # add second hidden layer
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

    # #-# You so NOT have to modify the code below this line. #-#-#
    # instantiate the CNN
    model_scratch = Net()
    print (model_scratch)

    # check if CUDA is available
    use_cuda = torch.cuda.is_available()
    # move tensors to GPU if CUDA is available
    if use_cuda:
        model_scratch.cuda()

Net(
    (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1))
    (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1))
    (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1))
    (conv4): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1))
    (conv5): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1))
    (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (conv_bn1): BatchNorm2d(224, eps=3, momentum=0.1, affine=True, track_running_stats=True)
    (conv_bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv_bn3): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv_bn4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv_bn5): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv_bn6): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (fc1): Linear(in_features=6400, out_features=500, bias=True)
    (fc2): Linear(in_features=500, out_features=133, bias=True)
)

```

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

**Answer:** Five convolution layers are used all with the convolution of size = 3, stride = 1 and padding = 1.

Followed by a max pooling layer of 2\*2

Followed by Batch Normalization for optimizing and regularizing the network.

Also two connected linear layer at the end are used.

Relu activations are used after each layers.

These choices were made after different iteration processes of improving accuracy of the model. Removing the dropout increases the accuracy of the achitecture. Since batch normalization equally prevent overfitting, I think the absence of dropout cannot give rise to overfitting.

### 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 [19]: import torch.optim as optim
        ### TODO: select loss function
        criterion_scratch = nn.CrossEntropyLoss()
        ### TODO: select optimizer
        optimizer_scratch = optim.Adam(model_scratch.parameters())
```

### 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 [20]: 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 #
            #####
            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_loss)
                # 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 parameters
                loss.backward()
                # perform a single optimization step (parameter update)
                optimizer.step()
                # update training loss
                train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss))

            # validation
            valid_loss = 0.0
            for batch_idx, (data, target) in enumerate(loaders['valid']):
                # move to GPU
                if use_cuda:
                    data, target = data.cuda(), target.cuda()
                ## find the loss and update the model parameters accordingly
                ## record the average validation loss
                loss = criterion(output, target)
                valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data - valid_loss))

            # print training/validation statistics
            # calculate average loss over an epoch
            train_loss = train_loss / len(loaders['train'])
            valid_loss = valid_loss / len(loaders['valid'])

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

            # save the model if validation loss has decreased
            if valid_loss < valid_loss_min:
                valid_loss_min = valid_loss
                if not os.path.exists('checkpoints'):
                    os.makedirs('checkpoints')
                torch.save(model.state_dict(), save_path)
```

```

#####
# validate the model #
#####
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 = valid_loss + ((1 / (batch_idx + 1)) * (loss.data - valid_loss))

train_loss = train_loss/len(train_loader.dataset)
valid_loss = valid_loss/len(valid_loader.dataset)

# 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 ...'.format(
        valid_loss_min,
        valid_loss))
    torch.save(model.state_dict(), save_path)
    valid_loss_min = valid_loss

# return trained model
return model

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

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

```

```

Epoch: 1      Training Loss: 0.000725      Validation Loss: 0.005366
Validation loss decreased (inf --> 0.005366). Saving model ...
Epoch: 2      Training Loss: 0.000671      Validation Loss: 0.005148
Validation loss decreased (0.005366 --> 0.005148). Saving model ...

```

