Convolutional Neural Networks

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.

Why We're Here

In this notebook, you will make the first steps towards developing an algorithm that could be used as part of a mobile or web app. At the end of this project, your code will accept any user-supplied image as input. If a dog is detected in the image, it will provide an estimate of the dog's breed. If a human is detected, it will provide an estimate of the dog breed that is most resembling. The image below displays potential sample output of your finished project (... but we expect that each student's algorithm will behave differently!).

 Sample Dog Output

In this real-world setting, you will need to piece together a series of models to perform different tasks; for instance, the algorithm that detects humans in an image will be different from the CNN that infers dog breed. There are many points of possible failure, and no perfect algorithm exists. Your imperfect solution will nonetheless create a fun user experience!

The Road Ahead

We break the notebook into separate steps. Feel free to use the links below to navigate the notebook.

Step 0: Import Datasets

Make sure that you've downloaded the required human and dog datasets:

- Download the dog dataset (https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/dogImages.zip). Unzip the folder and place it in this project's home directory, at the location `/dogImages`.
- Download the human dataset (https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/lfw.zip). Unzip the folder and place it in the home diretcory, at location `/lfw`.

Note: If you are using a Windows machine, you are encouraged to use 7zip (http://www.7-zip.org/) 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 [12]:

```
%cd ./drive/My Drive/Colab Notebooks/dog_breed
!ls
```

```
[Errno 2] No such file or directory: './drive/My Drive/Colab Notebooks/dog
_breed'
/content/drive/My Drive/Colab Notebooks/dog_breed
dog_app.html   dogImages       haarcascades  lfw       my_images  saved_mo
dels
dog_app.ipynb  dogImages_.zip  images        lfw_.zip  README.md
```

In [13]:

```
import numpy as np
from glob import glob

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

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

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

In [0]:

```
# from google.colab import drive
# drive.mount('/content/drive/')

# %cd drive
# !ls
# !unzip dogImages_
# !unzip lfw_
```

In [14]:

```
!pip install opencv-python
```

```
Requirement already satisfied: opencv-python in /usr/local/lib/python3.6/d
ist-packages (3.4.7.28)
Requirement already satisfied: numpy>=1.11.3 in /usr/local/lib/python3.6/d
ist-packages (from opencv-python) (1.17.3)
```

Step 1: Detect Humans

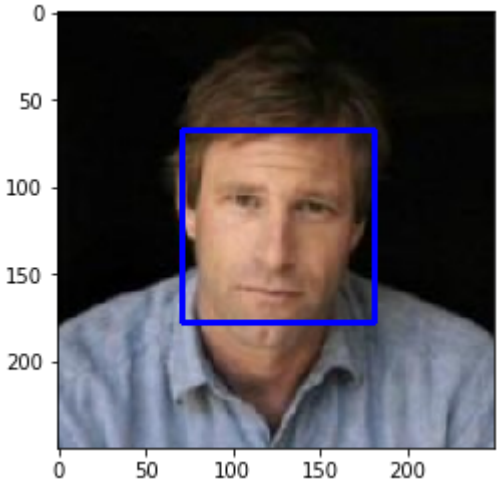
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers (http://docs.opencv.org/trunk/d7/d8b/tutorial_py_face_detection.html) to detect human faces in images.

OpenCV provides many pre-trained face detectors, stored as XML files on github (https://github.com/opencv/opencv/tree/master/data/haarcascades). 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 [15]:

```
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.

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 [1]:

```
# 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
```

(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 [0]:

```
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.

def face_detection_test(files):
    detection_cnt = 0;
    total_cnt = len(files)
    for file in files:
        detection_cnt += face_detector(file)
    return detection_cnt, total_cnt
```

In [18]:

```
print("detect face in human_files: {} / {}".format(face_detection_test(human_files_shor
t)[0], face_detection_test(human_files_short)[1]))
print("Percentage: ", float(face_detection_test(human_files_short)[0]/face_detection_te
st(human_files_short)[1]) )
print("detect face in dog_files: {} / {}".format(face_detection_test(dog_files_short)[0
], face_detection_test(dog_files_short)[1]))
print("Percentage: ", float(face_detection_test(dog_files_short)[0]/face_detection_test
(dog_files_short)[1]) )
```

```
detect face in human_files: 96 / 100
Percentage:  0.96
detect face in dog_files: 18 / 100
Percentage:  0.18
```

We suggest the face detector from OpenCV as a potential way to detect human images in your algorithm, but you are free to explore other approaches, especially approaches that make use of deep learning :). Please use the code cell below to design and test your own face detection algorithm. If you decide to pursue this *optional* task, report performance on `human_files_short` and `dog_files_short`.

In [0]:

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

Step 2: Detect Dogs

In this section, we use a pre-trained model (http://pytorch.org/docs/master/torchvision/models.html) to detect dogs in images.

Obtain Pre-trained VGG-16 Model

The code cell below downloads the VGG-16 model, along with weights that have been trained on ImageNet (http://www.image-net.org/), 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 (https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a).

In [20]:

```
import torch
import torchvision.models as models

# check if CUDA is available
use_cuda = torch.cuda.is_available()
print("cuda available? {0}".format(use_cuda))
```

```
cuda available? True
```

In [0]:

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

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

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

(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 (http://pytorch.org/docs/stable/torchvision/models.html).

In [0]:

```
from PIL import Image
import torchvision.transforms as transforms

def load_image(img_path):
    image = Image.open(img_path).convert('RGB')
    # resize to (244, 244) because VGG16 accept this shape
    in_transform = transforms.Compose([
                        transforms.Resize(size=(244, 244)),
                        transforms.ToTensor()]) # normalizaiton parameters from pytorch
doc.

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

In [0]:

```
def VGG16_predict(img_path):
    '''
    Use pre-trained VGG-16 model to obtain index corresponding to
    predicted ImageNet class for image at specified path

    Args:
        img_path: path to an image

    Returns:
        Index corresponding to VGG-16 model's prediction
    '''

    ## TODO: Complete the function.
    ## Load and pre-process an image from the given img_path
    ## Return the *index* of the predicted class for that image
    img = load_image(img_path)
    if use_cuda:
        img = img.cuda()
    ret = VGG16(img)
    return torch.max(ret,1)[1].item() # predicted class index
```

In [24]:

```
# predict dog using ImageNet class
VGG16_predict(dog_files_short[0])
```

Out[24]:

```
552
```

(IMPLEMENTATION) Write a Dog Detector

While looking at the dictionary (https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a), 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 [0]:

```
### 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 # true/false
```

In [26]:

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

```
False
False
```

In [0]:

```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.
def dog_detector_test(files):
    detection_cnt = 0;
    total_cnt = len(files)
    for file in files:
        detection_cnt += dog_detector(file)
    return detection_cnt, total_cnt
```

In [28]:

```
print("detect a dog in human_files: {} / {}".format(dog_detector_test(human_files_short
)[0], dog_detector_test(human_files_short)[1]))
print("detect a dog in dog_files: {} / {}".format(dog_detector_test(dog_files_short)[0
], dog_detector_test(dog_files_short)[1]))
```

```
detect a dog in human_files: 0 / 100
detect a dog in dog_files: 95 / 100
```

(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:

detect a dog in human_files: 0 / 100 thus 0%

detect a dog in dog_files: 95 / 100 thus 95%

We suggest VGG-16 as a potential network to detect dog images in your algorithm, but you are free to explore other pre-trained networks (such as Inception-v3 (http://pytorch.org/docs/master/torchvision/models.html#inception-v3), ResNet-50 (http://pytorch.org/docs/master/torchvision/models.html#id3), 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 [0]:

```
### (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	Black Labrador
 		

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

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

(IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

Use the code cell below to write three separate [data loaders (http://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader)](http://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader) for the training, validation, and test datasets of dog images (located at `dogImages/train`, `dogImages/valid`, and `dogImages/test`, respectively). You may find [this documentation on custom datasets (http://pytorch.org/docs/stable/torchvision/datasets.html)](http://pytorch.org/docs/stable/torchvision/datasets.html) to be a useful resource. If you are interested in augmenting your training and/or validation data, check out the wide variety of [transforms (http://pytorch.org/docs/stable/torchvision/transforms.html?highlight=transform)](http://pytorch.org/docs/stable/torchvision/transforms.html?highlight=transform)!

Preprocessing

In [0]:

```
import os
from torchvision import datasets
import torchvision.transforms as transforms
import torch
import numpy as np
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

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

batch_size = 20
num_workers = 0

data_dir = 'dogImages/'
train_dir = os.path.join(data_dir, 'train/')
valid_dir = os.path.join(data_dir, 'valid/')
test_dir = os.path.join(data_dir, 'test/')
```

User Standard Normalization Value

In [0]:

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

In [0]:

```
data_transforms = {'train': transforms.Compose([transforms.RandomResizedCrop(224),
                                    transforms.RandomHorizontalFlip(),
                                    transforms.ToTensor(),
                                    standard_normalization]),
                   'val': transforms.Compose([transforms.Resize(256),
                                    transforms.CenterCrop(224),
                                    transforms.ToTensor(),
                                    standard_normalization]),
                   'test': transforms.Compose([transforms.Resize(size=(224,224)),
                                    transforms.ToTensor(),
                                    standard_normalization])
                  }
```

Use ImageFolder to Load image_dataset

In [0]:

```
train_data = datasets.ImageFolder(train_dir, transform=data_transforms['train'])
valid_data = datasets.ImageFolder(valid_dir, transform=data_transforms['val'])
test_data = datasets.ImageFolder(test_dir, transform=data_transforms['test'])
```

In [0]:

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

Question 3: Describe your chosen procedure for preprocessing the data.

- How does your code resize the images (by cropping, stretching, etc)? What size did you pick for the input tensor, and why?
- Did you decide to augment the dataset? If so, how (through translations, flips, rotations, etc)? If not, why not?

Answer: I've applied RandomResizedCrop & RandomHorizontalFlip to just *train_data*. This will do both image augmentations and resizing jobs. Image augmentation will give randomness to the dataset so, it prevents overfitting and I can expect better performance of model when it's predicting toward test_data. On the other hand, I've done Resize of (256) and then, center crop to make 224 X 224. Since valid_data will be used for validation check, I will not do image augmentations. For the test_data, I've applied only image resizing.

(IMPLEMENTATION) Model Architecture

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

In [0]:

```
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

num_classes = 133 # total classes of dog breeds
```

In [35]:

```
import torch.nn as nn
import torch.nn.functional as F
import numpy as np

# define the CNN architecture
class Net(nn.Module):
    ### TODO: choose an architecture, and complete the class
    def __init__(self):
        super(Net, self).__init__()
        ## Define layers of a CNN
        self.conv1 = nn.Conv2d(3, 32, 3, stride=2, padding=1)
        self.conv2 = nn.Conv2d(32, 64, 3, stride=2, padding=1)
        self.conv3 = nn.Conv2d(64, 128, 3, padding=1)

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

        # fully-connected
        self.fc1 = nn.Linear(7*7*128, 500)
        self.fc2 = nn.Linear(500, num_classes)

        # drop-out
        self.dropout = nn.Dropout(0.3)

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

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

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

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

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

# instantiate the CNN
model_scratch = Net()
print(model_scratch)

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

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

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

Answer:

(conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))

activation: relu

(pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)

activation: relu

(conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))

activation: relu

(pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)

(conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))

(pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)

(dropout): Dropout(p=0.3)

(fc1): Linear(in_features=6272, out_features=500, bias=True)

(dropout): Dropout(p=0.3)

(fc2): Linear(in_features=500, out_features=133, bias=True)

explanations First 2 conv layers I've applied kernel_size of 3 with stride 2, this will lead to downsize of input image by 2. after 2 conv layers, maxpooling with stride 2 is placed and this will lead to downsize of input image by 2. The 3rd conv layers is consist of kernel_size of 3 with stride 1, and this will not reduce input image. after final maxpooling with stride 2, the total output image size is downsized by factor of 32 and the depth will be 128. I've applied dropout of 0.3 in order to prevent overfitting. Fully-connected layer is placed and then, 2nd fully-connected layer is intended to produce final output_size which predicts classes of breeds.

(IMPLEMENTATION) Specify Loss Function and Optimizer

Use the next code cell to specify a loss function (http://pytorch.org/docs/stable/nn.html#loss-functions) and optimizer (http://pytorch.org/docs/stable/optim.html). Save the chosen loss function as `criterion_scratch`, and the optimizer as `optimizer_scratch` below.

In [0]:

```
import torch.optim as optim

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

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

(IMPLEMENTATION) Train and Validate the Model

Train and validate your model in the code cell below. Save the final model parameters (http://pytorch.org/docs/master/notes/serialization.html) at filepath `'model_scratch.pt'`.