```
Epoch: 3      Training Loss: 0.000645      Validation Loss: 0.004906
Validation loss decreased (0.005148 --> 0.004906). Saving model ...
Epoch: 4      Training Loss: 0.000619      Validation Loss: 0.004875
Validation loss decreased (0.004906 --> 0.004875). Saving model ...
Epoch: 5      Training Loss: 0.000606      Validation Loss: 0.004753
Validation loss decreased (0.004875 --> 0.004753). Saving model ...
Epoch: 6      Training Loss: 0.000585      Validation Loss: 0.004582
Validation loss decreased (0.004753 --> 0.004582). Saving model ...
Epoch: 7      Training Loss: 0.000569      Validation Loss: 0.004478
Validation loss decreased (0.004582 --> 0.004478). Saving model ...
Epoch: 8      Training Loss: 0.000554      Validation Loss: 0.004282
Validation loss decreased (0.004478 --> 0.004282). Saving model ...
Epoch: 9      Training Loss: 0.000538      Validation Loss: 0.004394
Epoch: 10     Training Loss: 0.000528      Validation Loss: 0.004184
Validation loss decreased (0.004282 --> 0.004184). Saving model ...
Epoch: 11     Training Loss: 0.000515      Validation Loss: 0.004052
Validation loss decreased (0.004184 --> 0.004052). Saving model ...
Epoch: 12     Training Loss: 0.000500      Validation Loss: 0.003962
Validation loss decreased (0.004052 --> 0.003962). Saving model ...
Epoch: 13     Training Loss: 0.000494      Validation Loss: 0.003984
Epoch: 14     Training Loss: 0.000484      Validation Loss: 0.003941
Validation loss decreased (0.003962 --> 0.003941). Saving model ...
Epoch: 15     Training Loss: 0.000477      Validation Loss: 0.003844
Validation loss decreased (0.003941 --> 0.003844). Saving model ...
Epoch: 16     Training Loss: 0.000469      Validation Loss: 0.003809
Validation loss decreased (0.003844 --> 0.003809). Saving model ...
Epoch: 17     Training Loss: 0.000460      Validation Loss: 0.003711
Validation loss decreased (0.003809 --> 0.003711). Saving model ...
Epoch: 18     Training Loss: 0.000449      Validation Loss: 0.003758
Epoch: 19     Training Loss: 0.000441      Validation Loss: 0.003693
Validation loss decreased (0.003711 --> 0.003693). Saving model ...
Epoch: 20     Training Loss: 0.000437      Validation Loss: 0.003559
Validation loss decreased (0.003693 --> 0.003559). Saving model ...
Epoch: 21     Training Loss: 0.000433      Validation Loss: 0.003604
Epoch: 22     Training Loss: 0.000424      Validation Loss: 0.003567
Epoch: 23     Training Loss: 0.000420      Validation Loss: 0.003492
Validation loss decreased (0.003559 --> 0.003492). Saving model ...
Epoch: 24     Training Loss: 0.000412      Validation Loss: 0.003527
Epoch: 25     Training Loss: 0.000406      Validation Loss: 0.003435
Validation loss decreased (0.003492 --> 0.003435). Saving model ...
Epoch: 26     Training Loss: 0.000400      Validation Loss: 0.003508
Epoch: 27     Training Loss: 0.000397      Validation Loss: 0.003449
Epoch: 28     Training Loss: 0.000394      Validation Loss: 0.003392
Validation loss decreased (0.003435 --> 0.003392). Saving model ...
Epoch: 29     Training Loss: 0.000383      Validation Loss: 0.003293
Validation loss decreased (0.003392 --> 0.003293). Saving model ...
Epoch: 30     Training Loss: 0.000380      Validation Loss: 0.003335
```

### 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 [21]: def test(loaders, model, criterion, use_cuda):

    # monitor test loss and accuracy
    test_loss = 0.
    correct = 0.
    total = 0.

    model.eval()
    for batch_idx, (data, target) in enumerate(loaders['test']):
        # move to GPU
        if use_cuda:
            data, target = data.cuda(), target.cuda()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the loss
        loss = criterion(output, target)
        # update average test loss
        test_loss = test_loss + ((1 / (batch_idx + 1)) * (loss.data - test_loss))
        # convert output probabilities to predicted class
        pred = output.data.max(1, keepdim=True)[1]
        # compare predictions to true label
        correct += np.sum(np.squeeze(pred.eq(target.data.view_as(pred))).cpu().numpy())
        total += data.size(0)

    print('Test Loss: {:.6f}\n'.format(test_loss))

    print('\nTest Accuracy: %2d%% (%2d/%2d)' % (
        100. * correct / total, correct, total))

    # call test function
    test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)

Test Loss: 2.515823

Test Accuracy: 35% (293/836)
```

---

## Step 4: Create a CNN to Classify Dog Breeds (using Transfer Learning)

You will now use transfer learning to create a CNN that can identify dog breed from images. Your CNN must attain at least 60% accuracy on the test set.

### 1.1.12 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

Use the code cell below to write three separate **data loaders** for the training, validation, and test datasets of dog images (located at dogImages/train, dogImages/valid, and dogImages/test, respectively).

If you like, **you are welcome to use the same data loaders from the previous step**, when you created a CNN from scratch.

```
In [22]: ## TODO: Specify data loaders  
loaders_transfer = loaders_scratch
```

### 1.1.13 (IMPLEMENTATION) Model Architecture

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

```
In [23]: import torchvision.models as models  
import torch.nn as nn  
  
## TODO: Specify model architecture  
  
model_transfer = models.densenet161(pretrained=True)  
  
for param in model_transfer.parameters():  
    param.requires_grad = False  
  
num_neur = model_transfer.classifier.in_features  
model_transfer.classifier = nn.Linear(num_neur, 133)  
  
print(model_transfer)  
  
if use_cuda:  
    model_transfer = model_transfer.cuda()  
  
/opt/conda/lib/python3.6/site-packages/torchvision-0.2.1-py3.6.egg/torchvision/models/densenet.p  
Downloading: "https://download.pytorch.org/models/densenet161-8d451a50.pth" to /root/.torch/mod  
100%| 115730790/115730790 [00:01<00:00, 67299764.21it/s]  
  
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=True)
    (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=True)
    (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=True)
    (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=True)
    (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=True)
    (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=True)
    (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=True)
    (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=True)
    (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=True)
    (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=True)
    (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=True)
    (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=True)
    (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=True)
    (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=True)
    (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=True)
    (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=True)
    (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=True)
    (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=True)
    (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=True)
    (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=True)
    (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=True)
    (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=True)
    (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=True)
    (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=True)
        (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=True)
        (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=True)
        (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=True)
        (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=True)
    (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=True)
    (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=True)
    (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=True)
    (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=True)
    (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=True)
    (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=True)
    (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=True)
    (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=True)
    (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=True)
    (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=True)
    (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=True)
    (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=True)
    (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=True)
    (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=True)
    (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=True)
    (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=True)
    (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=True)
    (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=True)
    (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=True)
    (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=133, bias=True)
)
)

```

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

Answer: In a Densely Connected Networks every layer is connected to all its preceding layers and its succeeding ones, thereby creating a Dense Block. The authors claimed that their implementation performed better than previous state of the art on classification on ImageNet. A good reason I chose it. In addition, the implementation with connecting all layers of the network with each other, foster possibilities of testing new ideas with improved results.

As you can see, each time there is a convolution operation of the previous layer, it is followed by concatenation of the tensors. This is allowed as the channel dimensions, height and width of the input stay the same after convolution with a kernel size 3x3 and padding 1. The authors state that in this way the feature maps produced are more diversified and tend to have richer patterns. Also, another advantage will be better information flow during training.

Now to run some tests and see how the network it is performing, I optimize the model I use Stochastic Gradient Gradient Descent. I used batches of the images in size 20, with learning rate equal to 0.001 and momentum 0.90. I run the optimization method for 20 epochs. The total time for training the model was about 40 minutes. I was able to optimize the model on the GPU provided by the Udacity Workspace. The accuracy of over 80% I achieved convinced me the architecture is suitable.

### 1.1.14 (IMPLEMENTATION) Specify Loss Function and Optimizer

Use the next code cell to specify a [loss function](#) and [optimizer](#). Save the chosen loss function as `criterion_transfer`, and the optimizer as `optimizer_transfer` below.

```
In [24]: # specify loss function
criterion_transfer = nn.CrossEntropyLoss()
# specify optimizer
optimizer_transfer = optim.SGD(model_transfer.classifier.parameters(), lr=0.001, momentum=0.9)
```

### 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 model
model_transfer = train(20, loaders_transfer, model_transfer, optimizer_transfer, criterion_transfer)