In [0]:

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

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

        # train the model #

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

            # initialize weights to zero
            optimizer.zero_grad()
            output = model(data)

            # calculate loss
            loss = criterion(output, target)

            # back prop
            loss.backward()

            # grad
            optimizer.step()
            train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss
))
            if batch_idx % 100 == 0:
                print('Epoch %d, Batch %d loss: %.6f' %
                  (epoch, batch_idx + 1, train_loss))

        # validate the model #

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


        # print training/validation statistics
```

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

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

    # return trained model
    return model
```

In [38]:

```
# train the model

model_scratch = train(20, loaders_scratch, model_scratch, optimizer_scratch, criterion_
scratch,
                      use_cuda, 'saved_models/model_scratch.pt')
```

```
Epoch 1, Batch 1 loss: 4.879031
Epoch 1, Batch 101 loss: 4.886653
Epoch 1, Batch 201 loss: 4.880824
Epoch 1, Batch 301 loss: 4.872817
Epoch: 1        Training Loss: 4.870019         Validation Loss: 4.809074
Validation loss decreased (inf --> 4.809074).  Saving model ...
Epoch 2, Batch 1 loss: 4.754141
Epoch 2, Batch 101 loss: 4.809505
Epoch 2, Batch 201 loss: 4.796490
Epoch 2, Batch 301 loss: 4.785848
Epoch: 2        Training Loss: 4.785899         Validation Loss: 4.675137
Validation loss decreased (4.809074 --> 4.675137).  Saving model ...
Epoch 3, Batch 1 loss: 4.608727
Epoch 3, Batch 101 loss: 4.724247
Epoch 3, Batch 201 loss: 4.723772
Epoch 3, Batch 301 loss: 4.706271
Epoch: 3        Training Loss: 4.699392         Validation Loss: 4.531134
Validation loss decreased (4.675137 --> 4.531134).  Saving model ...
Epoch 4, Batch 1 loss: 4.482553
Epoch 4, Batch 101 loss: 4.617053
Epoch 4, Batch 201 loss: 4.600802
Epoch 4, Batch 301 loss: 4.595534
Epoch: 4        Training Loss: 4.593587         Validation Loss: 4.417414
Validation loss decreased (4.531134 --> 4.417414).  Saving model ...
Epoch 5, Batch 1 loss: 4.451584
Epoch 5, Batch 101 loss: 4.532630
Epoch 5, Batch 201 loss: 4.514453
Epoch 5, Batch 301 loss: 4.511855
Epoch: 5        Training Loss: 4.510175         Validation Loss: 4.302109
Validation loss decreased (4.417414 --> 4.302109).  Saving model ...
Epoch 6, Batch 1 loss: 4.413883
Epoch 6, Batch 101 loss: 4.449588
Epoch 6, Batch 201 loss: 4.454877
Epoch 6, Batch 301 loss: 4.447582
Epoch: 6        Training Loss: 4.445515         Validation Loss: 4.265918
Validation loss decreased (4.302109 --> 4.265918).  Saving model ...
Epoch 7, Batch 1 loss: 4.254455
Epoch 7, Batch 101 loss: 4.378865
Epoch 7, Batch 201 loss: 4.384565
Epoch 7, Batch 301 loss: 4.390982
Epoch: 7        Training Loss: 4.392698         Validation Loss: 4.206824
Validation loss decreased (4.265918 --> 4.206824).  Saving model ...
Epoch 8, Batch 1 loss: 4.437192
Epoch 8, Batch 101 loss: 4.350548
Epoch 8, Batch 201 loss: 4.345530
Epoch 8, Batch 301 loss: 4.331442
Epoch: 8        Training Loss: 4.334467         Validation Loss: 4.168550
Validation loss decreased (4.206824 --> 4.168550).  Saving model ...
Epoch 9, Batch 1 loss: 3.792347
Epoch 9, Batch 101 loss: 4.251232
Epoch 9, Batch 201 loss: 4.274724
Epoch 9, Batch 301 loss: 4.268054
Epoch: 9        Training Loss: 4.263001         Validation Loss: 4.046604
Validation loss decreased (4.168550 --> 4.046604).  Saving model ...
Epoch 10, Batch 1 loss: 4.094123
Epoch 10, Batch 101 loss: 4.217377
Epoch 10, Batch 201 loss: 4.211514
Epoch 10, Batch 301 loss: 4.214526
Epoch: 10       Training Loss: 4.209414         Validation Loss: 4.076917
Epoch 11, Batch 1 loss: 4.223356
Epoch 11, Batch 101 loss: 4.169202
```

```
Epoch 11, Batch 201 loss: 4.179853
Epoch 11, Batch 301 loss: 4.179676
Epoch: 11       Training Loss: 4.181222         Validation Loss: 3.939685
Validation loss decreased (4.046604 --> 3.939685).  Saving model ...
Epoch 12, Batch 1 loss: 3.602317
Epoch 12, Batch 101 loss: 4.080497
Epoch 12, Batch 201 loss: 4.110284
Epoch 12, Batch 301 loss: 4.111743
Epoch: 12       Training Loss: 4.116404         Validation Loss: 3.954967
Epoch 13, Batch 1 loss: 3.937528
Epoch 13, Batch 101 loss: 4.052757
Epoch 13, Batch 201 loss: 4.056012
Epoch 13, Batch 301 loss: 4.049047
Epoch: 13       Training Loss: 4.052325         Validation Loss: 3.852051
Validation loss decreased (3.939685 --> 3.852051).  Saving model ...
Epoch 14, Batch 1 loss: 4.004964
Epoch 14, Batch 101 loss: 4.014110
Epoch 14, Batch 201 loss: 4.010549
Epoch 14, Batch 301 loss: 4.014665
Epoch: 14       Training Loss: 4.015814         Validation Loss: 4.006102
Epoch 15, Batch 1 loss: 4.241508
Epoch 15, Batch 101 loss: 3.939792
Epoch 15, Batch 201 loss: 3.965302
Epoch 15, Batch 301 loss: 3.971589
Epoch: 15       Training Loss: 3.972163         Validation Loss: 3.837527
Validation loss decreased (3.852051 --> 3.837527).  Saving model ...
Epoch 16, Batch 1 loss: 3.313303
Epoch 16, Batch 101 loss: 3.905481
Epoch 16, Batch 201 loss: 3.900238
Epoch 16, Batch 301 loss: 3.904998
Epoch: 16       Training Loss: 3.906523         Validation Loss: 3.679406
Validation loss decreased (3.837527 --> 3.679406).  Saving model ...
Epoch 17, Batch 1 loss: 3.734645
Epoch 17, Batch 101 loss: 3.852474
Epoch 17, Batch 201 loss: 3.843322
Epoch 17, Batch 301 loss: 3.864138
Epoch: 17       Training Loss: 3.860472         Validation Loss: 3.680525
Epoch 18, Batch 1 loss: 3.656837
Epoch 18, Batch 101 loss: 3.786475
Epoch 18, Batch 201 loss: 3.817233
Epoch 18, Batch 301 loss: 3.823014
Epoch: 18       Training Loss: 3.826013         Validation Loss: 3.590614
Validation loss decreased (3.679406 --> 3.590614).  Saving model ...
Epoch 19, Batch 1 loss: 3.701981
Epoch 19, Batch 101 loss: 3.794490
Epoch 19, Batch 201 loss: 3.752643
Epoch 19, Batch 301 loss: 3.763506
Epoch: 19       Training Loss: 3.771371         Validation Loss: 3.613346
Epoch 20, Batch 1 loss: 3.593747
Epoch 20, Batch 101 loss: 3.743726
Epoch 20, Batch 201 loss: 3.714673
Epoch 20, Batch 301 loss: 3.737299
Epoch: 20       Training Loss: 3.737938         Validation Loss: 3.661290
```

In [39]:

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

Out[39]:

```
<All keys matched successfully>
```

(IMPLEMENTATION) Test the Model

Try out your model on the test dataset of dog images. Use the code cell below to calculate and print the test loss and accuracy. Ensure that your test accuracy is greater than 10%.

In [40]:

```
def test(loaders, model, criterion, use_cuda):

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

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

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

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

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

```
Test Loss: 3.744133


Test Accuracy: 14% (120/836)
```

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.

(IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

Use the code cell below to write three separate [data loaders (http://pytorch.org/docs/master/data.html#torch.utils.data.DataLoader)](http://pytorch.org/docs/master/data.html#torch.utils.data.DataLoader) 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 [0]:

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

(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 [42]:

```
import torchvision.models as models
import torch.nn as nn

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

```
Downloading: "https://download.pytorch.org/models/resnet50-19c8e357.pth" t
o /root/.cache/torch/checkpoints/resnet50-19c8e357.pth
100%|██████████| 97.8M/97.8M [00:03<00:00, 28.9MB/s]
```

In [0]:

```
for param in model_transfer.parameters():
    param.requires_grad = False
```

In [0]:

```
model_transfer.fc = nn.Linear(2048, 133, bias=True)
```

In [0]:

```
fc_parameters = model_transfer.fc.parameters()
```

In [0]:

```
for param in fc_parameters:
    param.requires_grad = True
```

In [47]:

```
model_transfer
```

Out[47]:

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3,
3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, cei
l_mode=False)
  (layer1): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=Fals
e)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_r
unning_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_r
unning_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=Fal
se)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=Fals
e)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=Fal
se)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_r
unning_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_r
unning_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=Fal
se)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=Fal
se)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_r
unning_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_r
unning_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=Fal
se)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
      (relu): ReLU(inplace=True)
```

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

```
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=Fa
lse)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (layer3): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=Fa
lse)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding
=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=F
alse)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track
_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2), bias=Fal
se)
        (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track
_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=F
alse)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding
=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=F
alse)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track
_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=F
alse)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding
=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=F
alse)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track
_running_stats=True)
      (relu): ReLU(inplace=True)
    )
```

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

```
      (downsample): Sequential(
        (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=Fa
lse)
        (1): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track
_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=F
alse)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding
=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=F
alse)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track
_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=F
alse)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding
=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=F
alse)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track
_running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=2048, out_features=133, bias=True)
)
```

In [0]:

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

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

Answer:

I picked ResNet as a transfer model because it performed outstanding on Image Classification. I looked into the structure and functions of ResNet. The core idea of ResNet is introducing a so-called “identity shortcut connection” that skips one or more layers. I guess this prevents overfitting when it's training.

I've pull out the final Fully-connected layer and replaced with Fully-connected layer with output of 133 (dog br

(IMPLEMENTATION) Specify Loss Function and Optimizer

Use the next code cell to specify a loss function (http://pytorch.org/docs/master/nn.html#loss-functions) and optimizer (http://pytorch.org/docs/master/optim.html). Save the chosen loss function as `criterion_transfer`, and the optimizer as `optimizer_transfer` below.

In [0]:

```
criterion_transfer = nn.CrossEntropyLoss()
optimizer_transfer = optim.SGD(model_transfer.fc.parameters(), lr=0.001)
```

(IMPLEMENTATION) Train and Validate the Model

Train and validate your model in the code cell below. Save the final model parameters (http://pytorch.org/docs/master/notes/serialization.html) at filepath `'model_transfer.pt'`.

In [0]:

```
# train the model
# train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer, criterion_trans
fer, use_cuda, 'model_transfer.pt')

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

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

        # train the model #

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

            # initialize weights to zero
            optimizer.zero_grad()
            output = model(data)

            # calculate loss
            loss = criterion(output, target)

            # back prop
            loss.backward()

            # grad
            optimizer.step()
            train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss
))
            if batch_idx % 100 == 0:
                print('Epoch %d, Batch %d loss: %.6f' %
                  (epoch, batch_idx + 1, train_loss))

        # validate the model #

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


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

```
            ))

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

    # return trained model
    return model
```

In [51]:

```
train(20, loaders_transfer, model_transfer, optimizer_transfer, criterion_transfer, use
_cuda, 'saved_models/model_transfer.pt')
```

```
Epoch 1, Batch 1 loss: 4.929161
Epoch 1, Batch 101 loss: 4.900809
Epoch 1, Batch 201 loss: 4.858445
Epoch 1, Batch 301 loss: 4.824739
Epoch: 1        Training Loss: 4.815086         Validation Loss: 4.621809
Validation loss decreased (inf --> 4.621809).  Saving model ...
Epoch 2, Batch 1 loss: 4.769074
Epoch 2, Batch 101 loss: 4.650020
Epoch 2, Batch 201 loss: 4.624834
Epoch 2, Batch 301 loss: 4.604022
Epoch: 2        Training Loss: 4.594440         Validation Loss: 4.370060
Validation loss decreased (4.621809 --> 4.370060).  Saving model ...
Epoch 3, Batch 1 loss: 4.566092
Epoch 3, Batch 101 loss: 4.456629
Epoch 3, Batch 201 loss: 4.428648
Epoch 3, Batch 301 loss: 4.401107
Epoch: 3        Training Loss: 4.395100         Validation Loss: 4.146312
Validation loss decreased (4.370060 --> 4.146312).  Saving model ...
Epoch 4, Batch 1 loss: 4.303094
Epoch 4, Batch 101 loss: 4.267818
Epoch 4, Batch 201 loss: 4.245560
Epoch 4, Batch 301 loss: 4.221259
Epoch: 4        Training Loss: 4.215495         Validation Loss: 3.920723
Validation loss decreased (4.146312 --> 3.920723).  Saving model ...
Epoch 5, Batch 1 loss: 4.138981
Epoch 5, Batch 101 loss: 4.102736
Epoch 5, Batch 201 loss: 4.080096
Epoch 5, Batch 301 loss: 4.050936
Epoch: 5        Training Loss: 4.042158         Validation Loss: 3.701551
Validation loss decreased (3.920723 --> 3.701551).  Saving model ...
Epoch 6, Batch 1 loss: 3.741824
Epoch 6, Batch 101 loss: 3.920786
Epoch 6, Batch 201 loss: 3.898877
Epoch 6, Batch 301 loss: 3.882885
Epoch: 6        Training Loss: 3.866145         Validation Loss: 3.505000
Validation loss decreased (3.701551 --> 3.505000).  Saving model ...
Epoch 7, Batch 1 loss: 3.668272
Epoch 7, Batch 101 loss: 3.753684
Epoch 7, Batch 201 loss: 3.739750
Epoch 7, Batch 301 loss: 3.715562
Epoch: 7        Training Loss: 3.709020         Validation Loss: 3.299601
Validation loss decreased (3.505000 --> 3.299601).  Saving model ...
Epoch 8, Batch 1 loss: 3.874454
Epoch 8, Batch 101 loss: 3.617947
Epoch 8, Batch 201 loss: 3.594281
Epoch 8, Batch 301 loss: 3.570467
Epoch: 8        Training Loss: 3.562160         Validation Loss: 3.107775
Validation loss decreased (3.299601 --> 3.107775).  Saving model ...
Epoch 9, Batch 1 loss: 3.446600
Epoch 9, Batch 101 loss: 3.443163
Epoch 9, Batch 201 loss: 3.448222
Epoch 9, Batch 301 loss: 3.424186
Epoch: 9        Training Loss: 3.426218         Validation Loss: 2.964848
Validation loss decreased (3.107775 --> 2.964848).  Saving model ...
Epoch 10, Batch 1 loss: 2.913977
Epoch 10, Batch 101 loss: 3.341584
Epoch 10, Batch 201 loss: 3.326594
Epoch 10, Batch 301 loss: 3.297400
Epoch: 10       Training Loss: 3.285944         Validation Loss: 2.801104
Validation loss decreased (2.964848 --> 2.801104).  Saving model ...
Epoch 11, Batch 1 loss: 3.514516
```

```
Epoch 11, Batch 101 loss: 3.188645
Epoch 11, Batch 201 loss: 3.185540
Epoch 11, Batch 301 loss: 3.167111
Epoch: 11       Training Loss: 3.165445         Validation Loss: 2.657069
Validation loss decreased (2.801104 --> 2.657069).  Saving model ...
Epoch 12, Batch 1 loss: 3.005798
Epoch 12, Batch 101 loss: 3.048082
Epoch 12, Batch 201 loss: 3.050039
Epoch 12, Batch 301 loss: 3.038995
Epoch: 12       Training Loss: 3.042513         Validation Loss: 2.521835
Validation loss decreased (2.657069 --> 2.521835).  Saving model ...
Epoch 13, Batch 1 loss: 3.518283
Epoch 13, Batch 101 loss: 2.966122
Epoch 13, Batch 201 loss: 2.956480
Epoch 13, Batch 301 loss: 2.933238
Epoch: 13       Training Loss: 2.930891         Validation Loss: 2.380705
Validation loss decreased (2.521835 --> 2.380705).  Saving model ...
Epoch 14, Batch 1 loss: 2.734737
Epoch 14, Batch 101 loss: 2.878083
Epoch 14, Batch 201 loss: 2.863528
Epoch 14, Batch 301 loss: 2.838707
Epoch: 14       Training Loss: 2.838131         Validation Loss: 2.268182
Validation loss decreased (2.380705 --> 2.268182).  Saving model ...
Epoch 15, Batch 1 loss: 2.844709
Epoch 15, Batch 101 loss: 2.785439
Epoch 15, Batch 201 loss: 2.776927
Epoch 15, Batch 301 loss: 2.750421
Epoch: 15       Training Loss: 2.745883         Validation Loss: 2.139792
Validation loss decreased (2.268182 --> 2.139792).  Saving model ...
Epoch 16, Batch 1 loss: 3.075635
Epoch 16, Batch 101 loss: 2.673582
Epoch 16, Batch 201 loss: 2.675809
Epoch 16, Batch 301 loss: 2.665081
Epoch: 16       Training Loss: 2.656976         Validation Loss: 2.048191
Validation loss decreased (2.139792 --> 2.048191).  Saving model ...
Epoch 17, Batch 1 loss: 2.743237
Epoch 17, Batch 101 loss: 2.600304
Epoch 17, Batch 201 loss: 2.599135
Epoch 17, Batch 301 loss: 2.574713
Epoch: 17       Training Loss: 2.568263         Validation Loss: 1.968246
Validation loss decreased (2.048191 --> 1.968246).  Saving model ...
Epoch 18, Batch 1 loss: 2.432633
Epoch 18, Batch 101 loss: 2.526377
Epoch 18, Batch 201 loss: 2.496937
Epoch 18, Batch 301 loss: 2.497822
Epoch: 18       Training Loss: 2.501013         Validation Loss: 1.886920
Validation loss decreased (1.968246 --> 1.886920).  Saving model ...
Epoch 19, Batch 1 loss: 2.445440
Epoch 19, Batch 101 loss: 2.412485
Epoch 19, Batch 201 loss: 2.433620
Epoch 19, Batch 301 loss: 2.424039
Epoch: 19       Training Loss: 2.422026         Validation Loss: 1.808883
Validation loss decreased (1.886920 --> 1.808883).  Saving model ...
Epoch 20, Batch 1 loss: 2.298903
Epoch 20, Batch 101 loss: 2.379572
Epoch 20, Batch 201 loss: 2.342648
Epoch 20, Batch 301 loss: 2.358676
Epoch: 20       Training Loss: 2.355334         Validation Loss: 1.762199
Validation loss decreased (1.808883 --> 1.762199).  Saving model ...
```

Out[51]:

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3,
3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, cei
l_mode=False)
  (layer1): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=Fals
e)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_r
unning_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_r
unning_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=Fal
se)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=Fals
e)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=Fal
se)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_r
unning_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_r
unning_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=Fal
se)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=Fal
se)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_r
unning_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_r
unning_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=Fal
se)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
      (relu): ReLU(inplace=True)
```

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

```
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=Fa
lse)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (layer3): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=Fa
lse)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding
=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=F
alse)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track
_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2), bias=Fal
se)
        (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track
_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=F
alse)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding
=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=F
alse)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track
_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=F
alse)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding
=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=F
alse)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track
_running_stats=True)
      (relu): ReLU(inplace=True)
    )
```

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

```
      (downsample): Sequential(
        (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=Fa
lse)
        (1): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track
_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=F
alse)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding
=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=F
alse)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track
_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=F
alse)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding
=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=F
alse)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track
_running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=2048, out_features=133, bias=True)
)
```

In [52]:

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

Out[52]:

```
<All keys matched successfully>
```

(IMPLEMENTATION) Test the Model

Try out your model on the test dataset of dog images. Use the code cell below to calculate and print the test loss and accuracy. Ensure that your test accuracy is greater than 60%.

In [53]:

```
test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 1.839142


Test Accuracy: 73% (617/836)
```

(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 [0]:

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

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

In [55]:

```
loaders_transfer['train'].dataset.classes[:10]
```

Out[55]:

```
['001.Affenpinscher',
 '002.Afghan_hound',
 '003.Airedale_terrier',
 '004.Akita',
 '005.Alaskan_malamute',
 '006.American_eskimo_dog',
 '007.American_foxhound',
 '008.American_staffordshire_terrier',
 '009.American_water_spaniel',
 '010.Anatolian_shepherd_dog']
```

In [56]:

```
class_names[:10]
```

Out[56]:

```
['Affenpinscher',
 'Afghan hound',
 'Airedale terrier',
 'Akita',
 'Alaskan malamute',
 'American eskimo dog',
 'American foxhound',
 'American staffordshire terrier',
 'American water spaniel',
 'Anatolian shepherd dog']
```

In [0]:

```
from PIL import Image
import torchvision.transforms as transforms

def load_input_image(img_path):
    image = Image.open(img_path).convert('RGB')
    prediction_transform = transforms.Compose([transforms.Resize(size=(224, 224)),
                                     transforms.ToTensor(),
                                     standard_normalization])

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

In [0]:

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

In [59]:

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

```
image_file_name: ./images/Curly-coated_retriever_03896.jpg,      predition
breed: Curly-coated retriever
image_file_name: ./images/Labrador_retriever_06455.jpg,          predition
breed: Chesapeake bay retriever
image_file_name: ./images/Labrador_retriever_06449.jpg,          predition
breed: Flat-coated retriever
image_file_name: ./images/Labrador_retriever_06457.jpg,          predition
breed: Golden retriever
image_file_name: ./images/American_water_spaniel_00648.jpg,      predition
breed: Curly-coated retriever
image_file_name: ./images/Brittany_02625.jpg,    predition breed: Brittany
image_file_name: ./images/sample_cnn.png,        predition breed: Brussels
griffon
image_file_name: ./images/Welsh_springer_spaniel_08203.jpg,      predition
breed: Welsh springer spaniel
image_file_name: ./images/sample_dog_output.png,         predition breed:
Greyhound
image_file_name: ./images/sample_human_output.png,       predition breed:
Bulldog
```

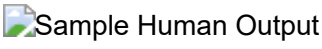
Step 5: Write your Algorithm

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!

 Sample Human Output

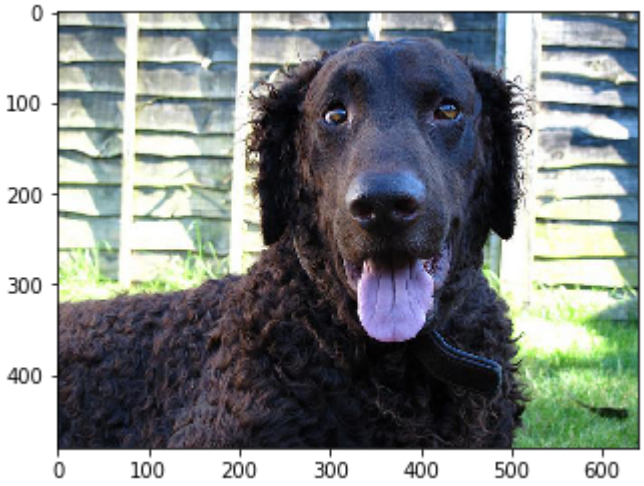
(IMPLEMENTATION) Write your Algorithm

In [0]:

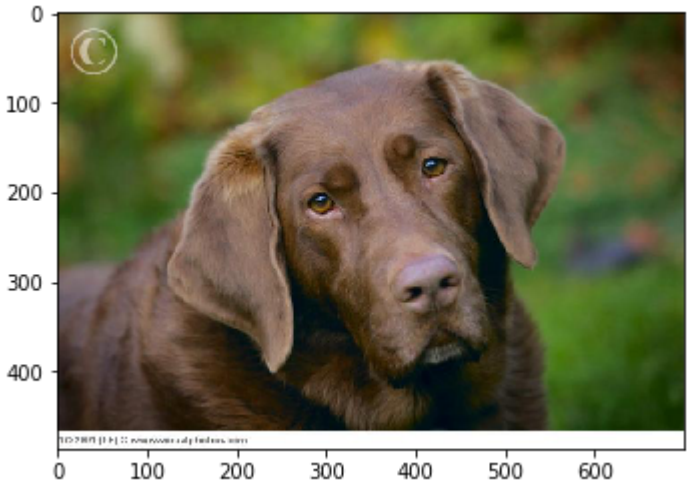
```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.

def run_app(img_path):
    ## handle cases for a human face, dog, and neither
    img = Image.open(img_path)
    plt.imshow(img)
    plt.show()
    if dog_detector(img_path) is True:
        prediction = predict_breed_transfer(model_transfer, class_names, img_path)
        print("Dogs Detected!\nIt looks like a {0}".format(prediction))
    elif face_detector(img_path) > 0:
        prediction = predict_breed_transfer(model_transfer, class_names, img_path)
        print("Hello, human!\nIf you were a dog..You may look like a {0}".format(predic
tion))
    else:
        print("Error! Can't detect anything..")
```

In [61]:

```
for img_file in os.listdir('./images'):
    img_path = os.path.join('./images', img_file)
    run_app(img_path)
```

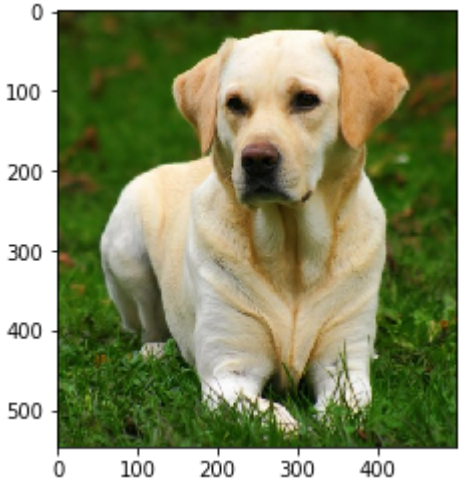


Dogs Detected!
It looks like a Curly-coated retriever



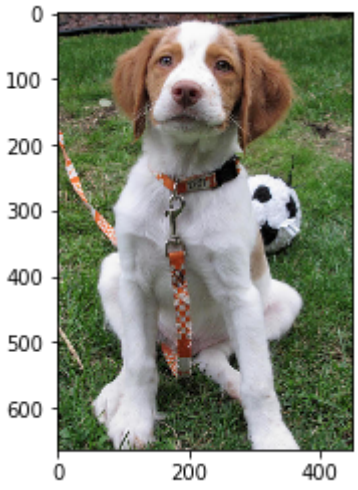
Dogs Detected!
It looks like a Chesapeake bay retriever

```
Dogs Detected!
It looks like a Flat-coated retriever
```

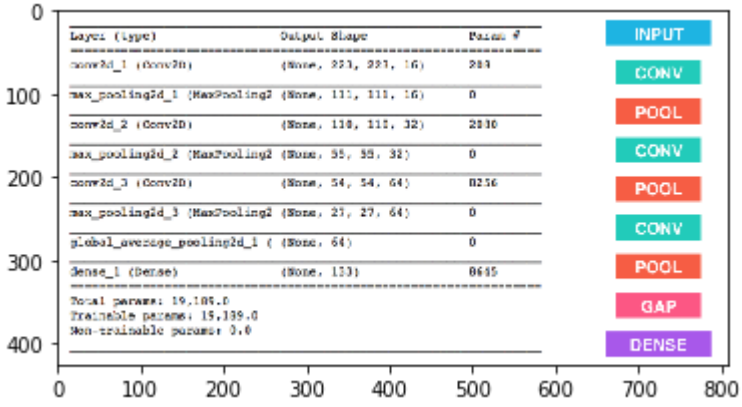



```
Dogs Detected!
It looks like a Golden retriever
```

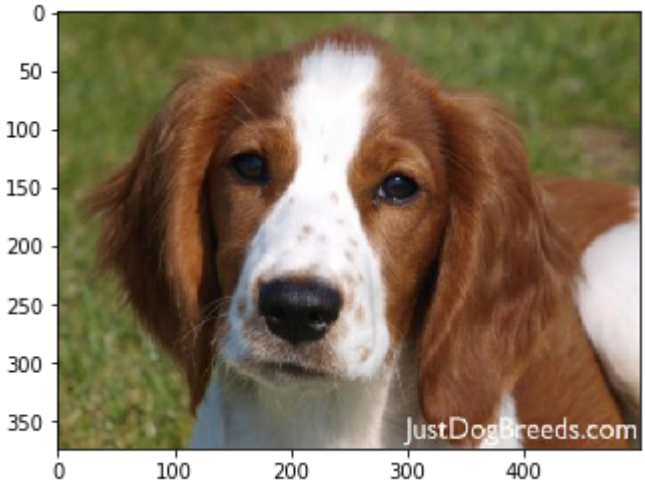

```
Dogs Detected!
It looks like a Curly-coated retriever
```



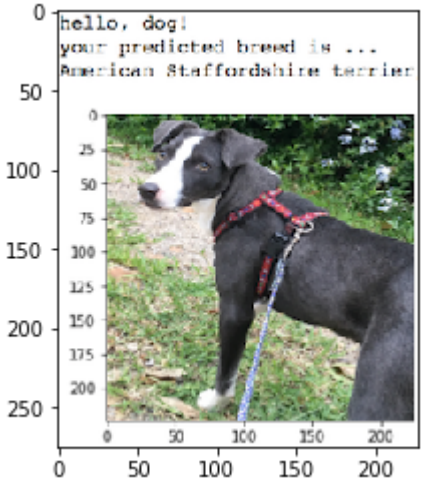
```
Dogs Detected!
It looks like a Brittany
```



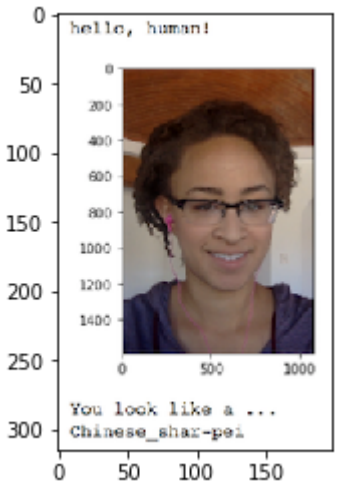

```
Error! Can't detect anything..
```




```
Dogs Detected!
It looks like a Welsh springer spaniel
```




```
Dogs Detected!
It looks like a Greyhound
```




```
Hello, human!
If you were a dog..You may look like a Bulldog
```

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?