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

Epoch: 1 Training Loss: 0.000607 Validation Loss: 0.003500  
Validation loss decreased (inf --> 0.003500). Saving model ...  
Epoch: 2 Training Loss: 0.000400 Validation Loss: 0.002252  
Validation loss decreased (0.003500 --> 0.002252). Saving model ...  
Epoch: 3 Training Loss: 0.000298 Validation Loss: 0.001614  
Validation loss decreased (0.002252 --> 0.001614). Saving model ...  
Epoch: 4 Training Loss: 0.000244 Validation Loss: 0.001309  
Validation loss decreased (0.001614 --> 0.001309). Saving model ...  
Epoch: 5 Training Loss: 0.000211 Validation Loss: 0.001134  
Validation loss decreased (0.001309 --> 0.001134). Saving model ...  
Epoch: 6 Training Loss: 0.000190 Validation Loss: 0.000985  
Validation loss decreased (0.001134 --> 0.000985). Saving model ...  
Epoch: 7 Training Loss: 0.000178 Validation Loss: 0.000913  
Validation loss decreased (0.000985 --> 0.000913). Saving model ...  
Epoch: 8 Training Loss: 0.000166 Validation Loss: 0.000833  
Validation loss decreased (0.000913 --> 0.000833). Saving model ...  
Epoch: 9 Training Loss: 0.000154 Validation Loss: 0.000774  
Validation loss decreased (0.000833 --> 0.000774). Saving model ...  
Epoch: 10 Training Loss: 0.000149 Validation Loss: 0.000755  
Validation loss decreased (0.000774 --> 0.000755). Saving model ...  
Epoch: 11 Training Loss: 0.000142 Validation Loss: 0.000715  
Validation loss decreased (0.000755 --> 0.000715). Saving model ...  
Epoch: 12 Training Loss: 0.000141 Validation Loss: 0.000708  
Validation loss decreased (0.000715 --> 0.000708). Saving model ...  
Epoch: 13 Training Loss: 0.000134 Validation Loss: 0.000654  
Validation loss decreased (0.000708 --> 0.000654). Saving model ...  
Epoch: 14 Training Loss: 0.000132 Validation Loss: 0.000649  
Validation loss decreased (0.000654 --> 0.000649). Saving model ...

```

Epoch: 15      Training Loss: 0.000125      Validation Loss: 0.000642
Validation loss decreased (0.000649 --> 0.000642). Saving model ...
Epoch: 16      Training Loss: 0.000125      Validation Loss: 0.000627
Validation loss decreased (0.000642 --> 0.000627). Saving model ...
Epoch: 17      Training Loss: 0.000122      Validation Loss: 0.000614
Validation loss decreased (0.000627 --> 0.000614). Saving model ...
Epoch: 18      Training Loss: 0.000123      Validation Loss: 0.000583
Validation loss decreased (0.000614 --> 0.000583). Saving model ...
Epoch: 19      Training Loss: 0.000121      Validation Loss: 0.000580
Validation loss decreased (0.000583 --> 0.000580). Saving model ...
Epoch: 20      Training Loss: 0.000118      Validation Loss: 0.000597

```

### 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 [26]: test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)

Test Loss: 0.429977

Test Accuracy: 86% (727/836)
```

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

          ### TODO: Write a function that takes a path to an image as input
          ### and returns the dog breed that is predicted by the model.
import matplotlib.image as mpimg
import random
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)
```

```

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

preprocess = transforms.Compose([
    transforms.RandomResizedCrop(224),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(10),
    transforms.ToTensor(),
    normalize
])
img_tensor = preprocess(img).float()
img_tensor.unsqueeze_(0)
img_tensor = Variable(img_tensor)
if use_cuda:
    img_tensor = Variable(img_tensor.cuda())
model_transfer.eval()
output = model_transfer(img_tensor)
output = output.cpu()
index = output.data.numpy().argmax()
return index, class_names[index], data_transfer['train'].classes[index]

```

```

In [39]: ### Display prediction
def display_predictions(img_path):
    #print (img_path)
    index, dog_breed, name = predict_breed_transfer(img_path)
    print("Dog detected!!")

    # display test image
    fig = plt.figure(figsize=(16,4))
    ax = fig.add_subplot(121)
    img = mpimg.imread(img_path)
    ax.imshow(img)
    plt.axis('off')

    # display sample of matching breed images
    subdir = '/'.join(['data/dog_images/valid', str(name)])
    file = random.choice(os.listdir(subdir))
    path = '/'.join([subdir, file])
    ax = fig.add_subplot(122)
    img = mpimg.imread(path)
    ax.imshow(img.squeeze(), cmap="viridis", interpolation='bicubic')
    plt.title(dog_breed)
    plt.axis('off')
    plt.show()

    # extract breed from image path

```

```

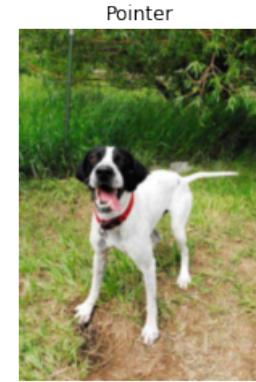
actual_breed = img_path.split('/')[-1].split('.')[1]
print(f"Actual Breed: {actual_breed}\n")
print(f"Predicted Breed: {dog_breed}\n")
print("\n\n")

In [40]: # Create list of test image paths
test_img = sorted(glob('/data/dog_images/test/*/*'))
# Shuffle the list and display first few rows
np.random.shuffle(test_img)
test_img[91:95]

for img_path in test_img[90:95]:
    display_predictions(img_path)

```

Dog detected!!

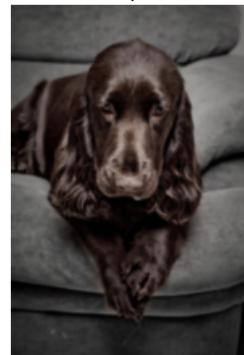


Actual Breed: Pointer

Predicted Breed: Pointer

Dog detected!!

Field spaniel



Actual Breed: Field\_spaniel

Predicted Breed: Field spaniel

Dog detected!!

Beagle



Actual Breed: Beagle

Predicted Breed: Beagle

Dog detected!!



English cocker spaniel



Actual Breed: English\_cocker\_spaniel

Predicted Breed: English cocker spaniel

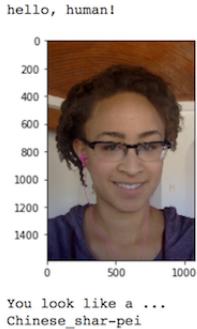
Dog detected!!



Chihuahua



Actual Breed: Chihuahua



Sample Human Output

Predicted Breed: Chihuahua

#### ## 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 [41]: *### TODO: Write your algorithm.*  
*### Feel free to use as many code cells as needed.*

```
def run_app(img_path):
    '''
```

*Use pre-trained model to check if the image at the given path contains a human being or a dog or none.*

*Args:*

*img\_path: path to an image*

*Returns:*

*print if a human face is detected or not*

```

    print the dog breed or show that neither human face nor a dog detected
    ...
pick_human = face_detector(img_path)
pick_dog = dog_detector(img_path)
index, dog_breed, name = predict_breed_transfer(img_path)

# display test image
fig = plt.figure(figsize=(16,4))

if(pick_human):
    print("Human detected!!")
    ax = fig.add_subplot(121)
    img = mpimg.imread(img_path)
    ax.imshow(img)
    plt.axis('off')

    # display sample of matching breed images
    subdir = '/'.join(['/data/dog_images/valid', str(name)])
    file = random.choice(os.listdir(subdir))
    path = '/'.join([subdir, file])
    ax = fig.add_subplot(122)
    img = mpimg.imread(path)
    ax.imshow(img.squeeze(), cmap="viridis", interpolation='bicubic')
    plt.title(dog_breed)
    plt.axis('off')
    plt.show()
    print("You resemble ... " + dog_breed)
    print("\n\n\n")
    return

elif(pick_dog):
    print("Dog detected!!")
    ax = fig.add_subplot(121)
    img = mpimg.imread(img_path)
    ax.imshow(img)
    plt.axis('off')

    # display sample of matching breed images
    subdir = '/'.join(['/data/dog_images/valid', str(name)])
    file = random.choice(os.listdir(subdir))
    path = '/'.join([subdir, file])
    ax = fig.add_subplot(122)
    img = mpimg.imread(path)
    ax.imshow(img.squeeze(), cmap="viridis", interpolation='bicubic')
    plt.title(dog_breed)
    plt.axis('off')
    plt.show()
    print("You resemble ... " + dog_breed)

```

```

        print("\n\n\n")
        return

    else:
        print('Error! The image is unfamiliar')
        ax = fig.add_subplot(121)
        img = mpimg.imread(img_path)
        ax.imshow(img)
        plt.axis('off')
        plt.show()
        print("\n\n\n")
        return

```

---

### ## 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:(Three possible points for improvement) The test accuracy of my algorithm is more than 80% which is very good.Algorithm is efficient in detecting if pictures contain a human face or a dog.

Some possible points of improvement:

More image datasets will improve training models. Data augmentations (position shifts, color channel shifts,flipping,rotating,cropping) will improve performance on test data

Hyper-parameters can be tuned to achieve optimal performance.

The model could be trained for more epochs.

Considering other optimizer like Adam optimizer to see if there is improvement.

```

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.

          # Get 10 random images of each type
          pick_human = np.random.randint(0,high=len(human_files),size=10)
          pick_dog = np.random.randint(0,high=len(dog_files),size=10)
          for file in np.hstack((human_files[pick_human], dog_files[pick_dog])):
              run_app(file)

```

Human detected!!