(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: (Four possible points for improvement)

1. More image datasets of dogs will improve training models. Also, more
2. Image augmentations trials (flipping vertically, move left or right, cropping, padding etc.) will improve performance on test data.
3. Hyper-parameter tunings: weight initializings, learning rates, drop-outs, batch_sizes, and optimizers will be helpful to improve performances.
4. Ensembles of models

In [0]:

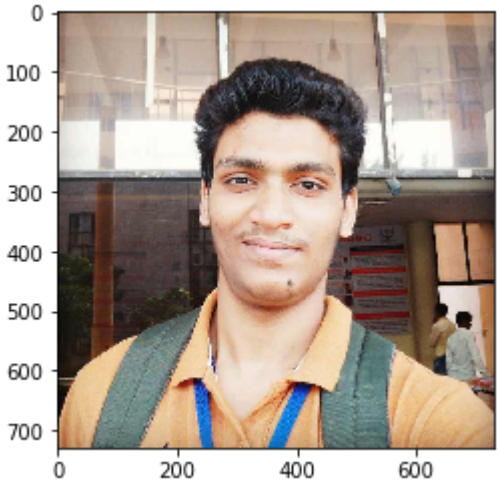
```
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.
```

In [0]:

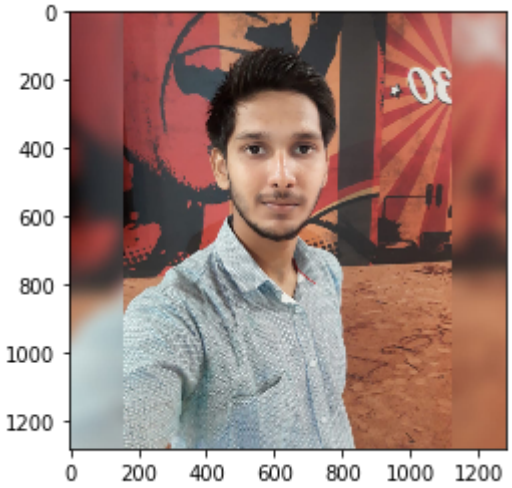
```
my_human_files = ['./my_images/govind.jpg', './my_images/gaurav.jpg', './my_images/chet
an.jpg']
my_dog_files = ['./my_images/german.jpg', './my_images/dog_yorkshire.jpg', './my_image
s/dog_retreiver.jpg']
```

In [102]:

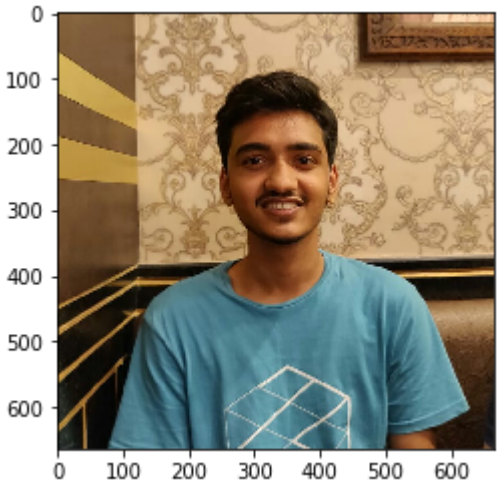
```
## suggested code, below
for file in np.hstack((my_human_files, my_dog_files)):
    run_app(file)
```



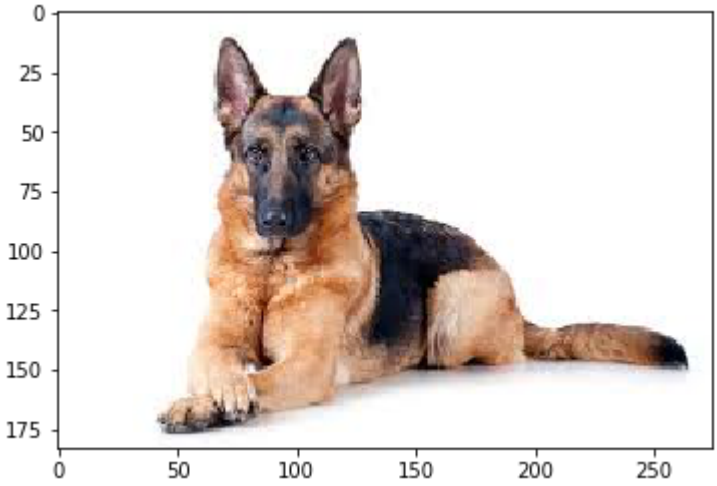

```
Hello, human!
If you were a dog..You may look like a Chinese crested
```




```
Hello, human!
If you were a dog..You may look like a Greyhound
```

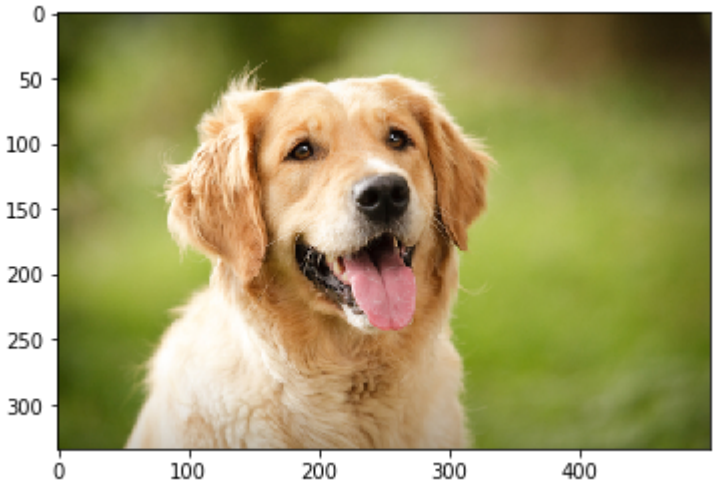



```
Hello, human!
If you were a dog..You may look like a Greyhound
```



Dogs Detected!
It looks like a German shepherd dog

Dogs Detected!
It looks like a Australian terrier



Dogs Detected!
It looks like a Golden retriever