Dachshund



You resemble ...Dachshund

Human detected!!

American water spaniel



You resemble ...American water spaniel

Human detected!!

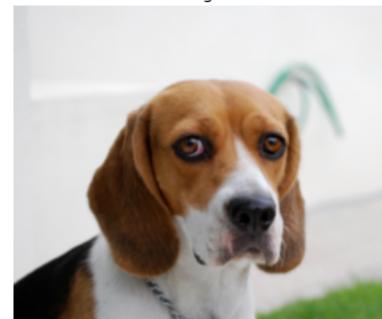
English springer spaniel



You resemble ...English springer spaniel

Human detected!!

Beagle



You resemble ...Beagle

Human detected!!

Kerry blue terrier



You resemble ...Kerry blue terrier

Human detected!!

English springer spaniel



You resemble ...English springer spaniel

Human detected!!

Beagle



You resemble ...Beagle

Human detected!!

Chinese crested



You resemble ...Chinese crested

Human detected!!

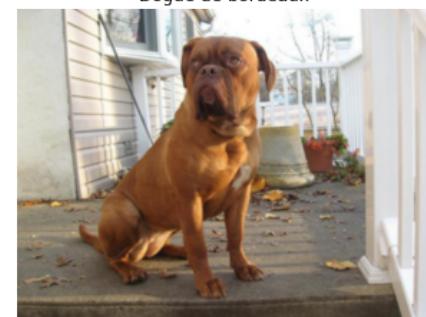
American water spaniel



You resemble ...American water spaniel

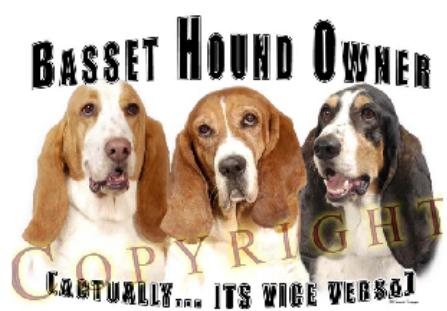
Human detected!!

Dogue de bordeaux



You resemble ...Dogue de bordeaux

Dog detected!!



You resemble ... Basset hound

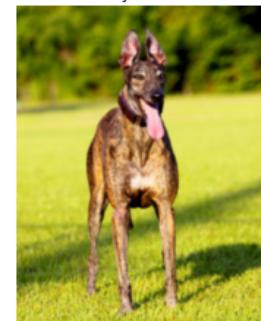
Dog detected!!



You resemble ... German shorthaired pointer

Dog detected!!

Greyhound



You resemble ... Greyhound

Dog detected!!

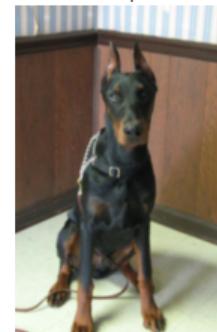
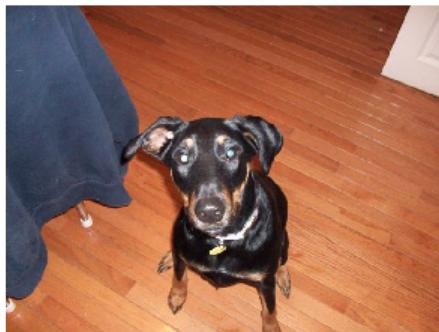
American staffordshire terrier



You resemble ... American staffordshire terrier

Dog detected!!

Doberman pinscher



You resemble ... Doberman pinscher

Dog detected!!

Australian cattle dog



You resemble ... Australian cattle dog

Dog detected!!

Lhasa apso



You resemble ... Lhasa apso

Dog detected!!

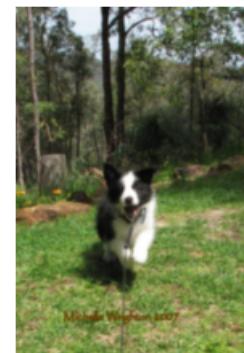
American eskimo dog



You resemble ... American eskimo dog

Dog detected!!

Border collie



You resemble ... Border collie

Dog detected!!

Chinese crested



You resemble ... Chinese crested

In [ ]: